Bilkent University

Department of Computer Science

Object-Oriented Software Engineering Design Report

CS 319 Project Group 3C: 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

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 when it is compared to inverse case.

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. Object design trade-offs

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.

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.

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.

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.

1.2.3. Criteria

1.2.3.1. 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.

1.2.3.2. 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.

1.2.3.3. 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 criterion 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.

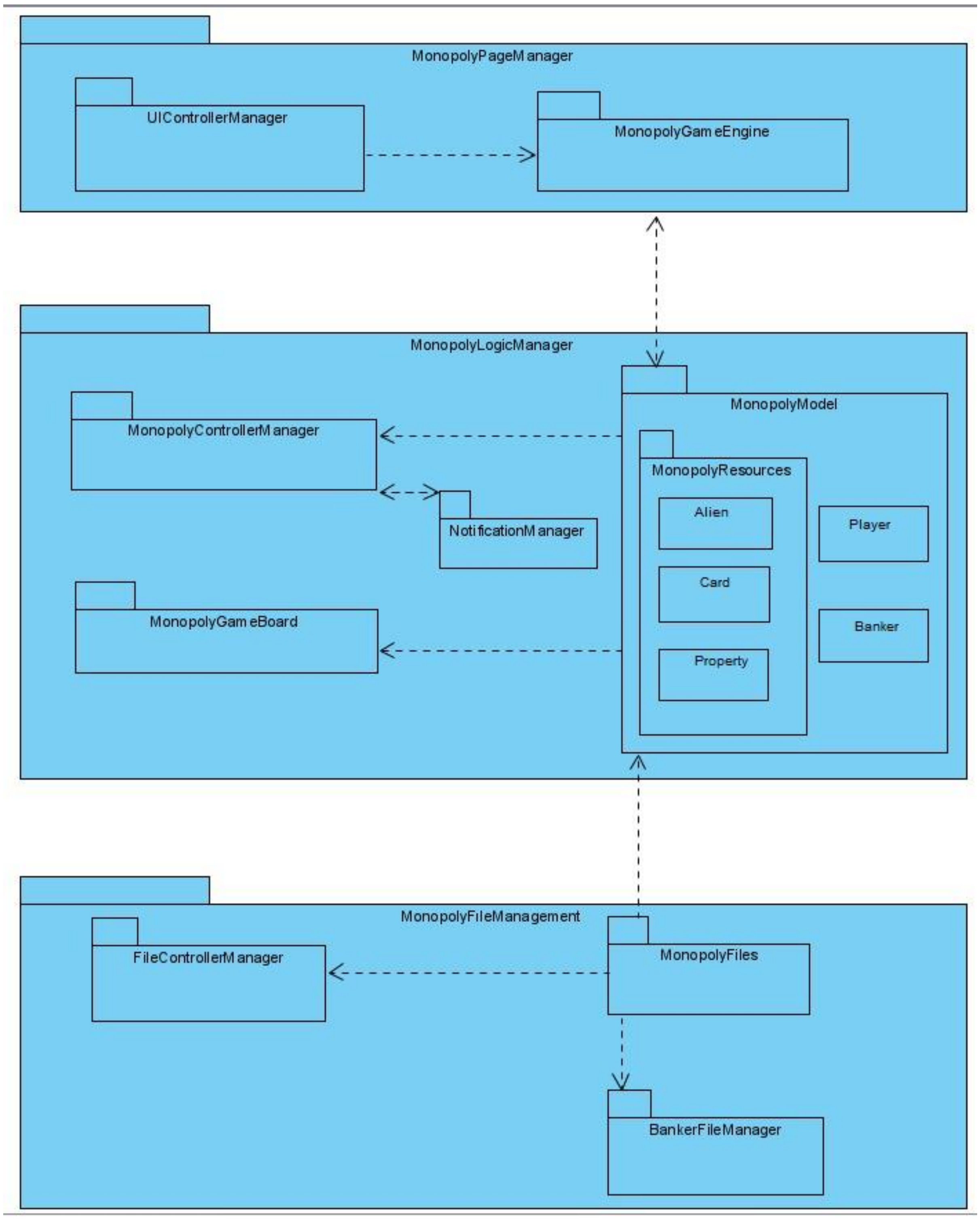


Figure 2.1 : Package diagram of Monopoly Space Edition Game

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

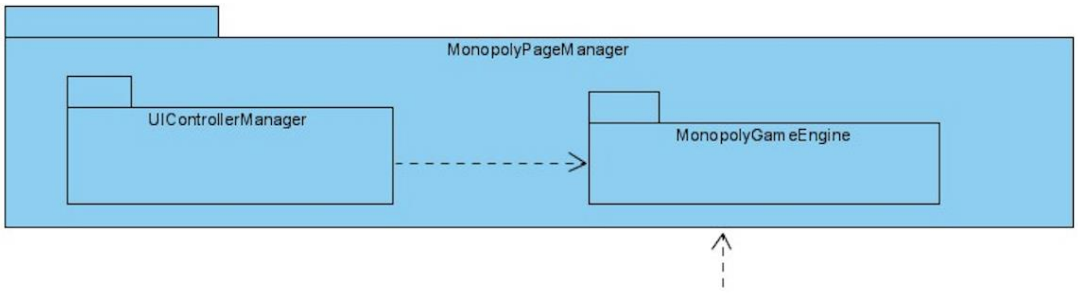


Figure 2.2 : First Layer of Monopoly Space Edition Game

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

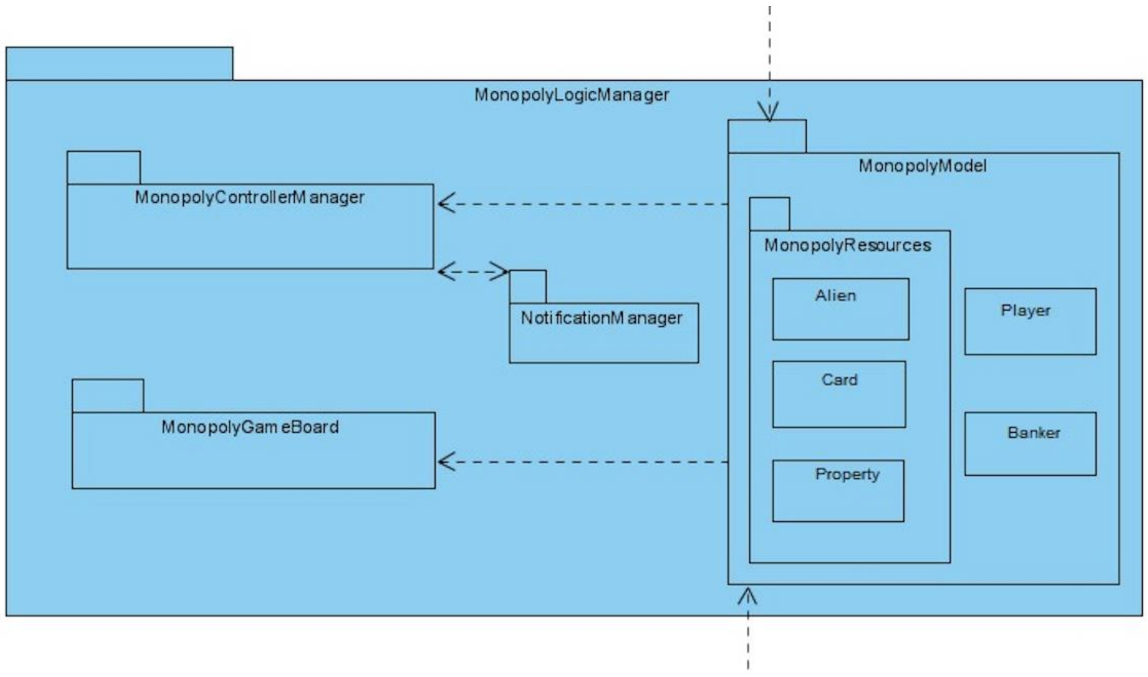


Figure 2.3 : Second Layer of Monopoly Space Edition Game

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 properties. The Monopoly board state is in the *MonopolyGameBoard* subsystem. In the *MonopolyModel* subsystem, there are game entities, Player, Alien, Card,Banker and Property. 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,Banker and

Property. In *MonopolyModel,* Card, Alien and Property 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 property) 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, Property and position of the Player on the Monopoly board via interaction with MonopolyModel.

