Bilkent University

Department of Computer Engineering

CS319 Term Project

3A - Monopoly

Design Report

Ali Taha Dinçer, İrem Ecem Yelkanat, Kutsal Bekçi, Saidcan Alemdaroğlu, Sultan Simge Kürkçü

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

Progress/Iteration 2 - Project Design Report
Dec 13, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Software Engineering course CS319.

Contents

1 Introduction

1.1 Purpose of the System

Monopoly is a square shaped board game where each player has a unique pawn and in each turn according to dice results they move their pawns. The players try to have properties and with time, they are able to build structures on their owned properties. The main purpose of the Monopoly game is to avoid going bankrupt which means losing all the money and the properties. Having structures on the properties will help the players have an income so that they won't go bankrupt easily. The game can be played with at least 2 and at most 8 players.

There exists two same shaped card allocations on the board, which are located for chance cards and community chest cards. These cards are drawn by the players and increases the joy of the game. There are different buyable property squares located along the board. The game starts when the dice are rolled and finishes when every player except one gets bankrupt.

In this project, the purpose is to implement Monopoly into the digital single player and multiplayer platform. The name of the digital version of the game implemented is "Yolopoly". The players can play against the computer to enhance their skills and clash with their friends in multiplayer game mode.

1.2 Design Goals

1.2.1 Usability

A usable program requires performing any task safely, effectively and efficiently. Since Yolopoly has various tasks in the playing process, it is quite sufficient to carry out events in the smallest time and space for the sake of usability. In addition, as the base game Monopoly is a rather complex game, it is aimed to have an expressive and easy to use UI for the players who want to play the game. Furthermore, players can learn the game by clicking the "How To Play" button in the main menu and the whole game can be playable with only a mouse or a trackpad.

1.2.2 Reliability

Functioning without failure is one of the most important points of Yolopoly game since there are accountings and specific calculations in the game as a great importance. Each calculation in states and backend processes need to be saved in case of any system failure. For online game options to be added to the game, the game Yolopoly will not require any private user data except a nickname.

1.2.3 Performance

The response time matters in terms of the performance of the game. In order for players to play Yolopoly without any tardiness, performance is taken into account. The data flow in the game needs to be efficient to prevent latency in the game to consider the game high performance. As only the data created by the In Game Manager is saved in the storage system, it is planned to have only 250 mb of space stored on the disk including the game itself. Furthermore, JVM options for running the game will be optimized and the game Yolopoly is planned to use only 512 mb of RAM storage. For planned online gameplay, response time of each move done by the players and sending a request to the Firebase server will not exceed 1 seconds. Each response and request will contain only the values used in In Game Manager. Therefore, the game will be playable with only a 1 Mbps bandwidth or higher.

1.2.4 Supportability

An extendable system requires the ability for modification or addition to the program without interrupting the dataflow of the program. Since Yolopoly has some sort of ground structure which shouldn't be changed, alterations and/or additions demanded can be applied easily. In addition, as stated in the section above, the game will require only 250 mb of storage from the disk and 512 mb of RAM from the computer. For online gameplay option, Yolopoly can be played with only 1 Mbps bandwidth. Furthermore, the game is implemented on Java 15 and can be played with any JRE that supports Java 7 and higher versions. Hence, the game will have extensive supportability.

1.2.5 Cost

In terms of development time, we want our system to have low cost. Since the time we have for the design and the implementation of the project is limited, we don't want our project to have complicated design that is hard to handle. In addition, since we have no budget for this project, we will not use any tools, database or software that will cost us.

1.2.6 Modifiability

We want our design to be easily modifiable. If a functionality will be changed or a new functionality will be added, it must be acclimated with ease. By making our game modifiable, making new changes will not require the current design to be changed extensively and the process will be less demanding.

1.2.7 Maintainability

We want our design to be easily maintainable. Our design should be repaired, improved and understood easily. If a new change will be performed to our design, it should not cause an error in other parts of the design. In addition, we want to find out the cause of a problem in a short amount of time, and the process will be less challenging.

1.2.8 Functionality

Monopoly is a game that has several rules and features, where the player can perform various actions. We tried to make our design to have all of the features, tried to add extra features to keep up the enthusiasm of the players regarding the game, and tried not to discard the features of the original game.

2 High Level Software Architecture

2.1 Subsystem Decomposition

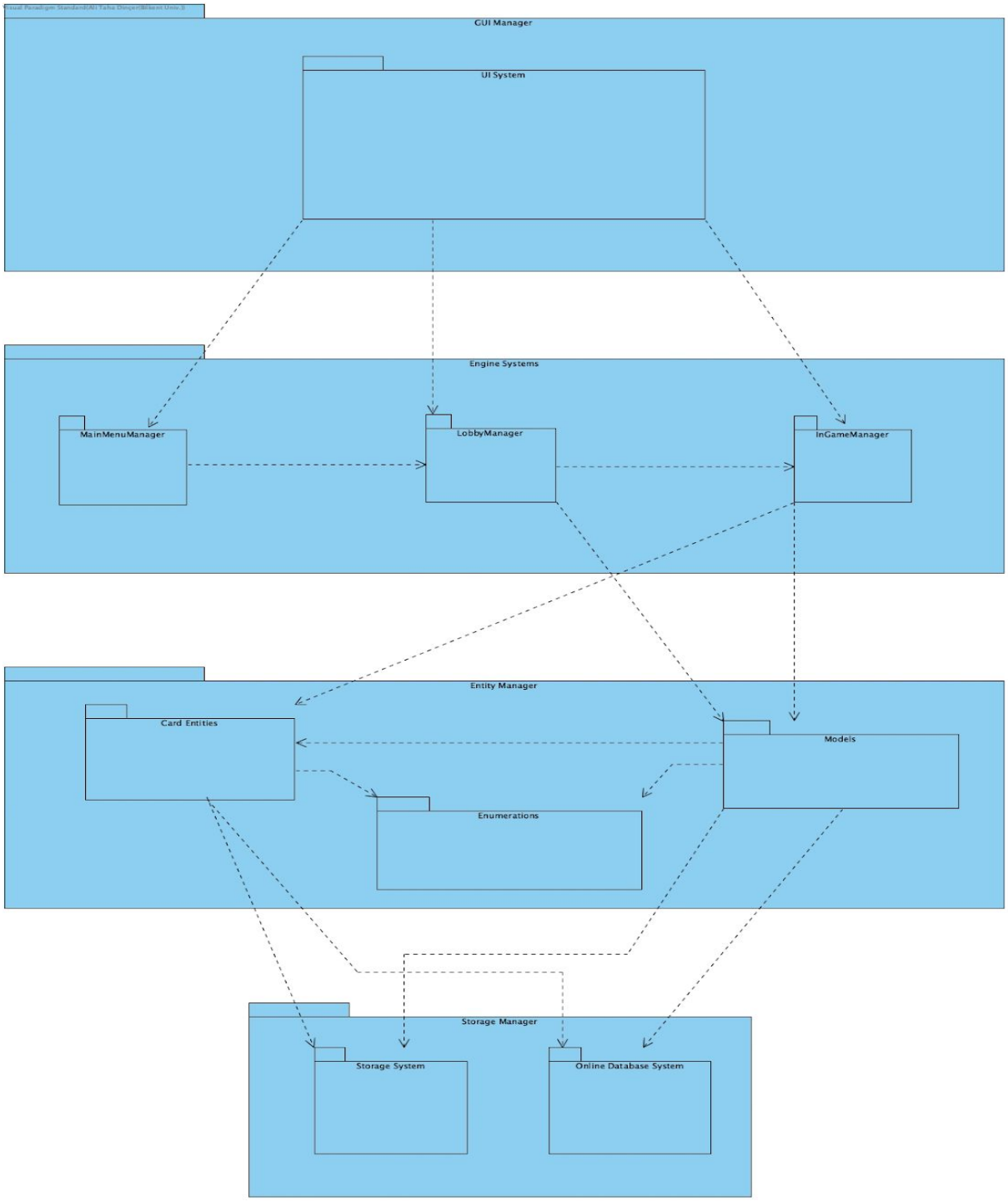

Figure 1: Subsystem Diagram

The system uses a four layer subsystem architecture and hence, is composed of four subsystems. The GUI Manager system deals with the player's navigation from the opening of the game to the playing process. According to coordination of the player the user interface connects to the Engine System and the operations are handled in the background in different types of managers in the Engine System. The Entity Manager is composed of entity objects in the system and controlled by the Engine System in the upper level. The Storage System deals with the storage of the information and online realtime database integrity in the program. Entities load and store data with the help of the Storage System.

2.2 Hardware/Software Mapping

Minimum version of Java 8 required in order to play the game. For the storage system, JSON will be used and for real time database integrity, Firebase will be used. The game will run on the players computer.

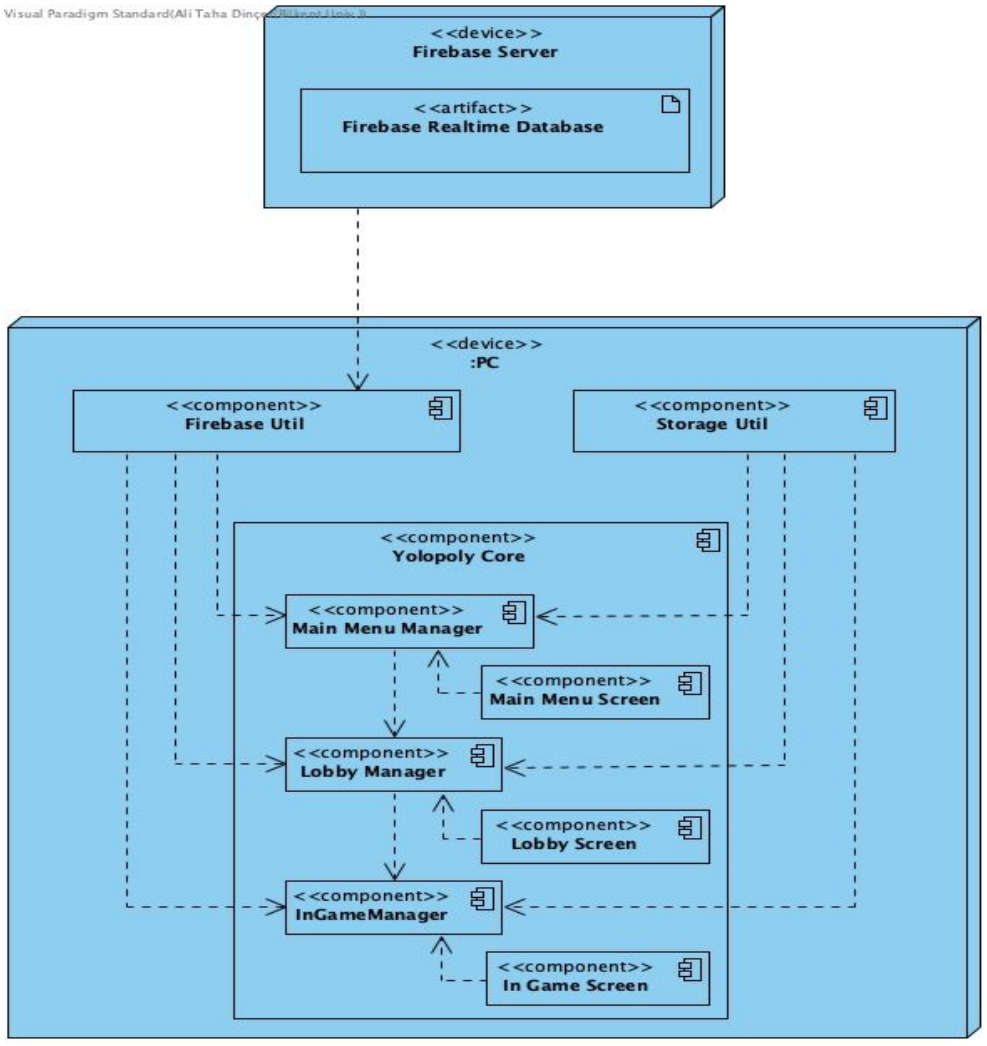

Figure 2: Hardware/Software Mapping

2.3 Persistent Data Management

The game Yolopoly is the digital implementation of the board game Monopoly. Hence, the game is meant to be multiplayer itself. Therefore, we planned to add an online multiplayer future to the game. However, in the first milestone of the project, it is decided to finish the single-player option of the game. Hence, if succeeded in single-player gameplay, it is planned that Google Firebase will be used to implement persistent data management in the game. Firebase serves a secure real-time database option, and Yolopoly is a turn based game. Hence, there will be no conflict or error during reading from and writing to the online database.

