Bilkent University

Department of Computer Science

Object-Oriented Software Engineering Design Report

CS 319 Project: Monopoly Space Edition

Design Report

Ahmet Işık, Elif Özer, Hande Sena Yılmaz, Mehmet Yiğit Harlak, Pelin Çeliksöz

Instructor: Eray Tüzün

TA (s): Barış Ardıç, Elgun Jabrayilzade, Emre Sülün

Table of Contents

Table of Figures

1. Introduction

1.1. Purpose of the system

Monopoly Space EDITION is the digitalised version of the well-known Monopoly board game with additional features. As an advantage of being in a digital environment, the addition of a wide range of new features will become possible.

Monopoly Space EDITION will be a multiplayer desktop game which can be played offline by two to four players. The game will have the space theme and all features will be modified according to that space theme. It is designed to be user-friendly, challenging and easy to learn for players and well documented and reusable for developers.

Monopoly Space EDITION includes an alien entity in addition to the usual Monopoly game. The alien in the game will be able to attack the players, if the players come to the position where the alien is on the board. As a result of this attack, players will try to get rid of the attack with the least damage by developing a strategy.

Each individual player's goal is to be in the game as other players go bankrupt. Last player who economically survives or in the best condition at the end of the game will be the winner. It aims to be entertainable and enjoyable and visually impressive from users point of view.

1.2. Design Goals

1.2.1. Trade-Offs

Functionality vs Usability

Since the game we designed is adapted from Monopoly, which is, a board game, no complex instructions or strategies are included. Therefore we cannot mention any advanced functions compared to most of the other games. Almost every individual can play it so it is usable and cannot be considered as functional as a result of the origin of the game and our design.

Performance vs Maintainability

In the design process of the game, a software easier to maintain has been a priority for us. Therefore we utilized inheritance and abstraction properties and aimed not to have similar functionalities for different objects which would cause confusions in a possible maintenance stage. However, separation of interfaces and classes would cause performance-wise negativities because of memory usage during the game .

Functionality vs Understandability

As mentioned in the first paragraph, design of the software we intend to design could not be described as a complex example for both developer and user side experiences due to less functionality. As a result of it, source code will be more readable for developers and the game will be easier to interpret for users.

Efficiency vs Rapid Development

Since implementation is planned to be made in a short period of time, we focused on other aspects of the software more than time efficiency of it in order to deliver a proper product in the final stage. Therefore, we can mention a rapidly developed software whose efficiency is not a prior concern.

1.2.2. Criteria

1.2.2.1. End-User Criteria

Usability

As Monopoly is a board game which has players from a wide range of ages and different cultures, the game we are going to implement will be playable for that many users as well. Therefore the game has been planned to be simple, user-friendly and understandable. User menu and options are designed in a way which does not confuse players. Also game flow is expected to be easily comprehend by users.

Ease of Learning

Usability and simplicity of the game has been explained in the previous paragraph. In addition to these criterias which have been met by the software, the game would also be played by users who have not sufficient knowledge about the game or people who have played board game version but not similar with digital versions. Help menu would guide users about rules and how to play. Users also will be guided throughout the game.

1.2.2.2. Maintenance Criteria

Modifiability

Three layer architecture is used for the subsystem structure design. As an advantage of this design, maintenance would become easier and only maintenance related parts of the software would be affected.

Reusability

Some of the subsystems which are related to UI are independent from the system. They are designed to be used in implementation of similar games at a further stage.

Good Documentation

Within the analysis and design documents we have created, documentation is one of our priorities. Our goal is to clearly and precisely specify the requirements, limitations and expectations regarding software we will create and describe components, roles and functions of the system we are going to create.

Portability

In order to reach as many users as possible, portability of the software is essential. As JavaFX library will be used in our software, our game would be platform independent and executable in platforms which include Java Virtual Machine (JVM). Therefore it will be available for various devices.

1.2.2.3. Performance Criteria

Fault Tolerance

As any other software product exists, our software is designed to overcome execution errors using proper exception handling mechanisms. Purpose of applying this criteria is to prevent failure of the entire system when one of the subsystems fails.

Response Time

To have a game which is playable and enjoyable for players, users should get response from the system after less than a reasonable time period. Therefore our goal is to update the status of the game according to user input within 1 second.

2. High-level software architecture

2.1. Subsystem decomposition

In order to design our system, we have decided to incorporate 3-Layer architectural style as a pattern for subsystem decomposition because it allows any of 3 layers (presentation, application and data layers) can be developed and modified by different teams/developers in response to changes.

