**BILKENT UNIVERSITY**

**ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER ENGINEERING**


# CS319

# Iteration 2

# Project Design Report

# Group 1F-Monopoly

**Göktuğ Gürbüztürk 21702383**

**Ege Şahin 21702300**

**Aybars Altınışık 21601054**

**Ahmet Feyzi Halaç 21703026**

**Bulut Gözübüyük 21702771**

# 1 Introduction

## 1.1 Purpose of the System

Monopoly is a local multiplayer game which aims to enjoy its players. The aim of players is to drive another player into bankruptcy. It will provide extra features compared to the traditional Monopoly board game such as minigames in certain locations in order to increase the entertainment of the digital version. Furthermore, in game music and sound effects will have a contribution to maximize in game experience. The theme of the game is also about the pandemic, which introduces virus infection and spreading features to the game.

## 1.2 Design Goals

### 1.2.1 Ease to use

Monopoly should present a good user experience in order not to confuse players. The buttons, pop-ups and descriptions will be well explained enough. Furthermore, there will also be small in game tutorials to inform users how to play the game according to the rules.

### 1.2.2 Portability

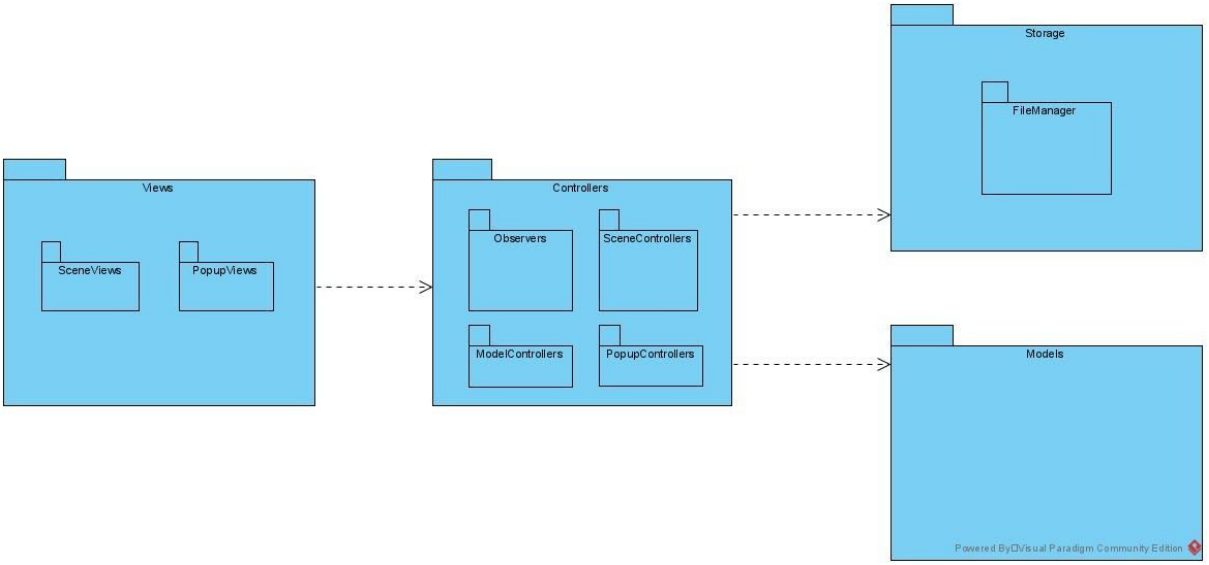
Since the Monopoly game will be written using Java programming language, it's going to be cross-platform software. The Java Virtual Machine provides us to run our game on all operating systems, so that many users will be able to play the game.

### 1.2.3 Good Documentation

Another requirement of a good design is having a good documentation. The code written should be easily understood by other developers so that any developer can analyze this project easily.