2.1.3. MonopolyFileManagement

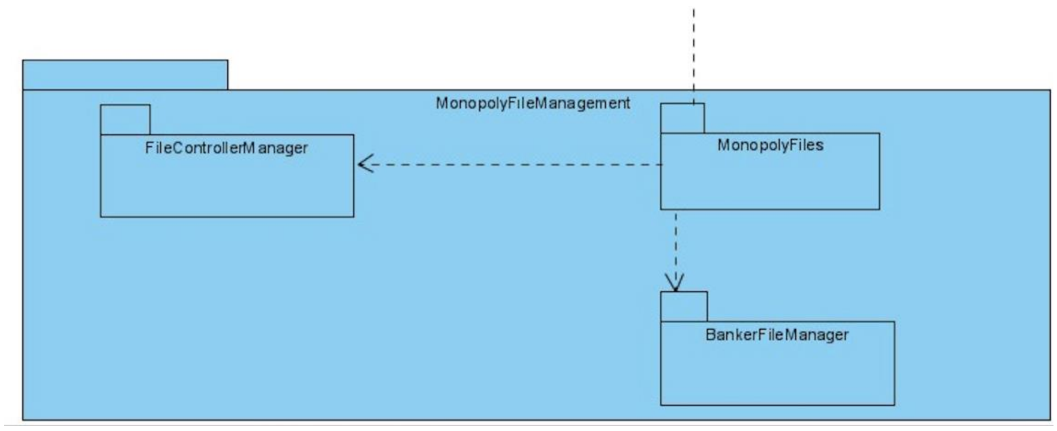


Figure 2.4 : Third Layer of Monopoly Space Edition Game

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 processed 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 property, building structure on the planet property) 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 required to Java Runtime Environment (JRE) to be runned. The game can be runned the platforms of Windows, macOS and Linux. By downloading the Java SE Runtime Environment 8u271 version which is appropriate for the users platform, the game will be able to be runned.

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. Locations are stored in a list of string. Token, Property and Card classes are also stored and updated throughout the game. 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. The game is played with multiple players, 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. Java needs to be installed 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 since text files will be used to store the data of the recent game and it will upload again. 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. Final Object Design

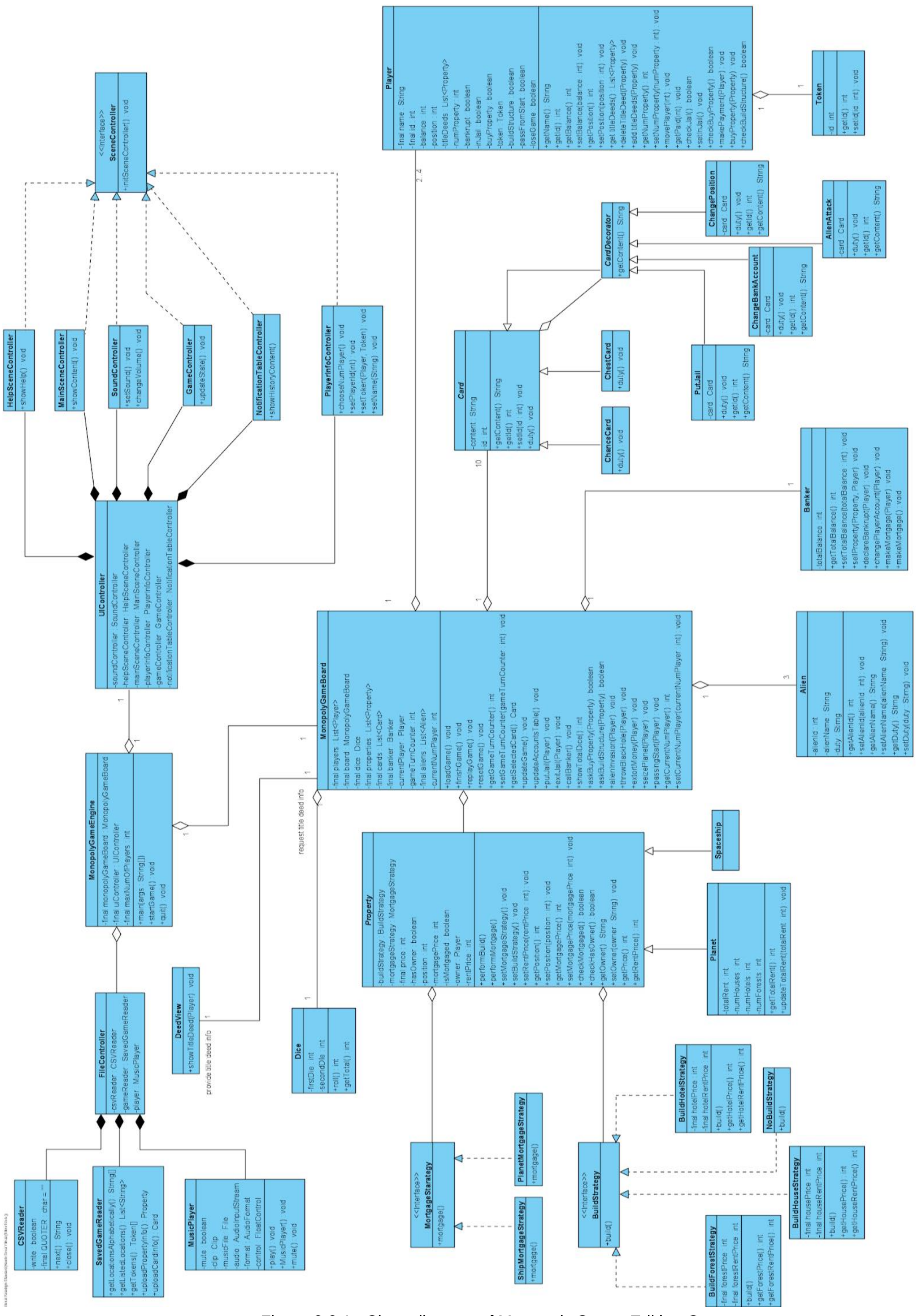


Figure 3.2.1 : Class diagram of Monopoly Space Edition Game

Note: To make Final Object Design visible, we added enlarged view of some parts under Final Object Design Details (before the explanation of diagram).

3.2. Design Patterns Used in Final Object Design

Decorator Pattern:

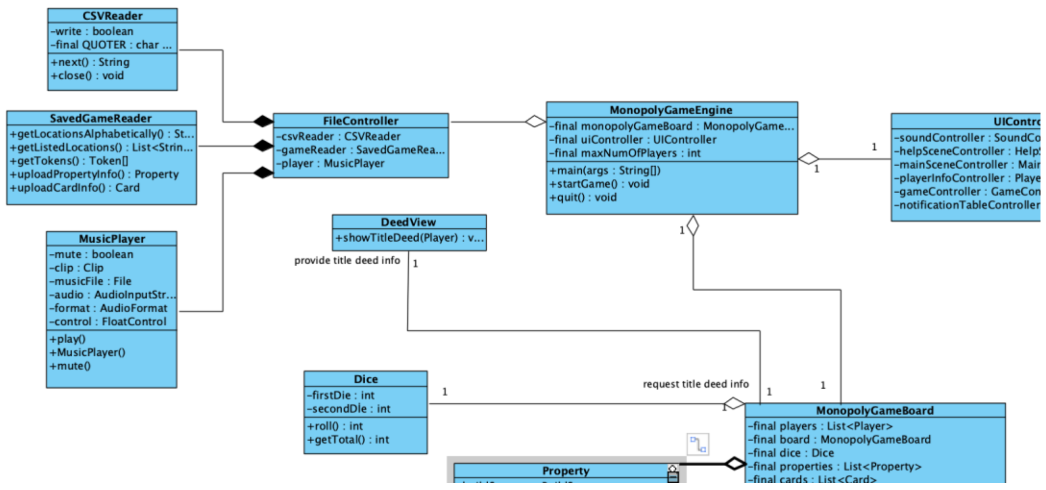
We used Decorator design pattern for our Card classes in our UML Class Diagram. In our system, Decorator design pattern provides an alternative to extending the behavior of Cards. There are both is-a and has-a relation between Card and CardDecorator classes. CardDecorator abstract class is used to wrap concrete component classes ChanceCard and ChestCard. Concrete decorator classes (PutJail, ChangeBankAccount, AlienAttack and ChangePosition) change behavior of ChanceCard and ChestCard classes and add new functionalities before using them. In Monopoly, there are different Chance cards and Chest cards. Instead of creating each card in many different classes, we used CardDecorator. Decorator design pattern provides us to wrap ChanceCard and ChestCard components with decorator classes. In order to create cards with different duties, we combine concrete decorator classes (PutJail, ChangeBankAccount, AlienAttack and ChangePosition) to create new duties for new cards. In short, by having a decorator design pattern, we provide new behavior for Card objects at runtime which adds functionality to our Monopoly Space Edition system.

