CS 319 - Object-Oriented Software Engineering

Fall 2020

Project Design Report Iteration 2

Monopoly

Group 3D

Cansu Moran - 21803665

Elif Gamze Güliter - 21802870

Melisa Taşpınar - 21803668

Öykü Irmak Hatipoğlu - 21802791

Yiğit Gürses - 21702746

Table Of Contents

1. Introduction

1.1 Purpose of the system

The purpose of our product is to provide an environment where people can create unique monopoly boards, share them, and play with each other. We plan to create a simple GUI that allows people without technical experience to design and create boards as they wish and play on them with their friends. The board configurations will be stored in single files which will allow easy sharing.

1.2 Design goals

1.2.1 Trade Offs

1.2.1.1 Development Time vs Performance:

Since monopoly is not a computationally costly game to run on a computer, we did not think performance would be a major issue, especially in modern computers. For this reason, we did not pick a language like C++ which is optimal for writing efficient code, but instead will use Java. We will not be spending too much resources on optimizing our code as long as the game runs within the boundaries of our performance criteria. This will allow us to develop faster.

1.2.1.2 Functionality vs Robustness:

In our particular case, adding a new functionality to the board editor can have big implications for robustness. All the editable features interact with each other and can create new corner cases. These ambiguous or conflicting corner cases might cause the game to crash or behave unexpectedly during a game session. To prevent this, we will have to limit the functionality of the board editor in some manner. We will be checking each edit to see if it can lead to ambiguous

cases and revert such edits with a warning message. It is very hard to think of all possible cases, therefore we will also need to limit the number and complexity of editable features. Otherwise it might be impossible for us to guarantee robustness in the long term.

1.2.1.3 Functionality vs Usability:

To make the editing process easier for players, we decided to limit the variety of edits a player can make. For example, we have different categories of squares which the user can choose from. Regarding the square chosen, the editable features will be different. For example, the user cannot edit the price of a joker square as joker squares do not usually have prices in the original Monopoly. If a user is given the power to change every detail in the game, it would take the user a long time to fully customize the game and the user probably will not be able to finish editing in one sitting. Therefore, by limiting the editing power of the user and thus the functionality of the editing option, we increase the usability of the game.

1.2.2 Criteria

1.2.2.1 Performance:

Monopoly is not a computationally costly game to run on a computer. The users are not negatively affected by low FPS either, as most of the screen is static. Also, our game will run on one computer and we will not need to deal with server connections and latency. All these should make it quite easy to achieve seamless gameplay in terms of lags and FPS. However, we still should not go overboard with inefficient code and features that require lots of resources. Each user input should be processed in less than 0.3 seconds and the system should be ready to receive new input. The sections that include non-static graphics such as the lucky wheel should have an FPS over 10.

1.2.2.2 Robustness:

Since we will be allowing users to modify the boards in many different ways, there will be lots of corner cases that would not have been an issue on the regular Monopoly boards. We need to make sure the system can gracefully handle all these cases. That is, the system should not crash during a game session no matter what the state of the board is or what the user input is. The system should also behave as expected (by the rules in the board configurations) in all

cases. This means we might need to deal with some conflicting modifications during the board editing stage by not allowing the user to create such situations. That is, within the boundaries of our editing system, the user should not be able to create a board that allows ambiguous or conflicting states to arise.

1.2.2.3 Portability:

We chose to write the game in Java and to make sure it can run on machines with up to date JVM. It is important to increase the range of people that can access our product. However, other than this, we had to consider the portability of boards. That is, we want a file system that allows users to drag and drop a “board configurations file” they got from a friend into their “boards” folder. Then, the game should be able to see this board and let the user modify it or play on it. To achieve this, we will need to find a way to store these configurations in a portable format that also works with different versions of the game.

1.2.2.4 Usability / User-friendliness:

The main purpose of our product is to allow people without technical experience or programming knowledge to create different boards and share them with friends. This means it is especially important that our system is user-friendly. A user that knows how to play regular monopoly should be able to get used to our GUI and intuitively play the default game mode. During a game, the board should display necessary information in a friendly and digestible manner. The players should be able to easily get used to new boards and understand the current state of the board without tracking what happened in the previous rounds. The board editor GUI should also be intuitive. When the user clicks on a board piece, all the editable configurations of that piece should be displayed. All these configurations should be modifiable through button clicks and in some rare cases text fields.

2. High-level software architecture

2.1 Subsystem decomposition

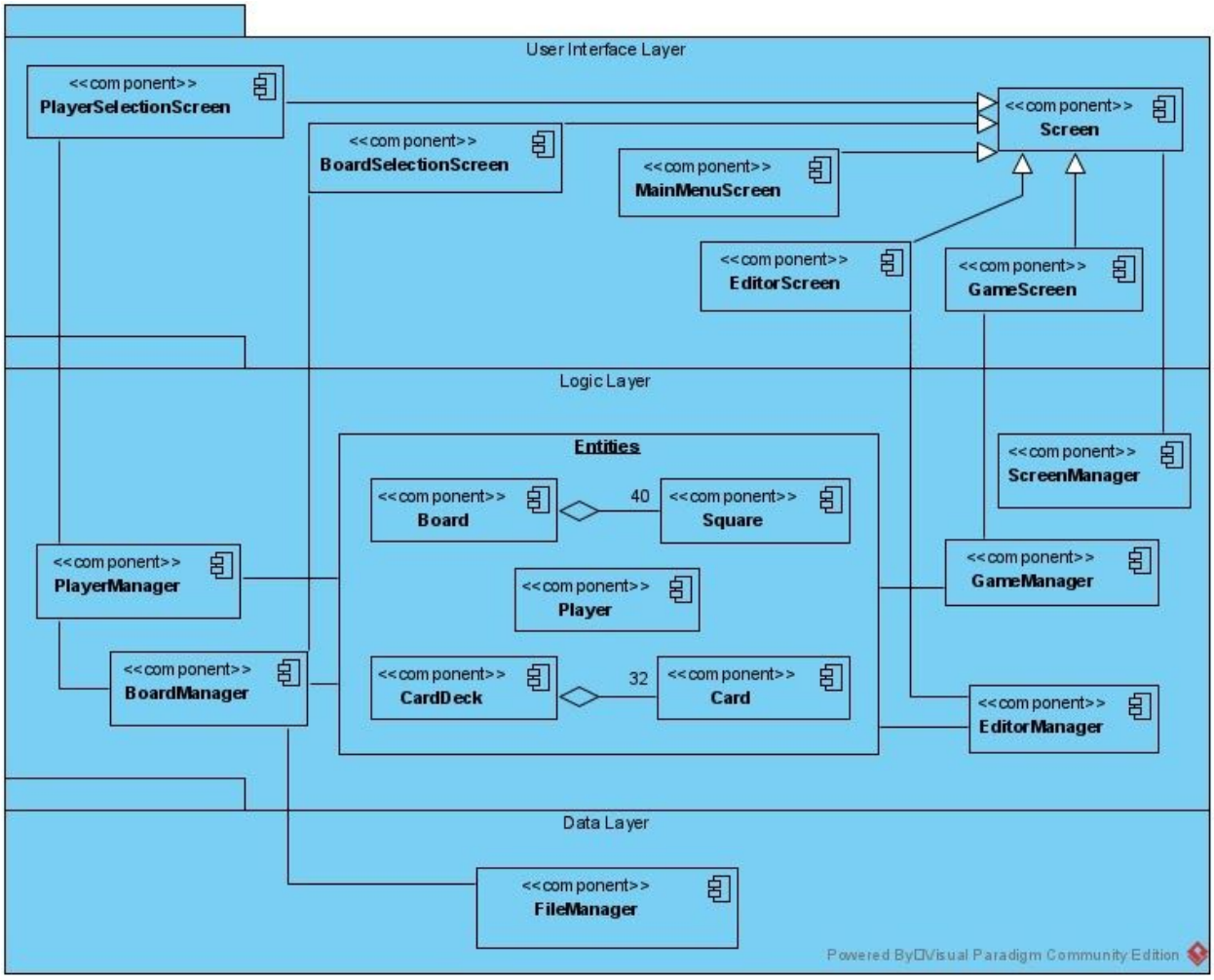

Figure 1. Subsystem Decomposition

While creating our subsystem decomposition (and also while designing the objects in the first place), we tried to apply the 3-layer architectural style in which we have a hierarchy of 3 layers called user interface (UI), logic and data layer. We chose to implement the 3-layer architectural style because it provides great maintainability and flexibility, while also allowing us to separate data, UI and logic components into different subsystems and reduces the decoupling between them. The Systems are decomposed in a way that objects that have similar functionalities are

grouped together while objects that have different functionalities are in different subsystems. As a result, we obtained three layers as follows:

User Interface Layer: This layer contains the components that have functionalities related to presentation. These components are responsible for taking user input and passing the necessary information to the logic layer, and for displaying the User Interface on the screen and updating the screen based on the current state of the logic layer components. One can also say changes in logic layer components trigger an update in related user interface components.

Logic Layer: This layer contains the components that have functionalities related to applications. They take user input from the User Interface layer and process it. After processing the input, they decide which pieces of the game state will be updated and what values they will take. The components of this layer call the functions of the Data layer based on user input.

