CS 319

Object-Oriented Software Engineering

Project Design Report

Monopoly

Group 2-H

Aziz Ozan Azizoğlu - 21401701
Hammad Khan Musakhel - 21801175
Veli Can Mert - 21602394
Gokhan Mullaoglu - 22001086
Mert Sayar - 21601435

1.Introduction

1.1. Purpose of the System

Monopoly is expected to provide users with high-performance, user-friendly and smooth game experience. Aim of users is to be the last player who does not go bankrupt.

At first, we focused on performance to make the game as fast as possible. We aim to provide users with high-performance game experience opportunities.

Then, we are interested to make the game as easy-to-use as possible. Users will not have difficulties in-game. Scoreboard, dice, deeds and etc are easy-to-access. Aim of the games is to entertain the users as much as possible without any confusion so “usability” is crucial for the games. Monopoly includes the buttons and information boxes in efficient places.

Lastly, we focused on visual appeal to give users smooth game experience. Modern and high quality looking games attract the attention of people and they make people feel included in the game more than other games. Monopoly is a high quality and modern looking game.

Our main purpose is to design a game with high-performance, smooth appearance and user-friendly interface to maksimize users’ enjoyment from the game.

1.2. Design Goals

Design goals are mainly related to non-functional requirements. Our non-functional requirements from the analysis report can be seen below.

1.2.1 Criteria

Performance:

Performance is one of the most important goals of our game design because it is important for users. We need to ensure that users have a unique game experience so the game needs to have high and stable performance. The frame rate will not drop below 30 frames per second, on an intel core 2 duo or higher processor machine. Rendering will be done according to the display frame, and the upper limit of crucial cells and objects visible on the screen will be set to 50.

Usability:

Menus, titles, buttons and scoreboard provide users with a new and more engaging form of Monopoly. We have a very organized and concise menu in terms of content. The menu has access to tutorials where all the game rules are displayed. We designed an easy-to-use system so that users can access game functions easily and they can enjoy the game without having difficulties.

Extensibility:

The code is organized in a way such that new/improvising features to the game are easily added. Most games need improvement and bug-fixing all the time so we take this into consideration and try to have a developable code design of the game. That’s why we use an object-oriented design pattern.

Reusability:

We designed the system piece by piece with respect to object-oriented programming. Therefore, we can use some classes in other games or any other similar systems without changing them. Within reusability, we designed the classes independent from the system.

User Experience:

Users need to keep track of money, deeds and some other things like chance cards & community chest cards. To provide a good gaming experience, we make the user interface understandable and simple as much as we can. We have a how to play/tutorial part to ease for users.

2. Software Architecture

2.1 Architectural Styles:

For the architectural style to design our project's basis on, we have elected to go with Model-View-Controller (MVC). The MVC has many effective advantages that favor us in the design making, especially the availability of its variants. The MVC's flexibility in use has allowed us to use another variant of it that is somehow not conventional.

The traditional MVC is the one where data flow is accessed between Model and View interchangeably; this results in the dependency between the views and their corresponding data models restricts the designers to decouple the UI from the actual game logic. In essence, a tightly coupled system with highly coupled subsystems are formed. We proceed with decomposition to attain low coupling and high coherence in the system which the traditional MVC restricts.

The version of MVC we have elected to use as a group is the one where there is no direct communication between Model and View, rather the Controller acts as the mediator between both. When View is received with an input, it notifies the Controller. Then, the Controller updates the relevant object models (in other words the corresponding object models). The Model then notifies the Controller which in turn updates the View. This variant of the MVC is termed as "Mediating-Controller MVC"; the model does not need to deal with the user's request. The controller tells the model what to do, and the controller interprets the request then retrieves the data that the request needs from the model.

This version of MVC enhances the design in terms of reliability as It reduces coupling. By using this variant, the Model and View are completely independent and unaware of one another thus both canoe associated at runtime. Another plus factor of this architectural style is the simplicity of model and view, yet the controller is a complex feature but can be managed as we have relative simplicity overall. Lastly, View is supplied with relevant and necessary updates from the controller, adapting the "push mechanism" rather than the "pull mechanism."

Ideally in this sort of set up we tend to have two controller objects: model controller and view controllers. However, we have yet to adapt to that style. Nonetheless, we have opted to keep one controller object to keep it relatively simple. Although, the idea of having two controller objects is at the back of our minds and can be implemented in the last iteration if no confidence vote was drafted on having one controller object.

2.2 Subsystem Decomposition:

We now undertake the process of decomposing subsystems. As stated above, to attain the ideal scenario, we need to have less coupling and high coherence. Decomposition lays out the general structure to further implement the design pattern, it is similar to having a blur print with defined components with no attributes yet defined or finalized for each sub-component. The Subsystem Decomposition is also planned on the architectural style, although here we define the larger pictures: where to place the different classes of class diagram.

As mentioned above, we have divided the general format into three layers: View, Controller, and Model. Starting with View, we chose to list the screen classes in it as they provide the user with an interface. The GameManager class, which serves the purpose of managing the game, is enlisted in Controller. Moreover, the object classes that represent the objects of the game are in the Model.

We have chosen the relevant packages based on their functionality towards the relevant layer and also the fact that each subsystem has a responsibility to invoke other systems so that the design/project can stay maintainable. In essence, we want to deliver the idea of Mediating Controller MVC as Data Observer is used in View to connect with the controller, GameManger class. The View takes inputs with KeyListener and MouseListener and then proceeds forward with notifying the Controller; the GameManager class is then directly linked with the Model and its objects and thus forming the Façade Design: any connection between UI and GameObjects is bridged by GameManager. This is very convenient in solving errors and also contributes in stabilizing, modifying, and extending the game.

