Bilkent University

Department of Computer Engineering

CS 319
Object-Oriented Software Engineering
Term Project

Section 2

Group 2F

Monopoly

Design Report

Project Group Members

Adil Meriç	21802660
Burak Öçalan	21703769
Doğa Tansel	21802917
Osman Batur İnce	21802609
Ömer Faruk Akgül	21703163

Supervisor: Eray Tüzün

Contents

1. Introduction

1.1 Purpose of the System

The purpose of our project is to make the game Monopoly. In Monopoly, players start out with some money and compete to buy, capitalize and conquer every location they possibly can by monopolizing those areas. It is a turn-based strategy game that allows the players to use different tactics. A player's purpose is to buy and sell his/her properties in such a way that the other players will go bankrupt trying to build and sell and he/she will be the last person standing. When this happens the game ends. The maps contain many locations such as properties, stations, jail, free parking, chance, community chests, tax administrations and more. The players all have money they can spend and they also acquire cards for properties and other locations as they purchase them. Despite its large amounts of functionality, Monopoly is supposed to be easy to understand and play. The system also aims to give players the option to customize their games and maps.

We set our design goals to reflect these aspects of the game. Our design goals are functionality, reliability, usability and modifiability.

1.2 Design Goals

Setting and prioritising design goals are essential in crafting a good design for any task in general, especially in programming. For that reason, design goals must be discussed and analyzed according to certain criteria and the goals with the highest priority must be focused on with great care. For our Monopoly game, we used end-user expectations, hardware constraints and performance constraints as our criteria.

Functionality

Our first top design goal is to have a very high amount of functionality. The players of the game will expect to be able to have variety in Monopoly and not have it be a repetitive and dull game. They will expect it to be suitable for their liking and perhaps will want to change some things about their playing, meaning they expect to be able to modify the game in a way where it provides them variety. For example, while playing this game on a computer, the players will want to have different modes or scenarios and even different maps (since they are not physically limited with

having one type of board or certain pieces). That is why the game must have high functionality for the users.

In terms of hardware constraints, one game including multiple players will be played on one computer and there will be no online servers connected to the game. This means we are not constrained by having to design the game on multiple areas of hardware and our hardware mapping will be fairly simplistic. Therefore, instead of having to focus on having the game run on multiple machines, we can focus more on what one game can do on one machine.

Finally, we have performance constraints. Monopoly is a game where you can perform many different actions on your turns, so it was logical for us to give this game a high level of functionality from the start.

Reliability

Our second design goal is to be able to build a reliable game. The end-user expectation for this is that the users do not want the game to be unreliable, where it easily becomes buggy, unresponsive or crashes. Monopoly is a long game in general and that is why the users will expect the game to be able to continue for at least one or more matches.

The game is again, not so difficult to implement in terms of hardware, as it will only have to run on a singular machine to be played. This means that when that hardware machine becomes unresponsive, the game should not crash and should be able to wait for the machine to become responsive again. Unless there is a major problem with the computer the game is being played on, the game should stay responsive and should be able to function. For example, network errors or other problems in a computer should not interfere with the playability of the game.

Since the game runs on a singular machine, it should be able to respond and do so quickly. As mentioned before, Monopoly is a long game and so the users would not want to waste time waiting for the game to respond. That is why it is a requirement that the game should be reliable in terms of performance.

Usability

The end-users will expect the game to be usable and easily playable. If the game is difficult to navigate or understand, the players will lose interest in the game. The user-interface should be simple and easy to use. The menus and the games

themselves should be easy to understand and follow or else the users will not be entertained. For example, it should be obvious to the players when someone's round ends and the other person's begins or when someone purchases a property (pop-ups or large text could be used for this).

As the hardware mapping is simple for the project, it should be easy to navigate and use the game, especially since it runs on only one machine. The game should not require almost any other hardware or software to get involved in the completion or initialization of the game.

Performance-wise, the game should be very quick to respond which means it should be quick to use. This will help the game flow and it will be easy to navigate the game when it has high performance.

Modifiability

Modifiability is important for the end-users in this way: when we get feedback from our users about the game or when we encounter a problem in our game, we as the developers should be able to modify the game easily so that whatever problem has been found can be easily fixed so that the game becomes playable again in a small amount of time. It is also important because if the game is modifiable, then the users' feedback (of course will be evaluated for priority) could be taken more seriously by the developers and maintenance.

The hardware does not put much constraints on the game fortunately, so we will be able to build a highly modifiable game which could be easily fixed or maintained throughout singular machines.

The game being on singular hardware also brings a challenge: if the machine is slow or unresponsive at times, it is important that the game can be modified to counter these sorts of performance problems.

2. High-level software architecture

2.1 Subsystem decomposition

In our design, we chose to implement three layers of architecture. These layers are the Interface Layer, the Logic Layer and the Data Layer.