Data Layer: This layer contains the component that has functionalities related to data storage and maintenance, namely the FileManager. Its responsibility is to read and write data on the local file system based on the inputs it gets from the logic layer. Here, the Logic layer calls the methods of the FileManager.

This decomposition seemed applicable as it allows us to extend or modify our design during development. For example after we have a working game, we might decide to modify the User Interface layer with different components. Let's say we will no longer have a main menu and we also reworked the UI designs. We will be able to apply such changes without touching the Data and Logic layers as long as the new components respect the previous interfaces. The same can be said for all layers. This will lead to agile development and also let us divide the work much easier. All in all, the 3-layer architectural style ensures our system is more maintainable and flexible.

2.2 Hardware/software mapping

In our program, we will be using JavaFX libraries. Since our program will be a Java application, the host machine needs to have a Java Runtime Environment for the program to run. For JavaFX libraries, the machine should have JRE installed. Our program won’t require any internet connection. We will be using a filesystem to store the data, therefore the host machine

should have available memory for the game. Using the JSON files that contain the board's information on the filesystem, the squares on the board will be implemented.

2.3 Persistent data management

Our version of Monopoly, Monopoly Worlds does not require portable data across multiple platforms and there is only one platform to access the data. Furthermore, the game will be a single user game on a local computer. Therefore, the data will be saved in a file system on the host machine's hard drive rather than a database. The board data that will be stored will be on JSON format.

The only persistent data will be the game boards which are edited and saved by the players. Therefore only properties of edited game boards will be stored in the file system. The player will have two different saving options for boards. After making the changes on the editor, the player can either save the board as a new board, in which case the template will remain unchanged and all the changes will be visible on the new board, or they can save the changes on top of the template which would overwrite the previous information about that board. The user won't be able to access the previous version of the board if they save the changes on top of the board. If the board is saved as a new board, a new board file will be created in the host machine's memory.

Data about gameplay such as the scores, winner, losers, places owned will not be saved in the file system. The game does not require any player data and login info thus no data will be saved in the file system about the players such as money, names and places bought . These data, about the game play and player have a lifetime of a single execution and if the user exits the game, all will be lost.

2.4 Access control and security

In the game, there are no other actors who are supposed to sustain the system other than the players. Since the game is not connected to a server and doesn't require internet connection, there won't be any security concerns regarding the game. Since there is only one type of actor using the program, which is the players, access control won't be needed in our system. We do not expect any access control problems such as user validation in order to perform any operation in the game since the game will not require any login requirements or user data and will be a multiplayer game from one computer. The game will not be an online game thus only the users who have access to the local computer will have access to the data saved in the file system.

2.5 Boundary conditions

Initialization:

During the initialization of the game, only the saved boards will be accessed. Since there is no user, score or previous game state information being stored, only the boards will be accessed from the filesystem. The program will be a jar file to increase portability. Since the game will be a jar file, no installations will be required. Program opens with the main menu during every execution. Since there is no login page that requires user information to be given, the user will be directly taken to the main page, where they can select whether they want to edit a board or play the game.

Termination:

The player can exit the program by clicking the shut down button of the window. If the escape key is pressed, an ingame menu opens, which also gives the option to exit the game. The escape key is only disabled on the main menu screen as the menu is already open. If the player exits the gameplay, whether by choosing to exit from the ingame menu or by shutting down the program, all the game data will be lost. In other words, once the user exits the game, they can't go back and continue their previous game. All the information stored during gameplay (the amount of money players have, which properties are bought on the board etc.) are only stored during the gameplay and they are cleaned up once the user exits the game.

On the other hand, edited boards are stored in a filesystem. However, if a user edits a board but doesn't save it before exiting the editor, all the changes are discarded.

Failure:

The game only reads from a file to access saved boards. Hence, if there is a failure during the file read, the boards will fail to load. If there is a failure during the gameplay that causes the game to shut down, all the gameplay data are lost, meaning the user can't access the game that was being played. The board itself (the properties of the squares) are not lost if the failure happens during the gameplay, as the board has to be previously saved for the user to be able to play it. However, if the failure happens during the execution of the editor, all the unsaved changes on the board are lost. There is no autosave mechanism for the editor. All the changes done after the last save are lost if a failure happens.

3. Architectural Style and Subsystem Services

3.1 User Interface Layer

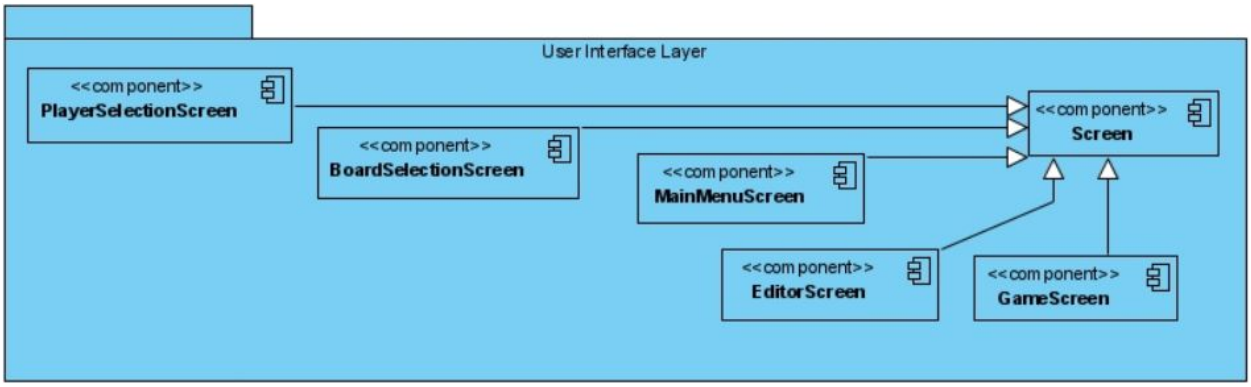

Figure 2. The User Interface Layer

This layer is responsible for two things:
1. Drawing the User Interface and related graphics to the screen based on current game state
2. Taking user input and transferring necessary information to the Logic layer components

These tasks are achieved through "Screen" components. All screens must have a draw method that fulfills the responsibility "1" when called. Also, screens themselves have components such as buttons that are attached to listeners. When these listeners receive user inputs, the screen will transfer the necessary information to the related Game Logic component to fulfill responsibility "2".

Screens also interact directly with the ScreenManager. If an input requires change of screens, necessary information is transferred to ScreenManager. Then the ScreenManager terminates the current screen and creates an instance of the requested screen.

3.2 Logic Layer

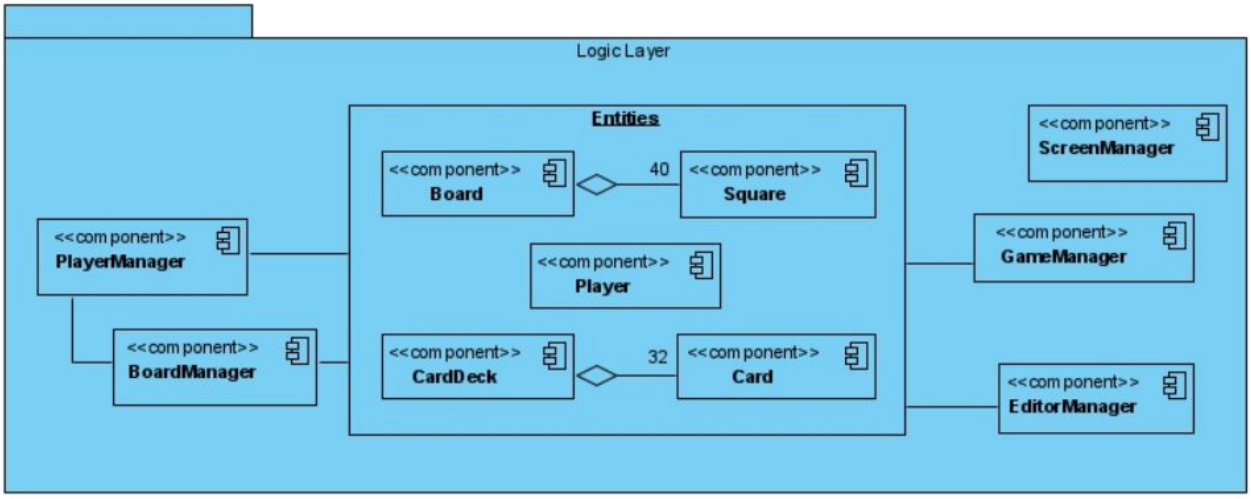

Figure 3. The Game Logic Layer

The Logic layer is mainly responsible for two things:

1. Taking user inputs from User Interface, processing them regarding the current game state and updating the corresponding entities
2. Calling the methods of FileManager from the Data layer when files should be read or written

GameManager and EditorManager components fulfill these responsibilities if the current screen is GameScreen or EditorScreen respectively. They interface with their respective screens to receive user inputs and update the necessary entities.

ScreenManager is responsible for handling the screen changes. It receives inputs from the current active screen. If the input necessitates a screen change, ScreenManager terminates the current screen and instantiates the new requested screen. An example of this happening could be going from the main menu screen to in game screen by clicking the play a game button.

We have grouped the entity components in order to make our subsystem diagram more readable by avoiding a big number of links in between components. These components, such as Player and Board, hold configurations/properties and provide them to components that need them.