The View is only connected to the Controller through Data Observer, and the Controller is connected to the Model (player factory and Game Board) through GameManager (Façade Principle). This results in low coupling and elaborative cohesion which is more flexible and extendable as the layers aren't interconnected with many associations and thus changes in one won't affect the other a lot. The Diagram 2.2.1 below defines the Subsystem Decomposition (on the next page) with View on top, Manger following it and Control in the last:

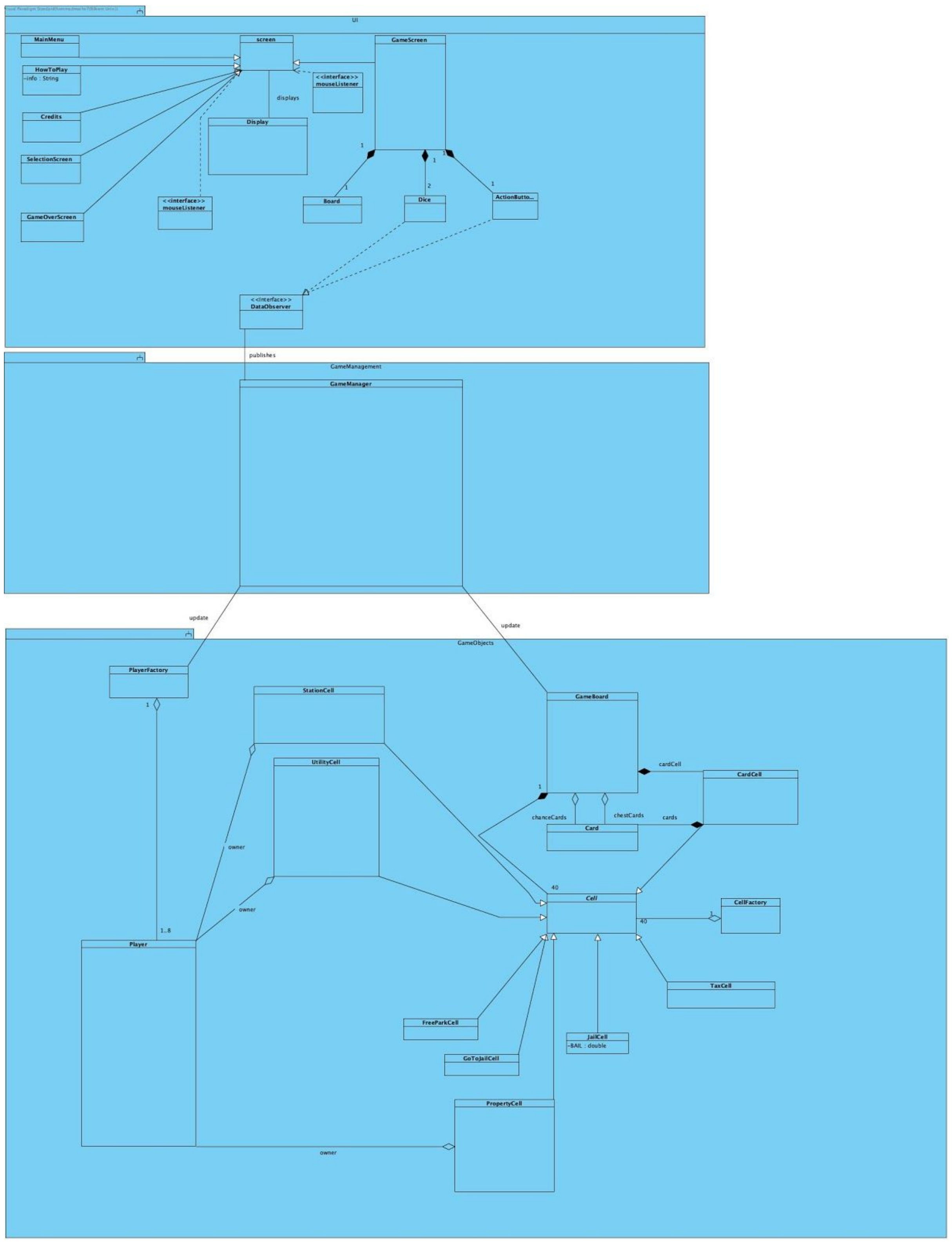

UI
MainMenu
HowToPlay
-info : String
Credits
SelectionScreen
GameOverScreen
screen
displays
Display
<<interface>>
mouseListener
<<interface>>
mouseListener
GameScreen
1
1
1
1
2
1
1
Board
Dice
ActionButto...
<<Interface>>
DataObserver
publishes
GameManagement
GameManager
update
update
GameObjects
PlayerFactory
1
1..8
Player
StationCell
UtilityCell
owner
owner
owner
PropertyCell
GameBoard
cardCell
CardCell
1
chanceCards
chestCards
cards
Card
40
Cell
40
1
CellFactory
TaxCell
FreeParkCell
GoToJailCell
JailCell
-BAIL : double

Figure 2.2.1

2.3 Hardware/Software Mapping:

Monopoly Game will be implemented in the Java programming Language. We will make use of JavaFX libraries. Thus, a software requirement for Monopoly Game would be JRE (Java Runtime Environment) with at least JDK 8 or newer to support JavaFX libraries.

Moreover, Monopoly Game will require a keyboard/mouse as a hardware requirement. Mouse will be used for Input/Output of the game. Users will control the menu selection, die generation, etc with the mouse. Consequently, the game's requirements in terms of hardware and software are collectively minimal.

We first thought of using a JSON file for storing maps but we didn't want to indulge with the web, hence we chose txt files to store maps, system preferences and storage of the game in general.

2.4 Persistent Data Management:

As we stated before, Monopoly Game wouldn't require any database or any complex data structure. Rather, it will store its classes in the hard-drive of the client (data will be kept in txt files). The txt files containing data regarding maps/aesthetic of the game board will not be allowed to be modified. However, data members such as setting presences and highest score will be kept in modifiable txt files, therefore users can change or update the data of these files during play time. To store the images, we will use .gif format (subject to change upon procession).

2.5 Access, Control, and Security:

There will not be any access control and security efforts in order to prevent unwanted acts as we have not linked the game to the web servers (as mentioned before, it doesn't require a database).

