CS 319 - Object-Oriented Software Engineering

Fall 2020

Project Design Report

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

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.

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.

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

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.

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.

Portability:

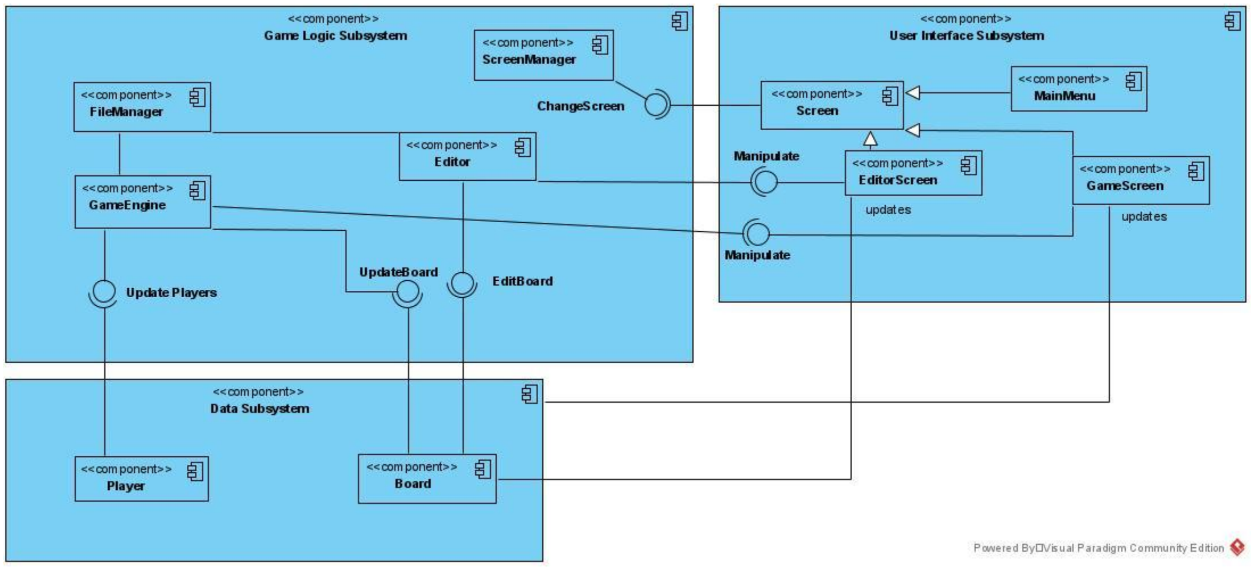
We chose to write the game in Java and to make sure it can run on machines with up to date JVM. It is important to increase the range of people that can access our product. However, other than this, we had to consider the portability of boards. That is, we want a file system that allows users to drag 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.

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



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. While doing this decomposition (and also while designing the objects in the first place), we tried to apply the MVC (Model, View, Controller) pattern. This allows us to separate data, UI and logic components into different subsystems and reduces the decoupling between them. Since all of the programming was done in Java and we did not have any server related components, a simple MVC was more favorable over a three layer approach for now. As a result, we obtained three subsystems as follows:

User Interface Subsystem: This subsystem contains the components that have functionalities similar to View in MVC pattern. These components are responsible for taking user input and passing the necessary information to the Game Logic Subsystem. They are also responsible for displaying the User Interface on the screen and update the screen based on the current state of the Data Subsystem components. One can also say changes in Data Subsystem components trigger an update in related user interface components.

Game Logic Subsystem: This subsystem contains the components that have functionalities similar to Controller in MVC pattern. They take user input from the User Interface Subsystem and process it. After processing the input, they decide which pieces of the game state will be updated and what values they will take. That is, they will update the Data Subsystem components based on user input. The exception is the ScreenManager which directly updates the User Interface by deciding which screen to show next. In some cases, FileManager will read or write data on the local file system.

Data Subsystem: This subsystem contains the components that have functionalities similar to Model in MVC pattern. They hold the current game state and are frequently modified by the Game Logic Subsystem. Changes in Data Subsystem also trigger changes in what the User Interface will display on the screen.

This decomposition allows us to extend or modify our design during development. For example after we have a working game, we might decide to completely rewrite the User Interface Subsystem 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 Game Logic subsystems as long as the new components respect the previous interfaces. The same can be said for all subsystems. This will lead to agile development and also let us divide the work much easier.