The Logic layer frequently updates the game state based on user inputs. Game State is represented by two components:

Players: Player components represent the Players and their state in the game such as their ID, position and balance.

Board: Board component holds the board configurations along with the in-game state of the board. These configurations include the number, name and type of squares and the currency type. The in-game state includes the owners of the properties. EditorManager can modify the board configurations while the GameManager can modify the in-game state. This allows us to use the same Board class type for both the game sessions and board editing.

The updates in these components can trigger changes in the User Interface since User Interface components decide what to draw on screen based on Logic components.

3.3 Data Layer

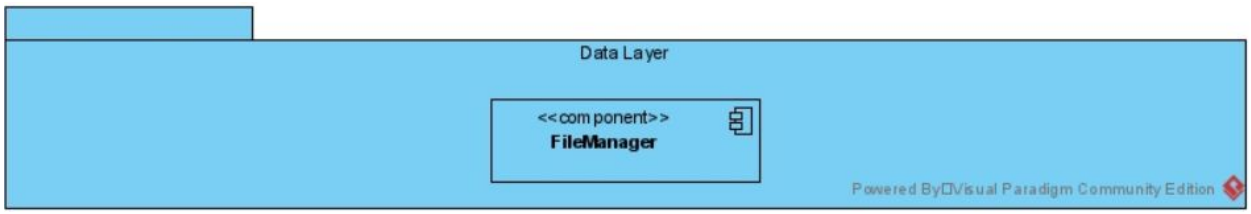

Figure 4. The Data Layer

Data layer consists of the FileManager, which is responsible for reading from and writing to the local file system. For instance, when the GameManager or EditorManager is initializing, they will need a new Board object. FileManager will read the board configurations from a file, parse it and give it to the Logic layer so that the corresponding Board object can be instantiated. Also, when the user decides to save a board, it will be able to write the configurations of the Board that EditorManager holds to the local file system for later use.

4. Low-level design

4.1 Design Patterns, Design Decisions and Object Design Trade-Offs

We have a sub-section called trade-offs (1.2.1) in which we discuss some design trade-offs. In this section, we will explain certain design patterns included in our program and the trade-offs of these design choices.

In our design, we have decided to use three different design patterns: Singleton, façade and factory. Even though we have chosen these design patterns due to their advantages, they still have certain disadvantages. The design choices and more detailed explanation of the chosen design patterns can be found below.

4.1.1 Singleton Design Pattern

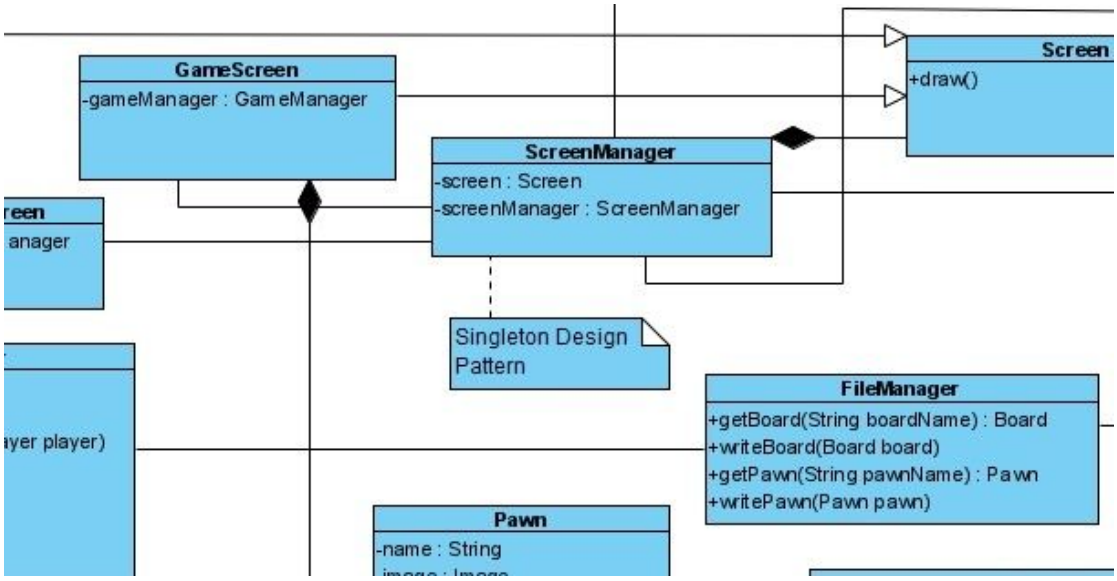

Figure 5. Singleton pattern as seen on ScreenManager

We have decided to use Singleton Design Pattern in the implementation of the ScreenManager to ensure that only one instance of this object is used. We have many different Screen objects

that have a reference to ScreenManager. Since there should only be one ScreenManager that manages these Screen objects, we decided to use Singleton so that all Screen objects refer to the same ScreenManager instance.

We didn't want to have a static class to ensure that there was only one instance of ScreenManager, as we wanted to add certain management implementations inside the manager. Therefore the Singleton pattern seemed suitable. While this design choice was done in order to avoid any reference problems that might occur in the mentioned classes, it still had its disadvantages. Using the Singleton pattern made testing harder, as it makes isolation of classes harder as it increases dependencies and coupling between classes.

4.1.2 Façade Design Pattern

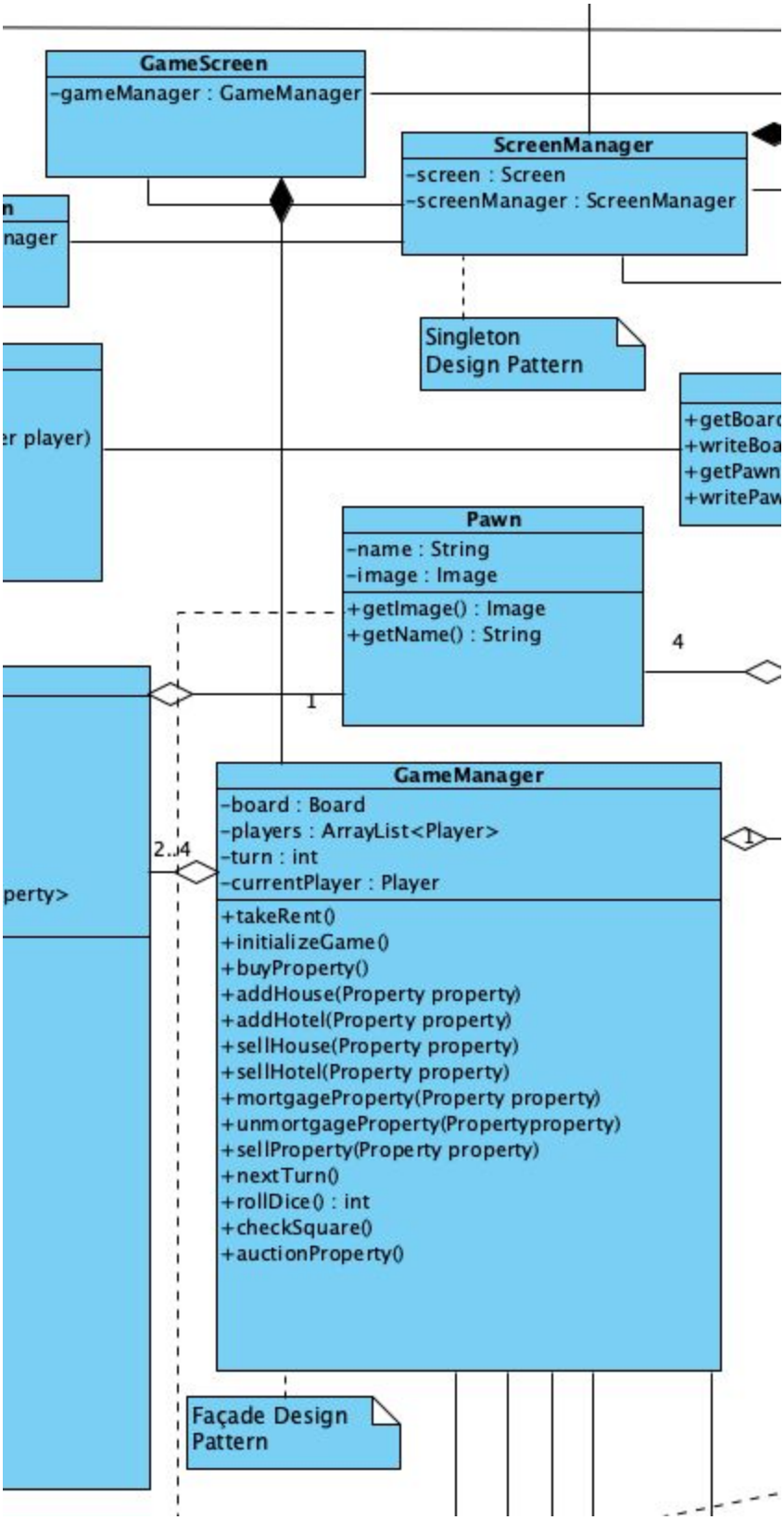

Figure 6. Façade design pattern as can be seen from GameManager and GameScreen

In the implementation of GameManager, the Façade Design pattern is used. GameManager supplies certain functions to GameScreen, without GameScreen knowing the implementation behind it. GameManager manages the board, squares on board, action of the squares and players. It keeps track of the game and updates the game. For example, GameScreen only calls BuyProperty() function from GameManager when the Buy button is clicked, without knowing that in the background GameManager is getting the square, getting the currentPlayer, updating Player balance, updating Square color and property owner. In our implementation, GameManager acts as the façade, controlling Board, Player, Property, Joker, ChanceandCommunityChest classes, whereas GameScreen acts as the client.