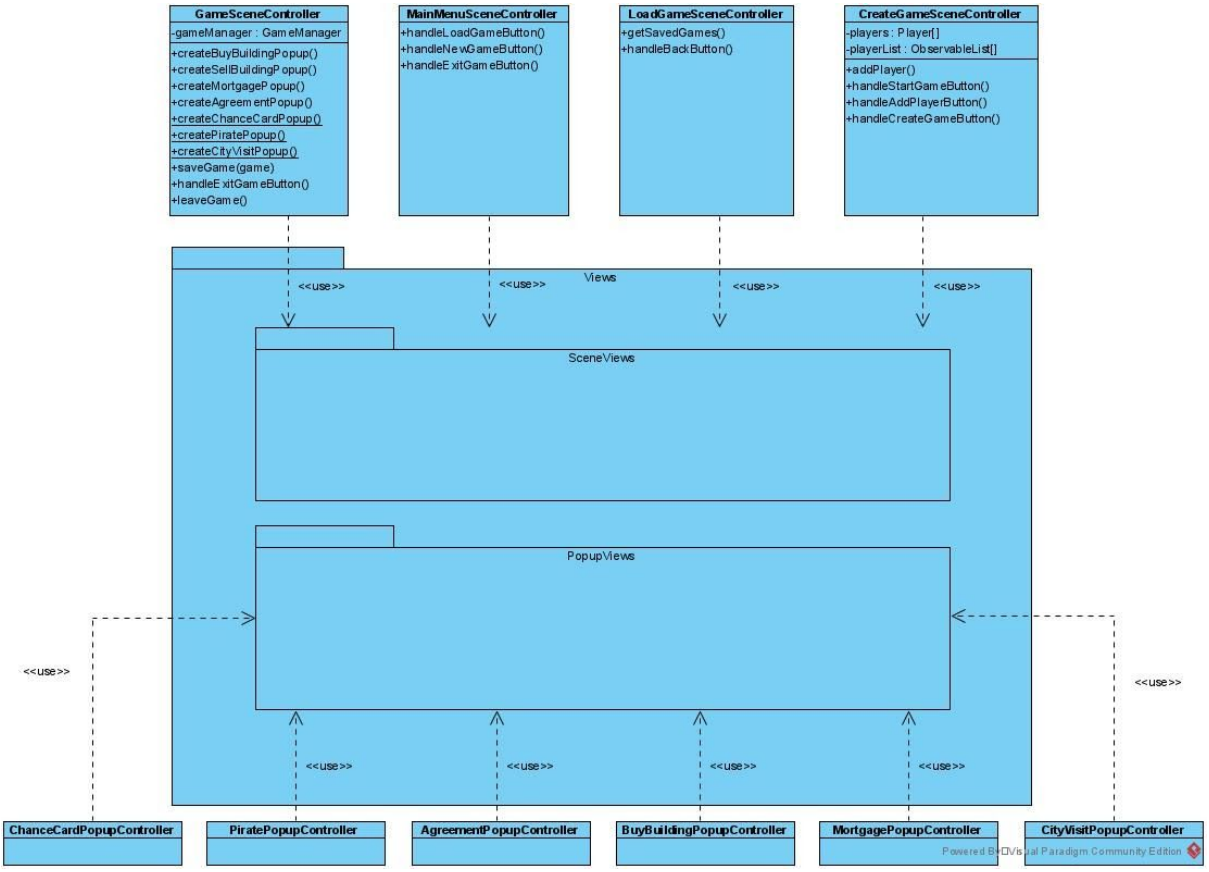
# 2 Proposed Software Architecture

## 2.1 Subsystem Decomposition



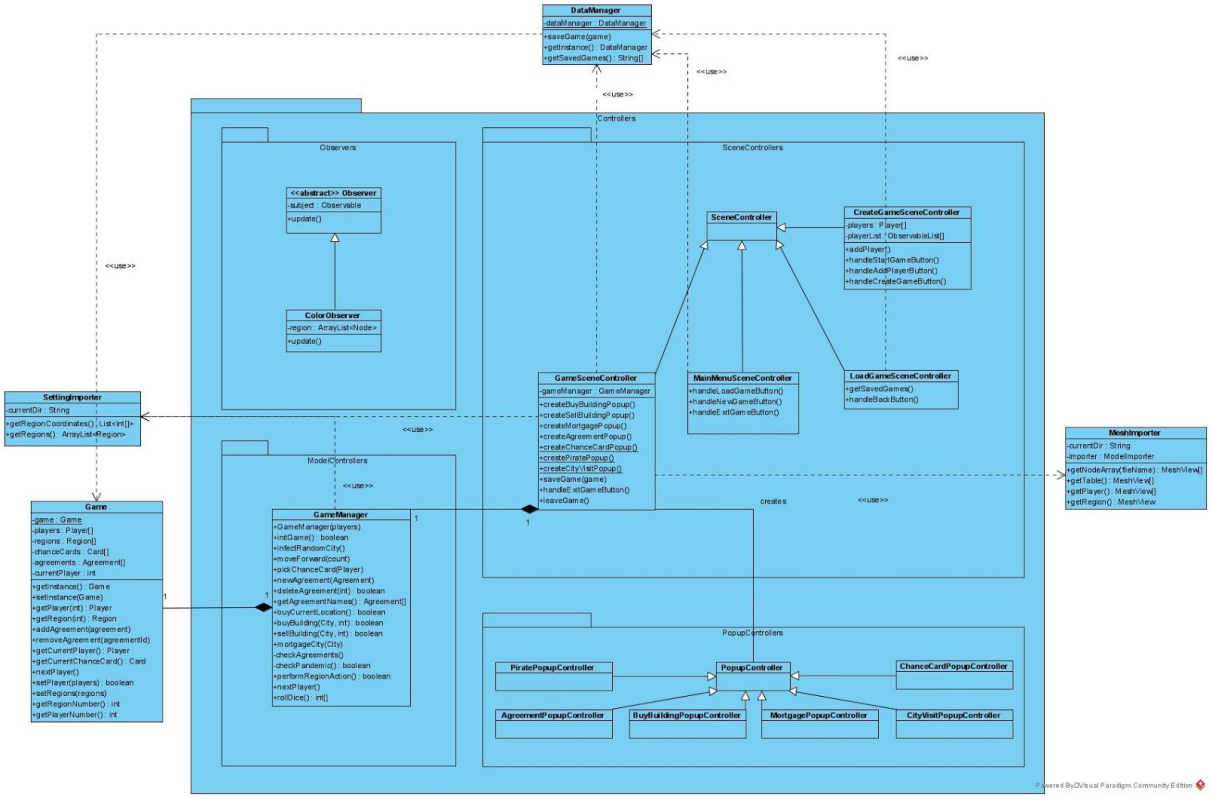

System is decomposed to four subsystems. Views subsystem handles the operations related to the user interface. Controllers subsystem consists of the main logic of the game and controls the models according to the game state. Storage subsystem manages file operations