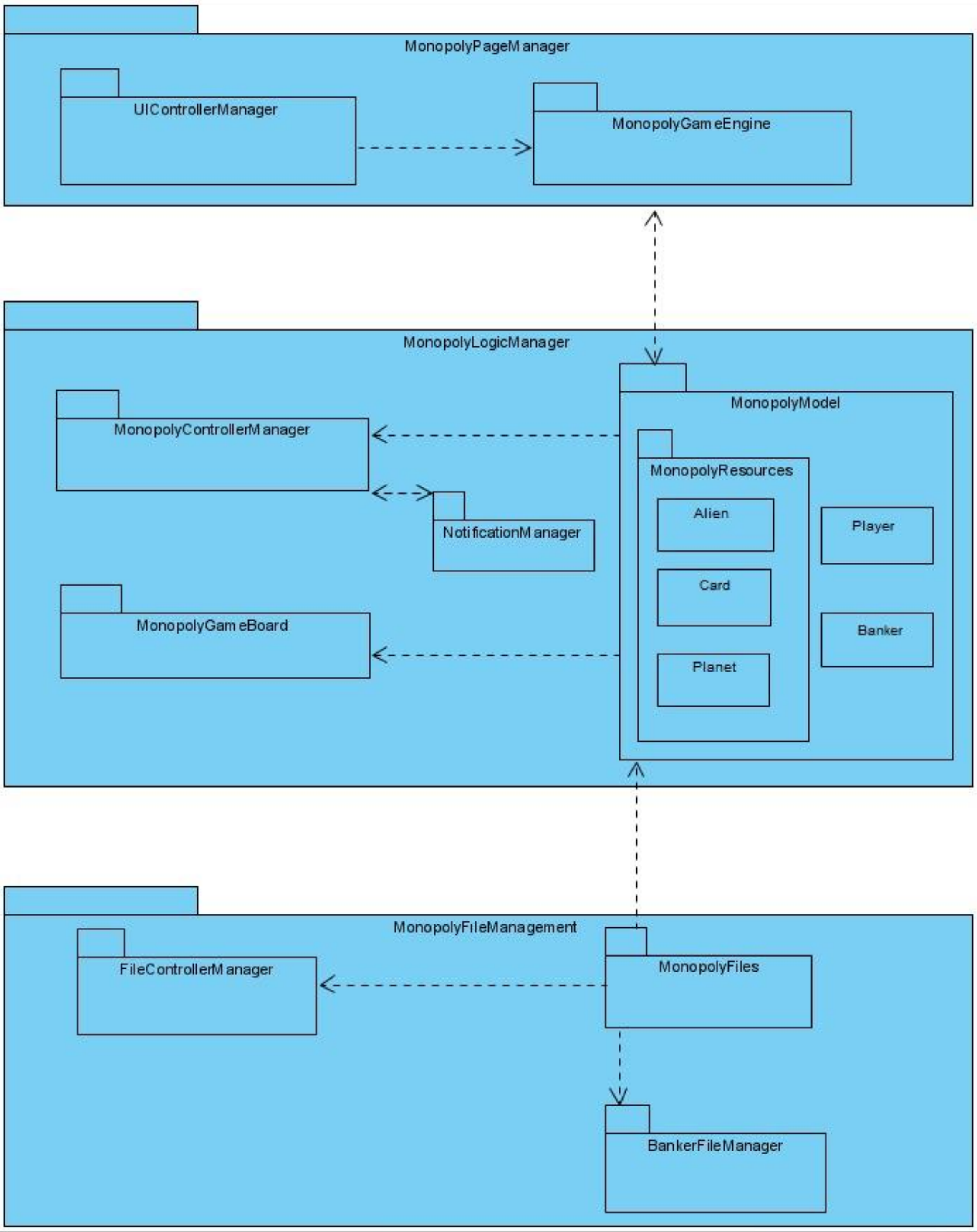


Figure 1 : Package diagram of Monopoly Space Edition Game

In our design, *MonopolyPageManage* represents presentation(client) layer. *MonopolyPageManager* consists of 2 subsystems called *UIControllerManager* and *MonopolyGameEngine*. These 2 subsystems ,in *MonopolyPageManager*, are managing with user interface and the page of the games. Our application layer is named as *MonopolyLogicManager* which contains entities in application domain and game controllers in the application level. *MonopolyLogicManager* contains subsystems named as *MonopolyModel, NotificationManager, MonopolyControllerManager and MonopolyGameBoard.* In our system, the data layer ,in 3-Layer architectural style, represented by *MonopolyFileManagement. MonopolyFileManagement* is responsible for managing player and data in the playing mode. The subsystems in this layer: *MonopolyFiles, FileControllerManager* and *BankerFileManager*.

2.1.1. MonopolyPageManager

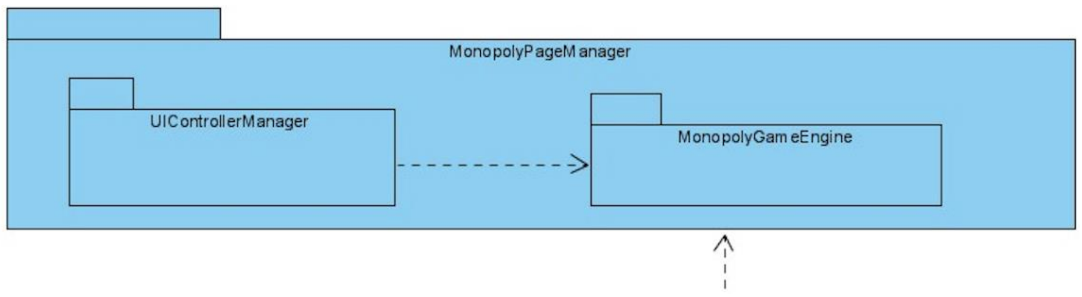


Figure 2 : First Layer of Monopoly Space Edition Game

MonopolyPageManager is responsible for the view of the Monopoly game including 2 subsystems for managing the views of scenes and UI components. Subsystems in this layer are called *UIControllerManager* and *MonopolyGameEngine*, which are responsible for managing the transformation of the scenes in the Monopoly game and also it interacts with the application layer. Interaction with the application layer is for the scenes in the Monopoly game so that scenes of Monopoly are updated based on the responses of the player in the game state.

2.1.1.1. UIControllerManager

This subsystem is responsible for UI components in Monopoly game. To handle UI components, *UIControllerManager* manages the controller logic of the Monopoly scenes. It includes controller classes for solution domain and enables transformation between scenes. Scenes are Main Page, Help Page, Choose Tokens and Players Page, Game Page, End Game Page.

2.1.1.2. MonopolyGameEngine

This subsystem works with the application layer and provides connections between the scenes and the Monopoly game logic so that the transformation of scenes in the Monopoly game are handled based on the game state.