By using Façade Design Pattern, we were able to isolate GameScreen from all the other classes involved with the gameplay. With this design pattern, GameScreen can only interact with GameManager to make the necessary updates to the screen. In case anything changes in Board, Player, Property, Joker or ChanceandCommunityChest, the GameScreen doesn't need to be updated, only changes that need to be done can be handled in GameManager, which creates a more reliable coding environment. However, the Façade Design Pattern has some disadvantages as well. Since the user inputs are taken from GameScreen, the process of implementing a game action becomes a longer process. For example if a player clicks the buy button, instead of GameScreen accessing Property directly, according to the input the corresponding function in GameManager needs to be called. Although the process becomes longer, it isolates GameScreen to be only responsible from taking input and updating screen view, instead of loading it with many unorganized lines of code about game logic. It also prevents any undesired changes happening in the game, as everything is controlled by GameManager.

4.1.3 Factory Design Pattern

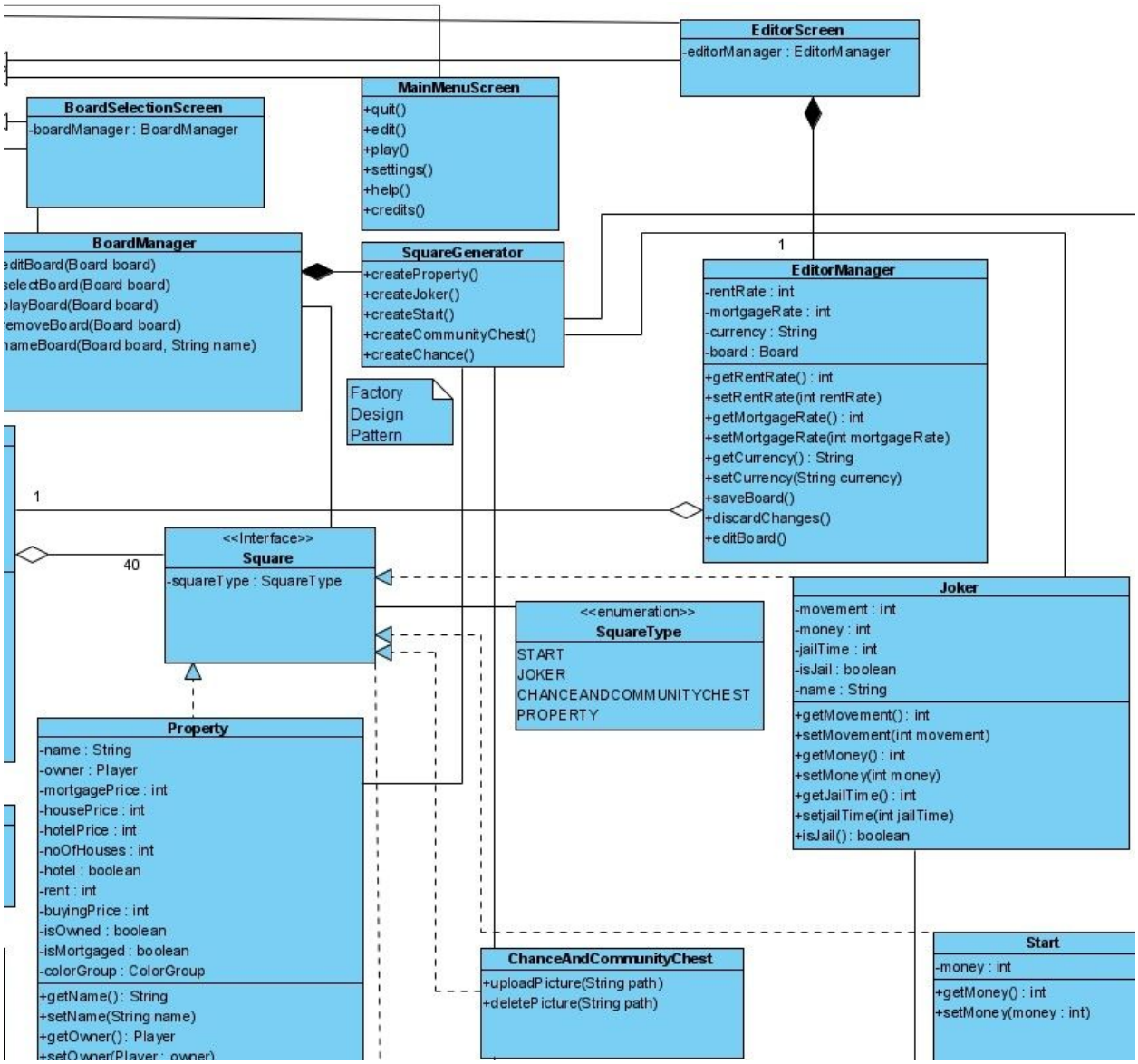

Figure 7. Factory Design Pattern as can be seen from SquareGenerator, Start, Property, Joker, ChanceAndCommunityChest, BoardManager classes and Square interface

Factory Design Pattern is used in the creation of squares according to the stored board information. SquareGenerator class acts as the factory, whereas BoardManager is the client that asks SquareGenerator to create a square. Square interface provides a unified interface, that all the products, which are Property, ChanceAndCommunityChest, Joker and Start can be referred together.

This design pattern was chosen because the board in our implementation is made up of different types of Square products. By using Factory Design Pattern, BoardManager doesn't

need to know how the squares are created, it only uses a common interface to refer to square objects created. So the creation of squares is abstracted, using the SquareGenerator. However, there are certain pitfalls to this design pattern. To implement this pattern, we had to add another class, which is the SquareGenerator that acts as the factory. Since the squares are generated in the SquareGenerator, the BoardManager is unaware of the implementation behind the squares. Though this might be a disadvantage in some scenarios, for our case, this was an advantage as we didn't need BoardManager to know the implementation behind squares.

4.2 Final object design

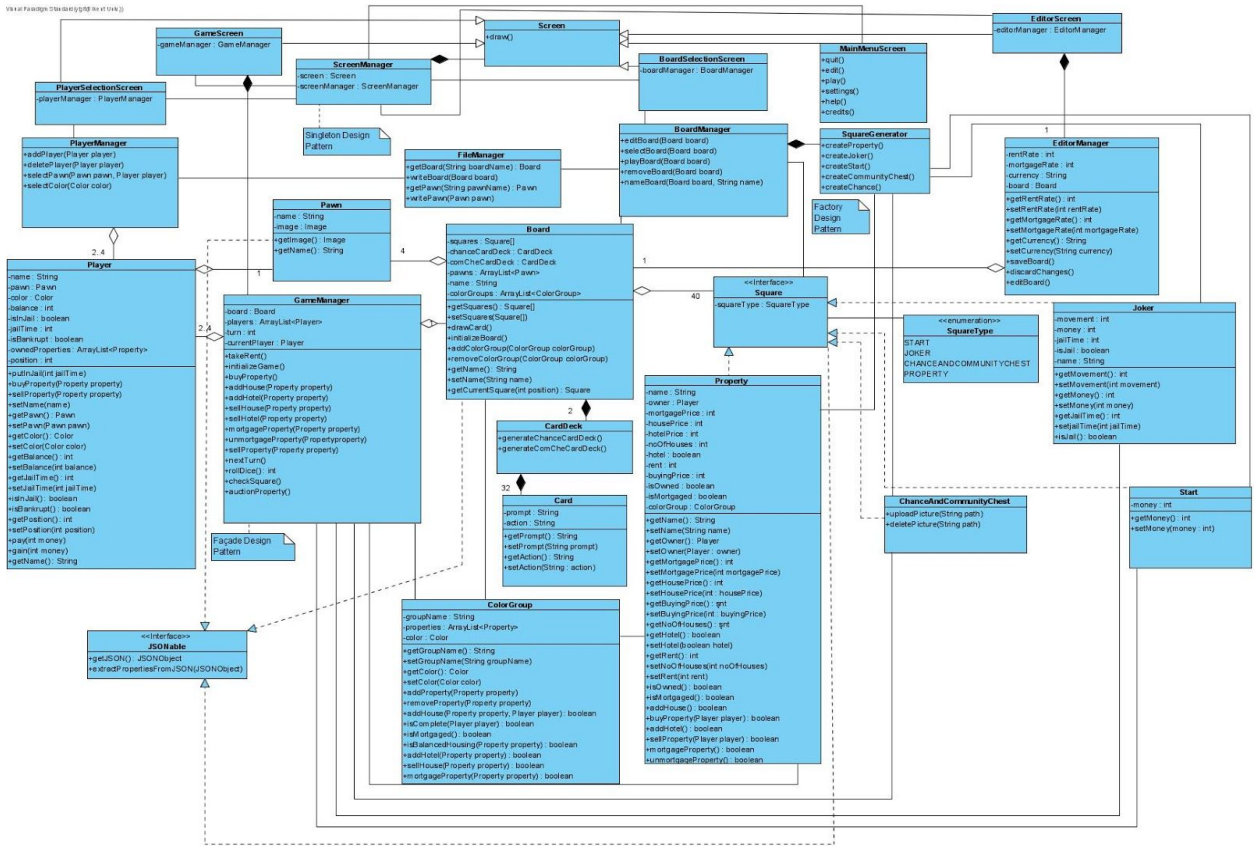

Figure 8. Complete object design diagram

For readability purposes, zoomed versions of the object design diagram can be seen below.