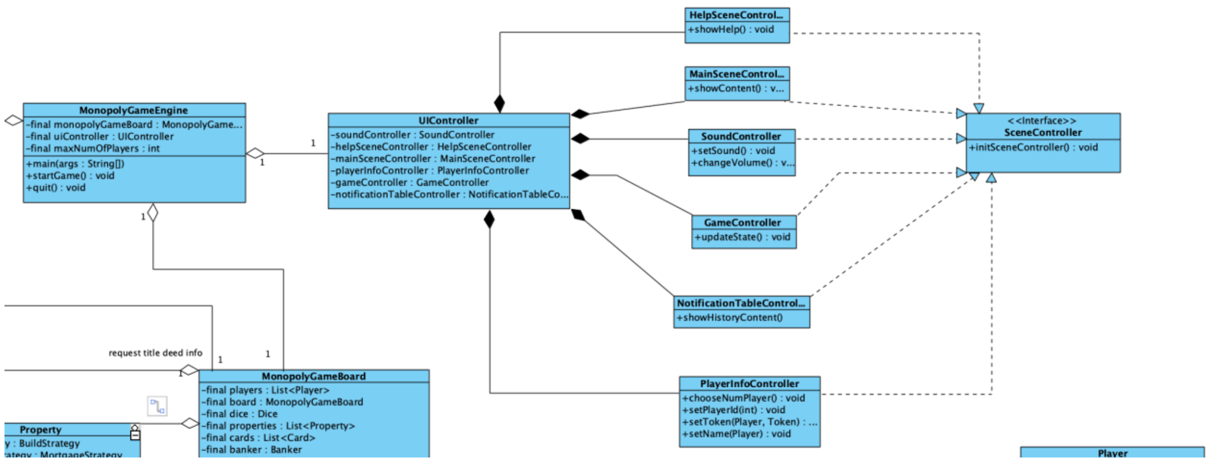
Strategy Pattern:

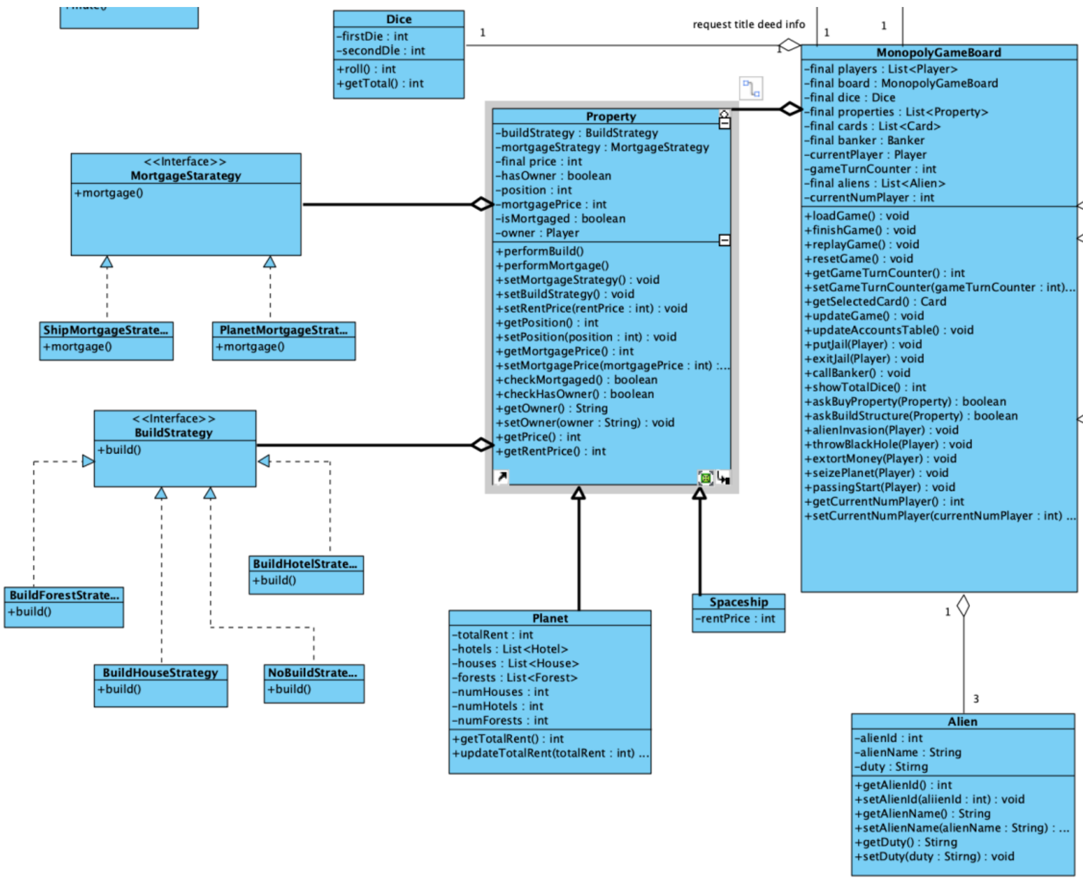
In order to encapsulate algorithms for code reuse the strategy design pattern is used for our Property class in our UML Class Diagram. The abstract property class represents the property location on the board. Planet and Spaceship classes extend Property class because they are properties with different features. The differences between the two are as follows, when the player comes to his own planet property, he can build a house or forest, while spaceship is not the case. There are also differences in the mortgage process between these two properties. While the player is mortgaging his planet property, the fee and the properties above are collected and the half price is given to the player by the bank and the title deed is taken by the bank. When the player wants to mortgage the spaceship property, the bank pays the player as much as the price of that title and buys the title deed. Two different interfaces have been created for these different behaviors: BuildStrategy and MortgageStrategy. BuildStrategy interface includes the build () method, and BuildForestStrategy, BuildHouseStrategy, BuildHotelStrategy and NoBuildStrategy, which implement this interface, determine how this build process will take place differently. NoBuildStrategy will be used for spaceship, since a property cannot be built on spaceship here. It implements the MortgageStrategy interface, the PlanetMortgageStrategy and ShipMortgageStrategy classes, and thanks to the different contents of the mortgage methods they contain, the mortgage process is realized differently in different properties. BuilStrategy and MortgageStrategy objects have been added to the Property class. This means that each property has a BuildStrategy and a MortgageStrategy. Thanks to the performBuild () and performMortgage () methods in the Property class, these operations are performed according to which property will build in which style or mortgage transaction. The setMortgageBuildStrategy () and setBuildStartegy () methods determine which build and mortgage processes the properties perform.