2.1.2. MonopolyLogicManager

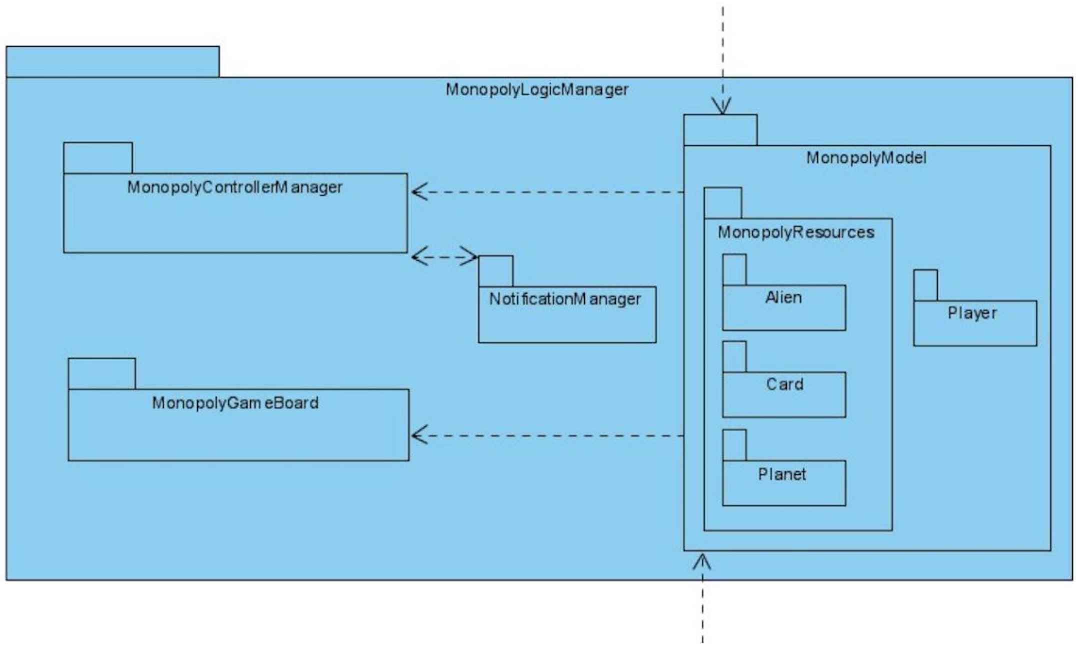


Figure 3 : Second Layer of Monopoly Space Edition Game

MonopolyLogicManager is to control the logic of the Monopoly and game states within their subsystems. This layer contains application domain controllers and application domain entities. The subsystem *MonopolyControllerManager* is the application domain controller which handles transformations in the Monopoly game state by managing notifications/messages between application domain entities. Application domain entities are Monopoly board, players, alien, cards and planets. The Monopoly board state is in the *MonopolyGameBoard* subsystem. In the *MonopolyModel* subsystem, there are game entities, Player, Alien, Card and Planet. The reason why there is another subsystem *MonopolyResources* in *MonopolyModel* is decreasing the coupling between subsystems.

2.1.2.1. MonopolyModel

This subsystem is responsible for creating abstractions of Monopoly game entities, Player, Card, Alien and Planet. In *MonopolyModel,* Card, Alien and Planet are in another subsystem called *MonopolyResources* to decrease coupling between subsystems. In this way, we reduce impacts of changes and errors in one subsystem on the other subsystem.

2.1.2.2. NotificationManager

This subsystem is to handle messages displayed on the Monopoly game based on the players' responses. When player gives a response(i.e. building forest on a planet) while playing the game, this message will be displayed on the game scene by interaction between subsystems *NotificationManager* and *MonopolyControllerManager*.

2.1.2.3. MonopolyControllerManager

This subsystem is for managing the controllers in the application level. The states in the Monopoly game is handled by this subsystem.

2.1.2.4. MonopolyGameBoard

This subsystem is responsible for board state in the Monopoly game and it includes Cards, Alien, Planet and position of the Player on the Monopoly board via interaction with MonopolyModel.

2.1.3. MonopolyFileManagement

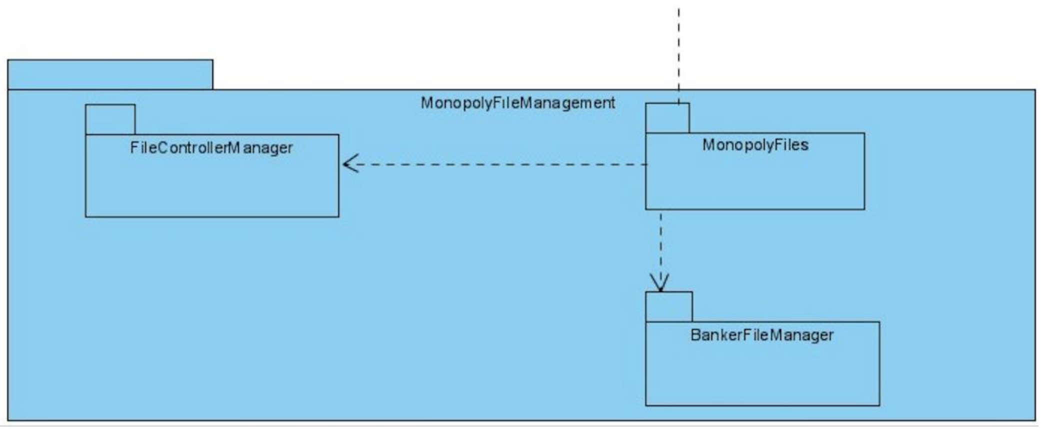


Figure 4 : Third Layer of Monopoly Space Edition Game

MonopolyFileManagement represents the data layer of the 3-Layer architectural style and it is responsible for handling the data, saving or loading files, inside the Monopoly game during the play mode. *MonopolyFileManagement* contains 3 subsystems: *MonopolyFiles, FileControllerManager* and *BankerFileManager*.

2.1.3.1. MonopolyFiles

This subsystem is responsible for connection between *MonopolyGameModel* and *MonopolyFileManagement*. This subsystem will be available at the beginning of the game to receive and update the game objects.

2.1.3.2. FileControllerManager

This subsystem is to handle processes of players' responses during play mode in Monopoly game. In the game, players' actions are considered as data and data is proceed by the controller classes inside this subsystem.

2.1.3.3. BankerFileManager

This subsystem is responsible for managing the data related to players’ bank accounts. In Monopoly play mode, players’ bank accounts change according to their actions (i.e. buying new planet, building constructions on the planet) and rules of the Monopoly (player receives money for each pass from the starting point). These changes are handled by the controller classed inside the *BankerFileManager.*

2.2. Hardware/software mapping

Monopoly Space Edition Game will be implemented in Java and the JavaFX library will be used. JRE (Java RunTime Environment) and JDK (Java Development Kit) are required for the implementation.The game requires a CPU and GPU with an average processor power. The buttons in the game will be activated with the mouse. The dice will be rolled with the mouse. Sound settings will be arranged using the mouse. At the start of the game, the keyboard will be used to enter the nicknames of the players into the system. Therefore, a mouse and keyboard are needed as hardware requirements. The game will be presented to users as a desktop application. Text files will be used to store data in the game. In this way, the necessary software will already be installed on the user's computer and a database will not be required.

2.3. Persistent data management

Monopoly Space Edition Game will be presented to the user as a desktop application, so a database is not required. The data that needs to be stored will be stored on the file system in the user's computer. The board of the game is written and stored in a way that cannot be changed while the game software is being made. Keeping data on the user's hard drive and prohibiting its updating is to prevent unnecessary data flow. Other visual icons that will not change during the game will be stored in the jar file. The movements of the players on the game board are updated and stored with the new version each time the dice are rolled. This storage will be done using text files. Text files will also be used for data that needs to be stored such as the players' score, the property they have purchased during the game, the money status of the players, game history, and this data will be open to update. For the visual symbols used in the game, the file with the extension .gif will be used and the images will be stored in this file. If the system suddenly goes out, players' information can be reloaded as it was when the system was opened.