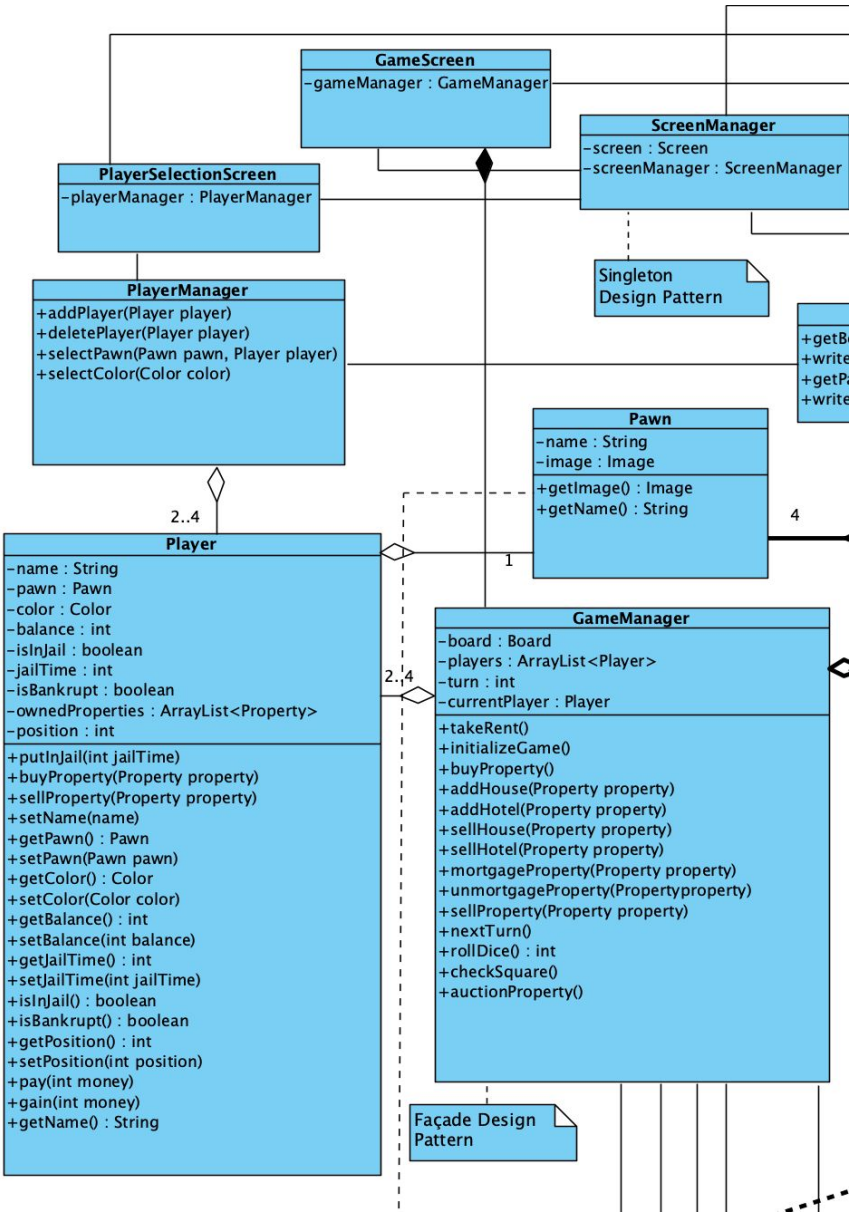

Figure 9. Zoomed object design diagram that focus on Player and GameManager related classes

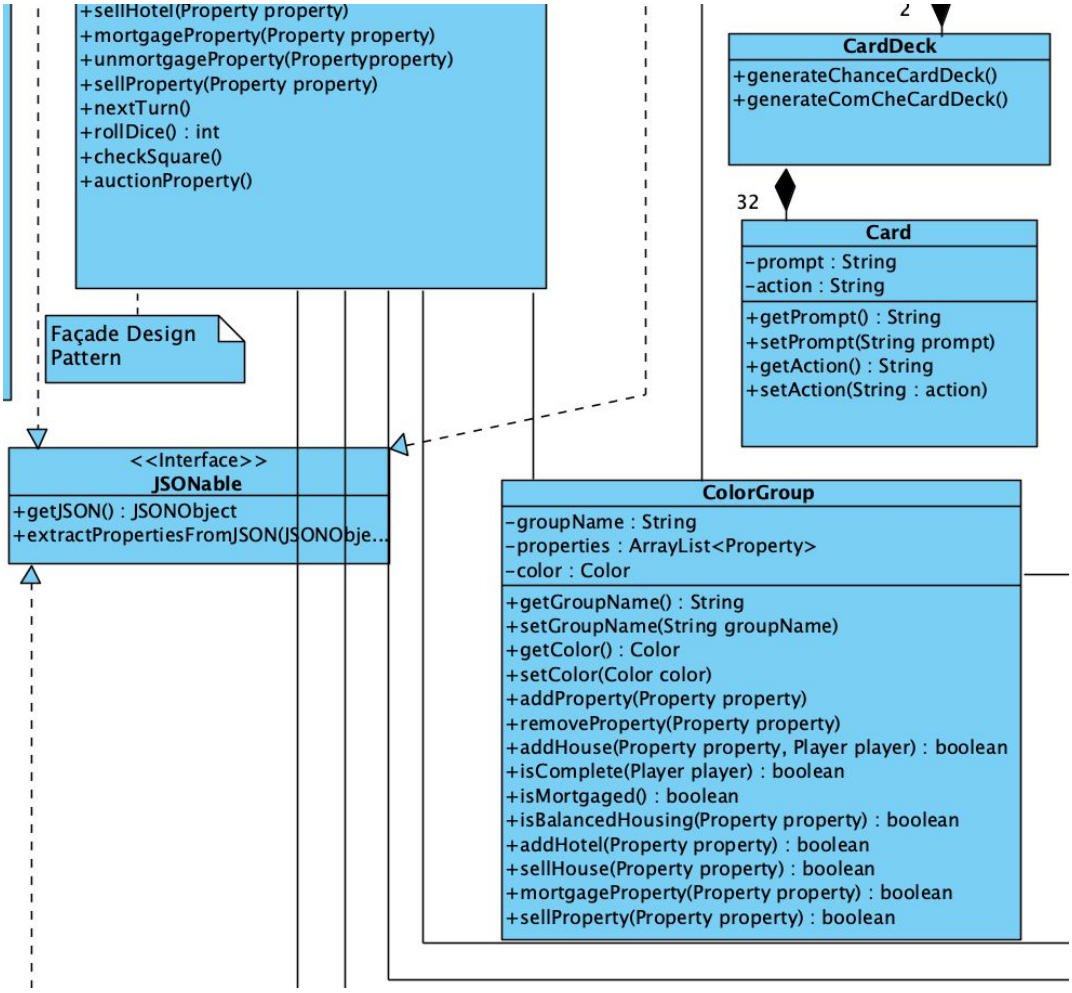

Figure 10. Zoomed object design diagram that focus on JSONable interface, and ColorGroup class