2.6 Boundary Conditions:

Monopoly Game will not require any installation but rather will have an executable .exe extension. The game can be copied from one computer to another.

The game can be closed by clicking the “Exit” button on the main menu. If there is an error, the game might proceed with partial or non loading of images; it can be fixed by fixing the corrupted .txt file by fixing the bugs/relevant errors personally. If the game collapses, settings/score will be lost.

3.Subsystem Services

3.1 Logic:

The game logics form a subsystem. Functions work with respect to the messages of the Manager subsystem. If the Manager subsystem sends instructions to Logic, the Logic subsystem makes changes and records them. If these updates are required to change in User Interface, Logic sends messages to the User Interface subsystem about the changes.

3.2 User Interface:

User Interface subsystem is a subsystem of Monopoly and displays the user interface of all systems according to the messages of the Manager subsystem. User Interface subsystem interacts with users and gets input from them. Then it sends them to the Manager subsystem and if these inputs/updates are required to change in User Interface, Manager sends messages to User Interface subsystem about the changes.

3.3 Manager:

Manager subsystem interacts both UI and Logic subsystems. It gets inputs from UI subsystem and it sends them to Logic subsystem. Then, if these inputs/updates are required to change in User Interface, Logic subsystem sends messages to Manager and Manager sends messages to User Interface subsystem about the changes.

4. Low-level design

4.1 Object Design Trade-offs:

Memory versus Performance:

In the game, we have created classes and used objects in an Object Oriented way. So, we think that performance will be efficient enough for us. We could say that memory can be a problem since we are using OO programming. But since we are making this game for only PC platforms, it won't be a big issue.

Features versus The Simplicity:

We wanted to add a feature for letting players play together on the same device to the game. We wanted to add this feature since the players mostly want to play together and paying attention to these types of accessibilities in the games when playing. But adding this new feature came with some complexity.

Rapid prototype versus Understandability:

Since we will add the feature explained above in our version of Monopoly, we will also try to add more understandability to the game interface for decreasing it's complexity but it will make the rapid prototype stage hard for us. We can also say that it "Costs" us design time.

4.2 Final object design:

Up to now, use of three different design patterns are planned. Subsystems are implemented related to planned design patterns.

- Singleton Design Pattern

Singleton design pattern will be used in the monopoly game for the controller object, which is the GameManager object in our case. There will be exactly one instance of the controller which will be responsible for the functional operation of Monopoly Game. Singleton design pattern will let us be able to call the exact single instance of the controller object to perform any logical or retrieval operations.

- Observer Design Pattern

Use of observer design pattern will let provide the communication between the view objects and the controller. Data flow will be two-sided. Observer interface will be between the GameManager object and sub-components of GameScreen objects such as Board, Dice and ActionButton objects as members of view objects. Controller will be publishing data and view objects will be updating the user interface according to the published data as listener objects.

- Factory Design Pattern

Factory design pattern is used in the implementation for Player and Cell objects. PlayerFactory and CellFactory classes will be responsible to create and keep instances of these classes. Cell class has multiple subclasses with different types. Factory class will let each instance's type be different. Since factory method creates a superclass but alters the type of objects to be created, it will be a useful design pattern for our implementation.

4.3 Packages:

4.3.1. Internal Library Packages:

Model Package
This package includes classes which are responsible to manage the storage for internal game objects and their storage.

View Package
This package includes the classes which are responsible to manage user interface.

Controller Package
This package includes the classes which are responsible to manage the system.

4.3.2. External Library Packages:

- **java.util**

This package contains the collections frameworks which will be used in many places as abstract List objects, ImmutableList objects and ObservableList objects.

- **javafx.scene.layout:**

This package provides User Interface layouts to fit GUI objects. We will use this for managing UI components.

- **javafx.event**

Provides basic framework for FX events, their delivery of inputs between different hardware components and handling.

- **javafx.scene.input**

Provides the set of classes for mouse and keyboard input event handling. We used this package to handle keyboard inputs and MouseEvents

- **javafx.scene.image**

Provides the set of classes for loading and displaying images. We use images to display luck and chest cards and icons for the cells.