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 Java Runtime Environment for program to run. For JavaFX libraries, the machine should have JDK 8 or a newer version installed. In our program, keyboard input will only be used on the editor to edit names and on the game to edit player names. All game controls will be done via mouse. Our program won't require any internet connection. We will be using filesystem to store the data, therefore the host machine should have available memory for the game.

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.

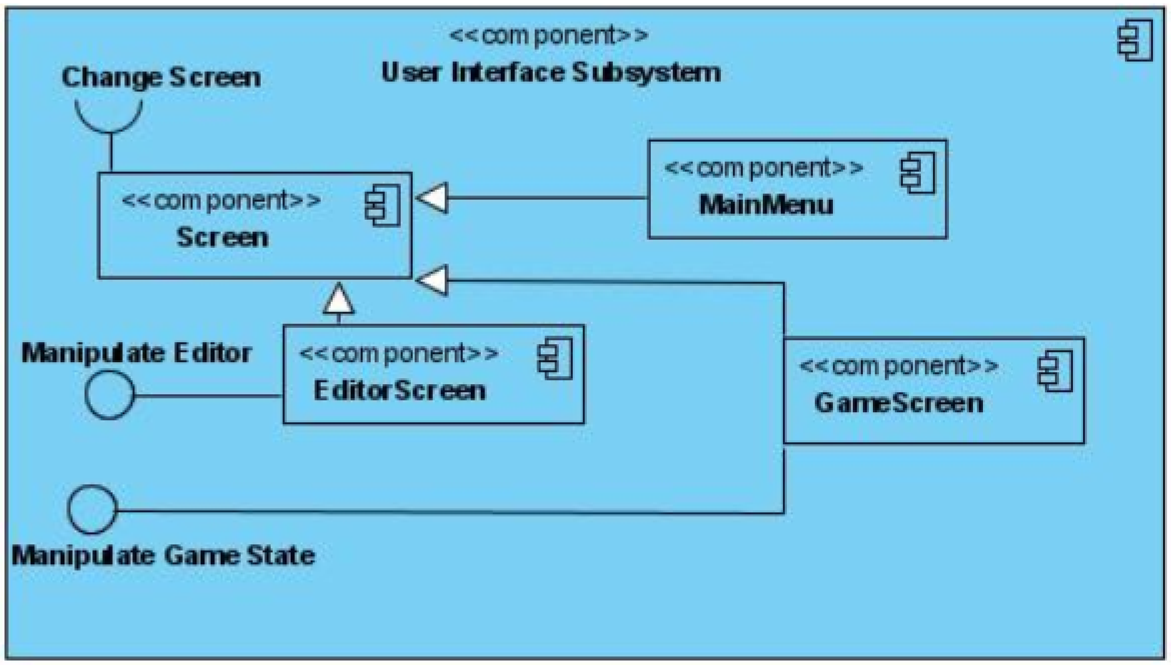
Termination:
The player can exit the program by clicking the shut down button of the window. If the escape key is pressed, an ingame menu opens, which also gives the option to exit the game. The escape key is only disabled on the main menu screen as the menu is already open. If the player exits the gameplay, 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. Subsystem Services