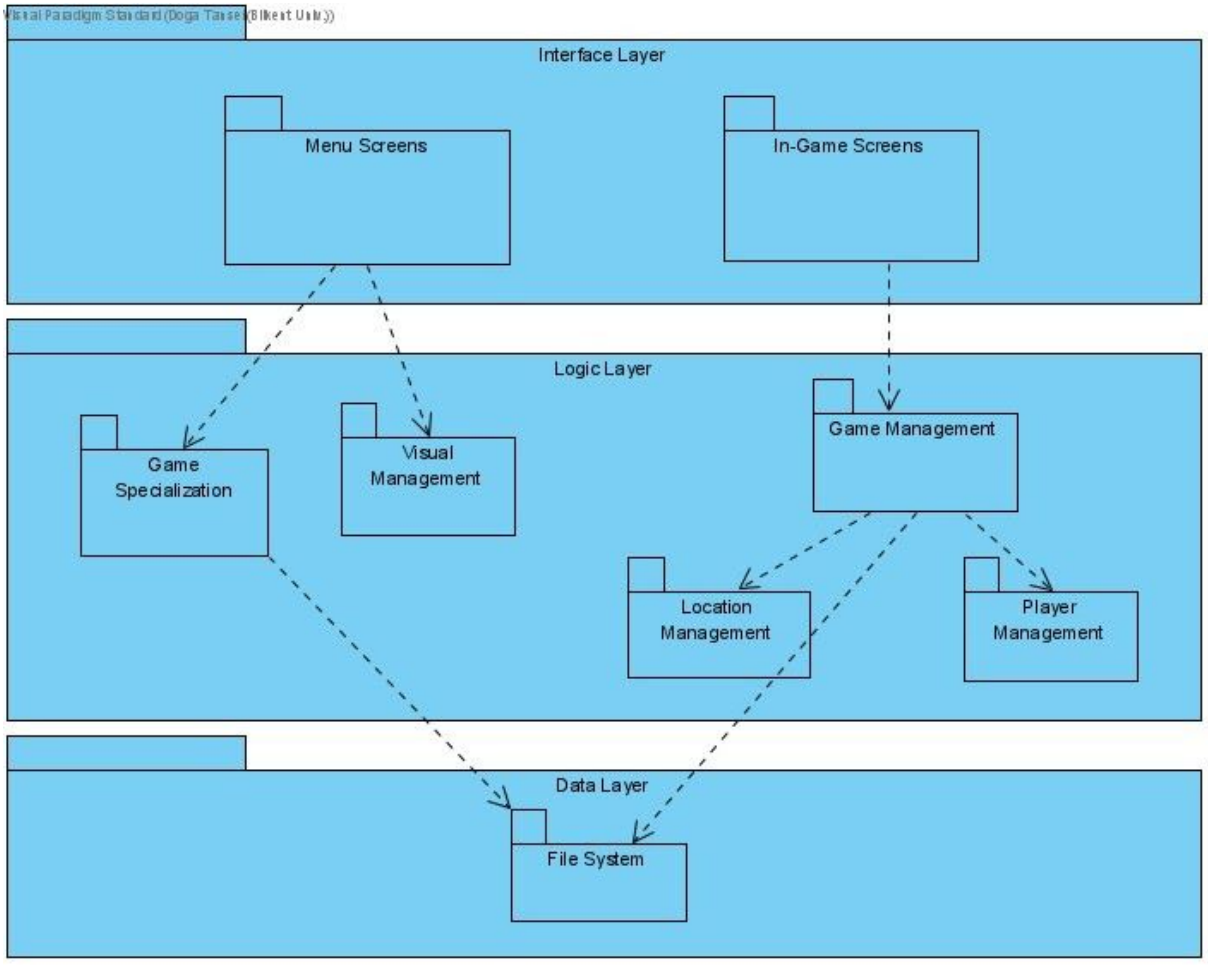

Figure 1: System Decomposition Package Diagram

The Interface Layer has the Menu Screens package which stores the files for controlling the main menu's UI such as the new game, load game, game editor, credits, settings options which allow the players to access the game and customize their games before starting. The In-Game Screens are for files controlling the UI of the games themselves such as the pop-ups or screens that are displayed throughout the games. The pause menu is also part of the in-game screens.

The Logic Layer is for the functions of the game. There are several packages. The Game Specialization and Visual Management packages are for files that control the customization and visual settings of the game. They communicate directly with the Menu Screens package in the Interface Layer. The Game Management package includes the more generalized files for the functionality of the gameplay. Two packages for Player Management and Location Management communicate with the Game Management package and they form a cohesive whole for the functionalities of gameplay. The Game Management package communicates with the In-Game Screens package in the Interface Layer.

Finally, the Data Layer includes one package called File System, storing the raw data which are the entity objects we will use in this project. The package communicates with the Game Specialization package for choosing the right objects for the customization of the game and the Game Management package for lending the functions of entity objects to the game manager.