4.4 Class Interfaces:

4.4.1 User Interface Subsystem:

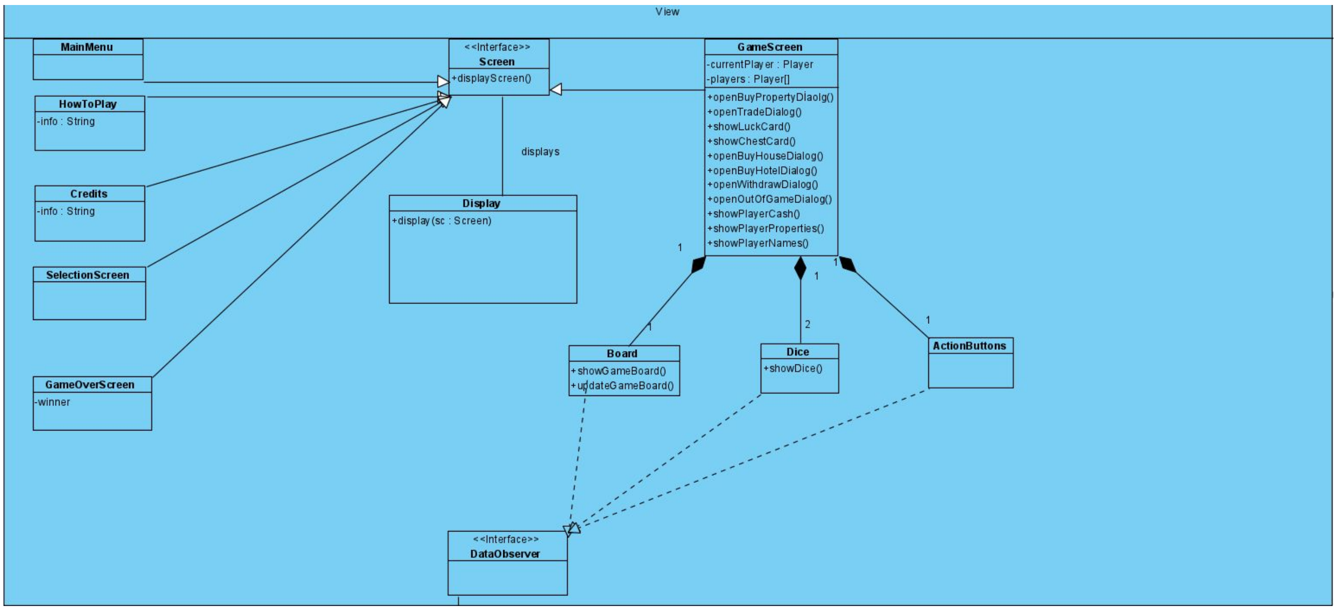


Figure 4.4.1.1: User Interface Subsystem

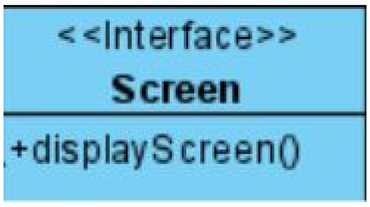


Figure 4.4.1.2: Screen Interface

Screen is the interface class to display different Screens.

Methods:

- **private void displayScreen():** displayScreen() method is a method of an interface which is implemented by 6 different classes. This method displays the related screen. Displayed screen can be either the main menu screen, how to play screen, credits screen, game over screen or game screen itself.

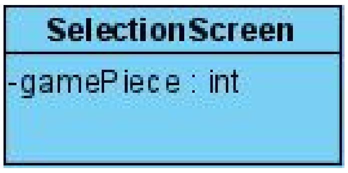


Figure 4.4.1.3: SelectionScreen Class

This class shows the selection screen for game pieces.

Attributes:

- **private int gamePiece:** This the attribute which shows the game pieces which can be selected by the players in the beginning of the game.

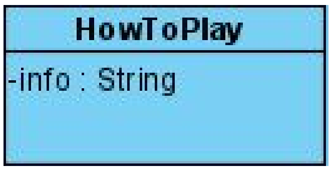


Figure 4.4.1.4: HowToPlay Class

This class shows the “how to play” screen.

Attributes:

private String info: This is the attribute which is the information & directions displayed in the how to play screen.

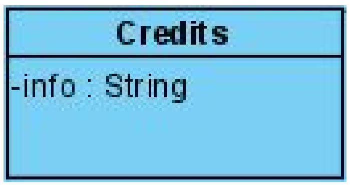


Figure 4.4.1.5: Credits Class

This class shows the “Credits” screen.

Attributes:

- **private String info:** This is the attribute which is the information & directions displayed in the credits screen.

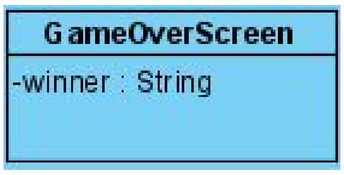


Figure 4.4.1.6: GameOverScreen Class

This class shows the “Game Over” screen.

Attributes:

- **private String winner:** This is the attribute which is the name of the winner whose name will be displayed in the credits screen.

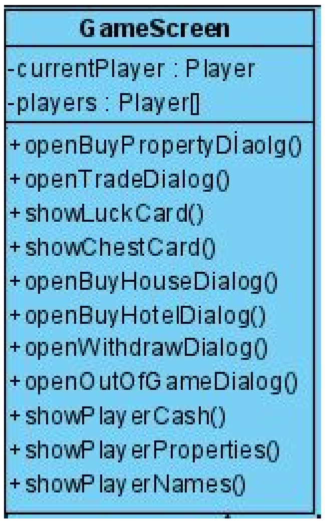


Figure 4.4.1.7: GameScreen Class

This class shows the whole Game screen.

Attributes:

- **private Player currentPlayer:** This attribute holds the Player object which has the turn.

- **private Player[] players:** This attribute holds the active players in the game in an array.

Methods:

- **private void openBuyPropertyDialog():** This method opens the dialog for the user in order to buy any property.

- **private void openTradeDialog():** This method opens the dialog for the user in order to complete a trade with any other user.

- **private void showLuckCard():** This method shows the luck card for a user.

- **private void showChestCard():** This method shows the chest card for a user.

- **private void openBuyHouseDialog():** This method opens the dialog for the user in order to buy a house for his/her property.

- **private void openBuyHotelDialog():** This method opens the dialog for the user in order to buy a house for his/her property.

- **private void withdrawDialog():** This method opens the dialog for the user in case the user decides to withdraw from the game.

- **private void outOfGameDialog():** This method opens the dialog for the user in case the user is out of the game. Related player may get bankrupt or withdraw from the game.
- **private void showPlayerCash():** This method continuously shows the player cash on the game screen.

- **private void showPlayerProperties():** This method continuously shows the player properties on the game screen.

- **private void showPlayerNames():** This method continuously shows the player names on the game screen.

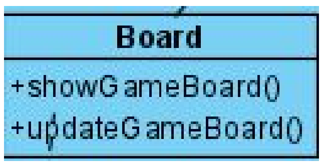


Figure 4.4.1.8: Board Class

This class performs visual representation of the game board.

Methods:

- **private void showGameBoard():** This method shows the current state of the game board.

- **private void updatesGameBoard():** This method updates the game board according to the current state of the game.

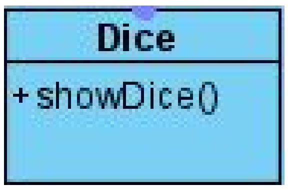


Figure 4.4.1.9: Dice Class

This class performs visual representation of dice on the game screen.

Methods:

- **private void showDice():** This method shows the current state of the visual dice objects on the game screen.

Figure 4.4.1.10: ActionButtonsClass

This class adds the action buttons such as “Withdraw” or “End Turn” on the game screen.