3.1 User Interface Subsystem

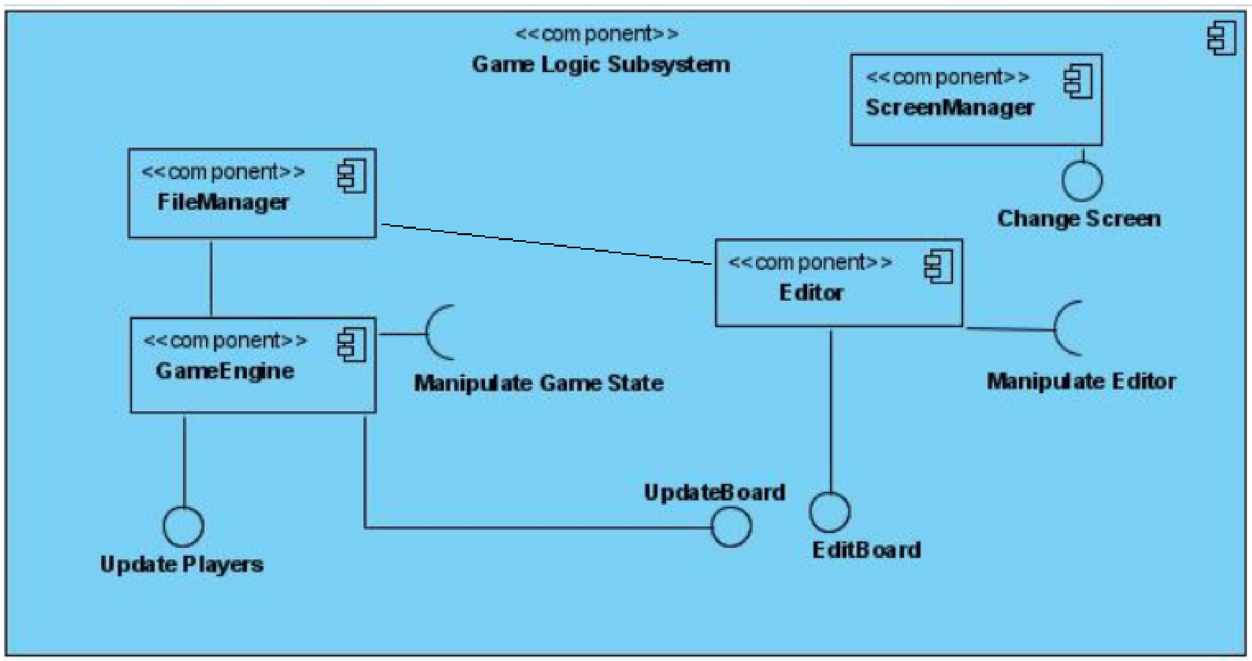


This subsystem is responsible for two things:
1. Drawing the User Interface and related graphics to the screen based on current game state
2. Taking user input and transferring necessary information to the Game Logic components

These tasks are achieved through “Screen” objects. 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 Game Logic Subsystem



The game logic subsystem is mainly responsible for two things:

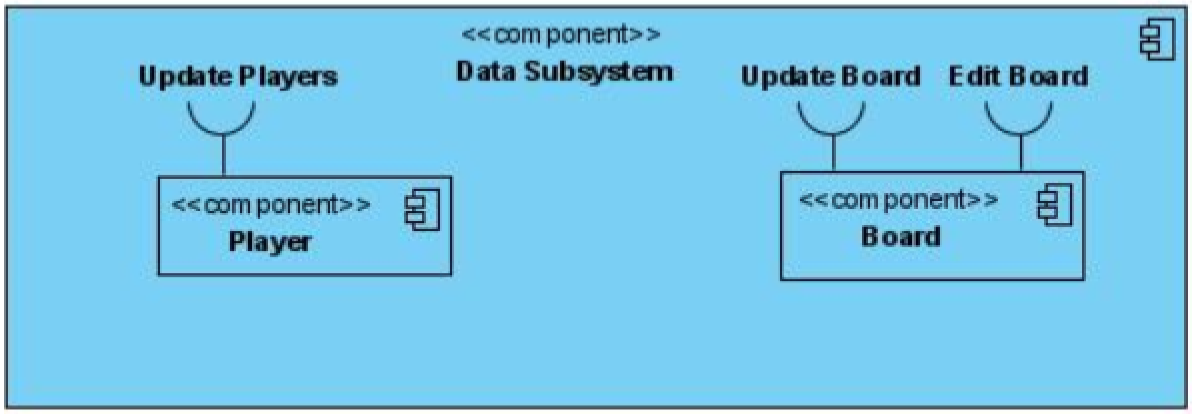
1. Taking user inputs from User Interface and processing them regarding the current game state
2. Updating the Data Subsystem components

GameEngine and Editor 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 data components.

FileManager is responsible for reading from and writing to the local file system. When the GameEngine or Editor is initializing, they will need a new Board object. FileManager will read the board configurations from a file and parse it to instantiate and return this Board object. Also, when the user decides to save a board, it will be able to write the Board object Editor holds to the local file system for later use.

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 main menu screen to in game screen by clicking the play a game button.

3.3 Data Subsystem



Data subsystem consists of objects that represent the game state. These components do not have any other responsibilities than to hold data and provide it to objects that need it. Game logic subsystem 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. These components are associated only with the GameEngine as the Board Editor is not concerned with players' in-game state.