2.2 Hardware / Software Mapping

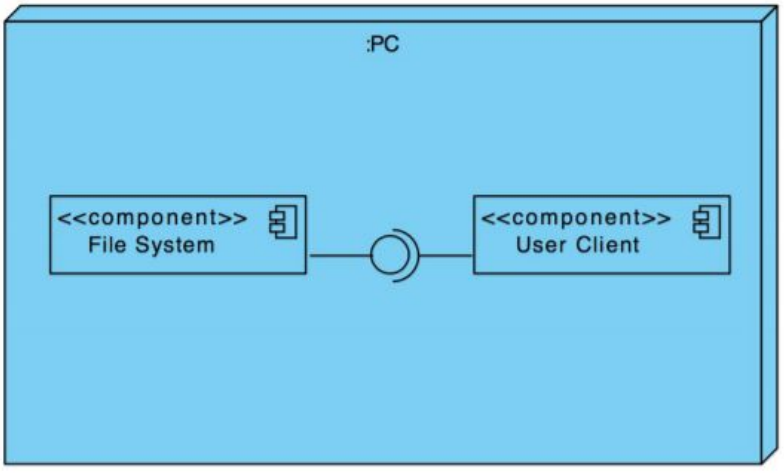

Figure 2: Deployment Diagram of the System

In terms of hardware/software mapping, our project is really simple as our Monopoly implementation will not have any kind of network connection requirement. This includes Local Area Networks (LANs), the Internet, any kind of connection to a global database and so forth. Therefore, the only remaining concern is selecting the virtual machine that our software is going to be built.

As our game will be implemented in Java, and we are going to use the JavaFX GUI library in advance, we need Java SE Runtime Environment 8 (JRE). The JavaFX

library is not built-in in newer versions of Java, therefore they might run into runtime problems.

In terms of virtual system, our application will run on Java Virtual Machine (JVM), however, as JavaFX 8 does not support mobile devices, OSes will be limited with desktop.

For Windows / MacOS / Linux Requirements:

- A monitor with a minimum 1366x768 resolution
- A keyboard and a mouse
- Java SE 8 must be installed.
- RAM: 128 MB
- Disk Space: Approximately 200 MBs

2.3 Persistent Data Management

Our Monopoly implementation will be a local, offline game so the game does not need any kind of database to store particular game data. We will store our data locally using JSON files as they are easy to read/write, lightweight, and environment independent. Saved games, which includes player data, table data, and custom created mods will be stored via JSON files. We want our game to be customizable as much as possible, therefore they might run into runtime problems.

In terms of virtual system, our application will run on Java Virtual Machine (JVM), however, as JavaFX 8 does not support mobile devices, OSes will be limited with desktop.

Reasons why we are using a file system are the data will not be transmitted between different devices and platforms, it will only exist on local platforms. There is only one program, our game, that can access the data. Our information density is low and our data does not require access at superb levels of detail. Moreover, there will be a single writer.

We will also include some image and sound files. The sound files are going to be stored in WAV format, however we did not come to a conclusion on the image format.

2.4 Access Control and Security

The Monopoly game that we designed is played by multiple users on a single computer. That is why, after installation of the game the user will not have to perform any action requiring internet connection such as user authentication during the initialization of the game. In other words, this version of the Monopoly game is designed in such a way that there will not be a need for online databases. Thus, the access permission scheme is not created since there is no data that is accessible only by specific users.

Saved games will be stored in the memory of the local computer. This storage can be accessed only by the owner of the computer as there is no other actor involved in it. In conclusion, as there will not be multiple actors interacting with each other, we will not deal with any access control and security issues.

2.5 Boundary conditions

Initialization Use Cases

StartMonopolyApplication: Double clicking the .jar file starts the application and activates the Menu Manager. The Main Menu Screen is displayed to the user and all navigational components on it (i.e. buttons) are activated. Then the user will be able to initiate the menu related use cases. Also sound and music will start playing and their levels will be adjusted to medium.

InitializeGame: The InitializeGame use case can be invoked by either the LoadGame use case, which loads the information from a previously saved game file, or processing the initialization settings selected by the user. With the initialization of a game, the Game Manager is created and takes the control of the game.. In-Game Screen is displayed to all players. Then the players will be able to initiate any gameplay related use cases.

Termination Use Cases

ExitGame: The ExitGame use case can be invoked by players during any time in the game with the Exit Game button. Before exiting the game, the player is asked whether or not to save the current game. As a response, the player may or may not select invoking the SaveGame use case. After that, the Game Manager terminates with all of its objects and the Main Menu Screen will appear on the screen. The control is given back to the Menu Manager.

ExitMonopolyApplication: Exiting the application is done by clicking the Quit button on the Main Menu interface. As a result, Menu Manager and all of its objects are terminated and so is the application.

Failure Use Cases

HandleLoadFailures: This use case is invoked by the LoadGame use case when it fails. This may be caused by several reasons. The file may not exist or may have a different file format than expected; in this case, the load request is discarded and the user is informed. The information in the file might be corrupted; in this case, the system informs the user about the corruption and asks permission for deleting the file.

HandleSaveFailures: This use case is invoked by the SaveGame use case when it fails. This may be caused by several reasons. A file with the same name might already exist; in this case, the user is asked whether or not to overwrite the file. The folder which is expected to keep the game files might be deleted or moved to another place for some reason; in this case, a new folder will be created to where it should be, and the save operation continues normally.