Considering the single player game option, the game will be played against the bots and the only human user is the player himself/herself. The implementation of the game relies on a state management system. Hence, every turn is handled with states and in each turn, data of the current game will be saved in a local storage system. This system will use the JSON files in order to store game data. In addition, there will be no conflict or error during writing to or reading from the inner storage system with the help of the exception handling system.

In both gameplay options, players are limited with the themes that served with the game. Hence, each image and audio resource used on the game is available on the PC of the players. In the online gameplay system, only the data in the In Game Manager will be transferred to the database system. Hence, for the online game option, each player will share the same In Game Manager system.

In addition to the information given above, the game will use different JSON files to load and store the saved games and settings data such as last played single player game data or music and audio settings data. The data of the saved games consists of all the information used in the In Game Manager system.

2.4 Access Control and Security

The first milestone during the implementation of the game Yolopoly, is to finish the single-player game option. As the game will be single-player during the next milestone, the game will be safe in means of any internet based security vulnerabilities.

In the online multiplayer game option, the Google Firebase real-time database will be used. For the online real-time database system, Google Firebase is selected because this system is trusted by users and maintained by Google, which is a huge and trusted company. As Firebase is maintained by Google and Firebase itself serves

a rule system for database systems, we are planning to write security related rules to avoid any internet based security vulnerabilities to the online database system.

2.5 Boundary Conditions

2.5.1 Initialization

The game will be shipped in a ".jar" file. Hence, any computer who has JRE (which supports Java 7+) installed can start the game by double clicking the game button. The initialization of the game can be divided into three sections.

The first section is the Main Menu section which is managed by the MainMenuManager system. This manager initializes settings of the game (JSON file including saved audio settings), instructions of the game (Images of the gameplay instructions of the Yolopoly) and "about us" information (JSON file including informatiın about developers).

The second section is the Game Settings-Lobby section which is managed by the Lobby Manager system when the player clicks either Single Player button or Multiplayer button in the Main Menu section. This manager initializes the Single Player settings which are bot count, pawns, mod and theme of the game and Multiplayer settings which are maximum player count, pawn for individual user, game mod and game theme. In both gameplay types, Lobby Manager transfers the mod of the game and data of the players including an order to the In Game Manager.

Third and last section is the In Game section, which is managed by the In Game Manager system when the player launches the game in the Game Settings-Lobby sections. This manager has a huge workload about initializing the game. The game board including each square, properties such as drawable cards and buyable cards, dice and bank is initialized in this section. For online multiplayer game options, as each player starts equally, only the In Game Manager data of the host of the game is transferred to the online database system and In Game Managers of the other players in the game is initialized from the data sent to the online database system.

By dividing initialization steps to different managers, we promise to achieve a better performance on the game by lowering the extensive use of system resources.

2.5.2 Save and Load

The game automatically saves the game at the end of each turn in the In-Game screen and saving and loading the game is only allowed for single-player games.

Users can save and exit from the game by clicking the corresponding button during the game and continue playing the game by clicking the according tile which includes game date, game mod and game theme. The game saves and loads only the data created and used by the In Game Manager system. The saved games are stored in a directory named "saves" and each save file is in JSON file format.

2.5.3 Termination

Direct termination of the game is possible by clicking the "close" button according to different operating systems but this is not a safe operation. In the Main Menu screen, players can safely exit from the game by clicking the Quit Game button. ın addition, players can terminate the operations of each manager. The operations of the In Game Manager can be terminated by exiting from the In-Game screen. The operations of Lobby Manager can be terminated by clicking the Return to Main Menu button. As explained, Main Menu Manager is terminated by clicking the Quit Game button in the Main Menu screen.

Terminating the managers by using in game buttons is the safest way as we aim to save every action to provide better reliability. However, we also aim to provide reliability when the "close" button provided by the operating system is pressed.

The importance of this "close" button is, in the online gameplay option, players can choose to close the game by pressing that button and hence the In Game Manager will not know that this player has exited or making decisions. Hence, only in the online game option, the In Game Manager will wait for 1 minute until the player does any action. If not, the corresponding player will be banned from the game and replaced with a bot player. By this, we aim to provide better security and reliability.

2.5.4 Exceptions and System Failures

During playing the game, it is possible to encounter errors and it is important to solve these issues before players have encountered them. In our implementation, we aim to provide a highly reliable system by using the exception handling system as best as possible. However, there are still some failures that can occur related to internet connection and operating systems. In case of poor or no network connection, we disconnect the according player from the game by checking if the user is doing anything in 1 minute period. In case of any inconsistency or corruption in the name of data, the corresponding manager will terminate itself and return an error which will be displayed to the player. The game is shipped in a ".jar" file, therefore there will be no exception or failure related to the resources such as images, audios or FXML files which are related to the UI of the game.

3 Subsystem Services

3.1 Entity-Model Subsystem

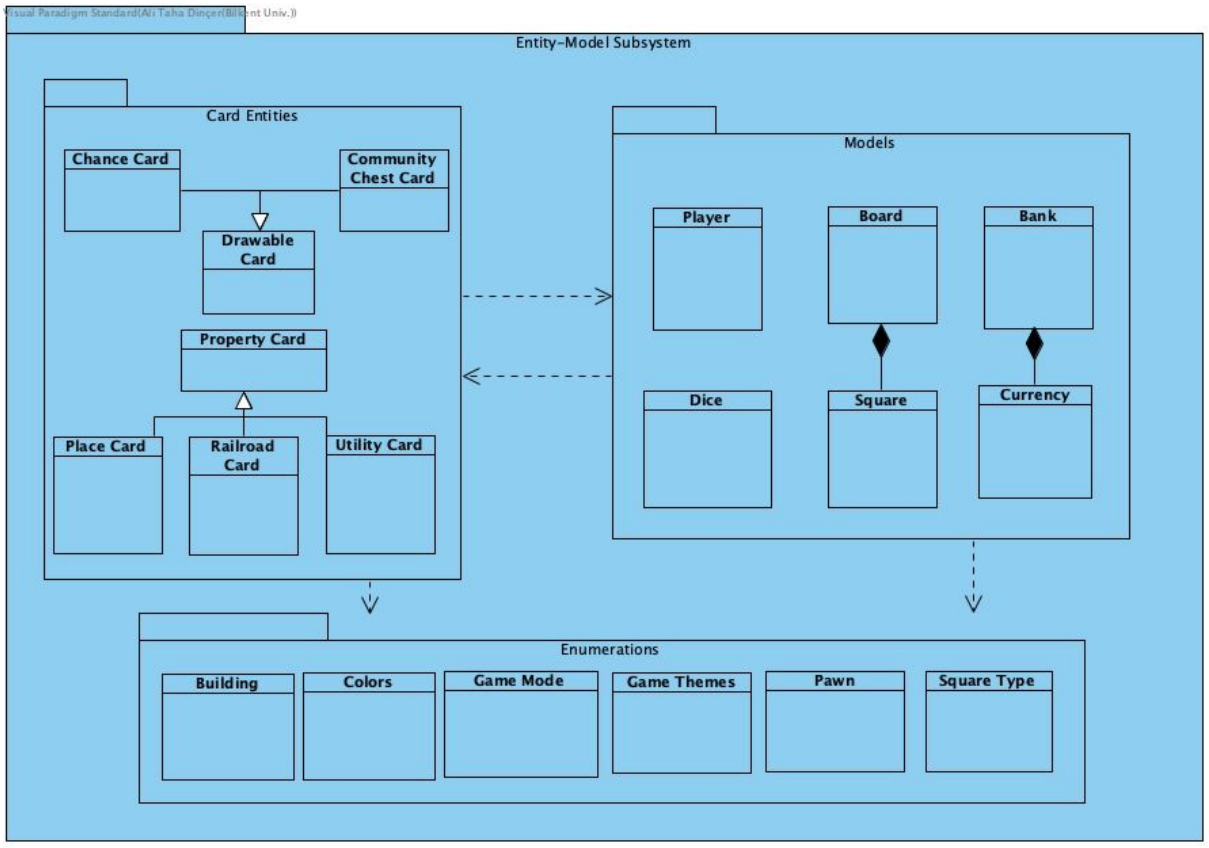

Figure 3: Entity-Model Subsystem Diagram

The Entity-Model Subsystem is divided into three main parts.

First part of the Entity-Model diagram contains the classes of the Drawable Card and Property Card. These classes are abstract and Drawable Card is associated with Chance Card and Community Chest Card classes and Property Card is associated with Place Card, Railroad Card and Utility Card classes.

The second part of the Entity-Model diagram contains base models of the game. These model classes are Player, Board, Square, Dice, Currency and Bank. In this package of the diagram, as the Board class consists of Squares, there is an association between these two classes and for only the Bankman Mode of the game, Bank class has currencies, hence, associated with the Currency class.

The third and last part of the Entity Model diagram is Enumerations. This package holds the enumeration classes that makes differentiation easier during the

implementation process of the Yolopoly. The enumerations are Building, Colors, Game Modes, Game Themes, Pawn and Square Type.

3.2 Controller-Engine Subsystem

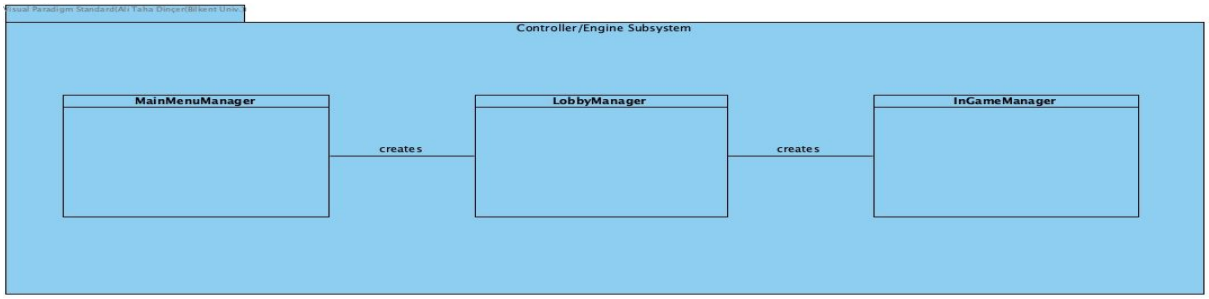

Figure 4: Controller-Engine Subsystem Diagram

The Controller-Engine Subsystem consists of three classes, which are Main Menu Manager, Lobby Manager and In Game Manager. Each of these managers are connected to the corresponding UI system and controls the information flow between models and user interface. Each controller creates the manager in the lower layer in respect to time the screen shown during runtime. Hence, each manager is able to communicate with the higher order manager. For example, before starting the game, an array of players are initialized in the Lobby Manager and the same array is resorted and used by the In Game Manager during the game. Main Menu Manager controls the Main Menu and related screens in the Game. Lobby Manager controls the game setup related screens in the game. Finally, the In Game Manager controls every data, events, functions and exceptions related to the In-Game Screen.