Figure 4.4.1.11: DataObserver Interface

This class performs the communication between view and controller objects.

4.4.2 Controller:

Our manager class interface consists of one general GameManager class that is responsible for the mouse actions, starting/ending the game, for giving turns to players, updating UI(user interface) and updating the attributes of the player classes.

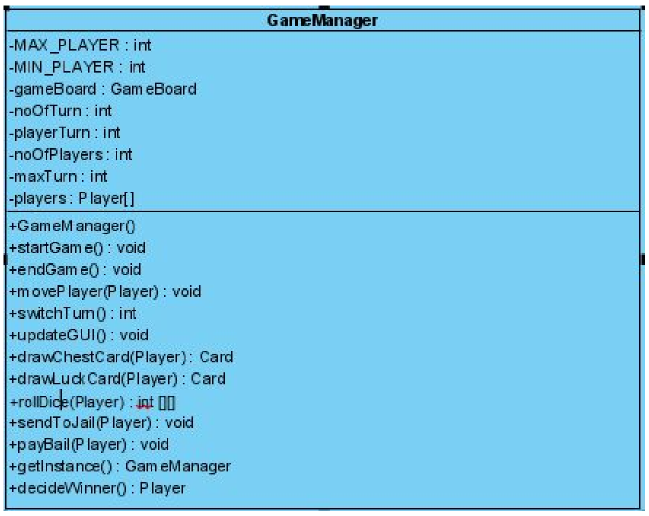


Figure 4.4.2.1: GameManager Class

Methods:

- **public void startGame():** This function starts the game.

- **public void endGame():** This function ends the game if needed.

- **public void movePlayer():** This function is for moving the player avatar after throwing dice.

- **public int switchTurn():** This function gives the turn to the next player.

- **public void updateGUI()**: This method is for updating the GUI when needed.

- **public Card drawChestCard(Player):** This method is for drawing a chest card.

- **public Card drawLuckCard(Player):** This method is for drawing a luck card.

- **public int[] rollDice(Player):** This function will be invoked when a player presses the “Roll Dice” button.

- **public void sendToJail(Player):** This function will be invoked when a player sits on the jail cell.
- **public void payBail(Player):** This function will be invoked when a player clicks the button and if they have enough money.

- **public GameManager getInstance():** This function returns the GameManager object instance.

- **public Player decideWinner():** This function decides and returns the winner of the game.

Attributes:

- **private final int MAX_PLAYER:** This attribute indicates the maximum number of players that can play the game for starting the game scene.
- **private final int MIN_PLAYER:** This attribute indicates the minimum number of players that can play the game for starting the game scene.
- **private GameBoard extension:** This attribute is an instance of the GameBoard. This is for the second island(game board) in the game.
- **private GameBoard gameBoard:** This attribute is an instance of the GameBoard class.
- **private Player[] players:** This attribute holds an array of player objects in the game session.
- **private int noOfTurn:** This attribute holds an integer for the total turn count that happened through the game for checking if the game can continue or the maximum turn limit exceeded.
- **private int playerTurn:** This attribute holds an integer for the current player that is having the turn at the moment.
- **private int noOfPlayers:**This attribute holds an integer for the total number of players that are playing the game in that session.