RefreshSaveFile: In order to decrease the possible effects of a game crash, the current game will be saved periodically. Whenever a player ends their turn, RefreshSaveFile use case will be invoked which will refresh the save file or create one if it doesn't exist. In this way, even if the game crashes, users will be able to reload their game.

3. Low-Level Design

3.1 Object Design Trade-offs

Cost vs Reliability

Most probably, none of the users would want to play a game which has many bugs or which constantly crashes. We as the developers also wouldn't want to produce a game that does not work properly. However, building an application without any bugs is a hard and costly job, especially if it includes many functionalities like our game. Despite its challenges, we are going to build a reliable game which handles any bugs and exceptional cases, even if it may cost us a lot of time.

Functionality vs Usability

In our game design, we are planning to build an easily understandable and usable game with a range of functionalities. Unfortunately, it is not possible to maximize both functionality and usability because they somewhat conflict with each other. So, we decided to maximize functionality in some parts of our game and maximize usability in other parts. In the game initialization, we are planning to focus on the functionalities which will enable the users to modify the game as they wish. To decrease the usability loss to minimum in the initialization part, we will also have an option to start the game with default settings. After the game starts with determined settings, the focus is going to shift towards usability. Any player who knows the rules of the original Monopoly board game should be able to easily understand and play the game without having to look at the user manual.

Development time vs Functionality

We think that the quality and the quantity of the functionalities provided to users is important. With more functionalities, users will be more flexible to find the best options for them. Also whenever they get bored from a certain configuration, they can try another one among various options. In order to give those opportunities to users, we will spend time implementing various functionalities.

Security vs Usability

Because our game does not have in-game purchases, we do not ask users for credit card information or any other private information. For this reason, we are focusing on the usability of our game without considering security issues.