3.3 View Subsystem

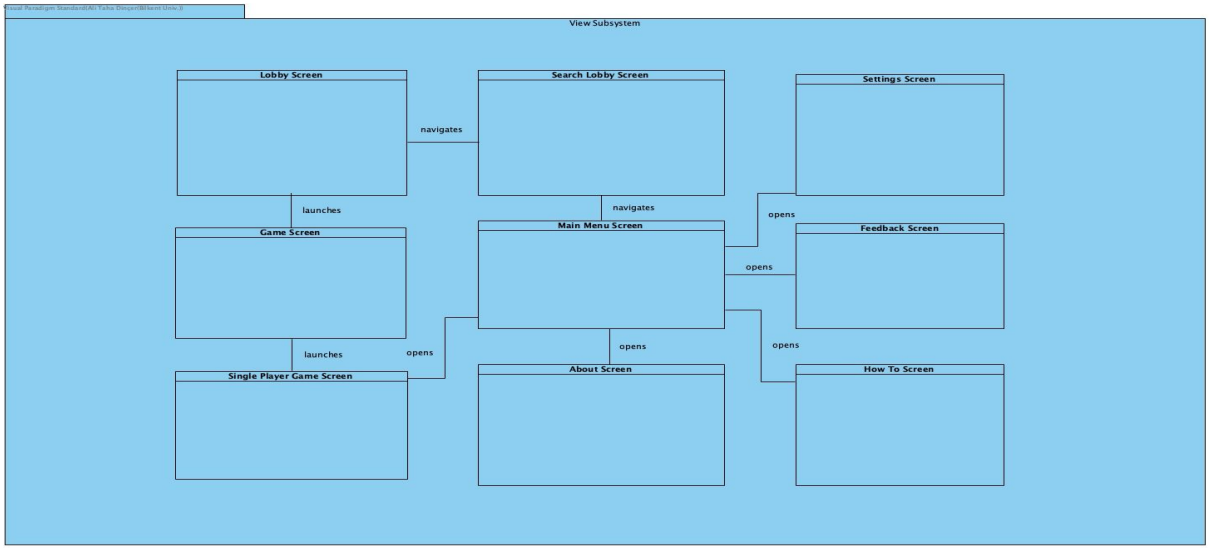

Figure 5: View Subsystem Diagram

The View Subsystem is divided into three systems conceptually. The division is determined by the action. Hence, there are screens that can be opened, screens that can be navigated to and screens that can be launched. In the middle of each class, everything starts with the Main Menu screen. Players can open Settings Screen, Feedback Screen, How To Screen and About Screen in the Main Screen. If the player decides to set up a game, he/she will be navigated to either Search Lobby Screen or Single Player Game Screen. If the player chooses to play a multiplayer game, after the Search Lobby Screen, the player will be navigated to the setup page of the multiplayer game screen, which is Lobby Screen. Finally, after the game setup is complete, Game Screen will be launched.

3.4 Storage Subsystem

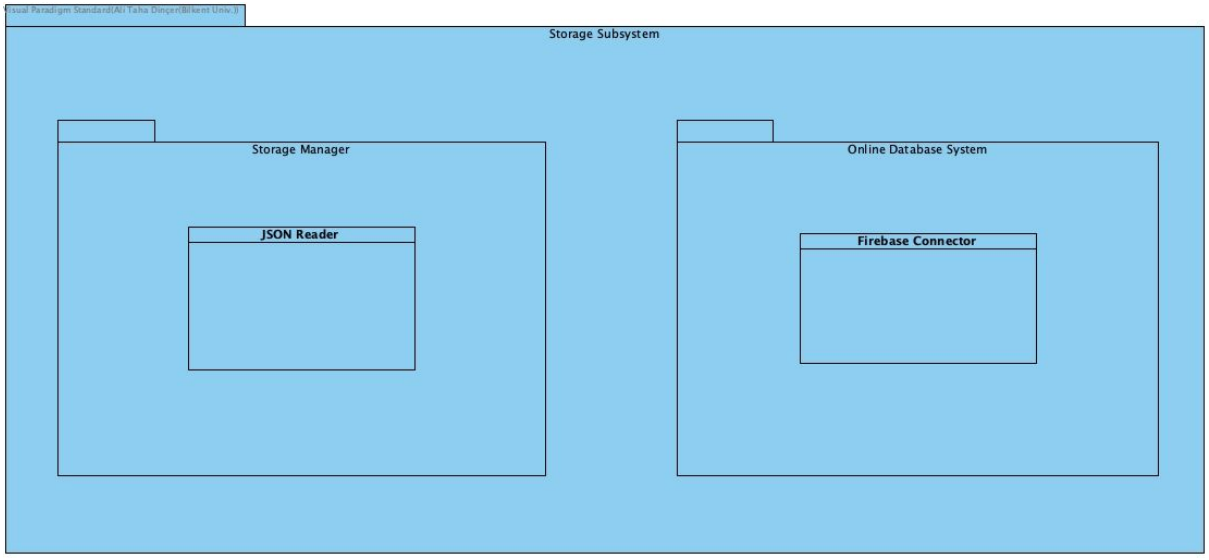

Figure 6: Storage Subsystem Diagram

The final subsystem is the Storage Subsystem. This subsystem is divided into two parts.

The first part is the Storage Manager, which handles the storage related operations such as saving and loading games and changing and saving settings of the game with JSON files. The operations made in that part of the subsystem is local-PC related operations.

The second and final part of this subsystem is Online Database System, which contains Firebase Connector class. As it can be understood, this class handles the communication between game and the online server database system. The operations can be sending requests such as In Game Manager data to the corresponding game lobby area created for the game in the Firebase database and

receiving responses such as changes made in the data of the In Game Manager of the game which is on the Firebase database and retrieving them.

4 Low-level Design

4.1 Object Design Trade-offs

4.1.1 Functionality vs. Cost

Since the time for the design and the implementation of the game is limited, we want the functions of the game as simple as possible. We will design the core logic of the game, however, the extra features that we desired to have in the game can be postponed or discarded. Hence, the time required for the software design and implementation can be reduced.

4.1.2 Usability vs. Security

We will store only the game data in the local file system. The system of the game does not hold any private data of the user other than the nickname of the user, and the system does not have signin & login functionality. Hence, we choose usability over security.

4.1.3 Efficiency vs. Portability

We prefer portability over efficiency as Java can run on any platform or environment provided that JVM is installed on the corresponding platform or environment but Java is slow because of time needed for the JVM operations such as start-up and memory management operations.

4.1.4 Functionality vs. Usability

Even though we are trying to design and implement the software as simple as possible, the Monopoly game has many rules and features, and we didn't want to discard these rules and features of the original game. The game setup screen and game play screen are more complex compared to other screens. However, this will be diminished by the help section with the explanations in "how-to" and guidebook in the help section.

4.2 Class Diagram

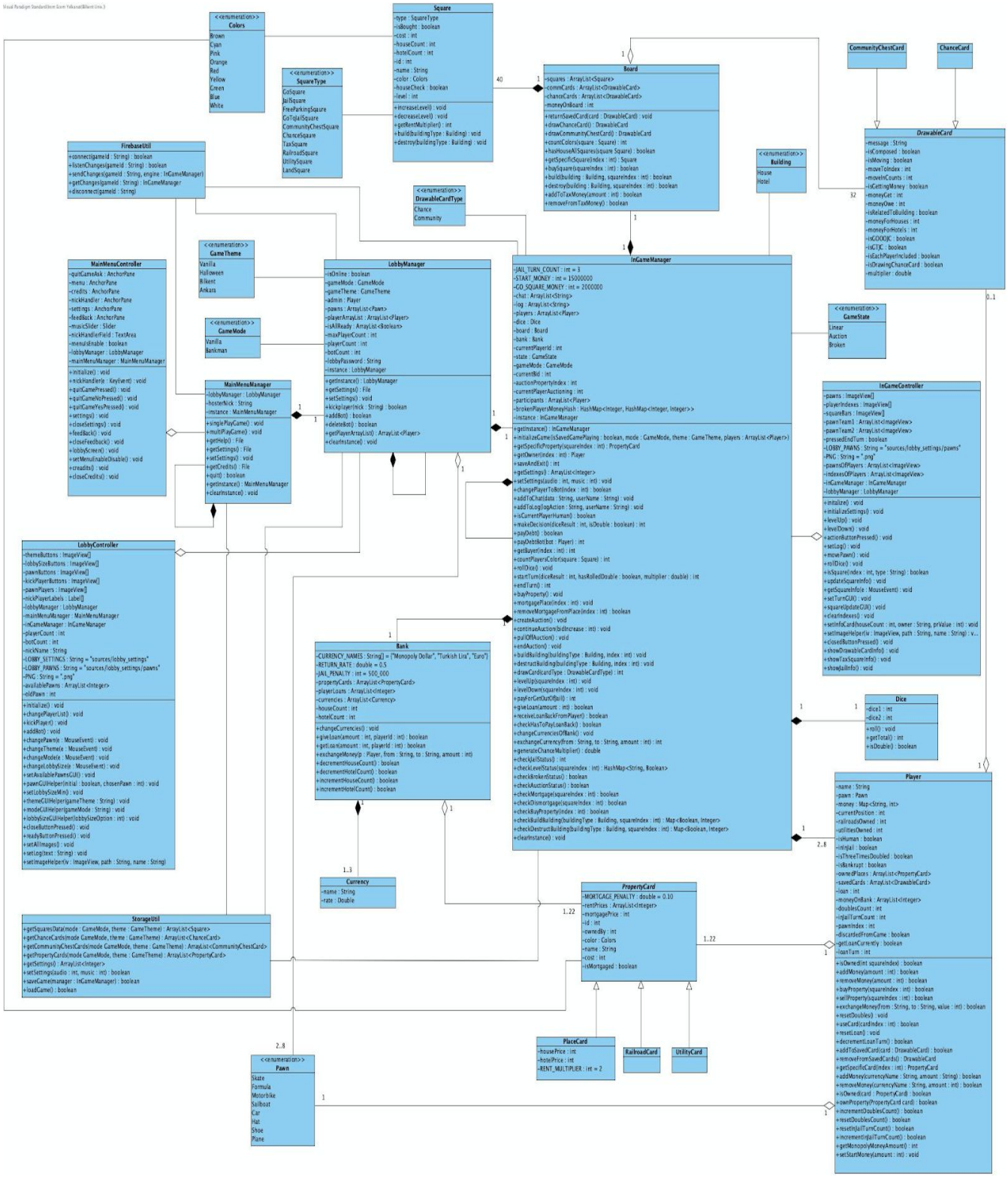

Figure 7: Object-Class Diagram

4.3 Design Patterns

4.3.1 Facade Design Pattern

We used the Facade Design Pattern regarding InGameManager class. As the game itself contains an immersive data flow and control system between the model and manager classes, having an universal controller class to handle this data flow makes the implementation easier regarding the back-end and front-end of the game. Having a single API like codebase enabled us to improve the direct communication between boundary objects and business logic.