2.4. Access control and security

Monopoly Space Edition Game does not require membership from players. It can be connected to the game and played on the desktop without any authentication. As no internet connection will be used to play the game, data leakage cannot be made from the player. Since this is a structure created during the software of the game, the player cannot change it later. During the software, information such as Monopoly Game Board, main menu, and the content of the cards will be written in a way that cannot be changed from the outside and the user will not be able to access or change them. Multiplayer is a game played, but there is no authorization in the game so all players have equal rights. A player does not have the right to play on behalf of another player or to change other players' data.

2.5. Boundary conditions

First boundary condition of the game is opening the game file. No installation is required on the computer to play the Monopoly Space Edition game. Since the game is in the form of a desktop application, it can be transferred to every computer as a .jar file. If there is a JVM on the transferred computer, the game can be run smoothly. Since there is no registration process in the game, there is no problem in connecting the players to the game at the beginning of the game. Thanks to the "Quit" button in the menu, you can exit the game. In other tabs that are opened, there will be a "Back" button, so the player will be able to return to the main menu. If you want to finish the game during the Monopoly game, it is enough to click the "Finish" button. If the system crashes during the game, the game will resume when it is reopened. If a player is bankrupt during the game, that player will be removed from the game (no authority to roll dice and no authority to move on the board.) If all players except one player are bankrupt, that remaining player wins the game and a notification is displayed on the screen that they have won the game. After this moment, no player is authorized to play the game. The game can be restarted with the "Replay" button on the screen, and you can exit the game with the "Quit" button. In a situation where all players in the game go bankrupt at the same time (this is impossible, but it may happen if the system fails), the system will terminate the game without showing any winning notification, since there will be no winner in the game. When playing the game, the token of that player moves as much as the unit that came on the dice, whoever it is the turn. Other players' tokens remain unchanged.

3. Low-level design

3.1. Object design trade-offs

3.1.1. Rapid Development vs. Functionality

Rapid development of the game may eliminate the functionality and decrease the performance due to the weak design and plan. Therefore, during the low level design, development of the system is detailly analysed and aimed to maximize functionality.

3.1.2. Space complexity vs. Time complexity

The game should occupy as much as less space to increase usability, adaptability and eliminate storage problems. Therefore, the algorithms and data structures are going to be designed to lessen the complexity and increase the efficiency for both performance and space.

3.1.3. Extensibility vs. Complexity

Extensibility of the program leads to the introduction of new classes, attributes relations and interfaces that causes to increase the program complexity. However, increasing complexity has priority on introduction of new featured classes and interfaces by means of performance. Therefore, extensibility is needed until complexity starts to give problems.

3.1.4. Backward Compatibility vs. Readability

Backward compatibility is a highly important feature that should be in a qualified program in order to increase usability. This feature should be supported with high readability of the code because backward compatibility can be implemented more easily when the readability of the code is high.

3.2. Final object design

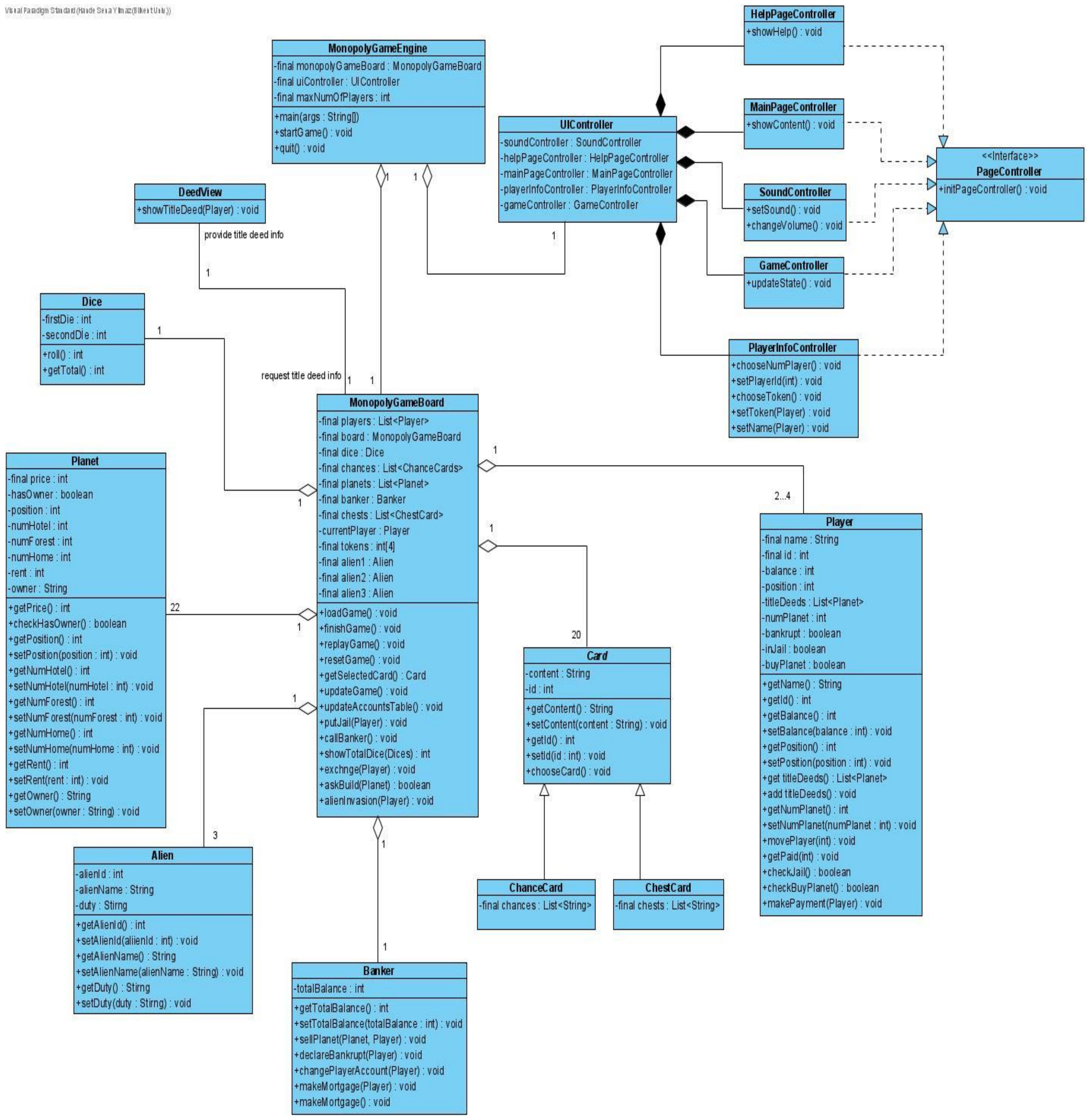


Figure 5 : Class diagram of Monopoly Space Edition Game

Abstraction of Final Object Design:

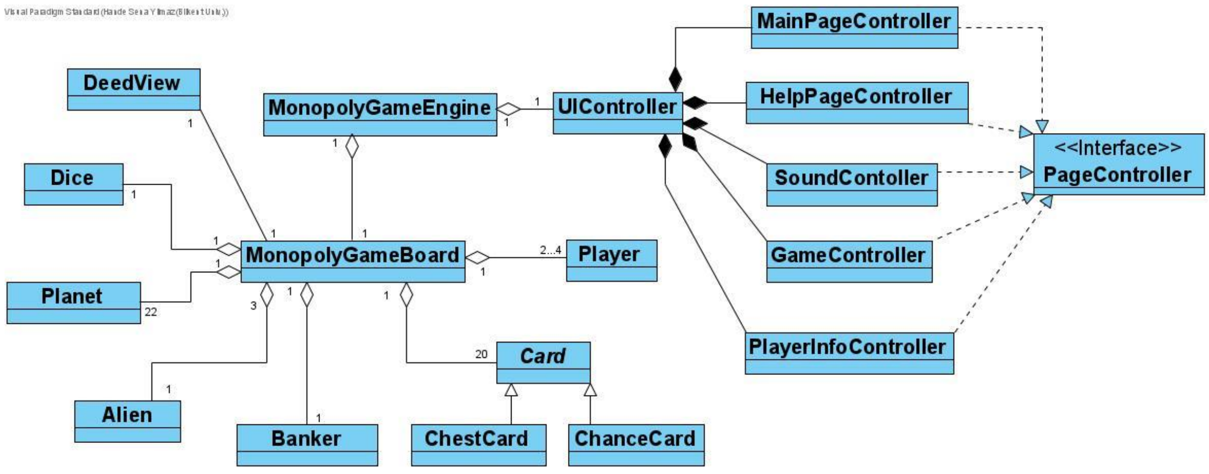
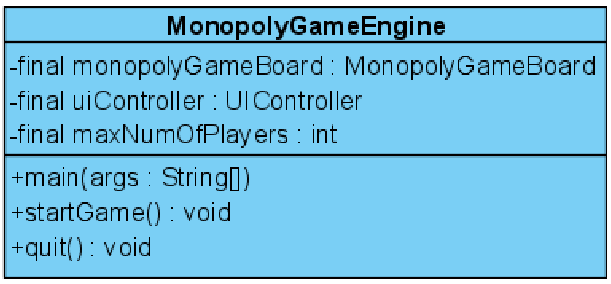


Figure 6: Object Design diagram of Monopoly Space Edition Game

3.2.1 MonopolyGameEngine Class



MonopolyGameEngine
-final monopolyGameBoard : MonopolyGameBoard -final uiController : UIController -final maxNumOfPlayers : int
+main(args : String[]) +startGame() : void +quit() : void

Figure 7 : MonopolyGameEngine Class of Monopoly Space Edition Game

MonopolyGameEngine class creates the connection between UIController and MonopolyGameBoard which are seen at the abstraction table. MonopolyGameBoard class is initialized in MonopolyGameEngine with an instance that allows ability to the object models.

3.2.1.1 Attributes of MonopolyGameEngine Class

private final MonopolyGameBoard monopolyGameBoard: Main table connecting all the objects.

private final UIController uiController: A connection between controllers and game board.

private final int maxNumOfPlayers: A restriction to the maximum number of players in the game.

3.2.1.2 Methods of MonopolyGameEngine Class

public static void main(String args[]) : Main method to run the game.

public void startGame(): Starts the game.

public void quit(): Closes the entire game screen.

3.2.2 MonopolyGameBoard Class

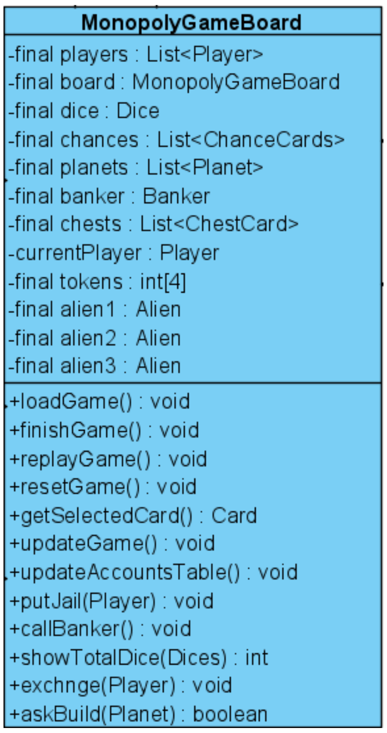


Figure 8 : MonopolyGameBoard Class of Monopoly Space Edition Game

This class represents the game board where the game actually takes place and which includes all the features of the game. It defines all aggregated objects as attributes and the process of the game with the functions given above.

3.2.2.1 Attributes of MonopolyGameBoard Class

private final List<Player> players: Represents all the players of the game.
private final MonopolyGameBoard board: Represents the game board where the game takes place.
private final Dice dice: Represents the dice being rolled during the game
private final List<ChanceCards> chances: Represents the chance cards on the game board.
private final List<Planet> planets: Represents all the planets on the game board.
private final Banker banker: Represents the banker of the game which is responsible for the money management.
private final List<ChestCard> chests: Represents the chest cards on the game board.
private Player currentPlayer: Represents the current player whose turn it is in the game
private final int[4] tokens: Represents the tokens of the player. There are at most 4 tokens for at most 4 players in the game.
private final Alien alien1: Represents the alien who takes the current player to black hole and keeps them waiting there for the next three turns.
private final Alien alien2: Represents the alien who makes the invasion with seizing one of the title deeds of the current player forever.
private final Alien alien3: Represents the alien who steals a certain amount of money from the current player.