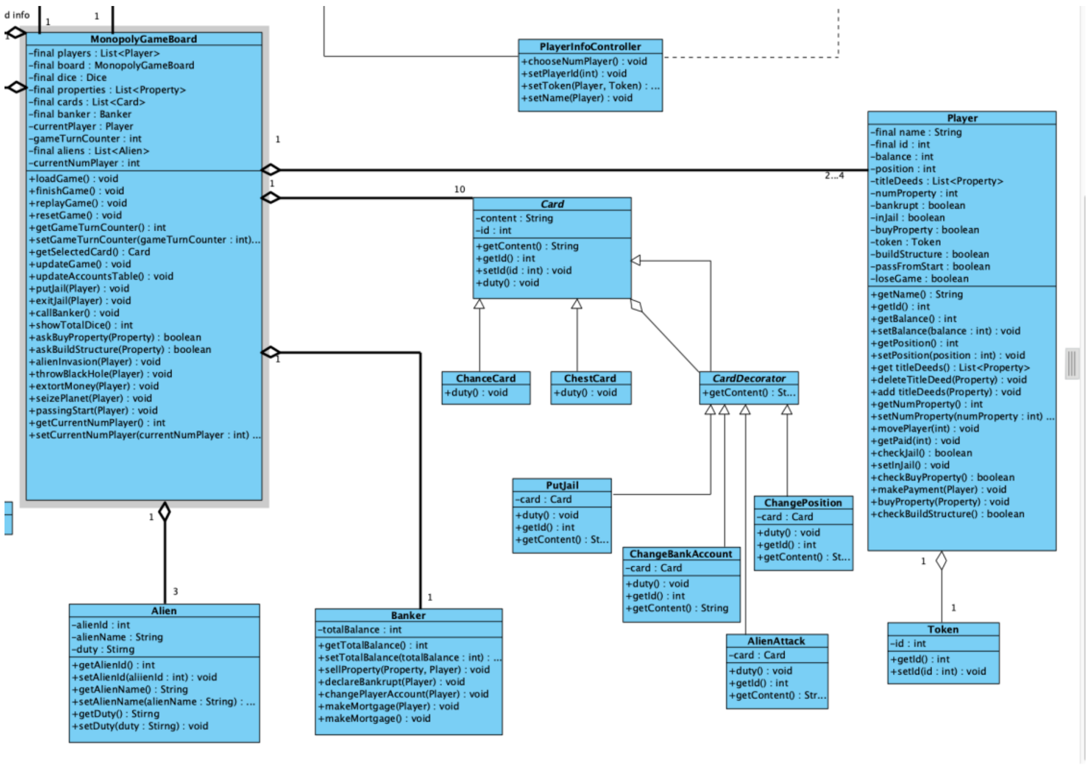
Final Object Design Details

Enlarged Images of Final Object-Design

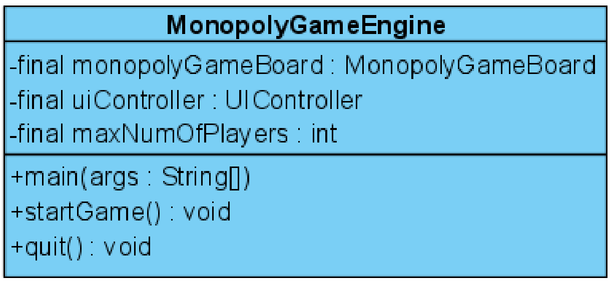



MonopolyGameEngine
-final monopolyGameBoard : MonopolyGame...
-final uiController : UIController
-final maxNumOfPlayers : int
+main(args : String[])
+startGame() : void
+quit() : void
UIController
-soundController : SoundController
-helpSceneController : HelpSceneController
-mainSceneController : MainSceneController
-playerInfoController : PlayerInfoController
-gameController : GameController
-notificationTableController : NotificationTableCo...
HelpSceneControl...
+showHelp() : void
MainSceneControl...
+showContent() : v...
SoundController
+setSound() : void
+changeVolume() : v...
GameController
+updateState() : void
NotificationTableControl...
+showHistoryContent()
PlayerInfoController
+chooseNumPlayer() : void
+setPlayerId(int) : void
+setToken(Player, Token) : ...
+setName(Player) : void
<<Interface>>
SceneController
+initSceneController() : void
request title deed info
1
MonopolyGameBoard
-final players : List<Player>
-final board : MonopolyGameBoard
-final dice : Dice
-final properties : List<Property>
-final cards : List<Card>
-final banker : Banker
Property
y : BuildStrategy
rategy : MortgageStrategy
Player


Dice
-firstDie : int
-secondDie : int
+roll() : int
+getTotal() : int
request title deed info
1
MonopolyGameBoard
-final players : List<Player>
-final board : MonopolyGameBoard
-final dice : Dice
-final properties : List<Property>
-final cards : List<Card>
-final banker : Banker
-currentPlayer : Player
-gameTurnCounter : int
-final aliens : List<Alien>
-currentNumPlayer : int
+loadGame() : void
+finishGame() : void
+replayGame() : void
+resetGame() : void
+getGameTurnCounter() : int
+setGameTurnCounter(gameTurnCounter : int)...
+getSelectedCard() : Card
+updateGame() : void
+updateAccountsTable() : void
+putJail(Player) : void
+exitJail(Player) : void
+callBanker() : void
+showTotalDice() : int
+askBuyProperty(Property) : boolean
+askBuildStructure(Property) : boolean
+alienInvasion(Player) : void
+throwBlackHole(Player) : void
+extortMoney(Player) : void
+seizePlanet(Player) : void
+passingStart(Player) : void
+getCurrentNumPlayer() : int
+setCurrentNumPlayer(currentNumPlayer : int) ...
<<Interface>>
MortgageStarategy
+mortgage()
ShipMortgageStrate...
+mortgage()
PlanetMortgageStrat...
+mortgage()
Property
-buildStrategy : BuildStrategy
-mortgageStrategy : MortgageStrategy
-final price : int
-hasOwner : boolean
-position : int
-mortgagePrice : int
-isMortgaged : boolean
-owner : Player
+performBuild()
+performMortgage()
+setMortgageStrategy() : void
+setBuildStrategy() : void
+setRentPrice(rentPrice : int) : void
+getPosition() : int
+setPosition(position : int) : void
+getMortgagePrice() : int
+setMortgagePrice(mortgagePrice : int) :...
+checkMortgaged() : boolean
+checkHasOwner() : boolean
+getOwner() : String
+setOwner(owner : String) : void
+getPrice() : int
+getRentPrice() : int
<<Interface>>
BuildStrategy
+build()
BuildHotelStrate...
+build()
BuildForestStrate...
+build()
BuildHouseStrategy
+build()
NoBuildStrate...
+build()
Planet
-totalRent : int
-hotels : List<Hotel>
-houses : List<House>
-forests : List<Forest>
-numHouses : int
-numHotels : int
-numForests : int
+getTotalRent() : int
+updateTotalRent(totalRent : int) ...
Spaceship
-rentPrice : int
Alien
-alienId : int
-alienName : String
-duty : Stirng
+getAlienId() : int
+setAlienId(aliienId : int) : void
+getAlienName() : String
+setAlienName(alienName : String) : ...
+getDuty() : Stirng
+setDuty(duty : Stirng) : void
1
3



3.2.1 MonopolyGameEngine Class



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

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.

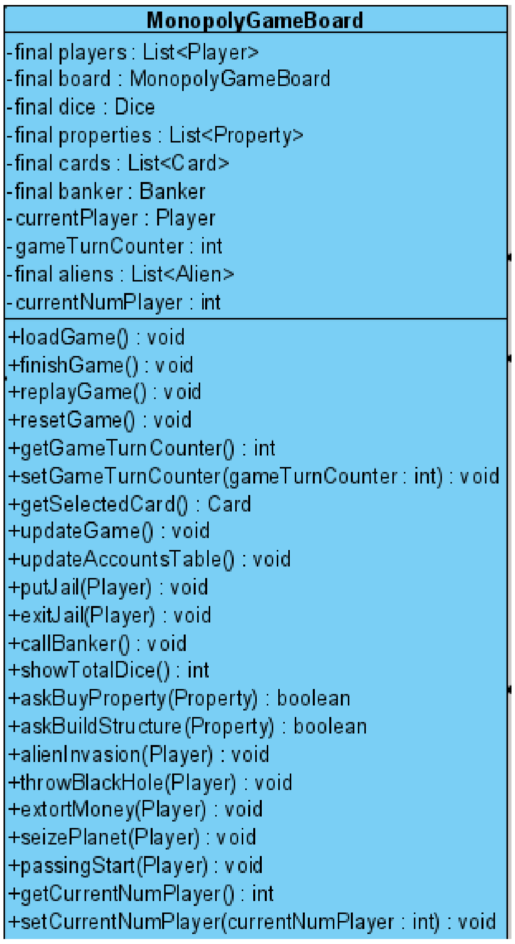
3.2.1.2 Methods of MonopolyGameEngine Class