4.3.2 Singleton Design Pattern

We used the Singleton Design Pattern regarding MainMenuManager, LobbyManager and InGameManager classes. These classes are designed considering the singleton design pattern as at any instant, there will not be more than one instance of these classes. Making these classes a singleton object makes us able to have greater access to the existing instances of those classes with the getInstance() method.

4.4 Packages

We use JavaFX to build the Graphical User Interface (GUI) of our game. The list of some packages that we use and their brief explanations are listed below.

- javafx.scene: Provides the core set of base classes for the JavaFX Scene Graph API.
- javafx.animation: Provides the set of classes for ease of use transition based animations.
- javafx.fxml: Contains classes for loading an object hierarchy from markup.
- javafx.application: Provides the application life-cycle classes.
- javafx.stage: Provides the top-level container classes for JavaFX content.
- javafx.event: Provides basic framework for FX events, their delivery and handling.
- java.util: Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

4.5 Class Interfaces

4.5.1 Player Class

Figure 8: Player Class Diagram

This class provides the ability to keep track of the player and everything connected to the Player Class. Player Class includes functions for the player actions. The functionality of the class simply directs the game flow.

4.5.1.1 Attributes of Player Class

- private String name: It is used to store the name of the player
- private Pawn pawn: It is used to keep the unique pawn of the player.

- private Map<String, int> money: It is used to keep the money currencies available on the game.
- private int currentPosition: It is used to keep the current position of the player.
- private int railroadsOwned: It is used to keep the number of railroads the player has.
- private int utilitiesOwned: It is used to keep the number of utilities the player owns.
- private boolean isHuman: It is used to differentiate whether the player is a real player or a bot.
- private boolean isInJail: It is used to know whether the player is in the jail section of the board or not.
- private boolean isThreeTimesDoubled: It is used to store whether the player doubled their dice three times or not in a turn.
- private boolean isBankrupt: It is used to store whether the player has gone bankrupt or not.
- private ArrayList<PropertyCard> ownedPlaces: It is used to keep the list of property cards that the player has owned so far.
- private ArrayList<DrawableCard> savedCards: It is used to keep the list of drawable cards that the player has drawn. Used to save "Get Out of Jail" cards.
- private int loan: It is used to keep the amount of money that the player loaned so far.
- private ArrayList<Currency, Integer> moneyOnBank: It is used to keep a list of the amount of money on the player in order to currency type.
- private int doublesCount: It is used to store the number of double dices the player rolled in the round.
- private int inJailTurnCount: It is used to store the number of turns that a player stayed in the jail.
- private int pawnIndex: It is used to keep the index of the pawn.
- private boolean discardedFromGame: It is used to keep track of whether the player is discarded from the game or not.
- private boolean getLoansCurrently: It is used to determine whether the player is getting a loan or not.

- private int loanTurn: It is used to keep track of the number of turns after the player gets a loan since there is a limitation for time to pay back.

4.5.1.2 Methods of Player Class

- public boolean isOwned(int squareIndex): It checks whether the square the player landed is owned by the player or not.
- public boolean addMoney(amount: int): It adds money on the player's bank account according to the actions on the round.
- public boolean removeMoney(amount: int): It removes money from the player's bank account according to the actions on the round.
- public boolean buyProperty(squareIndex: int): It makes the player to own a property in exchange of money on the player's account.
- public boolean sellProperty(squareIndex: int): It makes the player get rid of a property and get a specific amount of money on their account.
- public boolean exchangeMoney(from: String, to: String, value: int): It is used to keep track of an amount of money whether it is exchanged or not.
- public void resetDoubles(): It is used to reset doubles.
- public boolean useCard(cardIndex: int): It is used to use a card.
- public void resetLoan(): It is used to reset the loan of the player.
- public boolean decrementLoanTurn(): It is used to decrement the loan turn.
- public boolean addToSavedCard(Card: DrawableCard): It is used to add the saved cards to a card by the user.
- public DrawableCard removeFromSavedCards(): It is used to remove a card from saved cards list.
- public propertyCard getSpecificCard(index: int): It is used to get a specific property card with its index.
- public boolean addMoney(currencyName: String, amount: String): It is used to add money for the user.
- public boolean removeMoney(currencyName: String, amount: int): It is used to remove money from a user as a result of a purchase or payment.

- public boolean isOwned(card PropertyCard): It keeps track of whether a card is owned by a player or not.
- public boolean ownProperty(PropertyCard card): It is used to keep track of whether the property is owned by a player.
- public boolean incrementDoublesCount(): It is used to increment the count of doubles for the player.
- public boolean resetDoublesCount(): It is used to reset the count of doubles.
- public boolean resetInJailCount(): It is used to reset turn jail count.
- public boolean incrementInJailTurnCount(): It is used to increment the jail turn count by one.
- public int getMonopolyMoneyAmount(): It monitors the monopoly money amount of the player.
- public void setStartMoney(amount : int): It sets the start money of the player.

4.5.2 DrawableCard Class

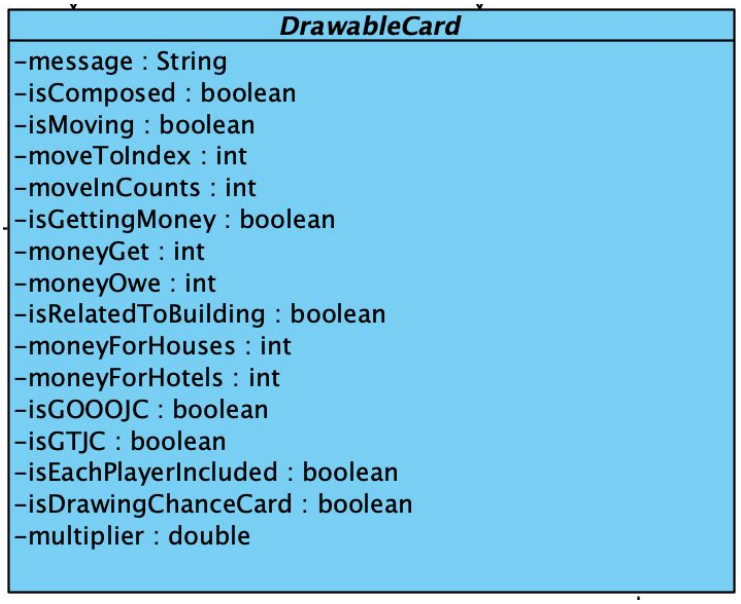

Figure 9: DrawableCard Class Diagram

Everything related to the cards is connected to DrawableCard class.

4.5.2.1 Attributes of DrawableCard Class

- private String message: It is used to store the message or instruction on the drawable card.
- private boolean isComposed: It is used to store the information about whether the card is composed or not.
- private boolean isMoving: It is used to store whether the player is moving or not as a result of drawed card.
- private int moveToIndex: It is used to store whether the player is moving to index or not as a result of card.
- private int moveInCounts: It keeps the counts of moves as a result of the card.
- private boolean isGettingMoney: It is stored to keep the information about whether the player is getting money as a result of the drawed card or not.
- private int moneyGet: It stores the amount of money the player will get as a result of the card.
- private int moneyOwe: It stores the amount of money the player will pay as a result of the card.
- private boolean isRelatedToBuilding: It stores whether the card drawn is related to a building or not.
- private int moneyForHouses: It is used to store the information that whether the player is paying money to the bank related to the houses owned.
- private int moneyForHotels: It is used to store the information that whether the player is paying money to the bank related to the hotels owned.
- private boolean isGOOJC: It is used to keep the information about whether the player picked the card related to getting out of jail or not.
- private boolean isGTJC: It is used to keep the information about whether the player picked the card related to going to jail or not.
- private boolean isEachPlayerIncluded: It is used to keep the information about whether the other players are included according to the card the player picked.
- private boolean isDrawingChanceCard: It keeps the information about whether the picked card is a chance card or not.
- private double multiplier: It is used for bankman specific variables.

4.5.3 InGameManager Class

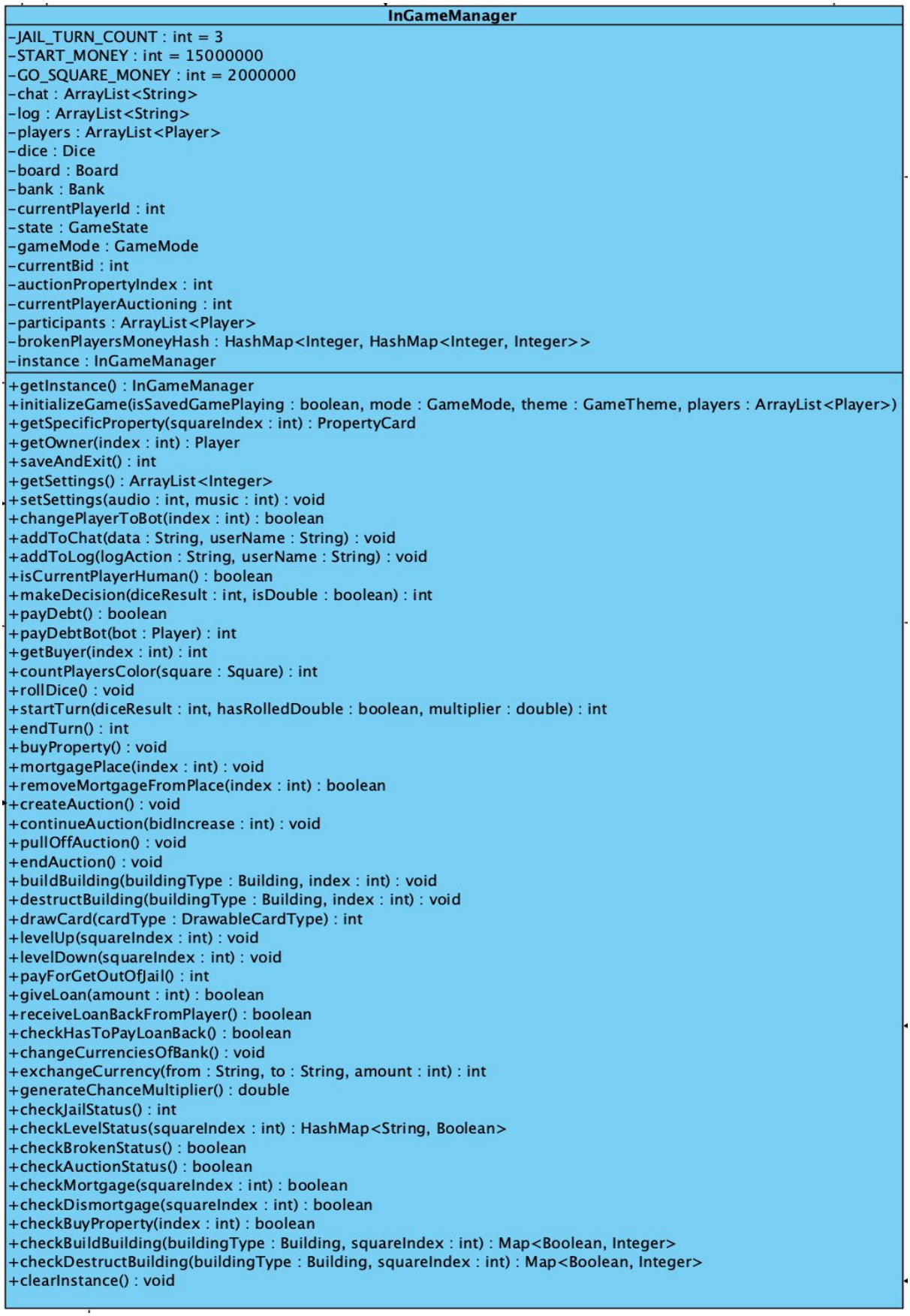

Figure 10: InGameManager Class Diagram

InGameManager class includes features and properties related to the screen which shows up while the players are playing the game. This class can be regarded as the game itself since players and their attitudes occur at this stage of the whole game.