on the disk. Models subsystem keeps game models in the memory.

### 2.1.1 Views Subsystem



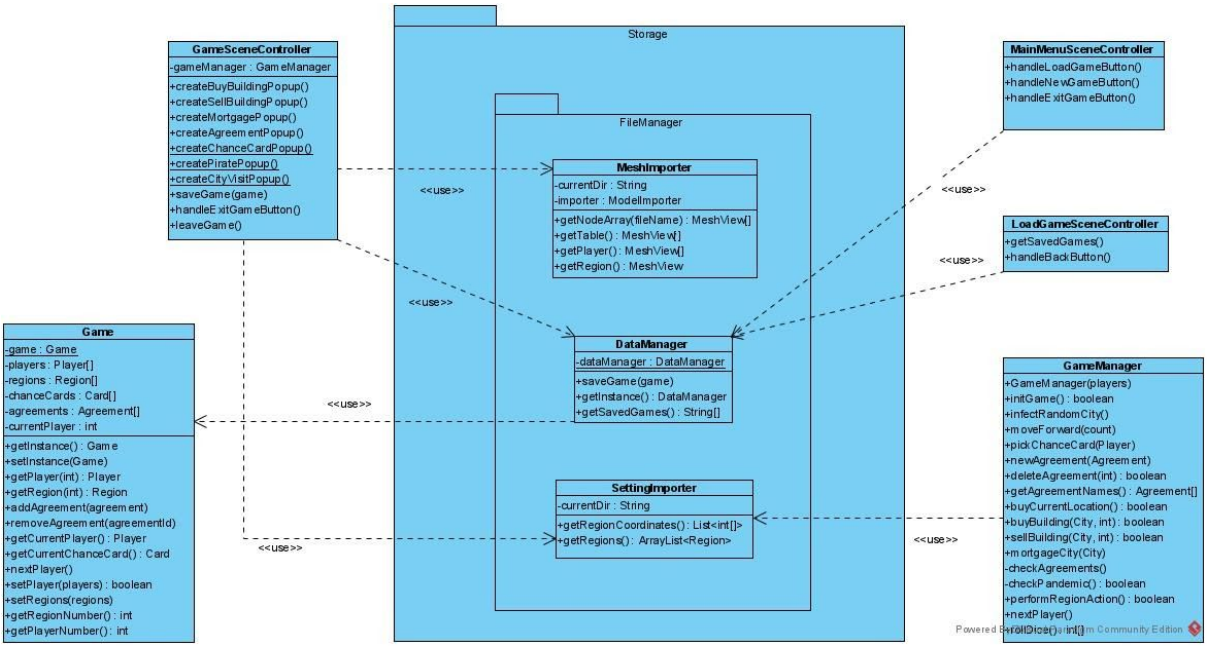

Views subsystem is composed of two packages. SceneViews has the Layout (UI) files which are related to the four scenes. PopupViews has the Layout files which are related to the six popups. Due to layout files has .FXML extension. They are not included in the Subsystem Diagram.