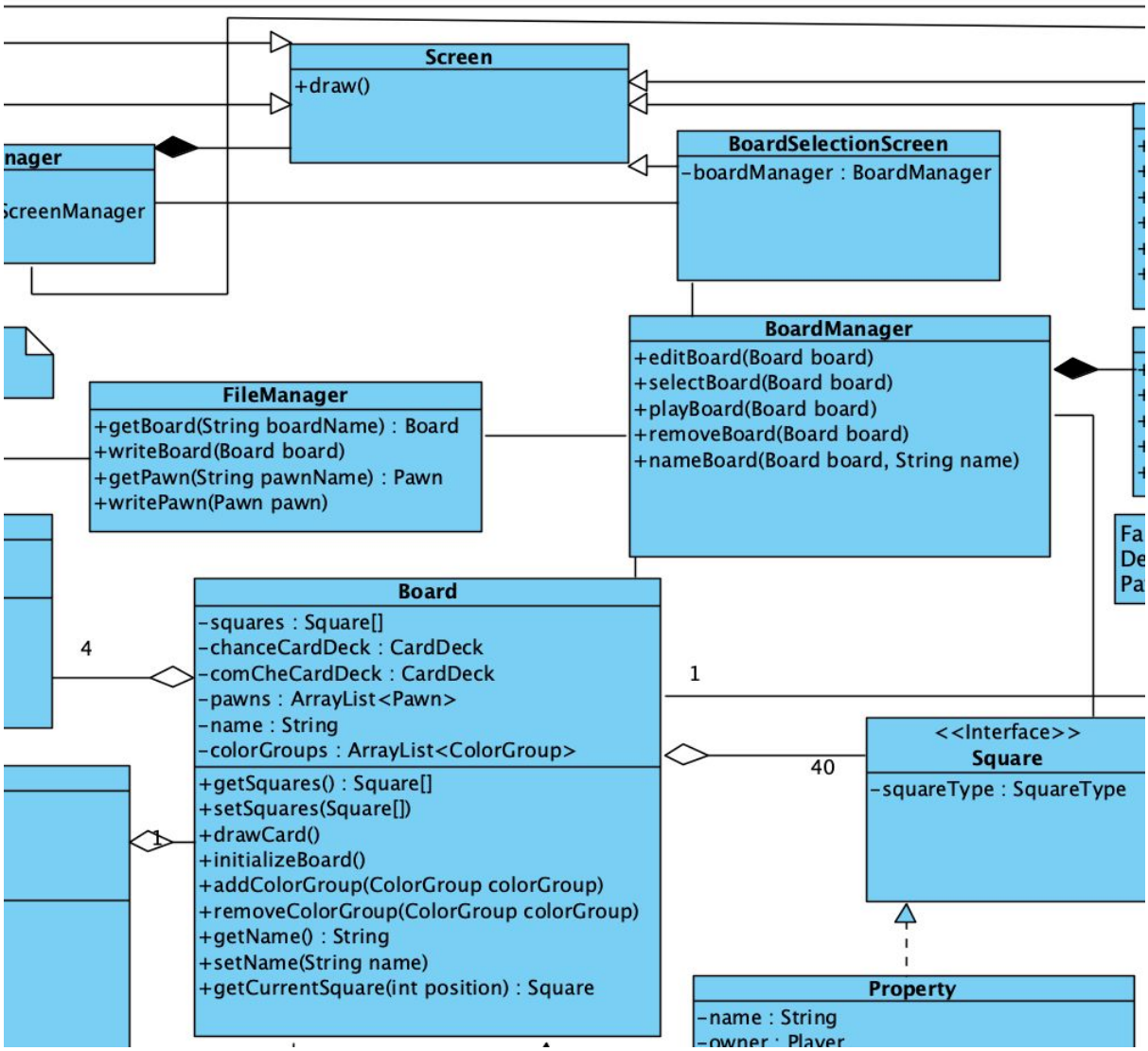

Figure 11. Zoomed object design diagram that focus on Board related classes and FileManager class

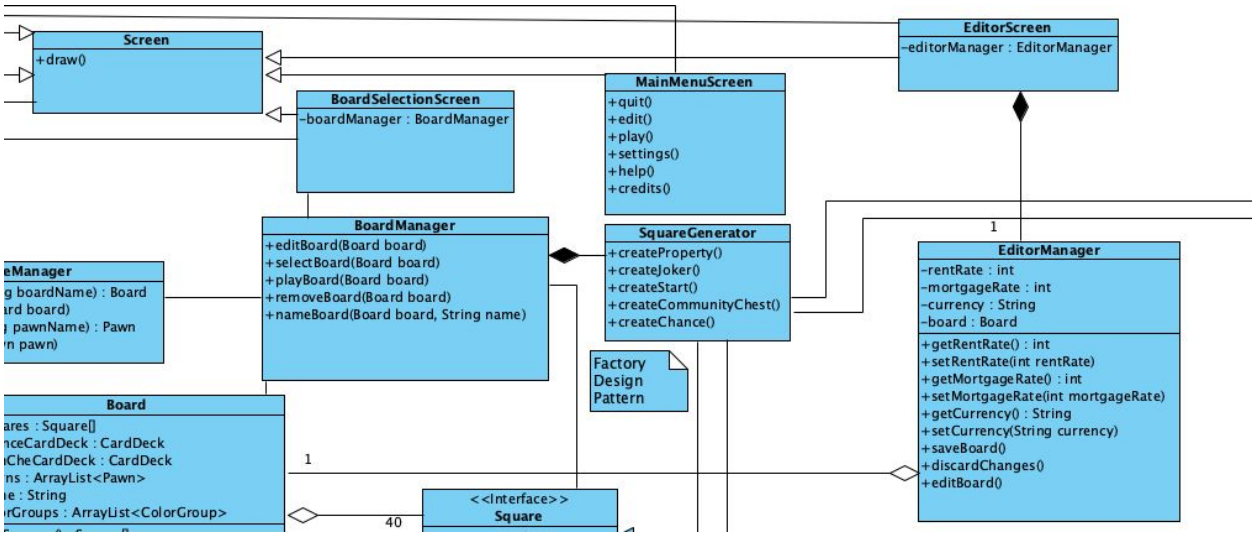

Figure 12. Zoomed object design that focus on SquareGenerator, MainMenuScreen, EditorScreen, and EditorManager classes

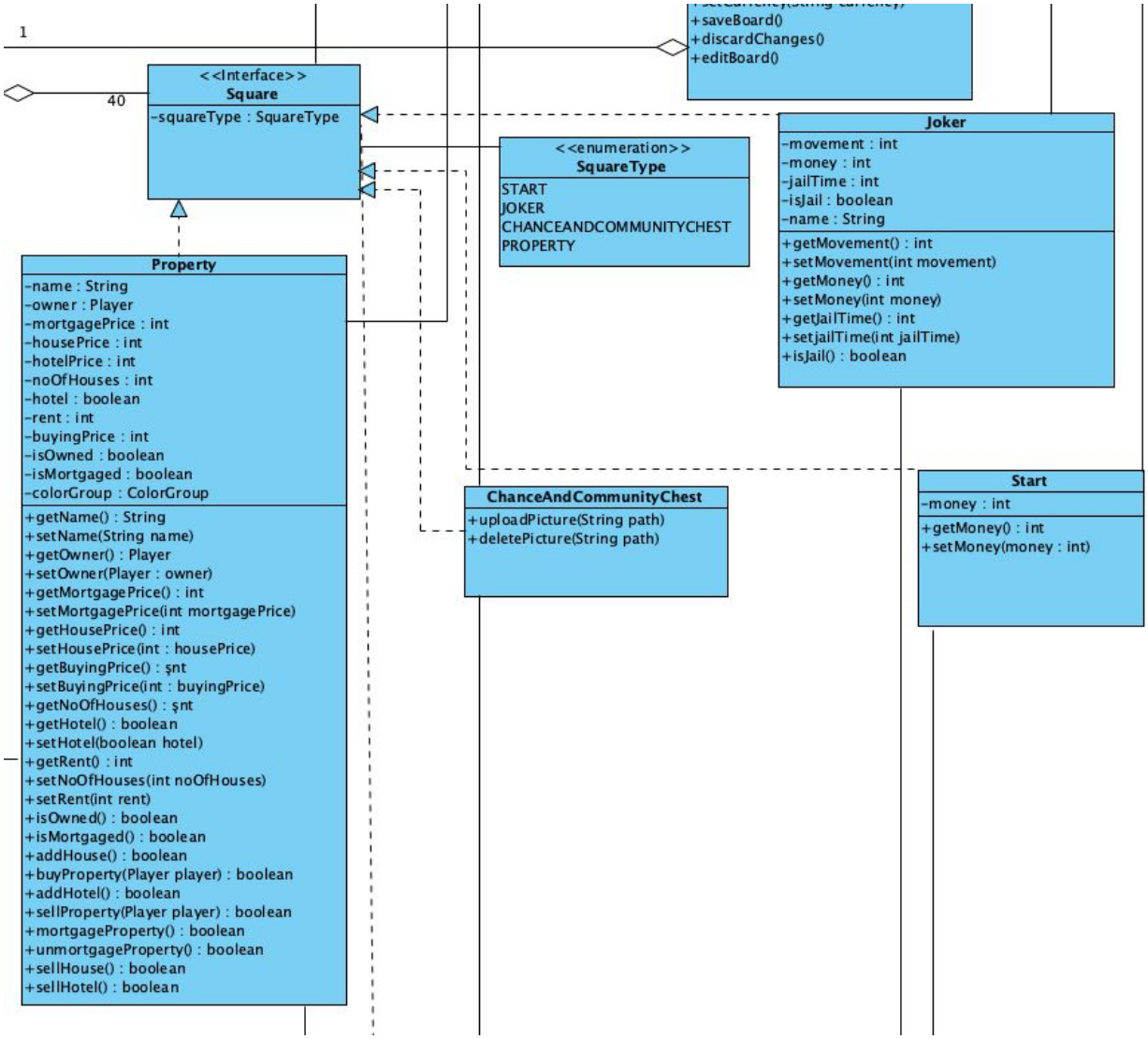

Figure 13. Zoomed object design diagram that focus on Property, Start, Joker, and ChanceAndCommunityChest classes, Square interface, and SquareType enumeration

4.3 Packages

4.3.1 External Packages

Javafx.application: This package is used to configure a JavaFX application.
Javafx.stage: This package is used for the window layer of the JavaFX user interface container.
Javafx.scene: This package is used for the contents layer of the JavaFX user interface container.
Javafx.event: This package is used for the event handling of the JavaFX application.
Javafx.fxml: This package is used to load the FXML file and to connect and create the components on the file.
org.json: This package is used to read and write the board configurations for the file system in JSON format

4.3.2 Internal Packages

Some packages were introduced by the developers to organize the classes and create a neater project.
Squares Package :

- Squares include ChanceCommunityChest, ColorGroup, Joker, Property, Square, and Start, SquareGenerator classes and SquareType interface.
- This package contains all the logic related to creating and modifying Square objects.