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



```
MonopolyGameBoard
-final players : List<Player>
-final board : MonopolyGameBoard
-final dice : Dice
-final properties : List<Property>
-final cards : List<Card>
-final banker : Banker
-currentPlayer : Player
-gameTurnCounter : int
-final aliens : List<Alien>
-currentNumPlayer : int
+loadGame() : void
+finishGame() : void
+replayGame() : void
+resetGame() : void
+getGameTurnCounter() : int
+setGameTurnCounter(gameTurnCounter : int) : void
+getSelectedCard() : Card
+updateGame() : void
+updateAccountsTable() : void
+putJail(Player) : void
+exitJail(Player) : void
+callBanker() : void
+showTotalDice() : int
+askBuyProperty(Property) : boolean
+askBuildStructure(Property) : boolean
+alienInvasion(Player) : void
+throwBlackHole(Player) : void
+extortMoney(Player) : void
+seizePlanet(Player) : void
+passingStart(Player) : void
+getCurrentNumPlayer() : int
+setCurrentNumPlayer(currentNumPlayer : int) : void
```

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<Property> properties: Represents the properties and spaceships as properties on the game board.
private final List<Card> cards: Represents all the chance and chest cards as cards on the game board.
private final Banker banker: Represents the banker of the game which is responsible for the money management.
private Player currentPlayer: Represents the current player whose turn it is in the game
private int gameTurnCounter: Represents the counter for each turn in the game.
private List<Alien> aliens: Represents the aliens who have different duties on the game board.
Private int currentNumPlayer: Represents the current total number of players playing the game (from 2 up to 4).

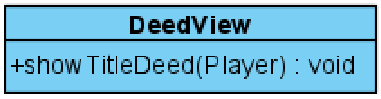
3.2.2.2 Methods of MonopolyGameBoard Class

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 int getGameTurnCounter(): Gets and returns the counter for the turns of the game.
public void setGameTurnCounter(int gameTurnCounter): Sets the counter for the turns of the 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 exitJail(Player player): Takes out the player from jail based on the exit requirements.
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 boolean askBuyProperty(Property p): Asks current player whether to buy the property currently on and return the answer as boolean.
public boolean askBuildStructure(Property p): Ask current player whether to build structures like hotel, forest, house he/she already possess and currently on and return the answer as boolean.
public void alienInvasion(Player p): Assigns a random alien from the game board to current player to make the invasion.
public void throwBlackHole(Player p): Makes the duty of one of the aliens which is throwing the current player to blackhole and lets them out of the game for the next three turns of that player.
public void extortMoney(Player p): Makes the duty of one of the aliens which is extorting some amount of money of the current player.
public void seizePlanet(Player p): Makes the duty of one of the aliens which is seizing a random planet from the current player’s properties.
public void passingStart(Player p): In each pass from start location on the board, it gives 2M money to the player given as parameter and updates the account.
public int getCurrentNumPlayer(): Gets and returns the current number of players playing the game.

public void setCurrentNumPlayer(int currentNumPlayer): Sets the current number of players playing the game.

3.2.3 DeedView Class

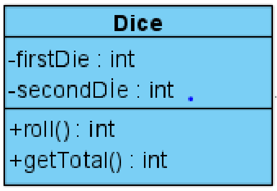


DeedView class is used to display players' property deeds and is connected to MonopolyGameBoard class.

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



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.

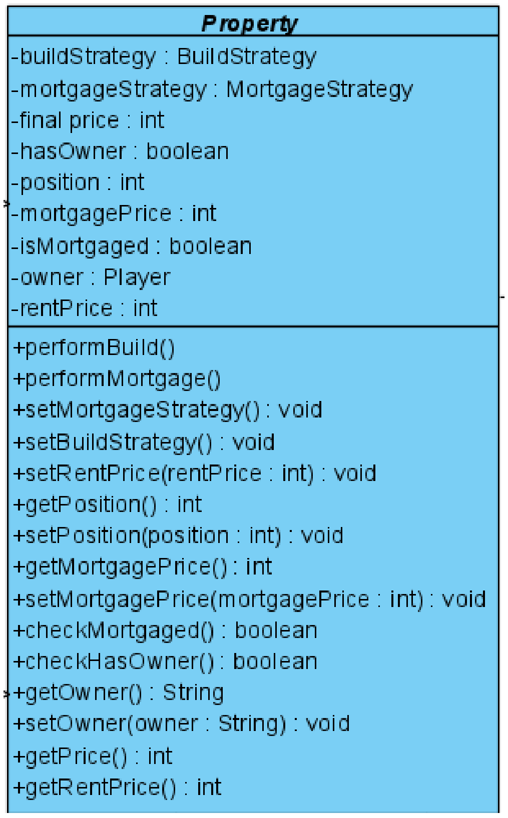
private int secondDie: It keeps the data value of the second die.

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 *Property* Abstract Class


Property
-buildStrategy : BuildStrategy
-mortgageStrategy : MortgageStrategy
-final price : int
-hasOwner : boolean
-position : int
-mortgagePrice : int
-isMortgaged : boolean
-owner : Player
-rentPrice : int
+performBuild()
+performMortgage()
+setMortgageStrategy() : void
+setBuildStrategy() : void
+setRentPrice(rentPrice : int) : void
+getPosition() : int
+setPosition(position : int) : void
+getMortgagePrice() : int
+setMortgagePrice(mortgagePrice : int) : void
+checkMortgaged() : boolean
+checkHasOwner() : boolean
+getOwner() : String
+setOwner(owner : String) : void
+getPrice() : int
+getRentPrice() : int

Property class is an abstract class to represent planets and spaceships as properties on the game board and also for title deeds of each player. It has an aggregation relation to MonopolyGameBoard class.

3.2.5.1 Attributes of *Property* Abstract Class

private BuildStrategy buildStrategy: Represents the build strategy used to build any of the structures or no structure as a part of strategy design pattern.

private MortgageStrategy mortgageStrategy: Represents the mortgage strategy used to represent mortgaged property as a part of strategy design pattern.

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

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

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

private int mortgagePrice: Represents the mortgage price of the property.

private boolean isMortgaged: Keeps track of whether the property is mortgaged.

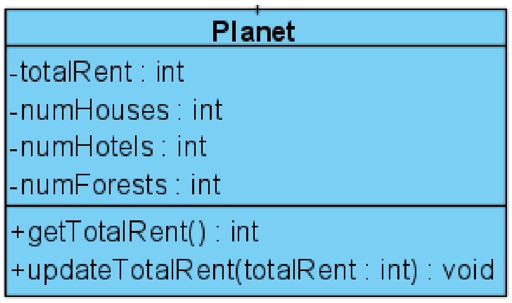
private String owner: Represents the owner of the property.

private int rentPrice: Represents the rent price of the property.

3.2.5.2 Methods of *Property* Abstract Class

public void performBuild(): Performs build operation for either planet or spaceship properties.
public void performMortgage(): Performs mortgage process for the defined property.
public void setMortgageStrategy(MortgageStrategy ms): Sets the mortgage strategy given as parameter for the property.
public void setBuildStrategy(BuildStrategy bs): Sets the build strategy given as parameter for the property.
public void setRentPrice(int rentPrice): Sets the rent price of the property.
public int getPosition(): Gets and returns the position of the property on the game board.
public void setPosition(int position): Sets the position of the property on the game board.
public int getMortgagePrice(): Gets and returns the mortgage price of the property.
public void setMortgagePrice(int mortgagePrice): Sets the mortgage price of the property.
public boolean checkMortgaged(): Checks and returns whether the property is mortgaged.
public boolean checkHasOwner(): Checks and returns whether the property has an owner.
public String getOwner(): Gets and returns the name of the owner of the property.
public void setOwner(String owner): Sets the name of the owner of the property.
public int getPrice(): Gets and returns the price of the property.
public int getRentPrice(): Gets and returns the rent price of the property.

3.2.6 Planet Class



Planet class represents the planets which is a major type of property on the game board and is a subclass of Property Abstract Class.