### 2.1.2 Controllers Subsystem



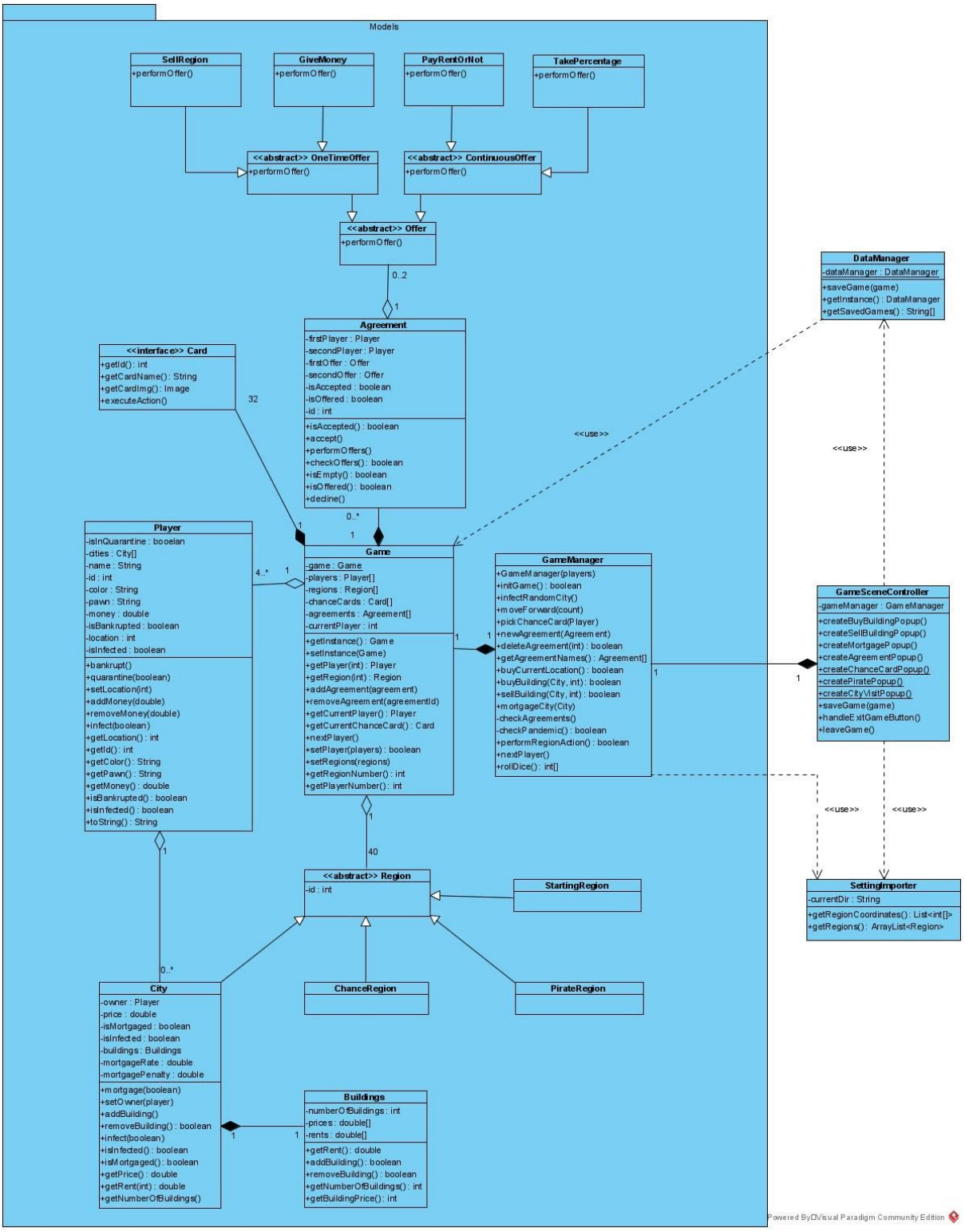

Controller subsystem is composed of four packages. SceneControllers and PopupController have the controller classes for each layout in SceneViews and PopupViews that is discussed in section 2.1.1. The ModelControllers package contains the classes that establish the connection between Models and Interface Controllers. Main logic of the game is managed by the GameManager class. Observers package contains classes related to the Observer design pattern.

### 2.1.3 Storage Subsystem




Storage subsystem is composed of one package. The FileManager package contains three classes; DataManager, MeshImporter, SettingImporter. DataManager class establishes a connection to the save files. Loading the game and saving the game operations will be conducted by this class. MeshImporter class imports the mesh files into the game scene. SettingImporter class imports Region information to start a game.