4.4.3 Model:

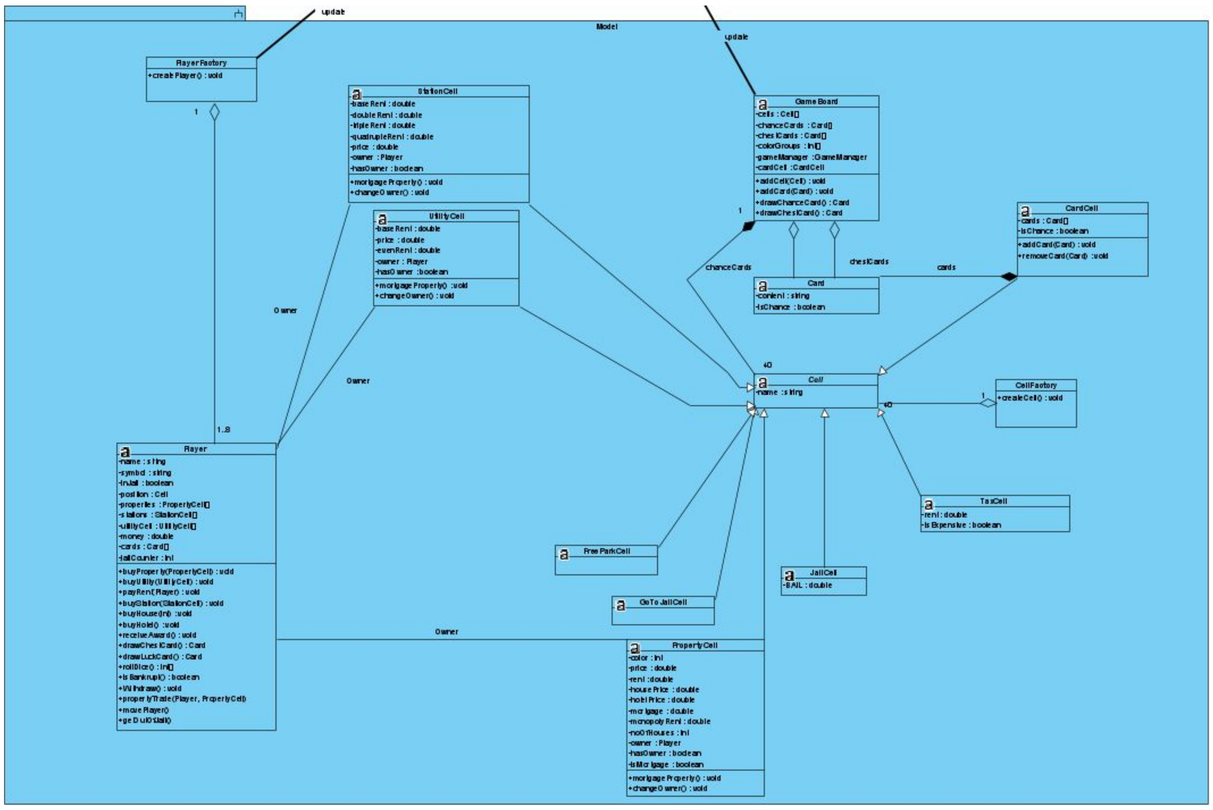


Figure 4.4.3.1: Model Components of the Class Diagram

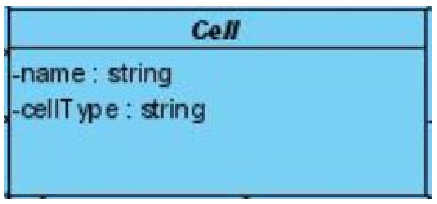


Figure 4.4.3.2: CellClass

This class is an abstract class that gives inheritance to cells.

Attributes:

- **private string name:** this attribute holds the cell name as a string.

- **private string cellType:** this attribute holds the type of the cell as a string.

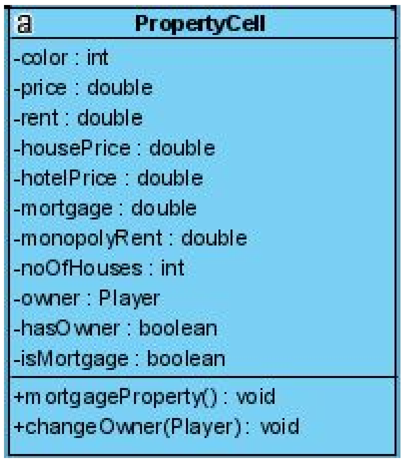


Figure 4.4.3.3: PropertyCellClass

Attributes:

- **private int color:** this attribute holds the color of property cells as an integer.
- **private int price:** this attribute holds the price of property cells as a double.
- **private double rent:** this attribute holds the rent of property cells as a double.
- **private double housePrice:** this attribute holds the amount that should be paid to build houses to property cells as a double.
- **private double hotelPrice:** this attribute holds the amount that should be paid to build hotels to property cells as a double.
- **private double mortgage:** this attribute holds the mortgage amount of property cells as a double.
- **private double monopolyRent:** this attribute holds the rent if a player achieves monopoly as a double.
- **private int noOfHouses:** this attribute holds the number of houses that are built to a property cell as an integer.

- **private Player owner:** this attribute holds the owner of a property cell as a Player type variable.

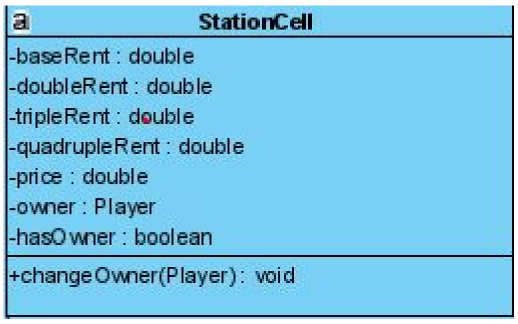
- **private boolean hasOwner:** This attribute holds whether the cell has an owner or not.

- **private boolean inMortgage:** This attribute holds whether the cell is in mortgage or not.

Methods:

- **public void mortgageProperty():** This method mortgages the property cell.

- **public void changeOwner(Player):** This method replaces the owner of the cell with the player which is taken as parameter.



Attributes:

- **public double baseRent:** This attribute holds the rent price if a player owns one station cell.

- **public double doubleRent:** This attribute holds the rent price if a player owns two station cells.

- **public double tripleRent:** This attribute holds the rent price if a player owns three station cells.

- **public double quadrupleRent:** This attribute holds the rent price if a player owns four station cells.

- **public double price:** This attribute holds the price of the cell.

- **public Player owner:** This attribute holds the owner of the cell.

- **public boolean hasOwner:** This attribute holds whether the cell has an owner or not.

Methods:

- **public void changeOwner(Player):** This method replaces the owner of the cell with the player which is taken as parameter.

GoToJailCell

Figure 4.4.3.4: GoToJailCell Class

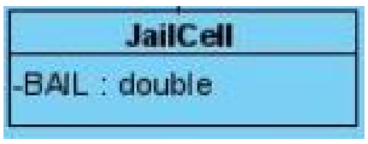


Figure 4.4.3.5: JailCelll Class

Attributes:

- **private final double BAIL:** this attribute holds the amount that should be paid to get out of jail as a double.

FreeParkCell

Figure 4.4.3.6: FreeParkCell Class

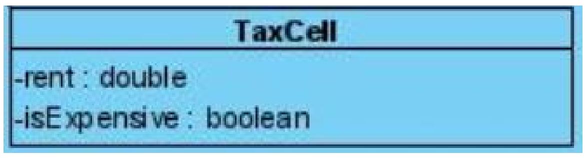


Figure 4.4.3.7: TaxCell Class

Attributes:

- **private double rent:** this attribute holds the amount should be paid when a player is in the tax cell as a double.

- **private boolean isExpensive:** this attribute holds to determine whether a tax cell is the expensive one or not as a boolean.

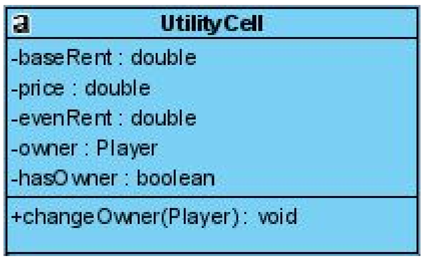


Figure 4.4.3.8: UtilityCell Class

Attributes:

- **private double baseRent:** this attribute holds the amount of rent if a player has one of utility cells as a double.

- **private double price:** this attribute holds the price of a utility cell as a double.

- **private double evenRent:** this attribute holds the amount of rent if a player has two or more Utility Cells as a double.