3.2.2.2 Methods of MonopolyGameBoard Class

public void loadGame(): Starts the game after the information of the players are given and entered.
public void finishGame(): Controls the requirements to end the whole game and finishes the game based on it.

public void replayGame(): Provides players with the option of replaying the entire game.
public void resetGame(): Provides players with the option of resetting the entire game.
public Card getSelectedCard(): Provides the current player with the pulling of a random card in the case of landing on the chest or chance card area on the game board.
public void updateGame(): Controls the each turn for each player and makes the update of the game.
public void updateAccountsTable(): Updates the account table for each player after each purchase and sale during the game.
public void putJail(Player player): Puts the current player to jail and applies the sanctions of imprisonment based on the rules of the game.
public void callBanker(): Calls the banker in the case of purchase and the sale or any money management requirement controlled by the banker.
public int showTotalDice(Dices dices): Rolls the dice and shows the result of it for the current player's turn in the game.
public void exchange(Player player): In the case of an exchange, provides the current player with the exchange of title deeds with another player.
public boolean askBuild(Planet planet): Asks current player to whether to purchase the landed planet and build based on the choice.

3.2.3 DeedView Class

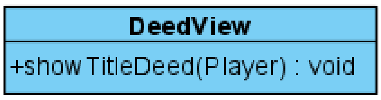


Figure 9 : DeadView Class of Monopoly Space Edition Game

DeedView class is used to display players' property deeds and is connected to MonopolyGameBoard class.

3.2.3.1 Methods of DeedView Class

public void showTitleDeed(Player player): This method lists the property deeds of the specified player at signature.

3.2.4 Dice Class

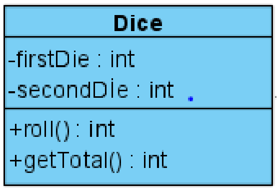


Figure 10 : DiceClass of Monopoly Space Edition Game

Dice class is directly connected to MonopolyGameBoard class to use rolled dice results in the game. The class contains die results as attributes and methods.

3.2.4.1 Attributes of Dice Class

private int firstDie: It keeps the data value of the first die.
private int secondDie: It keeps the data value of the second die.

3.2.4.2 Methods of Dice Class

public int roll(): This method assigns value from 1 to 6 for first and second die and get their sum as a result.
public int getTotal(): This method returns the roll result as an integer.

3.2.5 Planet Class

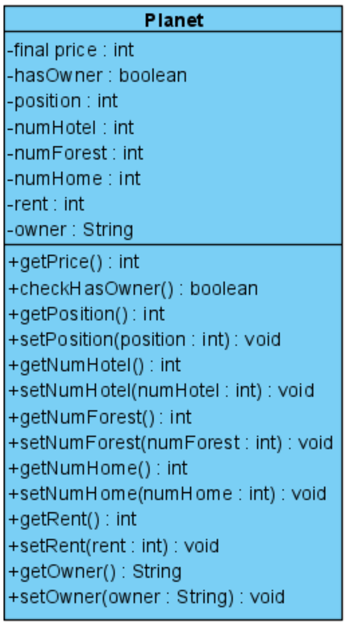


Figure 11 : PlanetClass of Monopoly Space Edition Game

Planet class represents the planets (title deeds) and has an aggregation association to MonopolyGameBoard class to represent the final list of the planets on the game board.

3.2.5.1 Attributes of Planet Class

private final int price: Represents the price of the planet.

private boolean hasOwner: Keeps track of whether the planet has an owner.

private int position: Represents the position of the planet on the game board.

private int numHotel: Represents the number of hotels purchased and built on the planet.

private int numForest: Represents the number of forests purchased and built on the planet.

private int numHome: Represents the number of homes purchased and built on the planet

private int rent: Represents the price of the rent of the planet.

private String owner: Represents the owner of the planet.

3.2.5.2 Methods of Planet Class

public int getPrice(): Gets and returns the price of the planet

public boolean checkHasOwner(): Checks whether the planet has an owner and returns it.

public int getPosition(): Gets and returns the position of the planet on the game board.

public void setPosition(int position): Sets the position of the planet on the game board.

public int getNumHotel(): Gets and returns the number of hotels purchased and built on the planet.

public void setNumHotel(int numHome): Sets the number of hotels purchased and built on the planet.

public int getNumForest(): Gets and returns the number of forests purchased and built on the planet.

public void setNumForest(int numHome): Sets the number of forests purchased and built on the planet.

public int getNumHome(): Gets and returns the number of homes purchased and built on the planet.

public void setNumHome(int numHome): Sets the number of homes purchased and built on the planet.

public int getRent(): Gets and returns the price of the rent of the planet.

public void setRent(int rent): Sets the price of the rent of the planet.

public String getOwner(): Gets and returns the owner of the planet.

public void setOwner(String owner): Sets the owner of the planet.

3.2.6 Alien Class

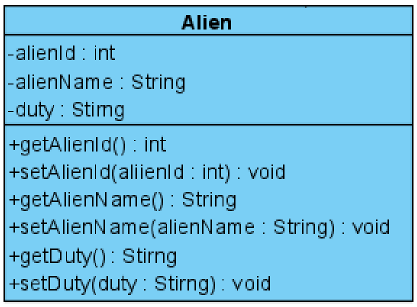


Figure 12 : Alien Class of Monopoly Space Edition Game

Alien class has a direct aggregation to MonopolyGameBoad to be able to connect Aliens to the game. The class is the base of the Alien object and stores its attributes and functions.

3.2.6.1 Attributes of Alien Class

private int alienId: It keeps the alien id which is identical for each alien object.
private String alienName: String keeps the alien name.
private String duty: Duty keeps the abilities of the aliens.

3.2.6.2 Methods of Alien Class

public int getAlienId(): Gets and returns the alien id.
public void setAlienId(int alienId): Sets an alien id.
public String getAlienName(): Gets and returns the alien name.
public void setAlienName(String alienName): Sets the alien name.
public String getDuty(): Gets and returns the duty of the alien.
public void setDuty(String duty): Sets a duty to the alien.

3.2.7 Banker Class

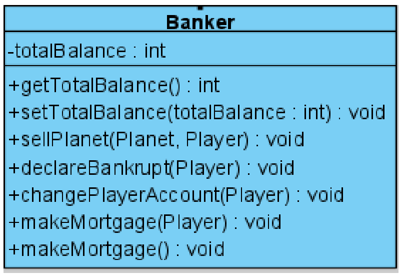

Banker
-totalBalance : int
+getTotalBalance() : int
+setTotalBalance(totalBalance : int) : void
+sellPlanet(Planet, Player) : void
+declareBankrupt(Player) : void
+changePlayerAccount(Player) : void
+makeMortgage(Player) : void
+makeMortgage() : void

Figure 13 : Banker Class of Monopoly Space Edition Game

Banker class has one to one relation with MonopolyGameBoard class and aggregates it. The class contains the specified attributes and methods to be able to conduct banking functions by a banker object.