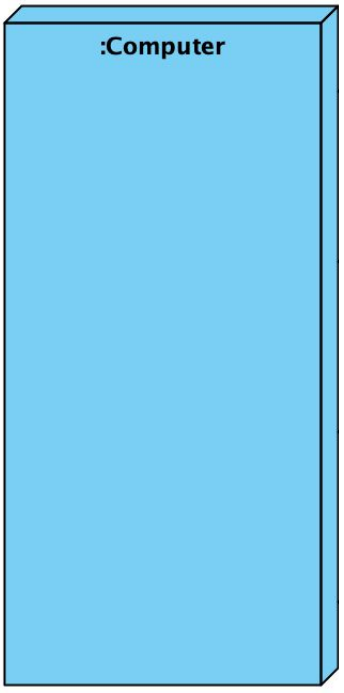
### 2.1.4 Models Subsystem



Models subsystem consists of classes related to the objects that will be kept in memory while playing the game.

## 2.2 Hardware & Software Mapping

The Monopoly game will be written using Java language so that Java Runtime Environment (JRE) is needed to run the game. Since Monopoly is a local multiplayer game, there is no need for an internet connection or server. The system of the game contains only one type of node which is Computer.



## 2.3 Persistent Data Management:

Following information must be stored in order to achieve the saving and loading functionality of a continuing game:

- All Players.
- All Regions. If the type of the region is city, following information must be stored:
    - number of buildings city have
    - owner (Player)
    - whether city is mortgaged
    - whether city is infected
- All chance cards, along with their ordering and the card at the top.
- All agreements that currently active. Each agreement consists of this information:
- Current player

Since Game object contains all this information and there is a single instance of the Game object in runtime (singleton), storing mentioned instance of the Game class is enough to store the game's state. This functionality can be achieved by making these classes implement Serializable interface:

- Player
- Game
- Region
- Agreement
- Offer
- Card

Making these Serializable is required in order to save Game object to a file or get a Game object from a saved file. After this, storing and loading of the Game object can be done with the help of ObjectInputStream and ObjectOutputStream classes of Java.

## 2.4 Access Control

In this system, there are two types of user which are host and player. However, the host is actually one of the players in the game. So, it can be said that there is only one user type which is a player during the game. There is no authorization in the system and all players have the same functionalities. All players are able to intervene in the game in their turns. It is not possible to perform transactions other than their own turns. Also, in this system, there are no server and remote storage. Therefore, players can access all data and functionalities in their own turns.

## 2.5 Global Software Control

Event-driven control flow is used in the application because the application is mainly controlled by user actions. Main loop always waits for external events and the program will continue according to those events. There are more than one event that can happen in a turn of players such as buying a region or proposing an agreement. According to choices of the player, functions will be called, objects will be modified and the game will be shaped.
In addition to that, centralized control flow is used in the game. Much of the dynamic behavior is placed on the game manager object. Game manager object communicates with most of the other objects and commands them.

## 2.6 Boundary Conditions

### 2.6.1 Initialization (Start-up)

The game will be started by using an executable program. The UI components of the main menu will be loaded when the player opens the executable file. If the player wants to create a new game, he/she selects the new game button and the adding player window is opened. When the players are added, the game will be initialized with game UI components and data. Data of the players and the game are stored in the program from this point until the game is closed. However, if the player continues to play a game from the saved one, he/she selects the saved version. Data of the saved game and it's states are saved in a file. Thus, when the player selects the saved version of the game, the data in the file are loaded into the program and the game starts with required UI components and stored data from saved values.

### 2.6.2 Termination (Shutdown)

The player terminates execution of the game either by saving or not saving. If he/she wants to save the game before shutdown, he/she clicks the save button and the data and states of the game are stored in a file. After that, the player can terminate the game by clicking the exit button. Thus, player who wants to play a saved game, he/she selects the saved version and the data of the game is loaded from the file. However, if he/she directly exits by clicking the exit button without saving, the game is automatically terminated and not saved. In addition to that the player can exit from the game while he/she is in the main menu screen by clicking the exit button directly.

### 2.6.3 Exceptions

#### 2.6.3.1 Software fault

Since writing bug-free software is difficult, there can be some bugs during the run-time. These bugs can cause the system to crash and be closed suddenly. This type of bug causes data loss because they are not saved while the game is closed due to run-time error. Since reliability is an important design goal for this system, it is needed for the system to react safely to these kinds of run-time errors. Therefore, one way to protect against these errors is saving data consistently. In order to achieve the protection, there will be a saving functionality for the players. Players will be able to save games any number of times during the game. In addition to that, games will be saved automatically in every three rounds and the data loss will be minimized. Thus, when the game is terminated due to errors, the player will be able to load the saved game and continue to play it.

# 3 Low Level Design

**Agreement Class:**

Agreement class contains the information about each agreement object.