Screens Package :

- Screen, EditorScreen, GameScreen, BoardSelectionScren, PlayerSelectionScreen, MainMenuScreen belong to this package.
- This package encapsulates the UI related classes.

Cards Package : This package contains the Card and CardDeck classes.

4.4 Class Interfaces

4.4.1 Player Class

Attributes:

- **private String name:** Player' name
- **private Pawn pawn:** Players' pawn
- **private int position :** Player's position on the board.
- **private Color color**: The color associated with the player
- **private int balance**: The amount of money that the player has
- **private boolean isInJail**: Boolean to indicate if the player is in Jail
- **private int jailTime**: The number of turns that the player will stay in the jail
- **private boolean isOut**: Boolean to indicate if the player is out of the game(resigned or bankrupt)
- **private boolean isBankrupt**: Boolean to indicate if the player is bankrupt
- **private ArrayList<Property> ownedProperties :** The list of the properties that the player owns.

Methods:

- **void putInJail(int jailTime)**: Method that puts the player to the jail for "jailTime" turns.
- **void buyProperty(Property property):** Method which is used for buying a property.
- **void sellProperty(Property property)**: Method which is used for selling an owned property.
- **Pawn getPawn()**: Method that returns player's pawn in the game.
- **void setPawn(Pawn pawn):** Method that sets the player's pawn.

- **int getBalance()**: Method that returns the amount of the money player has.
- **void setBalance(int balance):** Method that sets the amount of the money player has.
- **boolean isBankrupted()**: Method that returns “bankrupted” attribute
- **int getJailTime()**: Method that returns the number of the turns that the player has to wait inside the jail, if he/she is sent to the jail.
- **void setJailTime(int jailTime)**: Method that sets the number of the turns that the player has to wait inside the jail, if he/she is sent to the jail.
- **boolean isOut()**: Method that returns true if the player is out of the game(either bankrupt or resigned).
- **boolean isInJail()**: Method that returns true if the player is in Jail.
- **String getName()**: Method that returns “name” attribute
- **void setName(String name)**: Method that sets the player’s name
- **Pawn getPawn()**: Method that returns “pawn” attribute
- **void setPawn(Pawn pawn):** Method that sets the player’s pawn
- **Color getColor()**: Method that returns “color” attribute
- **void setColor(Color color)**: Method that sets the player’s associated color
- **int getMoney():** Method that returns the amount of money that the player has
- **void setMoney(int money)**: Method that sets the amount of money that the player has
- **int getPosition()**: Method that returns the position of the player in the game.
- **void setPosition()**: Method that sets the position of the player according to the dice roll.
- **void pay(int money):** Method that is used for the “payments” in the game such as penalties and rents.
- **void gain(int money):** Method that is used for “gaining money” in the game such as taking rent from another player or gaining money from a square.

4.4.2 Pawn Class

Attributes:

- **private int location**: Attribute that stores the location of the pawn in the game.
- **private Image image :** Attribute that holds the picture associated with the pawn.

Methods:

- **void setPath(String path)**: Method that sets the picture of the pawn using the path.
- **String getPicture()**: Method that returns the picture of the pawn.

4.4.3 PlayerManager Class

Methods:

- **void addPlayer(Player player)**: Method that adds a player to the game. This method will be used at the beginning of the game where players are added.
- **void deletePlayer(Player player)**: Method that deletes the player specified in the parameters from the game.
- **boolean uploadPicture(String path)**: Method that uploads a picture of a player from the local memory.
- **boolean selectColor(Color color):** Method that is used for the setting an associated color to a player.
- **void selectPawn(Pawn pawn):** Method to select the pawn of the player.

4.4.4 GameManager Class

Attributes:

- **private Board board**: The game board .
- **private ArrayList<Player> players**: Players of the game.
- **private int turn**: Integer number which indicates it is which player's turn.
- **private Player currentPlayer :** The player whose turn is the current turn according to the game.

Methods:

- **void takeRent(Player owner, Player payer, int rent)**: Method that takes rent from a player and adds that rent to another player's (owner's) balance attribute.
- **void initializeGame()**: Method that initializes the game at the beginning of the gameplay.
- **void buyProperty(Property property, Player player)** : Method which is used for a player to buy a property.
- **void addHouse(Property property):** Method which is used for adding a house on to a owned property.
- **void addHotel(Property property) :** Method which is used for adding a hotel on to a property.
- **void sellHouse(Property property):** Method which is used for selling a house .
- **void sellHotel(Property property)** : Method which is used for selling a hotel.
- **void mortgageProp(Property property)**: Method used for mortgaging the property.
- **void unmortgageProp(Property property)**: Method used for mortgaging the property.
- **void sellProperty(Property property)**: Method for selling a property.
- **void nextTurn()** : Method for moving to the next player's turn.

- **int rollDice()**: Method which is used for generating a number between 2-12 in order to move player's pawns.
- **void auctionProperty():** Method which is used for selling a property by auction
- **Square getCurrentSquare() :** Method which returns the square that the current player's pawn is currently landed on.
- **void checkSquare():** Method which is used for checking the actions in the current square and implementing them.

4.4.5 CardDeck Class

Methods:

- **void generateChanceCardDeck():** This method generates the chance card deck.
- **void generateComCheCardDeck():** This method generates the community chance card deck.

4.4.6 Card Class

Attributes :

- **private String prompt**: This attribute is the prompt written on a card.
- **private String action**: This attribute is the action of the card.

Methods:

- **String getPrompt():** This method returns the prompt on the card.
- **void setPrompt(String prompt)**: This method changes the prompt of the card to the value on the parameter.
- **String getAction():** This method returns the function of the card.
- **void setAction(String action):** This method sets the action ction of the card.

4.4.7 Board Class

Attributes:

- **private Square[ ] squares:** This attribute is an array of Squares that the board contains.
- **private CardDeck chanceCardDeck**: This attribute is the chance card deck that the board has.
- **private CardDeck comCheCardDeck:** This attribute is the community chest card deck that the board has.
- **private ArrayList<Pawn> pawns** : Attribute that holds a list of the pawns associated with the selected board.
- **private ArrayList<ColorGroup> colorGroups:** Attribute that holds a list of the color groups in the game.
- **private String name :** Attribute that holds the name of the board.

Methods:

- **Square[ ] getSquares()**: This method returns the array of the squares that are included on the board.
- **void setSquares(Square [ ] squares):** This method sets the array that contains the squares included on the board.
- **Card drawCard():** This method draws a card from a card deck
- **void initializeBoard()**: This method initializes the board by preparing the card decks and board for the new game.
- **void addColorGroup(ColorGroup colorGroup)**: The method that is used for adding a new color group to the game.
- **void removeColorGroup(ColorGroup colorGroup):** The method that is used for removing a new color group to the game.

- **String getName()**: The method that returns the name of the board.
- **void setName(String name):** The method that is used for setting the name of the board.
- **Square getCurrentSquare(int position) :** The method that returns the Square which is placed on the position in parameters.

4.4.8 EditorManager Class

Attributes:

- **private int rentRate :** Rent rate of the game.
- **private int mortgageRate :** Mortgage rate of the game.
- **private String currency** : Currency which will be used in the game.
- **private Board board** : Game board.

Methods:

- **int getRentRate()** : Method that returns the rent rate.
- **void setRentRate(int rentRate)** : Method that sets the rent rate.
- **int getMortgageRate()**: Method that returns the mortgage rate.
- **void setMortgageRate(int mortgageRate)** : Method that sets the mortgage rate.
- **String getCurrency():** Method that returns the currency of the game.
- **void setCurrency(String currency)** : Method that sets the currency of the game.
- **boolean saveBoard()** : Method which returns a boolean and saves the changes on the board.
- **boolean discardChanges()** : Method which returns a boolean and discards the changes on the board.
- **void editBoard()** : Method which is used in order to edit the board of the game.

4.4.9 Property Class

Attributes:

- **private String name:** This attribute is the name of the property.
- **private Player owner :** This attribute is the player who owns the property.
- **private Player owner:** This attribute is the Player owner of the property.
- **private int mortgagePrice:** This attribute is the mortgage price of the property.
- **private int housePrice**: This attribute is the price of a house that will be placed on the property.
- **private int hotelPrice**: This attribute is the price of a hotel that will be placed on the property.
- **private ColorGroup colorGroup :** The ColorGroup of the property.
- **private int buyingPrice**: This attribute is the buying price of the property.
- **private int noOfHouses:** This attribute is the number of houses on the property.
- **private boolean hotel**: This attribute says if there is a hotel on the property.
- **private int rent:** This attribute is the rent of the property.
- **private boolean isOwned:** This attribute shows if the property is owned.
- **private boolean isMortgaged:** This attribute shows if the property is mortgaged.

Methods:

- **String getName()**: This method returns the name of the property.
- **void setName(String name):** This method sets the name of the property to the value on parameter..
- **Player getOwner():** This method returns the Player which owns the property.