3.2.7.1 Attributes of Banker Class

private int totalBalance: It keeps the total balance of the banker.

3.2.7.2 Methods of Banker Class

private int getTotalBalance(): It returns the total balance attribute.

private void setTotalBalance(int totalBalance): It updates the total balance.

private void sellPlanet(Planet planet, Player player): The method is used for selling a planet to a player specified in the signature of the method.

private void declareBankrupt(Player player): Specifies the player gone bankrupt and disqualifying it from the game.

private void changePlayerAccount(Player player): Controls the player account given in the signature for the update of the account.

private void makeMortgage(Player player): Makes the mortgage process between players as a mortgaged player is specified in the signature.

private void makeMortgage(): Makes the mortgage process between a player to directly the bank.

3.2.8 *Card* Abstract Class

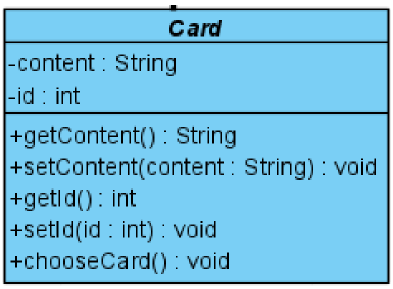


Figure 14 : Card Class of Monopoly Space Edition Game

Card class is an abstract class. It is the superclass of ChanceCard and ChestCard classes and has an aggregation association to MonopolyGameBoard class to represent the list of each type of cards on the game board. There are a total of 20 cards, 10 for chance cards and 10 for chest cards.

3.2.8.1 Attributes of *Card* Abstract Class

private String content: Represents the content of the card as string.

private int id: Represents the id of the card.

3.2.8.2 Methods of *Card* Abstract Class

public String getContent(): Gets and returns the content of the card.

public void setContent(String content): Sets the content of the card.

public int getId(): Gets and returns the id of the card.

public void setId(int id): Sets the id of the card.

public void chooseCard(): Makes a choice of a random card from the cards and displays the given content.

3.2.9 ChestCard Class

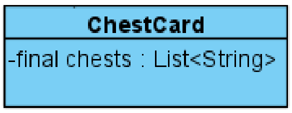


Figure 15 : ChestCard Class of Monopoly Space Edition Game

ChestCard is a subclass of Card Abstract Class and it represents the list of chest cards and their final content. There are a total of 10 chest cards on the game board.

3.2.9.1 Attribute of ChestCard Class

private final List<String> chests: Represents all the chest cards on the game board as a final list.

3.2.10 ChanceCard Class

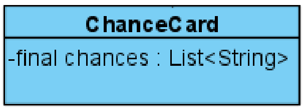


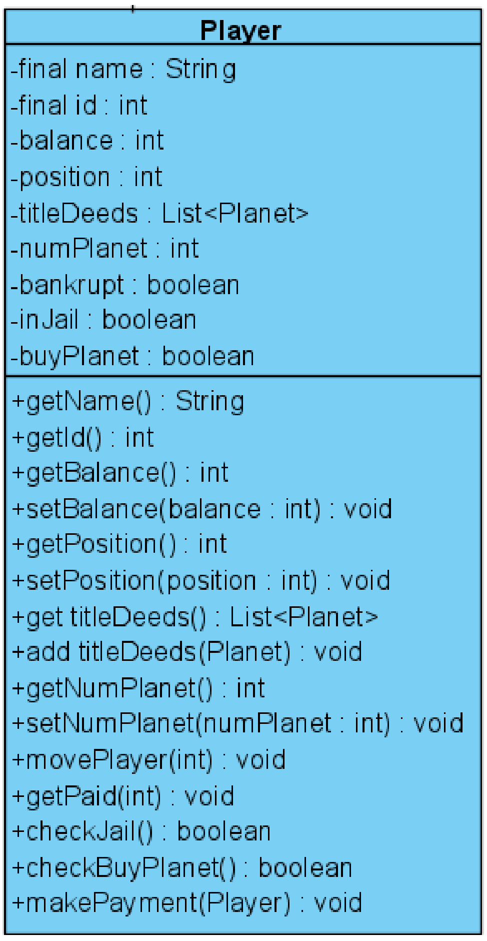
Figure 16 : ChanceCard Class of Monopoly Space Edition Game

ChanceCard is a subclass of Card Abstract Class and it represents the list of chance cards and their final content. There are a total of 10 chance cards on the game board.

3.2.10.1 Attribute of ChanceCard Class

private final List<String> chances: Represents all the chance cards on the game board as a final list.

3.2.11 Player Class



```
Player
-final name : String
-final id : int
-balance : int
-position : int
-titleDeeds : List<Planet>
-numPlanet : int
-bankrupt : boolean
-inJail : boolean
-buyPlanet : boolean
+getName() : String
+getId() : int
+getBalance() : int
+setBalance(balance : int) : void
+getPosition() : int
+setPosition(position : int) : void
+get titleDeeds() : List<Planet>
+add titleDeeds(Planet) : void
+getNumPlanet() : int
+setNumPlanet(numPlanet : int) : void
+movePlayer(int) : void
+getPaid(int) : void
+checkJail() : boolean
+checkBuyPlanet() : boolean
+makePayment(Player) : void
```

Figure 17 : Player Class of Monopoly Space Edition Game

Player class represents the player object of the game specifies its contribution. This class also has an aggregation association to MonopolyGameBoard class to represent the final list of the players and the current player on the game board.

3.2.11.1 Attributes of Player Class

private final String name: Represents the name of the player.

private final int id: Represents the id of the player.

private final int balance: Represents the total balance of the player.

private final int position: Represents the current position of the player on the game board.