4.5.3.1 Attributes of InGameManager Class

- private int JAIL_TURN_COUNT: It keeps the jail turn count.
- private int START_MONEY: It keeps the money to be started.
- private int GO_SQUARE_MONEY: It keeps the money that will be received when passing go square.
- private ArrayList<String> chat: It is used to store the list of chats in the game.
- private ArrayList<String> log: It is used to store the list of actions done by the users in order.
- private ArrayList<Player>players: It is used to store the list of players that contribute to playing the game.
- private Dice dice: It is used to keep the dice rolled in the round.
- private Board board: It is used to keep the board of the game.
- private Bank bank: It is used to keep the bank of the current game.
- private int currentPlayer: It is used to keep the current player among the players list.
- private GameState state: It is used to keep game state.
- private GameMode gameMode: It is used to keep game mode.
- private int currentBid: It is used to keep the current bid of an auction.
- private int auctionPropertyIndex: It keeps the index of property that will be on the auction.
- private int currentPlayerAuctioning: It keeps the current player auctioning right now.
- private ArrayList<Player> participants: It keeps the list of participants in the auction.

- private HashMap<Integer, HashMap<Integer, Integer>>: It keeps the debt of the players and where this debt has to go.
- private InGameManager instance: It takes the information in the game manager.

4.5.3.2 Methods of InGameManager Class

- public void startGame(isSavedGamePlaying : boolean): It initializes the game or loads the saved game.
- public InGameManager getInstance(): It gets the instance of in game manager.
- public initializeGame(isSavedGamePlaying : boolean, mode : GameMode, theme : GameTheme, players): ArrayList<Player>): It is used to initialize a game while keeping all the information of the game.
- public PropertyCard getSpecificProperty(squareIndex : int): It is used to get the card of a specific property.
- public Player getOwner(index : int): It keeps the owner of the property.
- public int saveAndExit(): It helps players to save the game and exit.
- public boolean changePlayertoBot(index : int): It is used to convert a player into a bot player.
- public void addToChat(data : String, userName: String): It is used to add a message to the chat part of the game.
- public void addToLog(logAction: String, userName: String): It is used to add a string to log.
- public boolean isCurrentPlayerHuman(): It keeps whether the player is a human or bot.
- public int makeDecision(diceResult: int, isDoubtle: boolean): It handles the game play of the bot.
- public int boolean payDebt(): It keeps the information about whether the player paid the debt or not.
- public int payDebtBot(bot: Player): It helps the bot to pay its debt.
- public int getBuyer(index: int): It gets the buyer of the specific index.
- public int countPlayersColor(square: Square): It keeps the count of the squares that are the same color for a specific user.

- public void rollDice(): It helps the player to roll the dice.
- public int startTurn(diceResult: int, hasRolledDouble: boolean, multiplier: double): It starts the turn.
- public int endTurn(): It ends the turn.
- public void buyProperty(): It helps a user to buy a property.
- public void mortgagePlace(index: int): It mortgages a place.
- public boolean removeMortgageFromPlace(index: int): It removes the mortgage on a place as a result of the required payment.
- public void createAuction(): It creates the auction for players.
- public void continueAuction(bidIncrease: int): It continues auction by a new bid by another player.
- public void pullOffAuction(): It helps a user to pull off from the auction.
- public void endAuction(): It ends the auction.
- public void buildBuilding(buildingType: building, index: int): It helps a user to build a building on a property square by the user.
- public void destructBuilding(buildingType : Building, index: int): It helps the user to destruct the building and get the money.
- public int drawCard(cardType: DrawableCardType): It helps the user to draw a card when needed.
- public void levelUp(squareIndex: int): It increases the level of a property square.
- public void levelDown(squareIndex: int): It decreases the level of a property square.
- public int payForGetOutOfJail(): It gets the payment from the player to get out of the jail.
- public bool giveLoan(amount:int): It helps the bank to give a loan to the user.
- public bool receiveLoanBackFromPlayer(): It gets the money that is given to the player previously from the player.
- public boolean checkHasToPayLoanBack(): It keeps the information about whether the player has paid their loan back to the bank.

- public void changeCurrenciesOfBank(): It changes the currency rate of the currencies at the end of the turn if the mode of the game is selected as Bankman.
- public int exchangeCurrency(from: String, to: String, amount: int): It exchanges the amount of money between given parameters "to" and "from" for the current player.
- public double generateChanceMultiplier(): It generates a chance multiplier if the mode of the game is set to Bankman.
- public int checkJailStatus(): It checks the current players jail status and returns it.
- public HashMap<String,boolean> checkLevelStatus (squareIndex: int): It checks the given square index and returns the value of the possible build count that can be built in that square.
- public boolean checkBrokenStatus(): It checks if any of the players has got into bankruptcy or not.
- public boolean checkAuctionStatus(): It checks if the auction is ended or not.
- public boolean checkMortgage(squareIndex: int): It checks if the corresponding player can mortgage the corresponding square index.
- public boolean checkDismortgage(squareIndex: int): It checks if the corresponding player can dismortgage the corresponding square index.
- public boolean checkBuyProperty(index: int): It checks the square in the given index and returns the value of if it is buyable or not.
- public Map <boolean, integer> checkBuildBuilding(buildingType: Building, squareIndex: int): It checks whether the corresponding player can build the given type of the building to the corresponding square.
- public Map <boolean, integer> checkDestructBuilding (buildingType: Building, squareIndex: int: It checks whether the selected index has the given building type and returns true if player has the corresponding building type in the corresponding square.
- public void clearInstance(): It clears the singleton instance created by the corresponding controller.

4.5.4 LobbyManager Class

Figure 11: LobbyManager Class Diagram

LobbyManager class includes attributes and functionalities of the lobby section of the game. Such options like theme and mode are selected here before the game actually starts. Ground structure of the game is shaped at this class and the LobbyManager class and InGameManager class are related to each other.

4.5.4.1 Attributes of LobbyManager Class

- private boolean isOnline: It is used to keep the information about whether the game is online or not.
- private GameMode gameMode: It is used to keep the game mode information.
- private GameTheme gameTheme: It is used to keep the game theme information.
- private Player admin: It is used to store the information about the host who created the lobby.
- private ArrayList<Pawn> pawns: It is used to keep the list of pawns used by the players.

- private ArrayList<Player> players: It is used to store the players who currently play the game.
- private ArrayList<Boolean> isAllReady: It is used to keep the data on whether the players are ready for the game in order to be able to start the game.
- private int maxPlayerCount: It is used to keep the maximum number of players in the lobby.
- private int playerCount: It is used to keep the current number of players in the lobby.
- private int playerCount: It is used to keep the current number of players in the lobby.
- private int botCount: It is used to keep the number of bot players in the lobby.
- private String lobbyPassword: It is used to keep the lobby password of the current game.
- private ArrayList<Integer> botIds: It is used to store bot players' id numbers.
- private LobbyManager instance: Instance of LobbyManager.

4.5.4.2 Methods of LobbyManager Class

- public LobbyManager getInstance(): It is used to get instance from the class.
- public File getSettings(): It is used to get settings from file.
- public void setSettings(): It is used to set settings in the file.
- public boolean kickplayer(nick : String): It discards the player from the game according to their nickname.
- public boolean addBot(): It adds bot player to the game.
- public boolean deleteBot(): It deletes bot player from the game.
- public Arraylist<Player> getPlayerArrayList(): It is used to get the list of players.
- public void clearInstance(): It is used to clear the instance of the class.

4.5.5 MainMenuManager Class

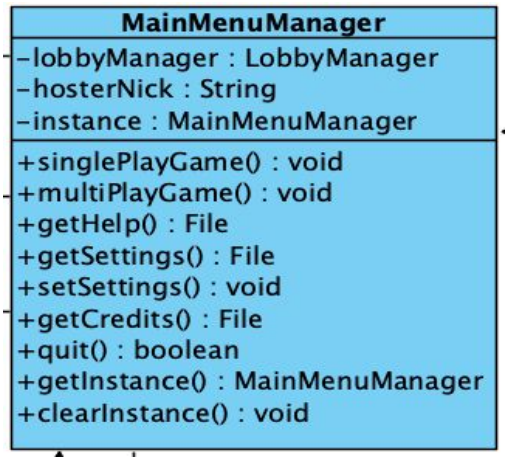

Figure 12: MainMenuManager Class Diagram

MainMenuManager class consists of the opening page of the game. Settings and information about the game is located here, before the creation of the game.

4.5.5.1 Attributes of MainMenuManager Class

- private LobbyManager lobbyManager: It is used to keep data from the lobby section of the game which is related to the lobby manager.
- private String hosterNick: It is used to store the nickname of the host who created the lobby, coming from the lobby screen controller.
- private MainMenuManager instance: It is used to store instance in the class.

4.5.5.2 Methods of MainMenuManager Class

- public void singlePlayGame(): It initializes the single player game.
- public void multiPlayGame(): It initializes the multiplayer game.
- public File getHelp(): It initializes the help section of the main menu.
- public File getSettings(): It gets the settings of the game.
- public void setSettings(): It changes the settings of the game to the desired settings.
- public File getCredits(): It gets the credits part of the game.
- public boolean quit(): It closes the game.

- public MainMenuManager getInstance(): It is used to get the instance of the class.
- public void clearInstance(): It is used to clear the instance of the class.

4.5.6 PropertyCard Class

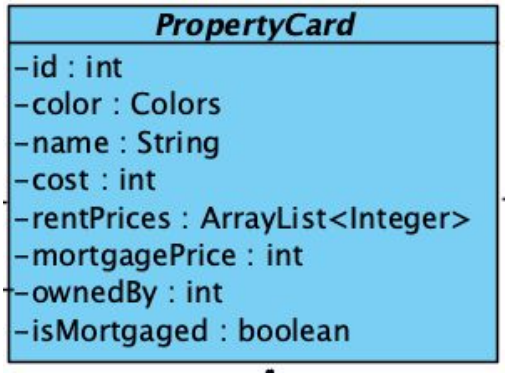

Figure 13: PropertyCard Class Diagram

PropertyCard class is the representative cards which are generated for the properties on squares. Every PropertyCard object has a unique id and some other properties.

4.5.6.1 Attributes of PropertyCard Class

- private int id: It keeps the id of the property.
- private Colors color: It keeps the color of the property cards.
- private String name: It keeps the name of the property card.
- private int cost: It keeps the cost of the property card.
- private ArrayList<Integer> rentPrices: It keeps the list of rent prices of properties.
- private int mortgagePrice: It keeps the mortgage price of the properties.
- private int ownedBy: It keeps the information about the owner of the property.
- private boolean isMortgaged: It keeps the information about whether the property is mortgaged or not.