3.2 Final Object Design

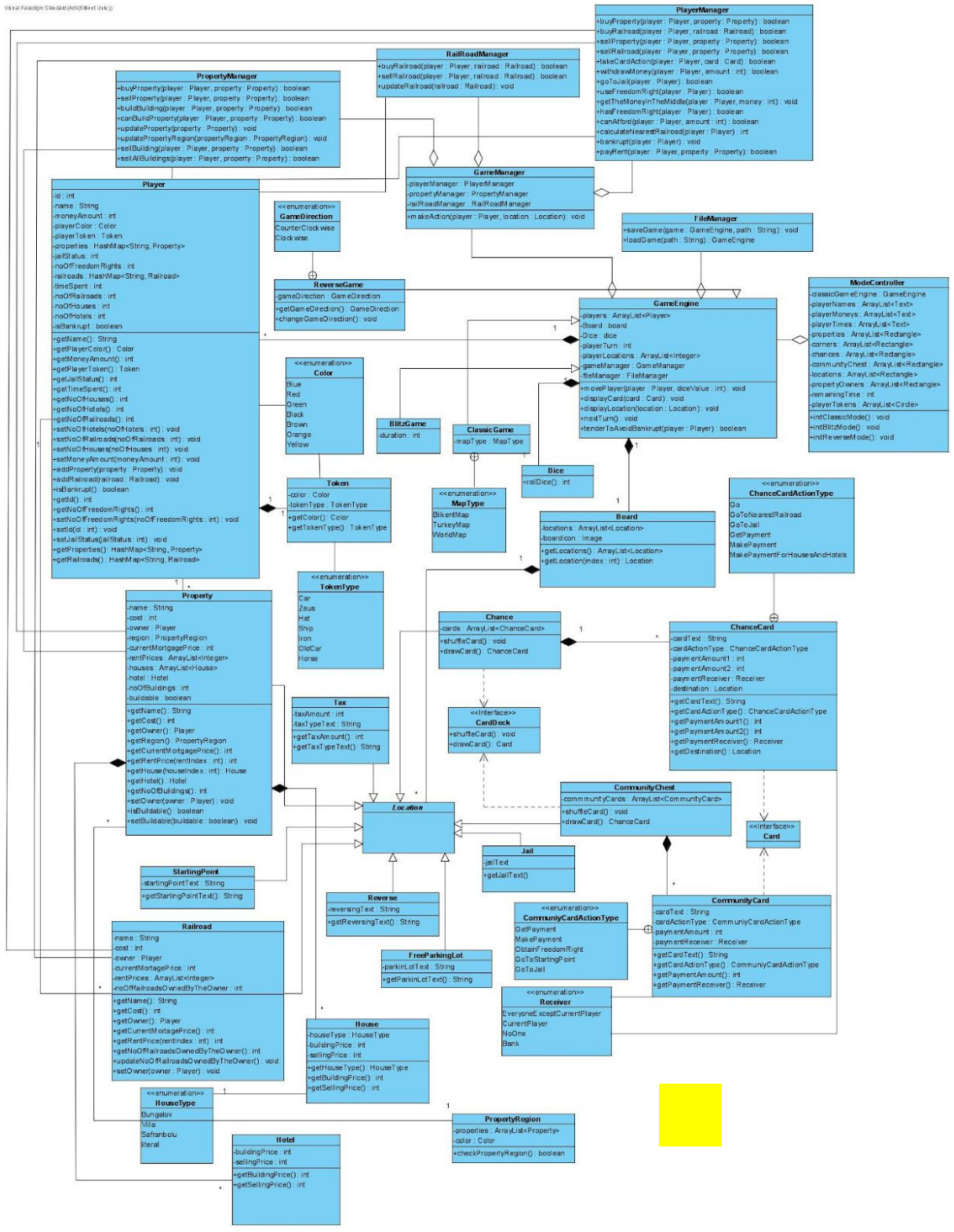

Figure 3: Final Object Design Class Diagram

3.3 Packages

In our Monopoly implementation, we can group our packages into two distinct categories. The first category is developer introduced packages that we have designed according to our subsystem decomposition, object design and class interfaces. The second type of packages are externally integrated packages that are JavaFX and JSON.simple library.

3.3.1 Developer Introduced Packages

In our project, ideally, we introduce packages that are in sync with our subsystem decomposition, object design and class interfaces. Therefore, we introduce packages ourselves.

1. **com.twoFMonopoly.menuScreens:** Includes the corresponding UI layouts for menu screens which can be summarised as main menu, new game menu, order determination menu, mod editor menu and so forth.
2. **com.twoFMonopoly.inGameScreens:** Includes the corresponding UI layouts for in-game screens which can be summarised as different map screens..
3. **com.twoFMonopoly.gameSpecialization:** Includes the general logic segments for game specialization actions in our game, such as in the mod editor screen where mod editor interacts with both menu screens and filesystem.
4. **com.twoFMonopoly.visualManagement:** Includes the general logic segments for menu screens as how it will navigate or how it will transmit data.
5. **com.twoFMonopoly.gameManagement:** Includes the general logic segments for in-game screens. This includes board, cards, houses and their interaction with game progress and UI.
6. **com.twoFMonopoly.playerManagement:** Includes the general logic segments for players in game and their interactions with other objects.
7. **com.twoFMonopoly.locationManagement:** Includes the general logic segments for locations such as properties, cards locations in game and their interactions with other objects.
8. **com.twoFMonopoly.fileSystem:** Manages saving data from filesystem and loading data to the filesystem. All connections between filesystem and the rest of the code is handled here.

3.3.2 Externally Integrated Packages

In our project, we will use JSON files to store information in the Filesystem subsystem in our data layer. As far as we have researched, we have decided to use JSON.simple library to manage the filesystem.

In terms of our project's GUI, we use JavaFX built-in library in Java SE 8. The packages that we are currently using are listed below other than Java 8 standard library, which can be found in this link.

9. **javafx.scene:** Includes the core set of base classes for JavaFX Scene Graph API
10. **javafx.application:** Includes the application life-cycle classes
11. **javafx.fxml:** Contains classes for loading an object hierarchy from an XML-like markup.
12. **javafx.event:** Includes basic framework for JavaFX events, their handling and delivery.
13. **javafx.stage:** Includes the top-level container classes for JavaFX content
14. **com.github.cliftonlabs.json_simple:** A free lightweight utility for deserializing and serializing Javascript Object Notation (JSON). We are planning to use the 3rd version.

3.4 Class Interfaces

In this section, we are describing the class interfaces of our code.

3.4.1 GameEngine Class

The GameEngine class is the main controller for our game. It keeps all entities and controls them upon the user's actions.

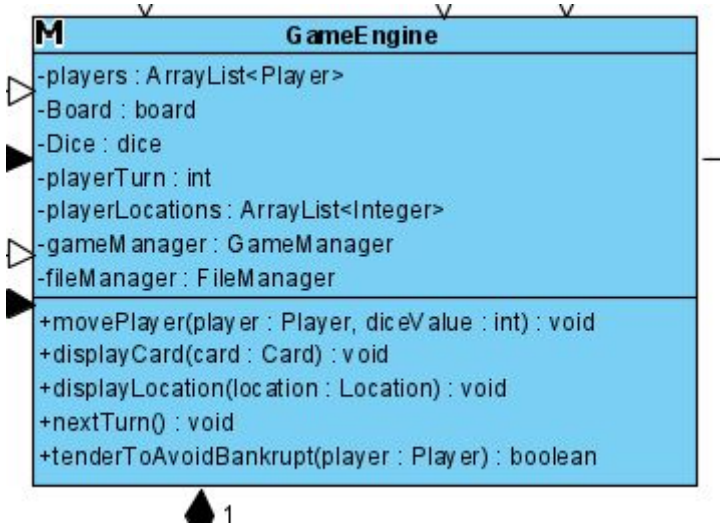

Figure 4: Game Engine Class Diagram

Attributes:

- **private ArrayList<Player> players:** This attribute keeps the list of all players in the game. Note that players' order in the list is the same as their turn order.

- **private Board board:** This attribute keeps the game board. Details of the board will be explained later.

- **private Dice dice:** This attribute represents the dice.

- **private int playerTurn:** This attribute keeps the index of the current player in the players array. It is incremented at each turn.