3.2.6.1 Attributes of Planet Class

private int totalRent: Represents the total rent of the planet.

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

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

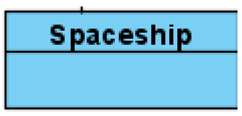
private int numHomes: Represents the number of homes purchased and built on the planet

3.2.6.2 Methods of Planet Class

public int getTotalRent(): Gets and returns the total rent of the planet

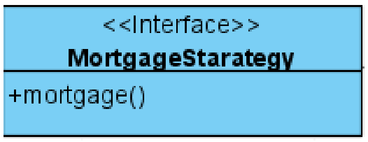
public void updateTotalRent(int totalRent): Updates and returns the total rent of the planet including the possible building structures.

3.2.7 Spaceship Class



Spaceship class represents the spaceship which is another type of property on the game board and is a subclass of Property Abstract Class.

3.2.8 MortgageStrategy Interface

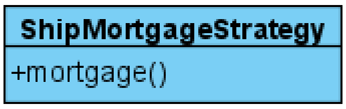


MortgageStrategy Interface defines different mortgage strategies for Property Class to operate the mortgage process for planets and spaceships on the game board.

3.2.8.1 Methods of MortgageStrategy Interface

public void mortgage(): Abstract method for mortgage.

3.2.9 ShipMortgageStrategy Class

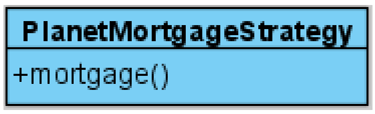


ShipMortgageStrategy Class defines a specific mortgage process for the spaceship properties.

3.2.9.1 Methods of ShipMortgageStrategy Class

public void mortgage(): Defines the mortgage process for spaceships specifically.

3.2.10 PlanetMortgageStrategy Class

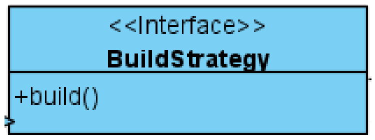


PlanetMortgageStrategy Class defines a specific mortgage process for the planet properties.

3.2.10.1 Methods of PlanetMortgageStrategy Class

public void mortgage(): Defines the mortgage process for planets specifically.

3.2.11 BuildStrategy Interface

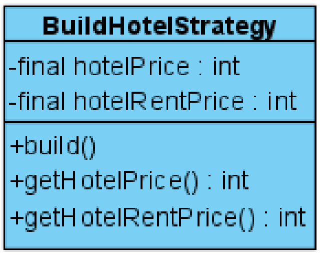


BuildStrategy Interface defines different building strategies like house, forest, hotel, and no building structures for Property Class to operate the building structures process only for planets on the game board.

3.2.11.1 Methods of BuildStrategy Interface

public void build(): Abstract method for building structures.

3.2.12 BuildHotelStrategy Class



BuildHotelStrategy Class defines a strategy for building hotel structures on a planet property.

3.2.12.1 Attributes of BuildHotelStrategy Class

private final int hotelPrice: Represents a single hotel structure building price.

private final hotelRentPrice: Represents a single hotel’s rent price.

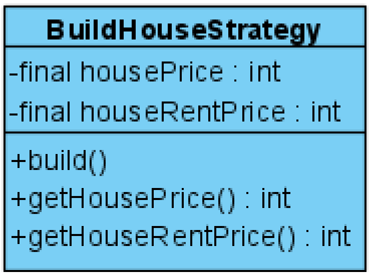
3.2.12.2 Methods of BuildHotelStrategy Class

public void build(): Operates building hotel structure.

public int getHotelPrice(): Gets and returns the hotelPrice attribute.

public int getHotelRentPrice(): Gets and returns the hotelRentPrice attribute.

3.2.13 BuildHouseStrategy Class



BuildHouseStrategy Class defines a strategy for building house structures on a planet property.

3.2.13.1 Attributes of BuildHouseStrategy Class

private final int houselPrice: Represents a single house structure building price.

private final houseRentPrice: Represents a single house’s rent price.

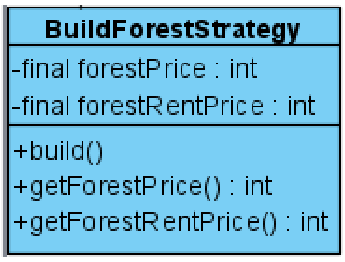
3.2.13.2 Methods of BuildHouseStrategy Class

public void build(): Operates building house structure.

public int getHousePrice(): Gets and returns the housePrice attribute.

public int getHouseRentPrice(): Gets and returns the houseRentPrice attribute.

3.2.14 BuildForestStrategy Class



BuildForestStrategy Class defines a strategy for building forest structures on a planet property.

3.2.14.1 Attributes of BuildForestStrategy Class

private final int forestPrice: Represents a single forest structure building price.

private final forestRentPrice: Represents a single forest’s rent price.

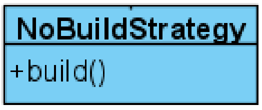
3.2.14.2 Methods of BuildForestStrategy Class

public void build(): Operates building forest structure.

public int getForestPrice(): Gets and returns the forestPrice attribute.

public int getForestRentPrice(): Gets and returns the forestRentPrice attribute.

3.2.15 NoBuildStrategy Class

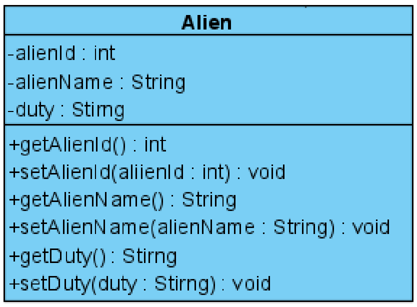


NoBuildStrategy Class defines a no building strategy for planets that has no building structures and all spaceships that never have building structures.

3.2.15.1 Methods of NoBuildStrategy Class

public void build(): Defines build operation for no structures.

3.2.16 Alien Class



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.

3.2.16.1 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.

3.2.16.2 Methods of Alien Class

public int getAlienId(): Gets and returns the alien id.

public void setAlienId(int alienId): Sets an alien id.

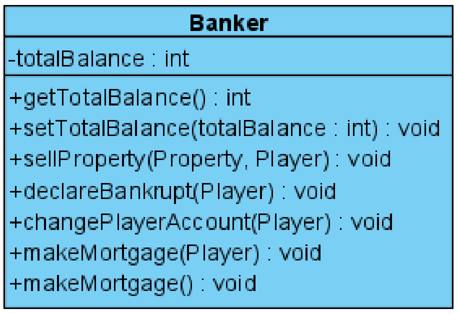
public String getAlienName(): Gets and returns the alien name.

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.

3.2.17 Banker Class


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

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.

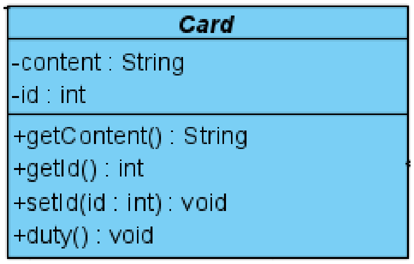
3.2.17.1 Attributes of Banker Class

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

3.2.17.2 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(Property property, Player player): The method is used for selling a property 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.

3.2.18 *Card* Abstract Class



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 10 cards, 5 for chance cards and 5 for chest cards.

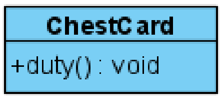
3.2.18.1 Attributes of *Card* Abstract Class

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

3.2.18.2 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 duty(): Fulfills the requirements written inside the chosen card.

3.2.19 ChestCard Class

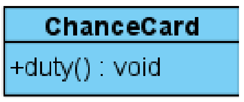


ChestCard is a subclass of Card Abstract Class and defined to represent a list of chest cards and their final content. There are a total of 5 chest cards on the game board.

3.2.19.1 Methods of ChestCard Class

public void duty(): Fulfills the requirements written inside the specific chest card.

3.2.20 ChanceCard Class

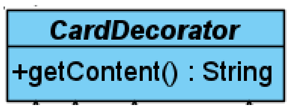


ChanceCard is a subclass of Card Abstract Class and defined to represent a list of chance cards and their final content. There are a total of 5 chance cards on the game board.