4.5.7 Square Class

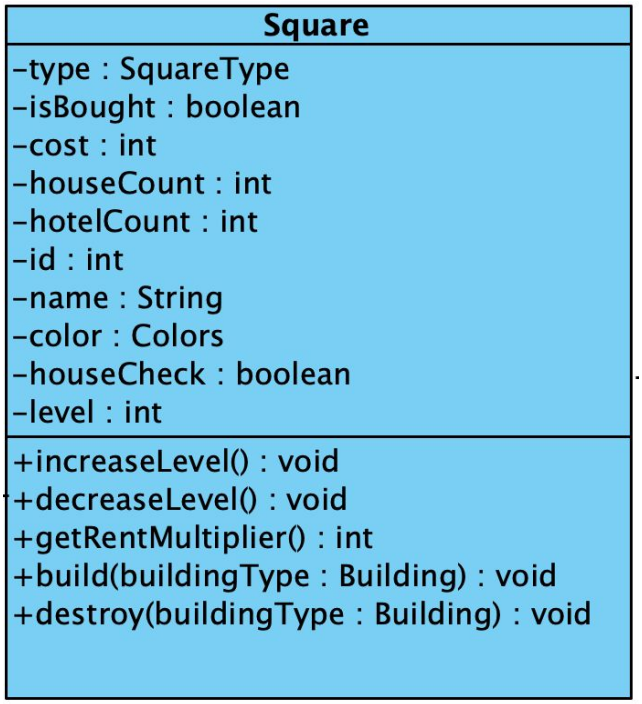

Figure 14: Square Class Diagram

Square class is used for the square representations of the board. It includes type of the square information and some other additional features related attributes.

4.5.7.1 Attributes of Square Class

- private SquareType type: It is used to keep the type of the square.
- private boolean isBought: It is used to keep the information about whether the square is bought by someone or not.
- private int cost: It is used to keep the current cost of the square.
- private int houseCount: It is used to keep the number of houses on a square.
- private int hotelCount: It is used to keep the number of hotels on a square.
- private int id: It keeps the id of the square.
- private String name: It keeps square name.
- private Colors color: It is used to keep the color of the square.
- private boolean houseCheck: It is used to keep check house data.

- private int level: It is used to keep the level of the square.

4.5.7.2 Methods of Square Class

- public void increaseLevel(): It is used to increase the level of the square.
- public void decreaseLevel(): It is used to decrease the level of the square.
- public int getRentMultiplier(): It is used to get rent multiplier to calculate rent.
- public void build(buildingType: Building): It is used to increase house or hotel number on the square.
- public void destroy(buildingType: Building): It is used to decrease house or hotel number on the square.

4.5.8 SquareType Enum Class

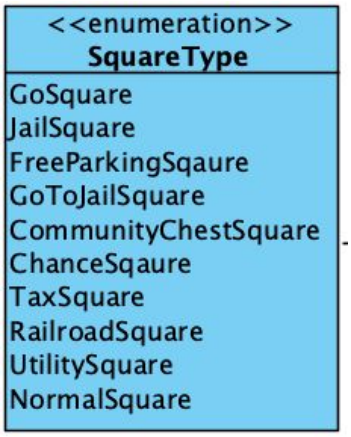

Figure 15: SquareType Enum Class Diagram

This enumeration class is used to store ten available square types on the board such as GoSquare, JailSquare, FreeParkingSquare.

4.5.9 GameTheme Enum Class

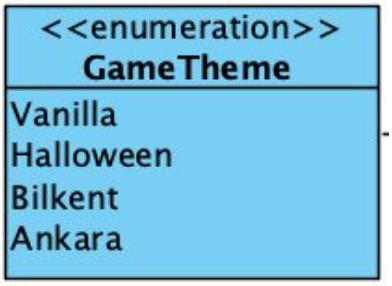

Figure 16: GameTheme Enum Class Diagram

This enumeration class is used to store different types of game themes available at the game which are Vanilla, Halloween, Bilkent and Ankara.

4.5.10 GameMode Enum Class

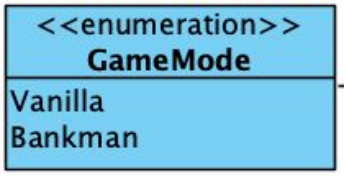

Figure 17: GameMode Enum Class Diagram

This enumeration class is used to store two types of game modes available in the game which are Vanilla and Bankman.

4.5.11 Pawn Enum Class

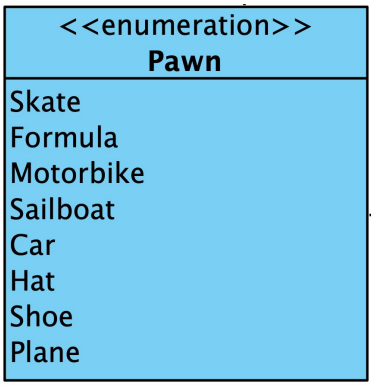

Figure 18: Pawn Enum Class Diagram

This enumeration class is used to keep the eight different types of pawns that can be chosen by the users. These eight types are Skate, Formula, Plane, Motorbike, Sailboat, Car, Hat and Shoe.

4.5.12 Board Class

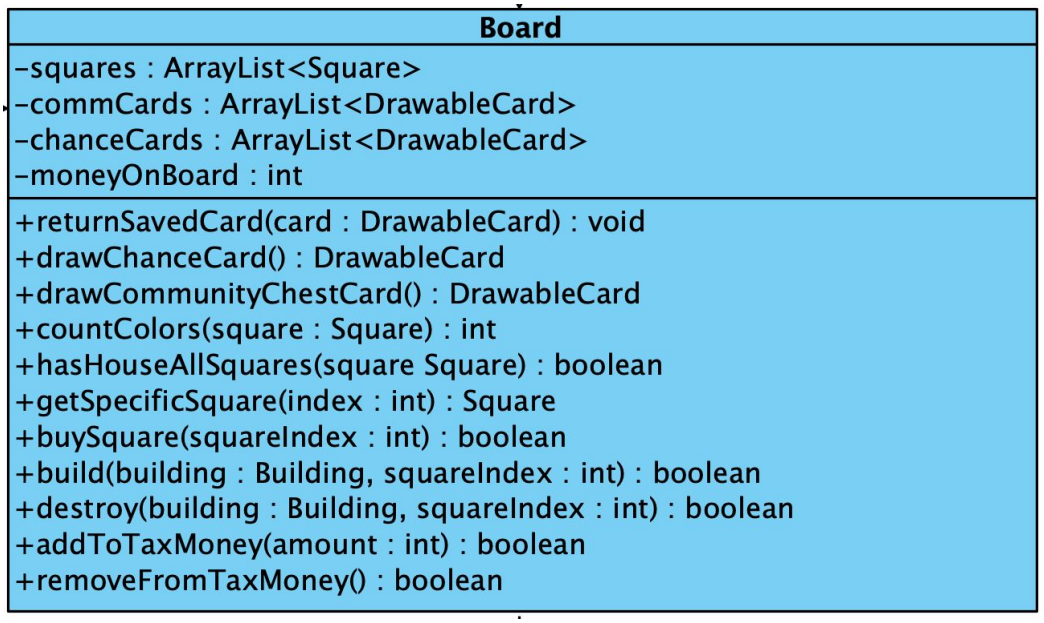

Figure 19: Board Class Diagram

Board class includes all kinds of attributes which can be located on the board. This class' attributes constitute the play.

4.5.12.1 Attributes of Board Class

- private ArrayList<Square> squares: It is used to keep the list of all the squares located on the board.
- private ArrayList<DrawableCard> commCards: It is used to keep the list of community chest cards available on the board at that time.
- private ArrayList<DrawableCard> chanceCards: It is used to keep the list of chance cards available on the board at that time.
- private int moneyOnBoard: It is used to keep the money from tax square.

4.5.12.2 Methods of Board Class

- public void returnSavedCard(card: DrawableCard): It is used to return a saved card.
- public DrawableCard drawChanceCard(): It is used to draw a chance card.
- public DrawableCard drawCommunityChanceCard(): It is used to draw a community chance card.
- public int countColors(square: Square): It counts the colors in the board according to the color given through the Square parameter.
- public boolean hasHouseAllSquares(square: Square): It is used to check all squares with the same color has a house.
- public Square getSpecificSquare(index: int): It returns the specific square through the given index parameter.
- public boolean buySquare(squareIndex: int): It is used to add an owner to square.
- public boolean build(building: Building, squareIndex: int): It is used to build a house on a square.
- public boolean destroy(building: Building, squareIndex: int): It is used to destroy a house on a square.
- public boolean addToTaxMoney(): It is used to add money to tax money.
- public boolean removeFromTaxMoney(): It is used to remove money from tax money.

4.5.13 Currency Class

Figure 20: Currency Class Diagram

Currency class represents the different types of currencies in the game.

4.5.13.1 Attributes of Currency Class

- private String name: It is used to keep the name of the currency.
- private Double rate: It is used to keep the rate of the currency.

4.5.14 Bank Class

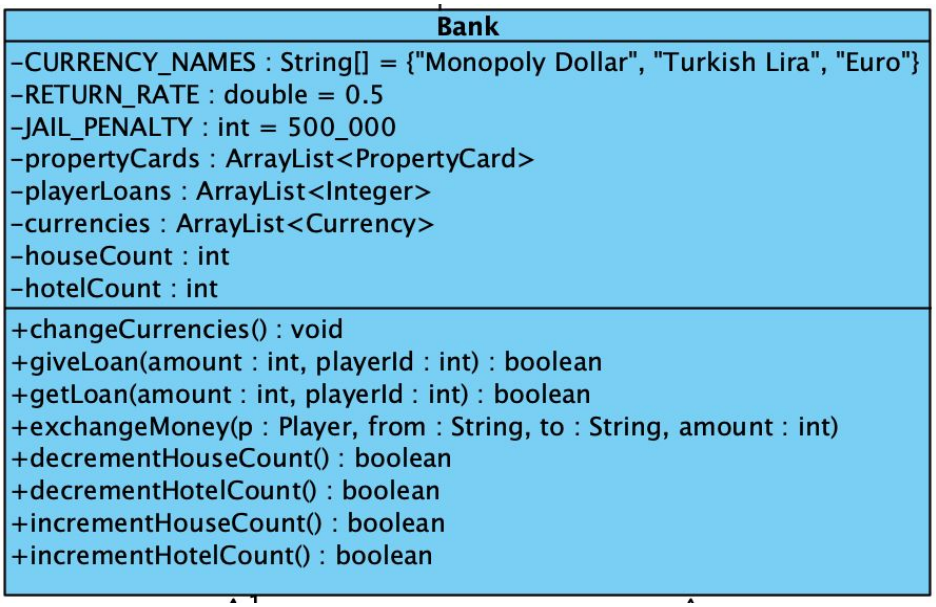

Figure 21: Bank Class Diagram

Bank Class includes actions and attributes that the players are able to do with the bank. It keeps track of cash flow in the game according to the actions of the players'.

4.5.14.1 Attributes of Bank Class

- private String[] CURRENCY_NAMES: It is used to keep the money types.
- private double RETURN_RATE: It is used to calculate return rate.
- private int JAIL_PENALTY: It is used to calculate jail penalty.
- private ArrayList<PropertyCard> propertyCards: It is used to keep the list of property cards to be used while purchasing.
- private ArrayList<Integer> playerLoans: It is used to keep the list of money loaned by the players.

- private ArrayList<Currency> currencies: It is used to keep the list of currencies.
- private int houseCount: It is used to keep house count.
- private int hotelCount: It is used to keep hotel count.

4.5.14.2 Methods of Bank Class

- public void changeCurrencies(): It changes the currencies and their rates.
- public boolean giveLoan(amount : int, playerId : int): It gives loan to the player.
- public boolean getLoan(amount : int, playerId : int): It gets back the money the bank loaned to the player.
- public boolean exchanceMoney(p: Player, from: String, to: String, amount: int): It exchances money between different currencies.
- public boolean decrementHouseCount(): It decreases house count.
- public boolean decrementHotelCount(): It decreases hotel count.
- public incrementHouseCount(): It increases house count.
- public incrementHotelCount(): It increases hotel count.