- **private ArrayList<Integer> playerLocations:** This attribute keeps the location indexes of player tokens in the board. It is modified while moving the tokens.

- **private GameManager gameManager:** Instance of game manager which will control the movements in the game.

Methods:

- **void movePlayer(Player player):** Moves the token of the current player according to the dice result.

- **void displayCard(Card card):** Tells the UI to display the certain card information to the screen.

- **void displayLocation(Location location):** Tells the UI to display the certain location information to the screen.

- **void moveTurn():** This method is called when the player clicks end turn in the user interface. When called, this method ends the turn of the current player, determines the next player and gives the turn to the next player.

3.4.2 Game Manager Class

The Game Manager is responsible for the control of other specific managers.

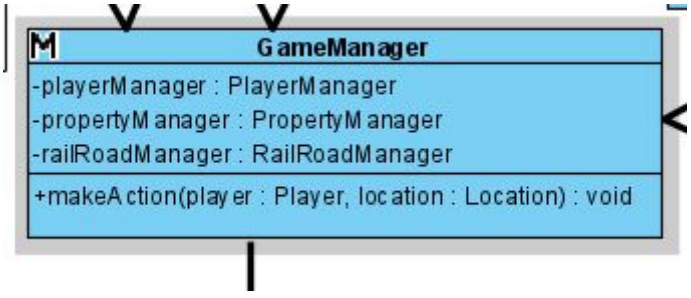

Figure 5: GameManager Class Diagram

Attributes:

- **private PlayerManager playerManager:** This attribute is responsible for player actions. We will explain detailly in the PlayerManager class interface.

- **private PropertyManager propertyManager:** This attribute is responsible for managing properties when players locate on a property. We will explain detailly in the PropertyManager class interface.

- **private RailRoadManager railRoadManager:** This attribute is responsible for managing properties when players locate on a property. We will explain detailly in the RailRoadManager class interface.

Methods:

- **public void makeAction(player: Player, location: Location):** This method is called after the player is moved to a certain location. It acts upon location type. It controls the player, property and rail road managers.

3.4.3 Property Manager Class

This class includes the property related methods.

Figure 6: PropertyManager Class Diagram

Methods:

- **boolean buyProperty(Player player, Property property):** It is called when a player tries to buy a certain property. If a player's current money is higher than the property's price, the property is added to the player and the price is taken from the player. If a player doesn't have enough money, a false is returned.

- **boolean sellProperty(Player player, Property property):** It is called when a player tries to buy a certain property. It takes the property from the player and gives the price of property to the player.

- **boolean buildBuilding(Player player, Property property):** It is called when the owner of a property wants to build a building. If the owner has enough money, and property doesn't have a maximum number of houses, then the building is built and the price is taken from the player. Else the action fails.

- **boolean canBuildProperty(Player player, Property property):** Checks the condition given above buildBuilding method.

- **void updatePropertyRegion(PropertyRegion propertyRegion):** Checks whether each property in the property region has the same owner. If so, all properties in this particular region are updated by the updateProperty function.

- **void updateProperty(Property property):** Updates the property by upgrading its price and rent.

- **sellBuilding(Player player, Property property):** The building number in the property is decreased, player takes the price of the building.

- **sellAllBuildings(Player player, Property property):** Sells all buildings at once.

3.4.4 RailRoad Manager Class

This class manages the railroad locations of the game.

Figure 7: Railroad Manager Class Diagram

Methods:

- **void buyRailroad(Player player, RailRoad railRoad):** Player buys the railroad like a property, except all other railroads' price is increased. updateRailroad method is called for that purpose

- **void sellRailroad(Player player, RailRoad railRoad):** Player sells the railroad like a property, except all other railroads' price is decreased. updateRailroad method is called for that purpose

- **void updateRailroad(RailRoad railRoad):** As we described in previous parts, railRoad prices are updated according to the number of railroads unoccupied.

3.4.5 Player Manager Class

This class is for managing the functionalities of the Player entity.

Figure 8: Player Manager Class Diagram

Methods:

- **boolean buyProperty(Player player, Property property):** This method is called with the buyProperty method of the property manager. There is a work allocation between those two methods, player related work is done in this method, property related work is done in the method in property manager.

- **boolean sellProperty(Player player, Property property):** This method is called with the sellProperty method of the property manager. There is a work allocation between those two methods, player related work is done in this method, property related work is done in the method in property manager.

- **boolean buyRailroad(Player player, Railroad railroad):** This method is called with the buyRailroad method of the railroad manager. There is a work

allocation between those two methods, player related work is done in this method, railroad related work is done in the method in railroad manager.

- **boolean sellRailroad(Player player, Railroad railroad):** This method is called with the sellRailroad method of the railroad manager. There is a work allocation between those two methods, player related work is done in this method, railroad related work is done in the method in railroad manager.