3.2.20.1 Methods of ChanceCard Class

public void duty(): Fulfills the requirements written inside the specific chance card.

3.2.21 *CardDecorator* Abstract Class

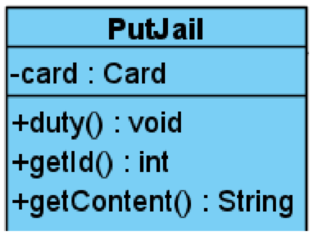


CardDecorator Abstract Class defines a decorator pattern for cards and it helps create different and combinated duties for each card type with the help of its concrete classes. It is a subclass of Card Abstract Class and it also has an aggregation relation to Card Abstract Class for decoration purposes.

3.2.21.1 Methods of *CardDecorator* Abstract Class

public String Content(): Inherited abstract class for concrete decorated classes.

3.2.22 PutJail Class



PutJail Class is a concrete class of *CardDecorator* Abstract Class to add a feature for putting a player to jail to a specific type of card.

3.2.22.1 Attributes of PutJail Class

private Card card: Represents a card that will be decorated with putting jail features.

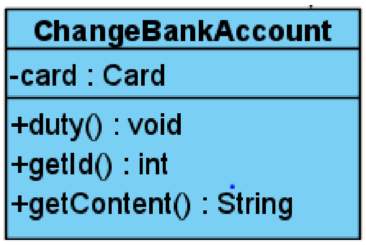
3.2.22.2 Methods of PutJail Class

public void duty(): Executes the requirement of the decorated card as its duty.

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

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

3.2.23 ChangeBankAccount Class



ChangeBankAccount Class is a concrete class of *CardDecorator* Abstract Class to add a feature for changing bank accounts of players to a specific type of card.

3.2.23.1 Attributes of ChangeBankAccount Class

private Card card: Represents a card that will be decorated with changing bank account features.

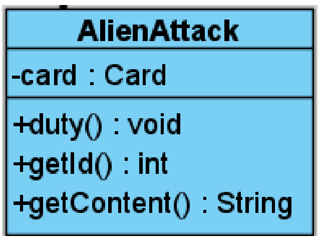
3.2.23.2 Methods of ChangeBankAccount Class

public void duty(): Executes the requirement of the decorated card as its duty.

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

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

3.2.24 AlienAttack Class



AlienAttack Class is a concrete class of *CardDecorator* Abstract Class to add a feature for different alien attacks to players, to a specific type of card

.

3.2.24.1 Attributes of AlienAttack Class

private Card card: Represents a card that will be decorated with alien attack features.

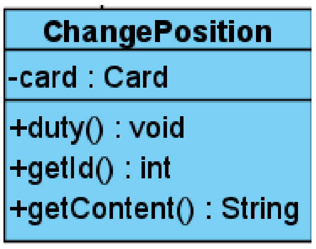
3.2.24.2 Methods of AlienAttack Class

public void duty(): Executes the requirement of the decorated card as its duty.

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

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

3.2.25 ChangePosition Class



ChangePosition Class is a concrete class of *CardDecorator* Abstract Class to add a feature for changing the position of players, to a specific type of card.

3.2.25.1 Attributes of ChangePosition Class

private Card card: Represents a card that will be decorated with changing position features of the players.

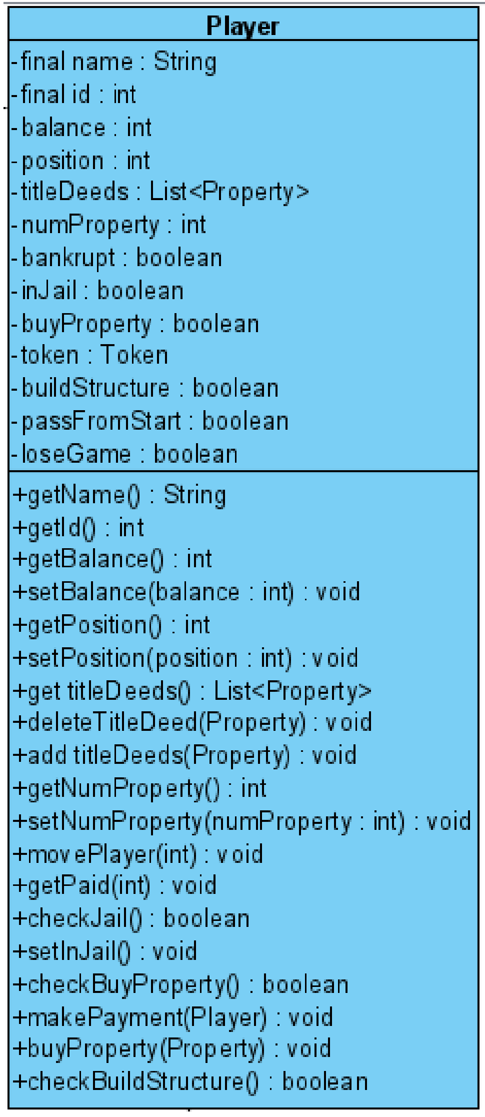
3.2.25.2 Methods of ChangePosition Class

public void duty(): Executes the requirement of the decorated card as its duty.

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

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

3.2.26 Player Class



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.

3.2.26.1 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.

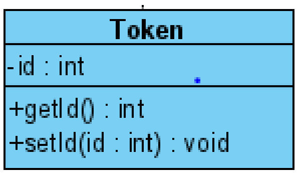
private List<Property> titleDeeds: Represents all the property’s title deeds which the player possesses as a list.
private int numProperty: Represents the number of properties which the player possesses.
private boolean bankrupt: Represents whether the player is bankrupt or not.
private boolean inJail: Represents whether the player is in jail or not.
private boolean buyProperty: Represents whether the player chooses to buy a property.
private Token token: Represents the token of the player.
private boolean buildStructure: Represents whether the player chooses to build structures.
private boolean passFromStart: Represents and keeps track of whether the player passes from the start partition on the game board.
private boolean looseGame: Represents and keeps track of whether the player lost the game yet.

3.2.26.2 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<Property> getTitleDeeds(): Gets and returns all the title deeds owned by the player as a list of properties.
public void addTitleDeeds(Property p): Adds the newly bought property given in the parameter by the player to the list of owned title deeds.
public void deleteTitleDeeds(Property p): Deletes the property given in the parameter from the list of owned title deeds by the player.
public int getNumProperty(): Gets and returns the total number of properties owned by the player.

public void setNumProperty(int numProperty): Sets the total number of properties 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 void setInJail(): Sets the jail condition for the player.
public boolean checkBuyProperty(): Checks whether the player wants to buy the property currently landed on.
public boolean checkBuildStructure(): Checks whether the player wants to build structures to the property he/she already possesses and currently landed on .
public void makePayment(Player player): Controls the payment made to another player specified in the signature.
public void buyProperty(Property property): Player buys the property given in the parameter.

3.2.27 Token Class



Token class represents the token object that each player takes to represent them on the game board during the game. Since each player has a token, Player Class has its instance as an attribute. Thus, it has an aggregation relation to Player Class.

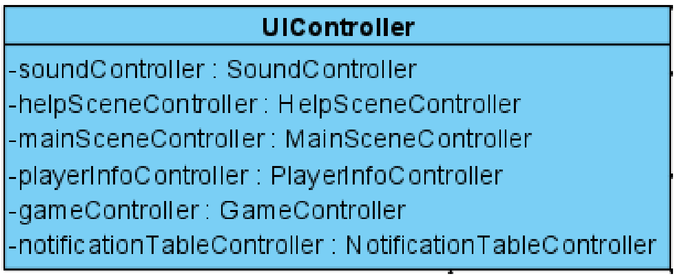
3.2.27.1 Attributes of Token Class

private int id: Represent the id of the token.

3.2.27.2 Methods of Token Class

public int getId(): Gets and returns the id of the token.
public int setId(int id): Sets the id of the token.

3.2.28 UIController Class


UIController
-soundController : SoundController
-helpSceneController : HelpSceneController
-mainSceneController : MainSceneController
-playerInfoController : PlayerInfoController
-gameController : GameController
-notificationTableController : NotificationTableController

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.