4.5.15 PlaceCard Class

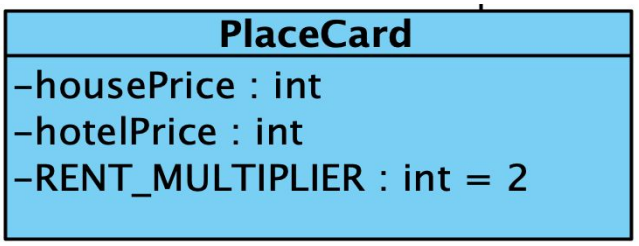

Figure 22: PlaceCard Class Diagram

PlaceCard class includes the information about the prices of the places.

4.5.15.1 Attributes of PlaceCard Class

- private int housePrice: It is used to keep the price of the house.
- private int hotelPrice: It is used to keep the price of the hotel.

- private int RENT_MULTIPLIER: It is used to double the rent price if the player has all of them.

4.5.16 RailroadCard Class

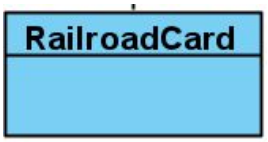

Figure 23: RailroadCard Class Diagram

This class is created in order to specify railroad cards.

4.5.17 Dice Class

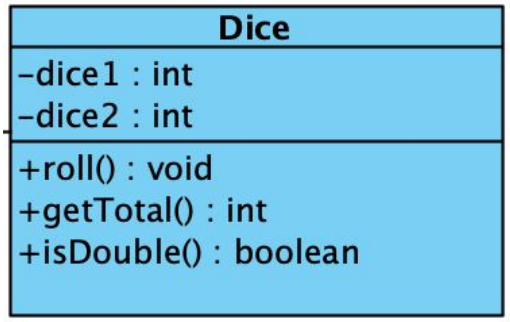

Figure 24: Dice Class Diagram

4.5.17.1 Attributes of Dice Class

- private int dice1: It is used to keep the value of the first dice rolled.
- private int dice2: It is used to keep the value of the second dice rolled.

4.5.17.2 Methods of Dice Class

- public void roll(): It rolls the two dices.
- public int getTotal(): It calculates the sum of the values of two dices.
- public boolean isDouble(): It checks if the dice result is double or not.

4.5.18 Building Enum Class

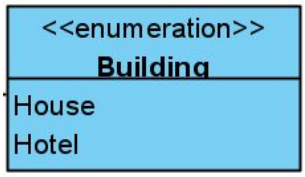

Figure 25: Building Enumeration Class Diagram

This enumeration class is used to specify two different building types which are house and hotel.

4.5.19 CommunityChestCard Class

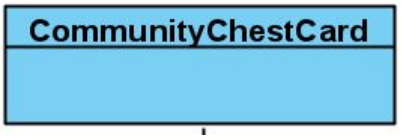

Figure 26: CommunityChestCard Class Diagram

This class is created for the community chest card type.

4.5.20 ChanceCard Class

Figure 27: ChanceCard Class Diagram

This class is created for the chance card type.

4.5.21 UtilityCard Class

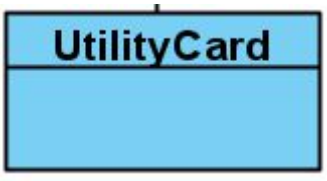

Figure 28: UtilityCard Class Diagram

This class is created for the utility card type.

4.5.22 Colors Enum Class

Figure 29: Colors Enumeration Class Diagram

This enumeration class is used to specify nine different colors for property cards and squares.

4.5.23 MainMenuController Class

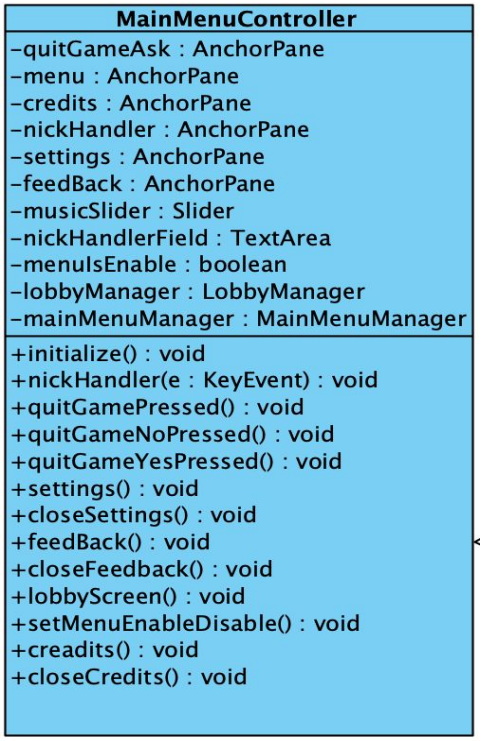

Figure 30: MainMenuController Class Diagram

This class is used to control the Main Menu screen of the game. Basically a controller class.

4.5.23.1 Attributes of MainMenuController Class

- private AnchorPane quitGameAsk: It is used to ask whether the player wants to quit the game or not.
- private AnchorPane menu: It is used to show the main menu.
- private AnchorPane credits: It is used for showing the credits of the game.
- private AnchorPane nickHandler: It is used for getting the nickname from the player.
- private AnchorPane settings: It is used for showing the settings of the game.
- private AnchorPane feedBack: It is used for showing the feedBack screen of the game.
- private Slider musicSlider: It is used for music settings.
- private TextArea nickHandler: It is used for the field to get the nickname from the player.
- private boolean menuIsEnable: It is used for keeping track of the menu status.
- private LobbyManager lobbyManager: Instance of LobbyManager class.
- private MainMenuManager: Instance of MainMenuManager class.
- private Stage primaryStage: It is used to connect the entry screen and the game play section of the game.

4.5.23.2 Methods of MainMenuController Class

- public void initialize(): It initializes the Main Menu Screen through corresponding FXML file.
- public void nickHandler(e: KeyEvent): Handles the nick input operation between changes of screen.
- public void quitGamePressed(): It is used when the quit game button is clicked.
- public void quitGameYesPressed(): It is used when the quit game question is answered as yes by the player.

- public void quitGameNoPressed(): It is used when the quit game question is answered as no by the player.
- public void lobbyScreen(): It is used to change the screen.
- public void settings(): It is used to open the settings dialog box.
- public void closeSettings(): Saves the current state of the settings and closes the dialog.
- public void feedBack(): It is used to open the feedback dialog box.
- public void closeFeedback(): It closes the feedback dialog.
- public void setMenuEnableDisable(): It's purpose is to show the menu staying on the blurred background.
- public void credits(): It is used to open credits dialog box.
- public void closeCredits(): It is used to close the credits dialog box.

4.5.24 LobbyController Class

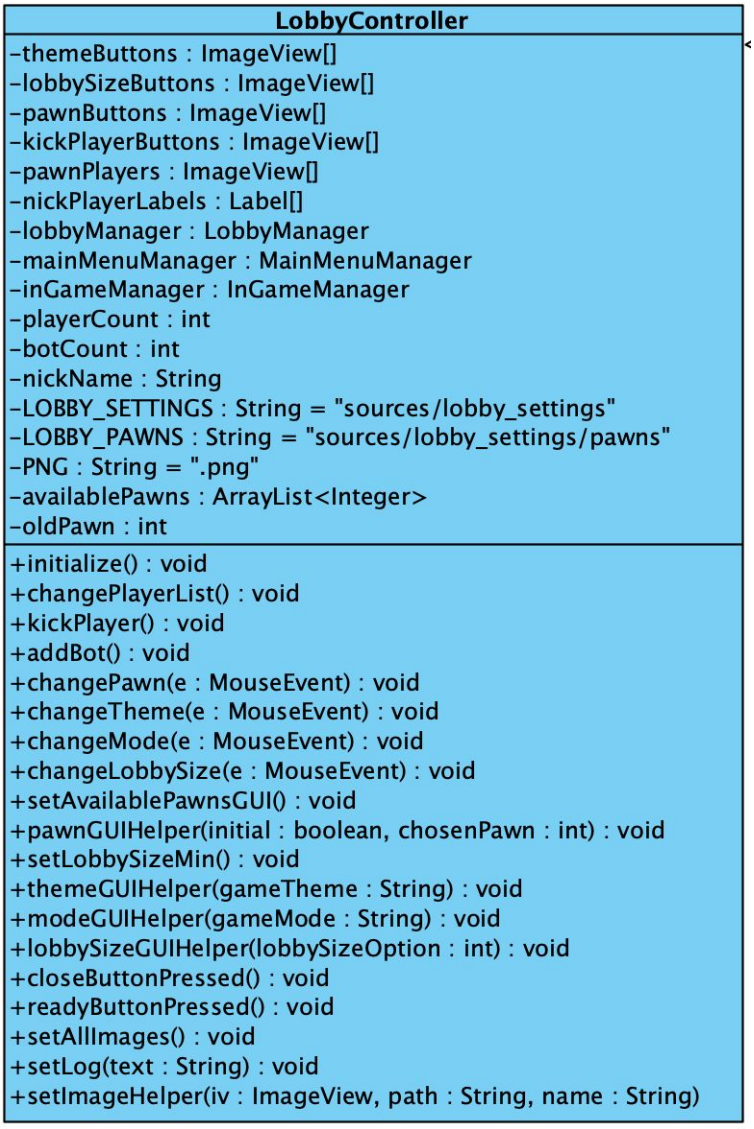

Figure 31: LobbyController Class Diagram

This class is used to control the Game Settings/Lobby screen of the game. Basically a controller class.

4.5.24.1 Attributes of LobbyController Class

- private ImageView[] themeButtons: It is used for showing the theme selection buttons.
- private ImageView[] lobbySizeButtons: It is used for showing the lobby size selection buttons.
- private ImageView[] pawnButtons: It is used for showing the pawn selection buttons.

- private ImageView[] kickPlayerButtons: It is used for showing the kick player buttons.
- private ImageView[] pawnPlayers: It is used for showing the selected and unselected pawn buttons.
- private Label[] nickPlayerLabels: It is used for showing the nicks of the labels.
- private LobbyManager lobbyManager: Instance of LobbyManager class.
- private MainMenuManager mainMenuManager: Instance of MainMenuManager class.
- private InGameManager inGameManager: Instance of InGameManager class.
- private int playerCount: It is used for keeping track of the player count in the lobby.
- private int botCount: It is used for keeping track of the bot player count in the lobby.
- private String LOBBY_SETTINGS: It is used for storing the directory path to lobby settings images.
- private String LOBBY_PAWNS: It is used for storing the directory path to pawn images.
- private String PNG: It is used for storing png extension.
- private ArrayList<Integer> availablePawns: It is used for keeping track of used and unused pawns and indexes.
- private int oldPawn: It is used for storing the old index of the selected pawn in case of changing the pawn as to add it back to available pawns.

4.5.24.2 Methods of LobbyController Class

- public void initialize(): It initializes the corresponding FXML of the controller.
- public void changePlayerList(): It changes the player list for each player or bot entered to lobby.
- public void kickPlayer(): It removes the selected player or bot from the player list.
- public void addBot(): It adds bot players to the game.

- public void changePawn(e: MouseEvent): It changes the pawn of the player to the according pawn clicked.
- public void changeTheme(e: MouseEvent): It changes the theme of the game.
- public void changeMode(e: Mouse Event): It changes the mode of the game.
- public void changeLobbySize(e: MouseEvent): It changes the maximum players size in the lobby.
- public void setAvailablePawns(e: MouseEvent): It changes the available pawn status when a player selects or changes their pawn.
- public void pawnGUIHelper(initial: boolean, chosenPawn: int): Helper function that is used on setAvailablePawns() function.
- public void setLobySizeMin(): Makes lobby size minimal during the initialization of the game.
- public void closeButtonPressed(): Handles closing the screen when close button is pressed.
- public void readyButtonPressed(): Handles change of the GUI when the player pressed ready during the Lobby Screen.
- public void setAllImages(): Helper function to set the images of the pawns in the screen.
- public void setLog(text: String): It adds the string written by the user to the log.
- public void setImageHelper(iv: ImageView, path: String, name: String): Helper function that used in setAllImages() function.