- **boolean takeCardAction(Player player, Card card):** After a player comes to one of the community chest or chance locations, this method is called with the selected card and the current player. Player simply takes action according to what is written in the card

- **boolean withdrawMoney(Player player, int amount):** When player withdraws money, the amount is taken from the player with that method

- **boolean goToJail(Player player):** When one of the conditions for the player to go to the jail occurs, this method is called to represent the players' going to jail.

- **boolean useFreedomRight(Player player):** When the player is doomed to jail, if he has a freedom right, he can use it and escape from jail. This method is called when a player asks for freedom. According to the existence of the freedom right card, the user escapes from the jail sentence or not.

- **boolean hasFreedomRight(Player player):** Checks the condition mentioned above.

- **boolean canAfford(Player player, int amount):** Checks whether the player has at least the given amount of money or not.

- **int calculateNearestRailroad(Player player):** This method is specifically created for a certain community card. Method finds the nearest railroad to the player in the board and returns the distance between them as integer, note that negative integers can also be returned which means the nearest railroad is at the back of the player.

3.4.6 Player Class

This is the class for the Player entity.

Figure 9: Player Class Diagram

Attributes:

- **int id:** represents the id of the player in the game. Each player has a specific id and this id is used to process the player.

- **String name:** The nickname of the player which is selected by the player itself during the initialization of the game.

- **int moneyAmount:** This attribute represents the amount of money that the player has during the gameplay.

- **Color playerColor:** The color of player is also selected by the player itself during initialization. This color is used in the game in many different places. For example, the color of the token of the player is playerColor. Also whenever the player buys a new location, this location gets colored with this color.

- **Token playerToken:** The token of the player is used to represent at which location the player is. As said earlier, the token's color is the same as playerColor.

- **HashMap <String, Property> properties:** Represents the properties that the player owns. It is designed as a hashmap in order to make it easier to search through the properties of a player. Because there are limited locations in a board, a very small sized hash map is enough.

- **int jailStatus:** Represents the jail status of a player. It is 0 when the player is not in jail. 1 means there is one turn for the player to get out of the jail. 2 and 3 have a similar meaning with 1.

- **int noOfFreedomRights:** Represents the number of freedom rights. They are gained from the cards and can be spent to escape from the jail. One card is used for one escape from jail.
- **HashMap <String, Property> railroads:** Represents the railroads that the player owns. It is designed as a hashmap in order to make it easier to search

through the railroads of a player. Because there are limited locations in a board, a very small sized hash map is enough.

- **int timeSpent:** Blitz mode requires keeping remaining time for each of the players. This attribute works for Blitz. It keeps the total time spent by a certain player.

- **int noOfRailroads:** Keeps the number of railroads that the player owns.

- **int noOfProperties:** Keeps the number of properties that the player owns.

3.4.7 Property Class

This class represents the Property entity.

Figure 10: Property Class Diagram

Attributes:

- **string Name:** This attribute represents the name of the property. It is displayed on the board with this name.

- **int cost:** This attribute represents the cost of the property. Players who step on that property need to pay this cost to buy this property.

- **Player owner:** Represents the owner of the property. Only the owner has the rights to do actions on the property.

- **PropertyRegion region:** Represents the region that the property is inside. Note that the region is important when a player makes a monopoly of this region.

- **int currentMortgagePrice:** This attribute represents the current mortgage price. If the owner of this property wants to mortgage the property, he takes this amount of money in return.

- **ArrayList<Integer> rentPrices:** This attribute represents the rent prices of this property. The rent prices are kept in an array list.

- **ArrayList<House> houses:** The array list houses keep the houses built in the property.

- **Hotel:** Represents the hotel built in the property, if there is one.

- **int noOfBuildings:** This attribute represents the number of buildings (total of houses and hotel) in the property.

- **boolean buildable:** This attribute represents whether a building can be built on the property. Note that if its number of buildings reaches to the maximum number, a new building cannot be built.

3.4.8 Railroad Class

This class is the Railroad entity.

Figure 11: Railroad Class Diagram

Attributes:

- **string Name:** This attribute represents the name of the railroad. It is displayed on the board with this name.

- **int cost:** This attribute represents the cost of the railroad. Players who step on that railroad need to pay this cost to buy this railroad.

- **Player owner:** Represents the owner of the railroad. Only the owner has the rights to do actions on the railroad.

- **int currentMortgagePrice:** This attribute represents the current mortgage price. If the owner of this railroad wants to mortgage the railroad, he takes this amount of money in return.

- **ArrayList<Integer> rentPrices:** This attribute represents the rent prices of this railroad. The rent prices are kept in an array list.

- **int noOfRailroadsOwnedByTheOwner:** This attribute represents the number of railroads owned by the owner of this railroad. It can also be updated by updateNoOfRailroadsOwnedByTheOwner method.

3.4.9 Chance Class

This class represents the Chance location on the board.

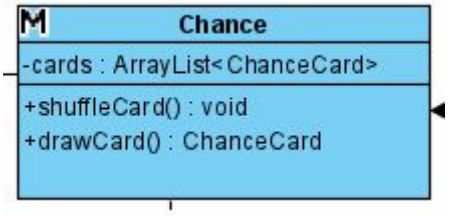

Figure 12: Chance Class Diagram

Attributes:

- **ArrayList<ChanceCard> cards:** Represents the pack of chance cards.