- **void setOwner(Player player):** This method sets the Player who owns the property to the value on parameter.
- **int getMortgagePrice()**: This method returns the mortgage price of the property.
- **void setMortgagePrice(int mortgagePrice):** This method sets the mortgage price of the property to the value on parameter.
- **int getHousePrice():** This method returns the price of a single house which will be placed on the property.
- **void setHousePrice(int housePrice)**: This method sets the price of the house which will be placed on the property to the value on parameter.
- **int getBuyingPrice():** This method returns the buying price of the property.
- **void setBuyingPrice(buyingPrice):** This method sets the buying price of the property to the value on parameter.
- **int getNoOfHouses()**: This method returns the number of houses on the property.
- **void setNoOfHouses(int noOfHouses):** This method sets the number of houses on the property to the value on parameter.
- **boolean getHotel():** This method returns if there is a hotel on the property.
- **void setHotel(boolean hotel):** This method sets the existence of a hotel on the property.
- **int getRent()**: This method returns the rent of the property.
- **void setRent(int rent)**: This method sets the rent of the property to the value on parameter.
- **boolean isOwned():** This method returns true if the property is owned and returns false otherwise.
- **boolean isMortgaged():** This method returns true if the property is mortgaged and returns false otherwise.

- **boolean addHouse()**: This method is used to add a house to the property. Returns true if the house can be added, returns false otherwise.
- **boolean addHotel():** This method is used to add a hotel to the property. Returns true if the hotel can be added, returns false otherwise.
- **boolean buyProperty(Player player)**: This method is used when a player buys a property that his/her pawn lands on.
- **boolean sellProperty(Player player):** This method is used when a property is sold to a player.
- **boolean mortgageProperty()**: This method is used when a property is wanted to be mortgaged.
- **boolean unmortgageProperty()**: This method is used when a property is wanted to be unmortgaged.

4.4.10 Square Interface

Attributes:

- **private String squareType:** This attribute stores the square type of the square whether it is a property square of a joker square etc.

4.4.11 ColorGroup Class

Attributes:

- **private String groupName**: This attribute is the name of the color group.
- **private ArrayList<Property> properties**: This attribute is the list of properties which belong to the color group.
- **private Color color**: This attribute represents the list of RGB values that represent the color of the color group.

Methods:

- **String getGroupName():** This method returns the name of the color group.
- **void setGroupName(String groupName)**: This method sets the name of the color group to the String given as the parameter.
- **Color getColor():** This method returns the integer arraylist representing the color of the group.
- **void setColor(Color color):** This method changes the color of the group to the value given as the parameter.
- **void addProperty(Property property)**: Method that adds property to the ColorGroup's properties arraylist.
- **void removeProperty(Property property)**: Method that removes a property from the ColorGroup's properties arraylist.
- **boolean addHouse(Property property, Player player):** This method checks whether a player can add a house on a specified property in the ColorGroup by calling the other helper methods explained below.
- **boolean isComplete(Player player):** This method checks whether the player owns all of the properties inside a color group.
- **boolean isMortgaged():** This method checks whether any of the properties in ColorGroup is mortgaged or not.
- **boolean isBalancedHousing(Property property):** This method checks whether there is a balanced housing according to the Monopoly rule book on a property in ColorGroup.
- **boolean addHotel(Property property):** This method checks whether it is possible to build a hotel on to the specified property in the ColorGroup.

- **boolean sellHouse(Property property):** This method checks whether it is possible to sell a house built on the specified property according to the balance rules about selling houses in a ColorGroup.
- **boolean mortgageProperty(property property):** This method checks whether it is possible to mortgage a property according to the balance rules about mortgaging properties in a ColorGroup.
- **boolean sellProperty(Property property):** This method checks whether it is possible to sell a property according to the balance rules about selling properties in a ColorGroup.

4.4.12 BoardManager Class

Methods:

- **void editBoard(Board board)**: This method opens the edit mode for the board given as the parameter.
- **void selectBoard(Board board):** This method selects the board given as a parameter which is taken from the board selection step.
- **void playBoard(Board board)**: This method starts the game with the board given as the parameter.
- **void removeBoard(Board board):** This method removes the board given as the parameter from the file system.
- **void nameBoard(Board board, String name):** This method changes/sets the name of the board, given as one of the parameters to the name given as the other parameter.

4.4.13 FileManager Class

Methods :

- **Board getBoard(String boardName):** The method which finds and returns the saved board with the given name from the local file system.
- **void writeBoard(Board board) :** The method which saves the given board to the local file system.
- **Pawn getPawn(String pawnName):** The method which finds and returns the saved pawn with the given name from the local file system.
- **void writePawn(Pawn pawn) :** The method which saves the given pawn to the local file system.

4.4.14 GameScreen Class

Attributes:

- **private GameManager GameManager**: This attribute is the Game Manager that the game uses to regulate game controls.

4.4.15 ScreenManager Class

Attributes:

- **private Screen screen:** This attribute stores an instance of the Screen.
- **private ScreenManager screenManager:** This attribute is the ScreenManager that is used to regulate screen controls.

4.4.16 Screen Class

Methods:

- **void draw():** This method draws the components of the screen.

4.4.17 ChanceAndCommunityChest Class

Methods:

- **boolean uploadPicture(String path):** This method uploads a picture to put on either a chance card or a community chest card. Returns true if the picture is successfully uploaded.
- **boolean deletePicture(String path):** This method deletes the picture on a chance or community chest card. Returns true if the deletion is successful.

4.4.18 EditorScreen Class

Attributes:

- **private EditorManager editorManager :** This attribute stores an instance of the EditorManager.

4.4.19 MainMenu Class

Methods:

- **void quit():** This method quits the game.
- **void edit():** This method takes the user to edit the board section.
- **void play()**: This method takes the user to play the game section.
- **void settings():** This method takes the user to the settings.
- **void help()**: This method takes the user to the help page.
- **void credits():** This method takes the user to the credits page.

4.4.20 Joker Class

Attributes:

- **private int movement:** Movement(number of squares) that the Joker square requires as a property when a player's pawn lands on it.
- **private int money :** The amount of money that the Joker square takes or gives to the player whose pawn is landed on it.
- **private int jailTime:** The amount of tours to be spent in jail that the Joker Square requires if a player's pawn lands on it.
- **private boolean isJail:** Boolean which is used to indicate whether that Joker square requires suspencion (like jail).
- **private String name:** The name of the Joker square.

Methods:

- **int getMovement() :** Method which returns the amount of squares to move the player's pawn.
- **void setMovement(int movement):** Method which sets the amount of squares to move.
- **int getMoney() :** Method which returns the amount of money to take from or give to the players who lands on the Joker square.
- **void setMoney(int money) :** Method which sets the amount of money to take from or give to the players who lands on the Joker square.
- **int getJailTime() :**Method which returns the amount of tours to which players who land on Joker square have to stay in Jail .

- **void setJailTime(int jailTime):** Method which sets the amount of tours to suspend the players who land on the Jail.
- **boolean isJail()**: Method that returns true if the Joker square is a Jail.

4.4.21 Start Class

Attributes:

- **private int money:** The amount of money which will be given to the players everytime they pass the Start square.

Methods:

- **int getMoney() :** The method which returns the money which will be given to the players everytime they pass the Start square.
- **void setMoney(int money)** : The method which sets the amount money which will be given to the players everytime they pass the Start square.

4.4.22 PlayerSelectionScreen Class

Attributes:

- **private PlayerManager playerManager :** This attribute is the instance of PlayerManager class.

4.4.23 BoardSelectionScreen Class

Attributes **:**

- **private BoardManager boardManager :** This attribute is the instance of BoardManager class.

4.4.24 SquareGenerator Class

Methods:

- **void createProperty() :** Method which is used to create an instance of the Property squares.
- **void createJoker():** Method which is used to create an instance of the Joker squares.
- **void createStart():** Method which is used to create an instance of the Start square.
- **void createChance()**: Method which is used to create an instance of the Chance square.
- **void createCommunityChest():** Method which is used to create an instance of the Community Chest square.

4.4.25 SquareType Enumeration

This enumeration includes SquareTypes START, JOKER, CHANCEANDCOMMUNITYCHEST, and PROPERTY. These types indicate the type of the square.

4.4.26 JSONable Interface

Methods:

- **JSONObject getJSON() :** This method saves the properties of the class inside a JSONobject and returns that JSONObject in order to save it in the local file system.

- **extractPropertiesFromJSON(JSONObject):** This method extracts the properties of a class instance from a JSONObject.

5 Improvement Summary

- In this iteration, we added Design patterns to our object design. The patterns we added are façade, singleton and factory patterns. We discussed the reason why we decided to use these patterns, in addition to the pitfalls of the chosen design patterns.

- In this iteration we also updated our class diagram with new classes and interfaces. We added JSONable and Square interfaces and added new classes such as SquareGenerator, Start and BoardSelectionScreen classes.

- We added new attributes and operations and changed some of the existing ones in the class diagram.

- We updated our subsystem decomposition with a more detailed design.

- In the first iteration, we decided to apply MVC architectural style, but in this iteration we changed our architectural style to the 3-Layer Architectural Style with the concern of more maintainable and flexible design.