**Attributes:**

- **private Player firstPlayer:** The player who offers the agreement.
- **private Player secondPlayer:** The player who is offered an agreement.
- **private Offer firstOffer:** Offer of the first player.
- **private Offer secondOffer:** Offer that is asked from the second player.
- **private boolean isAccepted:** This is true if the second player accepts the offer, false otherwise.
- **private boolean isOffered:** This is true if the agreement is seen by the second player.
- **private String name:** Name of the agreement.
- **private int id:** Id of the agreement**.**

**Methods:**

- **public void performOffers():** If the agreement is accepted by the second player and both first offer and second offer are one time offers, then agreement is performed and deleted from the agreements array of the game class. If one or two of the offers are continuous, this agreement continues to exist in the agreements array and is checked in every turn by

checkAgreements() method of GameManager class and performAgreement() function is called when necessary. This function calls performOffer() methods of the offer objects.

- **public boolean checkOffers(City city):** Returns whether there is an agreement on the city.
- **public boolean isEmpty():** Returns whether agreement is completed.
- **public boolean isOffered():** Returns whether second play is accept the offer.
- **public boolean isAccepted():** Returns true if the second player accepts the offer, false otherwise.
- **public void accept(boolean bool):** If the bool is true agreement will be accepted, otherwise it will be declined.

**Offer Class:**

This is an abstract class which will be used as two types of offer. (One time offer, continuous offer)

**Methods:**

- **public abstract void performOffer(Player firstPlayer, Player secondPlayer):** This method will be used in children classes of Offer class.

**OneTimeOffer Class:**

This is an abstract class which will be used as two types of offer. (SellRegion, GiveMoney)

**Methods:**

- **public abstract void performOffer(Player firstPlayer, Player secondPlayer):** This method will be used in children classes of OneTimeOffer class.

**ContinuousOffer Class**

This is an abstract class which will be used as two types of offer. (PayRentOrNot, TakePercentage)

**Methods:**

- **public void performOffer(Player firstPlayer, Player secondPlayer):** This method will be used in children classes of ContinuousOffer class.

**SellRegion Class:**

This class is inherited from the OneTimeOffer class.

**Methods:**

- **public void performOffer(Player firstPlayer, Player secondPlayer):** This method is called by performAgreement() method of agreement class and changes the owner of the city.

**GiveMoney Class:**

This class is inherited from the OneTimeOffer class.

**Methods:**

- **public void performOffer(Player firstPlayer, Player secondPlayer):** This method is called by performAgreement() method of agreement class, transfers money from one player to another.

**PayRentOrNot Class:**

This class is inherited from the ContinuousOffer class**.**

**Methods:**

- **public void performOffer(Player firstPlayer, Player secondPlayer):** This method is called by performAgreement() method of agreement class, forming a partnership about paying rent.

**TakePercentage Class:**

This class is inherited from the ContinuousOffer class.

**Attributes:**

- **private City city:** The city which will be shared
- **private double percentage:** The percentage of how city will be shared

**Methods:**

- **public void performOffer(Player firstPlayer, Player secondPlayer):** This method is called by performAgreement() method of agreement class, distributes the profit of a region(s) between two players.
- **public City getCity():** returns the city object

**Card Class:**

Chance cards are the object of this class.

**Attributes:**

**Methods:**

- **public void getId:** Getter method of card's id.
- **public void getCardName():** Getter method of card name.
- **public void getCardImg():** Getter method of card image file name.
- **public void executeAction():** Gets the car id and the player and changes player object's attributes according to the action of the chance card. This method is called by the pickChanceCard(Player player) method of GameManager class.

**Region Class:**

This is an abstract class which is inherited by City class, ChanceRegion class and SpecialRegion class.

**Attributes:**

- **Int id:** Id of the region.

**Methods:**

- **public int getId():** Return the id of the city.

**City Class:**

This class is inherited from region class.

**Attributes:**

- **private Buildings buildings:** Buildings in the city.
- **private Player owner:** Owner of the city.
- **private double[] rents:** Array of rents. (rent without building on it, rent with 1 house, rent with 2 houses...)
- **private double price:** Price that one must pay to the bank to buy the building.
- **private boolean isMortgaged:** True is the city is mortgaged.
- **private boolean isInfected:** True is the city is infected.
- **private double mortgageRate:** is a rate that the owner will obtain when the corresponding city is mortgaged
- **private double mortgagePenalty:** The penalty that will be paid when the owner cancelled mortgage on the city