4.5.25 InGameController Class

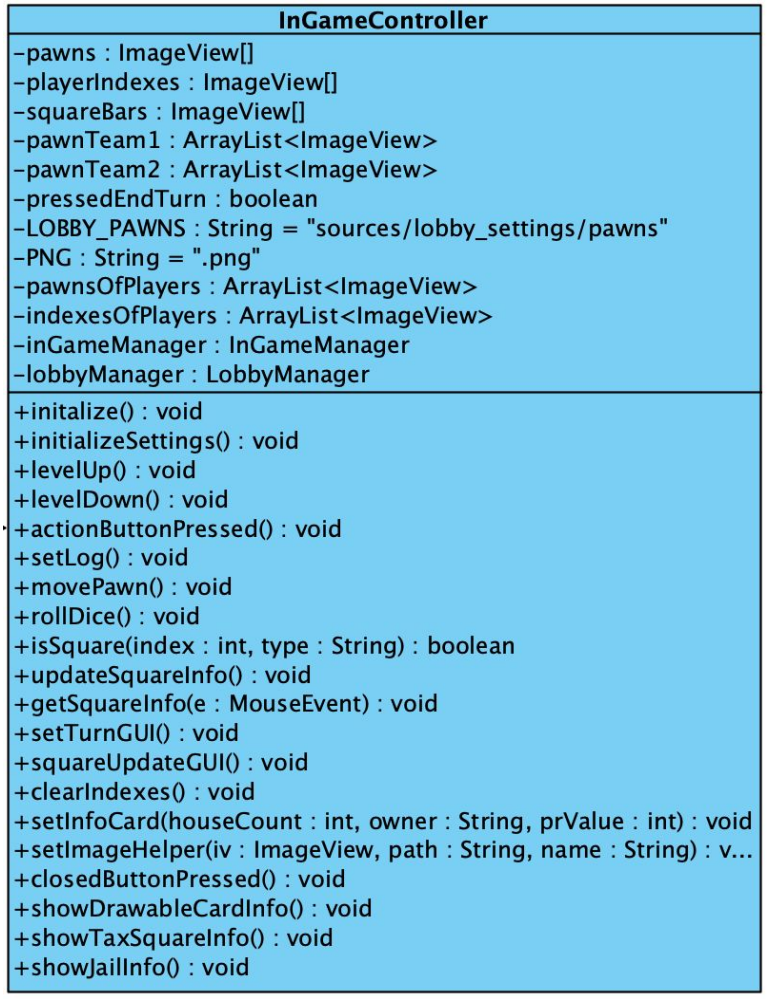

Figure 32: InGameController Class Diagram

This class is used to control the In-Game screen of the Game. Basically a controller class.

4.5.25.1 Attributes of InGameController Class

- private ArrayList<ImageView> indexes: It is used to list the squares on the board.
- private ArrayList<ImageView> sqBars: It is used to list the squares with either hotels or houses built on them.
- private ArrayList<ImageView> propertyInfo: It is used to list the information of properties.
- private ArrayList<Label> labels: It is used to list the labels.
- private ImageView[] pawns: It is used for storing the pawn images.

- private ArrayList<ImageView> pawnTeam1: It is used for calculating movement of a pawn.
- private ArrayList<ImageView> pawnTeam2: It is used for calculating movement of a pawn.
- private boolean pressedEndTurn: It is used for tracking if the ent turn button has been pressed or not.
- private String LOBBY_PAWNS: It is used for storing the directory of pawn images.
- private String PNG: It is used for storing png extension.
- private ArrayList<ImageView> pawnsOfPlayer: It is used for making operations of the players' pawns.
- private ArrayList<ImageView> indexesOfPlayer: It is used for showing and indicating the current players' pawn.
- private InGameManager inGameManager: Instance of InGameManager class.
- private LobbyManager lobbyManager: Instance of LobbyManager class.

4.5.25.2 Methods of InGameController Class

- public void initialize(): It is used to initialize the in game screen of the game.
- public void initializeSettings(): It is used to initialize the setting of the game.
- public void levelUp(): It is used to update the GUI of the game when a player is selected to improve its properties.
- public void levelDown(): It is used to update the GUI of the game when a player is selected to impair its properties.
- public void actionButtonPressed(): It is used to complete the turn of the player.
- public void setLog(): It is used to change the GUI of the log in the game screen.
- public void movePawn(player: Player, pawn: ImageView, moveCount: int, pawnsIsLonged: boolean): It is used to move the pawn from its current place.
- public void rollDice(): It is used to trigger the roll the dice animation in the GUI.
- public boolean isSquare(index: int, type: String): Helper function to determine the clicked index is not a corner square.
- public void updateSquareInfo(): It is used to update the square information.

- public void getSquareInfo(): It is used to show the information of the clicked square.
- public void setTurnGUI(): It is used to initialize the GUI in each turn.
- public void squareUpdateGUI(): It is used to change the look of a square when a square is leveled up or leveled down.
- public void clearIndexes(): It is used to clear the squares shown in GUI.
- public void setInfoCard(houseCount: int, owner: String, prValue): It is used to open a dialog box containing the information of the square clicked.
- public void setImageHelper(iv: ImageView, path: String, name: String): Helper function that is used in setInfoCard function.
- public void closedButtonPressed(): This function quits the InGameScreen.
- public void showDrawableCardInfo(): This function shows the GUI of the card that player has drawn from either Chance Cards or Community Chest Cards.
- public void showTaxSquareInfo(): This function shows the GUI of the money collected from taxes when the FreePark square is clicked.
- public void showJailInfo(): This function shows the GUI of the current jail status when the Jail square is clicked.

4.5.26 DrawableCardType Enum Class

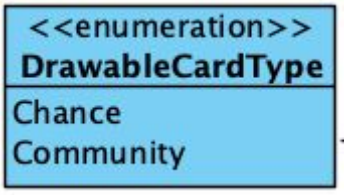

Figure 33: DrawableCardType Enumeration Class Diagram

This enumeration class is used to specify different types of Drawable Cards. Used by the InGameManager class.

4.5.27 GameState Enum Class

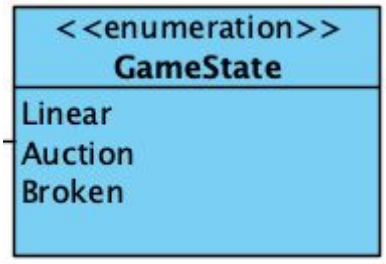

Figure 34: InGameController Class Diagram

This enumeration class is used to specify the game state values that are used by the InGameManager class.

4.5.28 StorageUtil Class

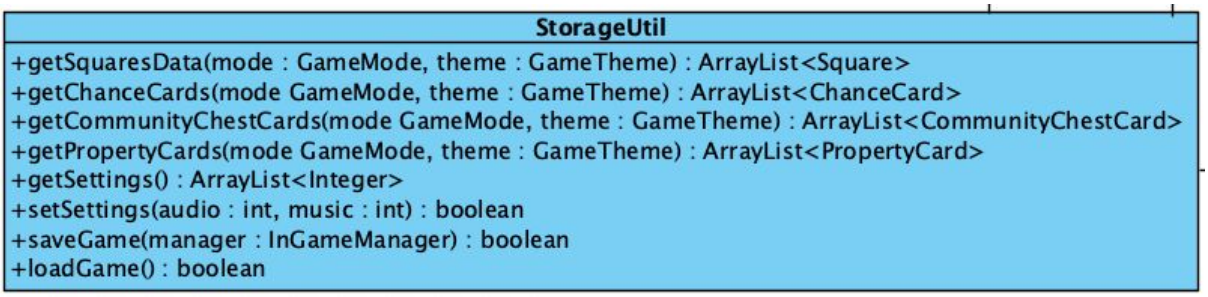

Figure 35: StorageUtil Class Diagram

This class is used to handle local storage operations made in the game.

4.5.28.1 Methods of StorageUtil Class

- public ArrayList<Square> getSquaresData(mode:GameMode, theme:GameTheme): This function reads the corresponding JSON file according to the parameters given by the function and gets the Square data from the JSON file read and returns it.
- public ArrayList<ChanceCard> getChanceCards(mode:GameMode, theme:GameTheme): This function reads the corresponding JSON file according to the parameters given by the function and gets the Chance Cards data from the JSON file read and returns it.
- public ArrayList<CommunityChestCard> getCommunityChestCards(mode:GameMode, theme:GameTheme): This function reads the corresponding JSON file according to the parameters given by the

function and gets the Community Chest Cards data from the JSON file read and returns it.

- public ArrayList<PropertyCard> getPropertyCards(mode:GameMode, theme:GameTheme): This function reads the corresponding JSON file according to the parameters given by the function and gets the Property Cards data from the JSON file read and returns it.
- public ArrayList<Integer> getSettings(): This function reads the JSON file that the settings of the game such as music and audio volume values and returns them as an ArrayList.
- public boolean setSettings(audio: int, music: int): This function saves the value of the audio and music sounds set by the player and saves the values to the corresponding JSON file.
- public boolean saveGame(inGameManager: InGameManager): This function saves the currently playing game to a JSON file.
- public boolean loadGame(): This function loads the selected game from the JSON file and utilizes an InGameManager object.

4.5.29 FirebaseUtil Class

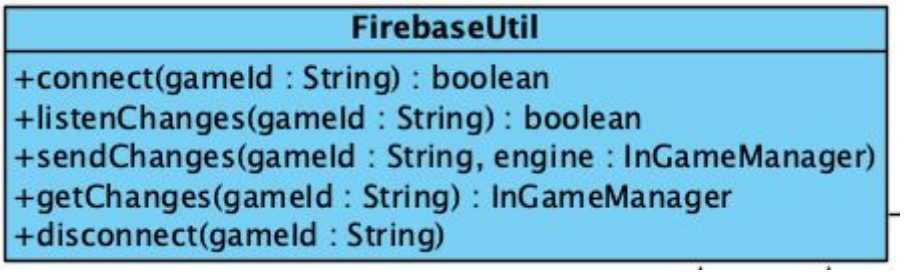

Figure 36: StorageUtil Class Diagram

This class is used to communicate with the Firebase online database system.

4.5.29.1 Methods of FirebaseUtil Class

- public boolean connect(gameId: String): This function connects the game to the Firebase online database system.
- public boolean listenChanges(gameId: String): This function listens to the changes made in the Firebase online database and returns true if changes made and false otherwise.

- public boolean sendChanges(gameId: String, engine: InGameManager): This function sends the given manager to the corresponding field in the Firebase online database.
- public InGameManager getChanges(gameId: String): This function is used with listenChanges method. This function gets the value changes in the database as an InGameManager object and modifies the values of the InGameEngine of the player's game.
- public void disconnect(gameId: string): This function disconnects the game from the database in order to save resource usage and improve security.

5. References

- Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and Iterative Development,by Craig Larman, Prentice Hall, 2004, ISBN: 0-13-148906-2
- *Object-Oriented Software Engineering*, by Timothy C. Lethbridge and Robert Laganiere, McGraw-Hill, 2001, ISBN: 0-07-709761-0
- https://docs.oracle.com/