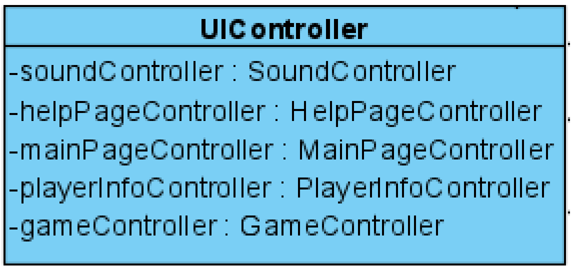
private List<Planet> titleDeeds: Represents all the planet's title deeds which the player possesses as a list.

private int numPlanet: Represents the number of planets which the player possesses.
private boolean bankrupt: Represents whether the player is bankrupt or not.
private boolean inJail: Represents whether the player is in jail or not.
private boolean buyPlanet: Represents whether the player chooses to buy a planet.

3.2.11.2 Methods of Player Class

public String getName(): Gets and returns the name of the player.
public int getId(): Gets and returns the id of the player.
public int getBalance(): Gets and returns the total balance of the player.
public void setBalance(int balance): Sets the total balance of the player.
public int getPosition(): Gets and returns the current position of the player on the game board.
public void setPosition(int position): Sets and updates the position of the player on the game board based on the given signature.
public List<Planet> getTitleDeeds(): Gets and returns all the title deeds owned by the player as a list of planets.
public void addTitleDeeds(Planet): Adds a newly bought planet by the player to the list of owned title deeds.
public int getNumPlanet(): Gets and returns the total number of planets owned by the player.
public void setNumPlanet(int numPlanet): Sets the total number of planets owned by the player.
public void movePlayer(int a): Makes the movement of the player on the game board based on the given integer in the signature.
public void getPaid(int a): Gets the payment of the player which is taken from other players or the banker.
public boolean checkJail(): Checks whether the player is currently in the jail.
public boolean checkBuyPlanet(): Checks whether the player wants to buy the planet currently landed on.
public void makePayment(Player player): Controls the payment made to another player specified in the signature.

3.2.12 UIController Class


UIController

-soundController : SoundController

-helpPageController : HelpPageController

-mainPageController : MainPageController

-playerInfoController : PlayerInfoController

-gameController : GameController

Figure 18 : UIController Class of Monopoly Space Edition Game

This class provides a user interface controller to the game and it has an aggregation association to MonopolyGameEngine class to connect the user interface components.

3.2.12.1 Attributes of UIController Class:

private SoundController soundController: Represents and stores a SoundController object.

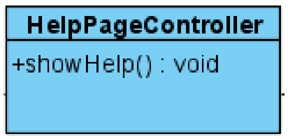
private HelpPageController helpPageController: Represents and stores a HelpPageController object.

private MainPageController mainPageController: Represents and stores a MainPageController object.

private PlayerInfoController playerInfoController: Represents and stores a PlayerInfoController object.

private GameController gameController: Represents and stores a GameController object.

3.2.13 HelpPageController Class


HelpPageController

+showHelp() : void

Figure 19 : HelpPageController Class of Monopoly Space Edition Game

This class is to provide the controller for the help page to the UIController Class.

3.2.13.1 Methods of HelpPageController Class:

public void showHelp(): Displays the content of the help page where the rules of the game is explained.

3.2.14 MainPageController Class

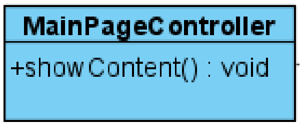


Figure 20 : MainPageController Class of Monopoly Space Edition Game

This class is to provide the controller for the main page to the UIController Class.

3.2.14.1 Methods of MainPageController Class:

public void showContent(): Displays the content of the main page.

3.2.15 SoundController Class

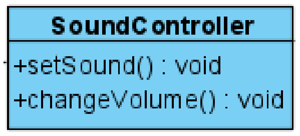


Figure 21 : SoundController Class of Monopoly Space Edition Game

This class controls the sound settings of the entire game and provides an attribute to UIController Class.

3.2.15.1 Methods of SoundController Class:

public void setSound(): Sets the sound of the entire game to default.
public void changeVolume(): Changes the volume of the sound of the entire game.

3.2.16 GameController Class

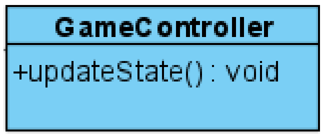


Figure 22 : GameController Class of Monopoly Space Edition Game

This class provides a controller for the game and declares an object for the UIController Class.

3.2.16.1 Methods of GameController Class:

public void updateState(): Updates the state of the game.

3.2.17 PlayerInfoController Class

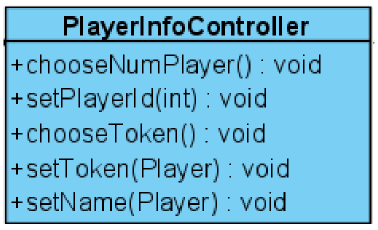


Figure 23 : PlayerInfoController Class of Monopoly Space Edition Game

This class provides a controller for the players' information before starting the game and declares an object for the UIController Class.

3.2.17.1 Methods of PlayerInfo Class:

public void chooseNumPlayer(): Provides the user to choose the number of players for the game.

public void setPlayerId(int): Sets the player id for each player.

public void chooseToken(): Chooses the token representing each player.

public void setToken(Player): Sets the chosen tokens to each player.

public void setName(Player): Sets the name of each player.

3.2.18 PageController Interface

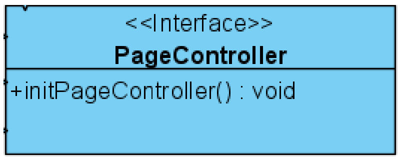


Figure 24 : PageController Class of Monopoly Space Edition Game

PageController is an interface for PlayerInfoController, GameController, SoundController, MainPageController, and HelpPageController classes.

3.2.18.1 Methods of PageController Interface:

public void initPageController(): Initializes the page controller.

3.3. Packages

The packages in the game are designed with respect to packages at the subsystem decomposition part. Therefore, the packages are MonopolyGameManager, MonopolyLogicManager and MonopolyFileManagement that all of them have their own subsystem packages in it. These packages are used in order to controlize the access between subsystems and test the subsystems via different frameworks.

3.4. Class Interfaces

The design has PageController interface implemented by all controllers which all aggregate the user interface controller. UIController is connected to the game engine and thanks to the controller classes implementing hasPageController interface, changes are done in the game engine and the game runs properly. These controller options are PlayerInfoController, MainPageController, SoundController etc. given at the class diagram.

4. Glossary & References

[1]] Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.

[2] “Package Diagram,” *VP Gallery.* [Online]. Available: https://www.visual-paradigm.com/VPGallery/diagrams/Package.html [Accessed:Nov. 25, 2020]