- **private Player owner:** this attribute holds the owner of a utility cell as a Player type variable.
- **private boolean hasOwner:** This attribute holds whether the cell has an owner or not.

Methods:

- **public void changeOwner(Player):** This method replaces the owner of the cell with the player which is taken as parameter.

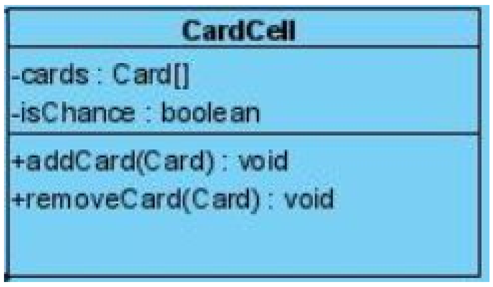


Figure 4.4.3.9: CardCell Class

Attributes:

- **private Card[] cards:** this attribute holds the cards as a card object array.

- **private boolean isChance:** this attribute holds to determine whether a card is a chance card or a chest card as a boolean.

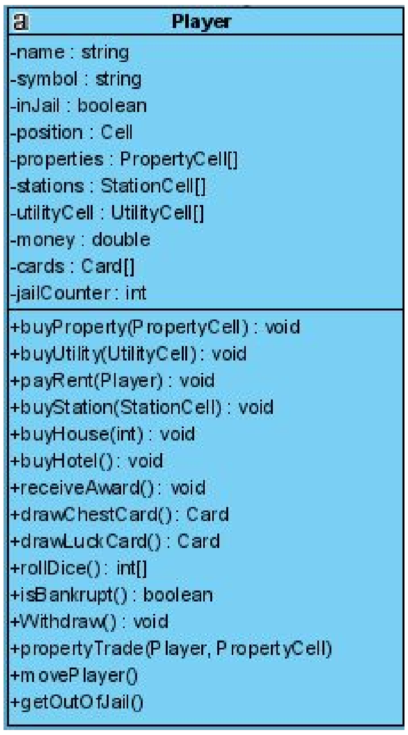
Methods:

- **public void addCard(Card):** this method is used for adding a card to the card cell.

- **public void removeCard(Card):** this method is used for removing a card from the card cell.

CellFactory

Figure 4.4.3.10: CellFactory Class



Player

-name : string
-symbol : string
-inJail : boolean
-position : Cell
-properties : PropertyCell[]
-stations : StationCell[]
-utilityCell : UtilityCell[]
-money : double
-cards : Card[]
-jailCounter : int

+buyProperty(PropertyCell) : void
+buyUtility(UtilityCell) : void
+payRent(Player) : void
+buyStation(StationCell) : void
+buyHouse(int) : void
+buyHotel() : void
+receiveAward() : void
+drawChestCard() : Card
+drawLuckCard() : Card
+rollDice() : int[]
+isBankrupt() : boolean
+Withdraw() : void
+propertyTrade(Player, PropertyCell)
+movePlayer()
+getOutOfJail()

Figure 4.4.3.10: Player Class

This class represents the player object in monopoly.

Attributes:

- **private string name:** This attribute holds a string for player names.

- **private string symbol:** This attributes holds a string for selected player symbols

- **private bool inJail:** This attribute stores a boolean that returns true if the player is in jail.

- **private Cell position:** This attribute holds the current position of the player.

- **private PropertyCell[] properties:** This attribute holds a list for players' own properties from the board.

- **private StationCell[] stations:** This attribute holds a list for players' own stations.

- **private UtilityCell[] utilityCell:** This attribute holds a list for UtilityCell that the player has.

- **private double money:** This attribute stores the player money.

- **private Card[] cards:** This attribute holds a list of cards that the player has.

- **private int jailCounter:** This attribute holds an integer value for every turn that the player is in jail.

Methods:

- **public void BuyProperty(PropertyCell):** This method will be invoked when the player clicks the BuyProperty Button. Buys the selected property if Player money is enough.

- **public void BuyStation(StationCell):** This method will be invoked when the player buys the current cell and if the current cell is an instance of a station cell.

- **public void BuyUtility(UtilityCell):** This method will be invoked when the player clicks the BuyUtility Button. Buys the selected UtilityBuilding if Player money is enough.

- **public void PayRent(Player):** This method will be invoked when the player is in the opponent's cell and needs to pay the rent.

- **public void BuyHouse(int):** This method will be invoked when the player clicks the BuyHouse button. Buys the selected house if Player money is enough and has property in the current cell index.

- **public void BuyHotel():** This method will be invoked when the player clicks the BuyHotel button.

- **public void getOutOfJail():** This method will be invoked if the player gets out of the jail and the inJail attribute will be set to false.

- **public void ReceiveAward():** This method will be invoked when the players finish the board and start from the starting point again.

- **public Card DrawChestCard():** This method will be invoked when the player is in the chest cell and can draw a chest card.

- **public Card DrawLuckCard():** This method will be invoked when the player is in the Luck Card cell and can draw a luck card.

- **public int RollDie():** This method will be invoked when the player presses the roll dice button.

- **public bool IsBankrupt():** This method returns a boolean for checking if the player is bankrupt.

- **public void Withdraw():** This method will be invoked when a player wants to surrender and clicks the Withdraw Game button.

- **public void PropertyTrade(Player, PropertyCell):** This method is for trading properties. It has two parameters for defining which player to trade with and which propertyCell to trade

- **public void MovePlayer():** This method will change the players current cell after the player rolls the dice.