**Methods:**

- **public void mortgage(boolean bool):** If the bool is true, mortgage the city.
- **public void setOwner(Player player):** This function sets the owner of the city when a player buys it or gets it with an agreement. So, this function is used in the buyCurrentLoction() function of the game class.
- **public void addBuilding(int count):** This method is called by the buyBuilding(city, numberOfBuildings) method of the GameManager class. Adds specified number of buildings to the city.
- **public void removeBuilding(int count):** This method is called by sellBuilding(city, numberOfBuildings) method of the GameManager class. Removes specified number of buildings to the city.
- **public void infect(boolean bool):** If bool is true infects the city, otherwise does nothing. This method is used by the infectRandomCity() method of the GameManager class.
- **public boolean isInfected():** Returns true if the city is infected false otherwise.
- **public boolean isMortgaged():** Returns true if the city is infected false otherwise.
- **public double getPrice():** Return the price of the city**.**

- **public double getRent():** Returns the rent of the city with respect to buildings on the city.
- **public int getNumberOfBuildings():** Returns the the number of buildings in the city.
- **public int getBuildingPrice(int count):** return the building price according to the count.
- **public Player getOwner():** Returns the owner of the city

**Buildings Class:**

**Attributes:**

- **int numberOfBuildings:** number of buildings
- **double[] prices:** price of the buildings
- **double[] rent:** rent according to the number of buildings

**Methods:**

- **public boolean addBuilding():** Adds building and returns true if operation is successful.
- **public boolean removeBuilding():** Removes building and returns true if operation is successful
- **public int getNumberOfBuildings():** Returns the number of buildings.
- **public double getBuildingPrice():** Returns the price of the specified building.

**ChanceRegion Class:**

This class is inherited from region class.

**StartingRegion Class:**

This class is inherited from region class.

**PirateRegion Class:**

This class is inherited from region class.

**GameSceneController Class:**

This class mainly controls the pop ups of the game and some other actions which are explained below.

**Attributes:**

- **private GameManager gameManager:** By using this attribute, GameScene class uses the GameManagerr class' methods.

**Methods:**

- **public void createBuyBuildingPopup():** This method is called when a player is buying a building. It creates a popup which includes details about

purchasing action. It also uses the buyBuilding(city, numberOfBuildings) method of the GameManager class.

- **public void createSellBuildingPopup():** This method is called when a player is selling a building. It creates a popup which includes details about selling action. It also uses the sellBuilding(city, numberOfBuildings) method of the GameManager class.
- **public void createMortgagePopup():** This method is called when a player is mortgaging a region. It creates a popup which includes details about mortgage action. It also uses the mortgageCity(City city) method of the GameManager class.
- **public void createAgreementPopup():** This method is called when a player is offering an agreement or offered an agreement. It creates a popup which includes details about agreement.
- **public static void createChanceCardPopup():** This method is used when the user comes to a chance region. This popup includes information about the chance card such as what the chance card asks.
- **public static void createMiniGamePopup():** This method is used when the user comes to a mini game region. This popup includes the mini game.
- **public static void createCityVisitPopup():** This method is used when the user comes to a city region. This popup includes information about the city such as how much is the rent or price.
- **public void saveGame(Game game):** This method writes the information about the game to a file. This file is used later to load the same game.
- **public void exitGame():** This method exits the game window.
- **public void leaveGame():** This method leaves the game and goes to the main menu.

**MainMenuSceneController Class:**

This class is used in the main menu.

**Methods:**

- **public void loadGame():** Directs player to load a game screen.
- **public void newGame():** Starts a new game, directs player to create game screen.
- **public void exitGame():** This function closes the game window and exits the game.

**CreatGameSceneController Class:**

This class is used to make the adjustments before the game starts.

**Attributes:**

- **private Player[] players:** Player will add the new players to this array before the game starts.

**Methods:**

- **public void addPlayer():** This method is used to add players to the players array.

**LoadGameSceneController Class:**

This class is used to show all loaded games on screen and load one.

**Methods:**

- **public void loadGame(saveName):** This function loads a previously saved game. Uses loadGame(string saveName) method of DataManager class.

**DataManager Class:**

This is a singleton class. This class is used by GameScene and MainMenuScene classes to save games and load games. It uses Game class to save the information when the game is saved.

**Attributes:**

- **private DataManager dataManager:** Singleton instance.

**Methods:**

- **public void saveGame(Game game):** This function saves the information of the game object to a file. This function is used by saveGame(Game game) function of GameScene class.
- **public void getInstance():** Gets the instance of game class and uses game class methods.
- **public void getSavedGames():** This function gets the saved game files.

**SettingImporter Class**

**Attributes:**

- **private String currentDir:** Directory of the project

**Methods:**

- **public List<int[]> getRegionCoordinates():** Returns the coordinates of the regions
- **public ArrayList<Region> getRegions():** Returns the regions

**Game Class:**

Game class contains all the information of the game such as players, regions, chance cards and agreements. When game is started, instance of the Game class is initialized before the game starts. This class is a Singleton.