3.2.28.1 Attributes of UIController Class:

private SoundController soundController: Represents and stores a SoundController object.

private HelpSceneController helpSceneController: Represents and stores a HelpSceneController object.

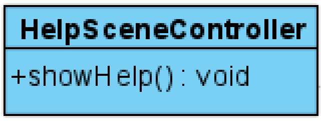
private MainSceneController mainsceneController: Represents and stores a MainSceneController object.

private PlayerInfoController playerInfoController: Represents and stores a PlayerInfoController object.

private GameController gameController: Represents and stores a GameController object.

private NotificationTableController notificationTableController: Represents and stores a NotificationTableController object.

3.2.29 HelpSceneController Class

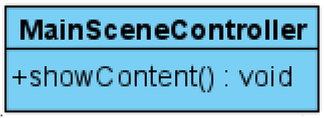


This class is to provide the controller for the help scene to the UIController Class.

3.2.29.1 Methods of HelpSceneController Class:

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

3.2.30 MainSceneController Class

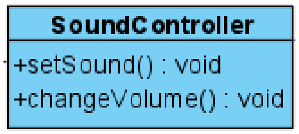


This class is to provide the controller for the main scene to the UIController Class.

3.2.30.1 Methods of MainSceneController Class:

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

3.2.31 SoundController Class



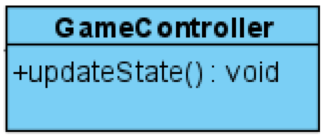
This class controls the sound settings of the entire game and provides an attribute to UIController Class.

3.2.31.1 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.

3.2.32 GameController Class

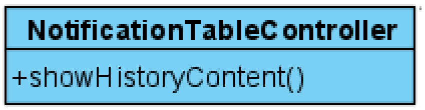


This class provides a controller for the game and declares an object for the UIController Class.

3.2.32.1 Methods of GameController Class:

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

3.2.33 NotificationTableController Class:

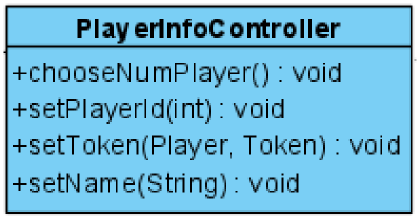


This class provides a controller for the notifications of the passed activities of the game and declares an object for the UIController Class.

3.2.33.1 Methods of NotificationTableController Class:

public void showHistoryContent(): Shows what has been done in the game and its reflection to players in each turn on the notification panel.

3.2.34 PlayerInfoController Class



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

3.2.34.1 Methods of PlayerInfoController Class:

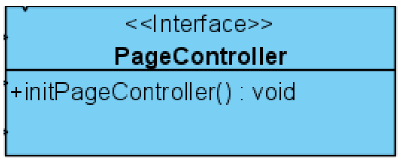
public void chooseNumPlayer(): Provides the user to choose the number of players for the game.

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

public void setToken(Player p, Token t): Sets the chosen tokens to each player.

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

3.2.35 PageController Interface

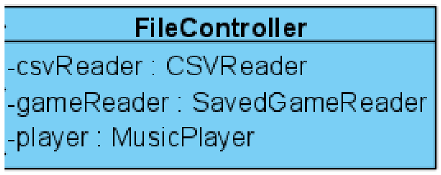


PageController is an interface for PlayerInfoController, GameController, SoundController, MainSceneController, HelpSceneController, and NotificationTableController classes.

3.2.35.1 Methods of PageController Interface:

public void initPageController(): Initializes the page controller.

3.2.36 FileController Class

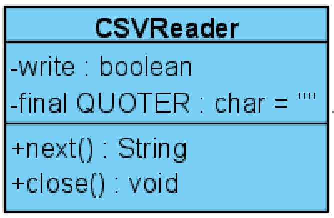


FileController class is a class that keeps the objects of file dependent classes and it is connected with the game engine class.

3.2.36.1 Attributes of FileController Class:

private CSVReader csvReader: Represents and stores a CSVReader object.
private SavedGameReader gameReader: Represents and stores a SavedGameReader object.
private MusicPlayer player: Represents and stores a MusicPlayer object.

3.2.37 CSVReader Class



CSVReader class is a class that operates the reading of a csv file. It extends BufferedReader and implements Serializable classes.

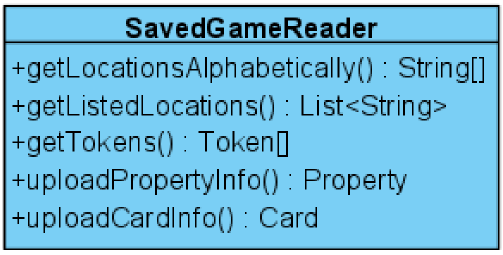
3.2.37.1 Attributes of CSVReader Class:

private boolean write: A boolean value to check the data is written before.
private final char QUOTER: A char consisted of a quotation mark and it is used in the reading process one by one.

3.2.37.2 Methods of CSVReader Class:

public String next(): It is the method to read inside of a CSV file.
public void close(): The method is an override method that is used to close the reader.

3.2.38 SavedGameReader

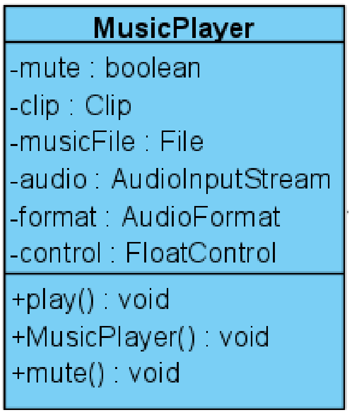

SavedGameReader
+getLocationsAlphabetically() : String[]
+getListedLocations() : List<String>
+getTokens() : Token[]
+uploadPropertyInfo() : Property
+uploadCardInfo() : Card

SavedGameReader class is a class that reads the text files of the game.

3.2.38.1 Methods of SavedGameReaderClass:

public String[] getLocationsAlphabetically(): The method is used to return the locations on the game board and keep them in a string array from the text file.
public List<String> getListedLocations(): The method returns a string list that consists of the locations on the game board and is used to find the player locations .
public Token[] getTokens(): The method takes the token names from a .txt file and creates a token array keeping the tokens in it.
public Property uploadPropertyInfo(): The method is used to upload the information of properties in the game from a .txt file.
public Card uploadCardInfo(): The method is used to upload the cards' information from a .txt file like what the cards do.

3.2.39 MusicPlayer Class



MusicPlayer class contains the methods and attributes in order to play and stop the music during the game.

3.2.39.1 Attributes of MusicPlayer Class:

private boolean mute: This attribute checks if the music is muted or not.
private Clip clip: This attribute is used in play method.
private File musicFile: This attribute is initialized in play method with a music file.
private AudioInputStream audio: This attribute is assigned to the audio format.
private AudioFormat format: This attribute is used in play method.
private FloatControl control: This attribute is used to arrange the volume.

3.2.39.2 Methods of MusicPlayer Class:

public void play(): This method takes the sound file and runs it.
public void MusicPlayer(): This is the constructor that takes the .wav file in it.
public void mute(): This method is used to mute the music by calling stop and start methods onto the clip attribute.

3.3. Packages

The packages in the game are designed with respect to packages at the subsystem decomposition part. Therefore, the packages are MonopolyFiles which is responsible for the connection between MonopolyGameModel and MonopolyFileManagement, FileControllerManager and BankerFileManager 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.

Because Strategy and Decorator design patterns are used in the design, there exists also more interfaces. MortgageStrategy is an interface that has two implementations for the planets and for the spaceships. Also, BuildStrategy is an interface that specifies the building types for the properties in the game which are forests, hotels, houses and no buildings for spaceships.

4. Improvement Summary

- 4.1. Design Patterns are introduced.
- 4.2. Final Object design is updated according to Design Patterns.
  - 4.2.1. Design patterns are represented.
  - 4.2.2. New attributes and methods are added.
  - 4.2.3. Name of attributes and methods are updated for consistency.
- 4.3. New controller classes are added to Object Design which are responsible for handling the saving or loading files to handle data in the Game.

5. 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]

[3] Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra - Head First Design Patterns-O’Reilly (2014) [Accessed:Dec. 7, 2020]