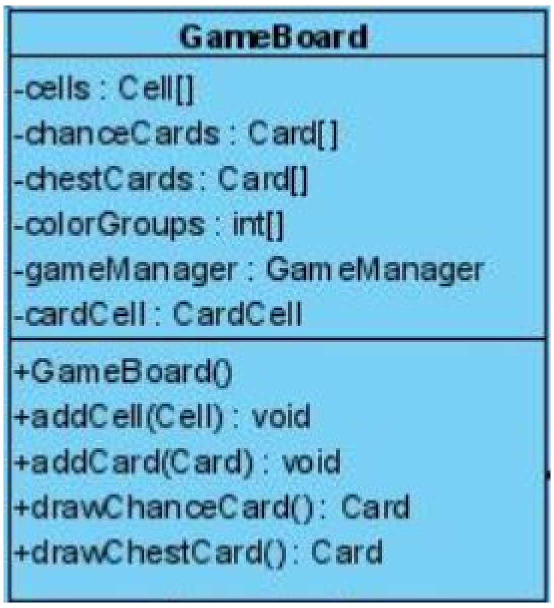


Figure 4.4.3.11: Player Class

This class represents the game board of the Monopoly Game.

Attributes:

- **private Cell[] cells:** This attribute contains a list of cells for every cell in the game board.

- **private Card[] chanceCards:** This attribute contains a list of chance cards for every chance card in the game board deck.

- **private Card[] chestCards:** This attribute contains a list of cards for every chest card in the game board deck.

- **private int[] colorGroups:** This attribute holds an integer list for the color groups.

- **private GameManager gameManager:** This attribute holds a reference to GameManager.

- **private CardCell cardCell:** This attribute holds a reference to CardCell.

Methods

- **public void AddCell(Cell):** This method will be called when the initialization of the game board.

- **public void AddCard():** This method will be invoked for adding the cards to the decks of the game board.

- **public Card DrawChanceCard():** This method picks a random chance card from the chance card deck.

- **Public Card DrawChestCard():** This method picks a random chest card from the chest card deck.

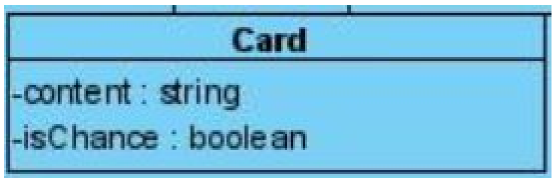


Figure 4.4.3.12: Card Class

Attributes

- **private string content:** This attribute holds the card content in a string.

- **private bool isChance:** This attribute holds a boolean for checking if it is a chance card or not

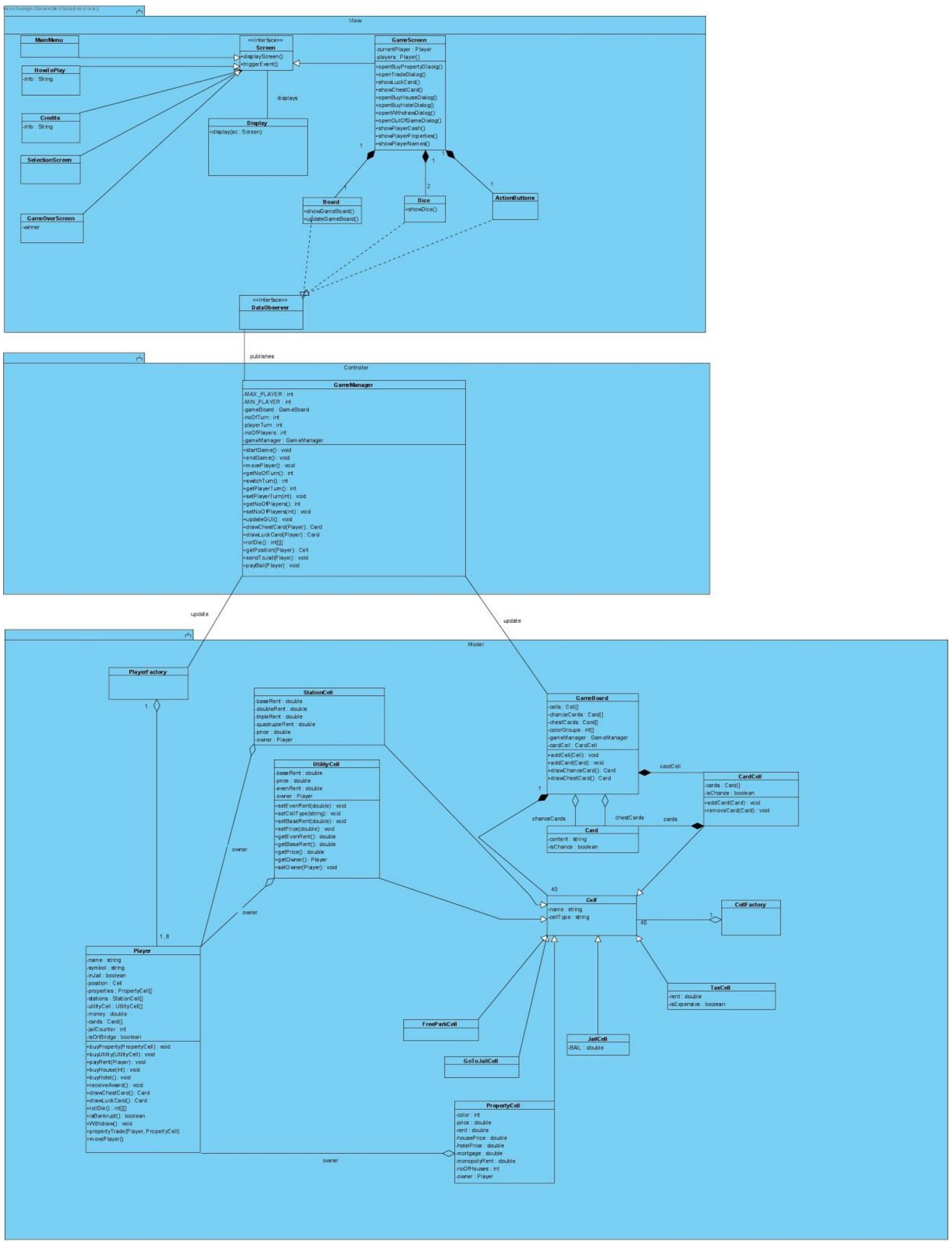


Figure 4.4.3.13 : Big picture of class diagram with subsystems.