**Attributes:**

- **private static Game game:** Singleton instance of Game class
- **private ArrayList<Player> players:** All players in the game
- **private ArrayList<Region> regions:** All regions in the game
- **private ArrayList<Card> chanceCards:** All cards in the game
- **private ArrayList<Agreement> agreements:** All continuing agreements in the game
- **private int currentPlayer:** Id of the current Player

**Methods:**

- **public Game getInstance():** Gets the singleton instance of Game class
- **public void setInstance(Game game):** Sets the Game if it is loaded from a file.
- **public Player getPlayer(int id):** Gets the player with specified id.
- **public Region getRegion(int id):** Gets the region with specified id.
- **public Region getRegionNumber():** Gets the number of the region.
- **public void addAgreement(Agreement agreement):** Adds a new agreement.
- **public void removeAgreement(String agreementName):** Removes the agreement with specified id.
- **public Player getCurrentPlayer():** Gets the current player.
- **public Card getCurrentChanceCard():** Gets the chance card at the top.
- **public void nextCard():** Increments the currentChanceCard by 1.
- **public void nextPlayer():** Increments the currentPlayer by 1.
- **public ArrayList<Agreement> getAgreements():** Returns the list of the agreements
- **public int getPlayerNumber():** Returns the number of the players

**GameManager Class**

Manages and edits the properties of the Game instance.

**Methods:**

- **public boolean initGame():** When a new game is created, initializes the game with the information obtained from files. Returns true if it is successful. Properties are initialized like this:
    - players are obtained from CreateGameScene
    - regions are read from Regions file
    - chanceCards are read from Cards file
    - agreements is initialized as an empty array
    - currentPlayer is 0
- **public void infectRandomCity():** Every two turns, resets all cities' infected information as false, randomly chooses a city and infects it.
- **public void moveForward(int count):** Gets the result of dice which current player rolled as argument and calculates the resulting location. In other words, it gets current player's location, adds dice result to it and takes the modulus of it by region count. After it sets the current player's location.
- **public void pickChanceCard():** Picks a card, calls executeAction method of the picked card for the specified player and increment currentChanceCard by 1.
- **public void newAgreement(Offer firstOffer, Offer secondOffer, Player firstPlayer, Player secondPlayer, String agreementName):** Adds a new agreement to the array in the properties when current player offers a new agreement.
- **public boolean deleteAgreement(String agreementName):** Deletes an agreement when it is done. An agreement can be deleted if both offers are one time or owner of a continuous offer is bankrupted.
- **public boolean buyCurrentLocation():** Checks whether current player can buy the city at the current location and if successful, adds that city to current player.
- **public boolean buyBuilding(City city, int count):** Calls canBuyBuilding method for current player and specified city and if it returns true, adds specified number of buildings to city and removes required money from player
- **public boolean sellBuilding(City city, int count):** Removes specified amount of building from specified city and adds money to the current player.
- **public void mortgageCity(City city, boolean bool):** Mortgages the specified city and adds money to current player

- **private void checkAgreements():** Checks all agreements every turn and executes agreements that is satisfied at that turn.
- **private boolean checkPandemic():** Every turn, checks whether all players are infected. If yes, puts all players into quarantine.
- **public void performRegionAction():** Performs the required action for a region when a player lands on it.
- **public int[] rollDice():** Returns the dice
- **public void nextTurn():** When a player clicks the end turn button, this method is called to pass to the next player turn.
- **public String[] getAgreementNames():** Returns the name of the agreements.

**Player Class:**

**Attributes:**

- **private boolean isInQuarantine:** True if player is in quarantine.
- **private ArrayList<City> cities:** All cities player owns.
- **private String name:** Name of the player.
- **private int id:** Id of the player.
- **private String color:** Color of the player.
- **private String pawn:** Pawn type of the player.
- **private double money:** Current money of the player.
- **private boolean isBankrupted:** True if player is bankrupted.
- **private int location:** Id of the region player is currently waiting.
- **private boolean isInfected:** True if player is infected.

**Methods:**

- **public void bankrupt():** Sets isBankrupted true.
- **public void quarantine(boolean bool):** Sets isInQuarantine to parameter.
- **public void setLocation(int regionID):** Sets location to parameter.
- **public void addMoney(double money):** Adds specified amount of money to the money property.
- **public void removeMoney(double money):** Removes specified amount of money from the money property.
- **public void addCity():** Add the city to the player
- **public void removeCity():** Remove city from the player
- **public ArrayList<City> getCities():** Returns the list of the cities