Methods:

- **void shuffleCard():** Shuffles the card pack kept as “cards” in this class. This shuffle work is done just after this class is created.

- **ChanceCard drawCard():** Represents the draw card action in the package. The card at the top of the card pack is drawn without replacement and returned.

3.4.10 Chance Card Class

This class represents the Chance Card entity of the game.

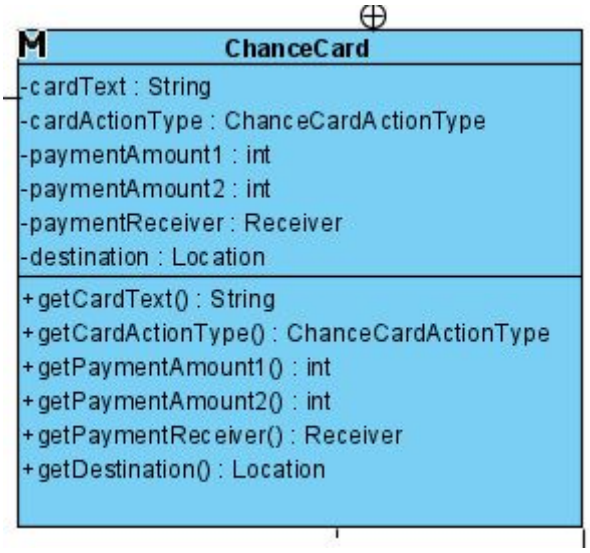

Figure 13: Chance Card Class Diagram

Attributes:

- **String cardText:** This is the text which describes what chance card does. Of course this string will not be the only explanation, but it is the most textual and non numeric description.

- **ChanceCardActionType cardActionType:** In our game, we have 5 different types of cards. These types are: “Go, GoToNearestRailroad, GoToJail, GetPayment, MakePayment, MakePaymentForHouseAndHotels” This variable keeps which type this card has.

- **int paymentAmount1:** This attribute keeps the first payment amount described in the card description.

- **int paymentAmount2:** This attribute keeps the second payment amount described in the card description.

- **Receiver paymentReceiver:** If a payment occurs as a result of the card action, this attribute keeps who will be the receiver of this payment. A payment receiver can be: “EveryoneExceptCurrentPlayer, CurrentPlayer, NoOne, Bank”.

- **Location destination:** As a result of the card action, the player might go to a location, or destination. This attribute keeps which location the player will go after the card action.

3.4.11 CommunityChest Class

This class represents the Community Chest location on the board.

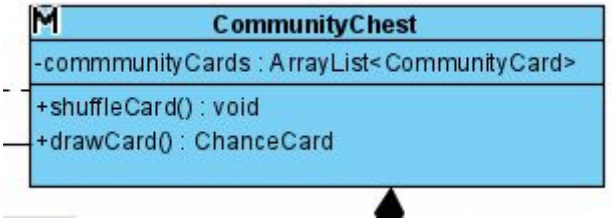

Figure 14: Community Chest Diagram

Attributes:

- **ArrayList<CommunityCard> cards:** Represents the pack of chance cards.

Methods:

- **void shuffleCard():** Shuffles the card pack kept as “cards” in this class. This shuffle work is done just after this class is created.

- **CommunityCard drawCard():** Represents the draw card action in the package. The card at the top of the card pack is drawn without replacement and returned.

3.4.12 Community Card Class

This class represents the Community Chest Cards of the game.

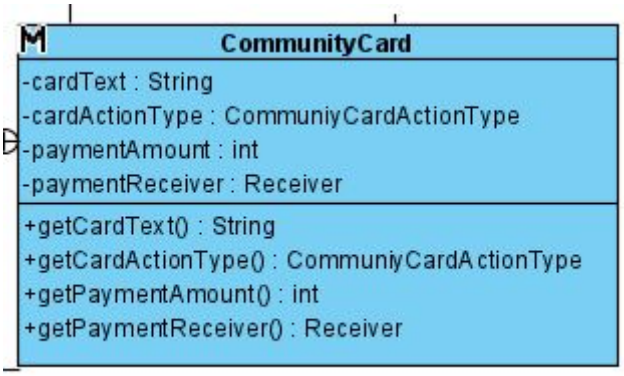

Figure 15: Community Chest Card Diagram

Attributes:

- **String cardText:** This is the text which describes what chance card does. Of course this string will not be the only explanation, but it is the most textual and non numeric description.

- **CommunityCardActionType cardActionType:** In our game, we have 5 different types of cards. These types are: “GetPayment, MakePayment, ObtainFreedomRight, GoToStartingPoint, GoToJail” This variable keeps which type this card has.

- **int paymentAmount:** This attribute keeps the payment amount described in the card description.

- **Receiver paymentReceiver:** If a payment occurs as a result of the card action, this attribute keeps who will be the receiver of this payment. A payment receiver can be: “EveryoneExceptCurrentPlayer, CurrentPlayer, NoOne, Bank”.