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 tiles and the currency type. The in-game state includes the owners of the properties. Editor can modify the board configurations while the GameEngine can modify the in-game state. This allows us to use the same Board class type for both the game sessions and board editing.

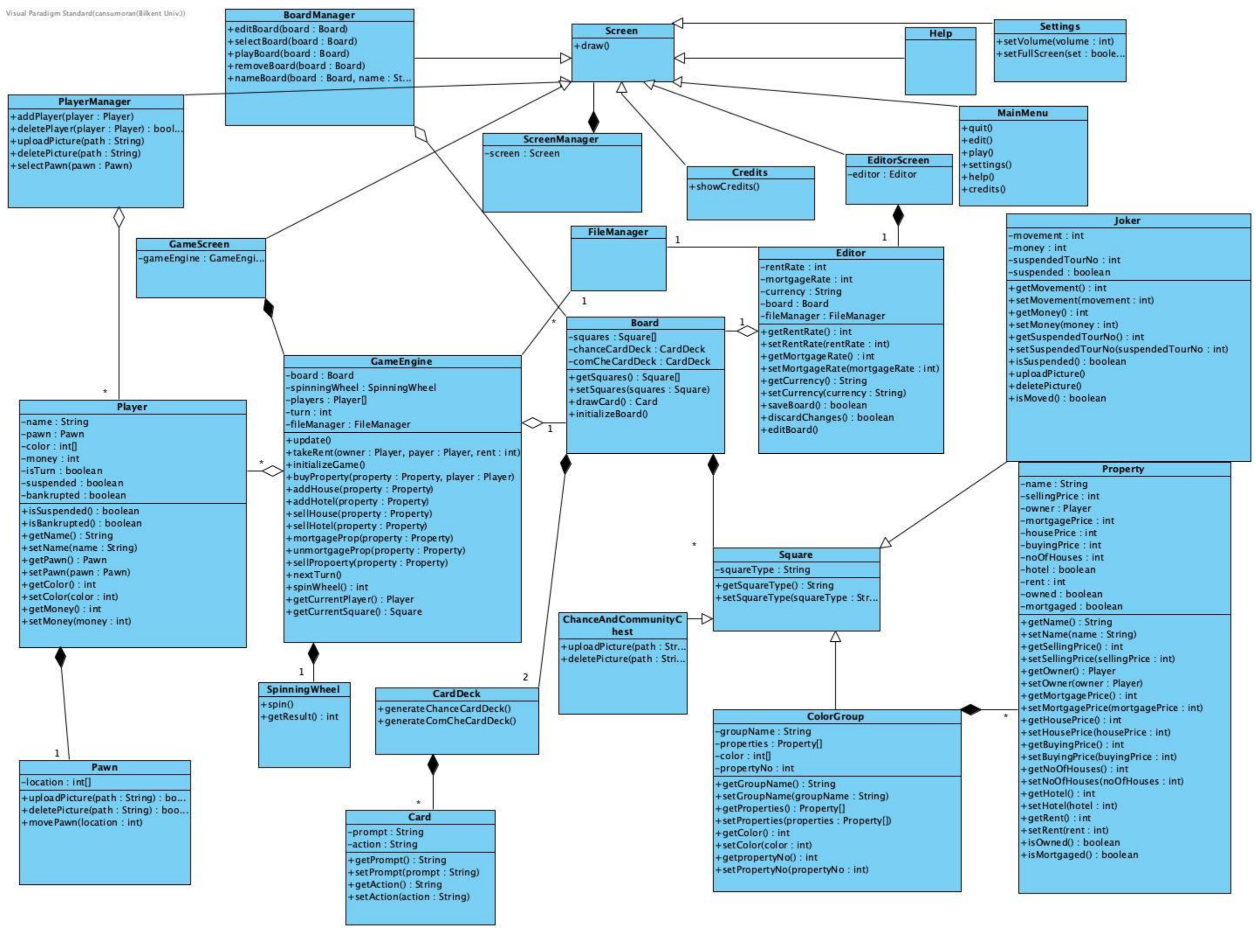
The updates in these components can trigger changes in the User Interface since User Interface components decide what to draw on screen based on Data components.

4. Low-level design

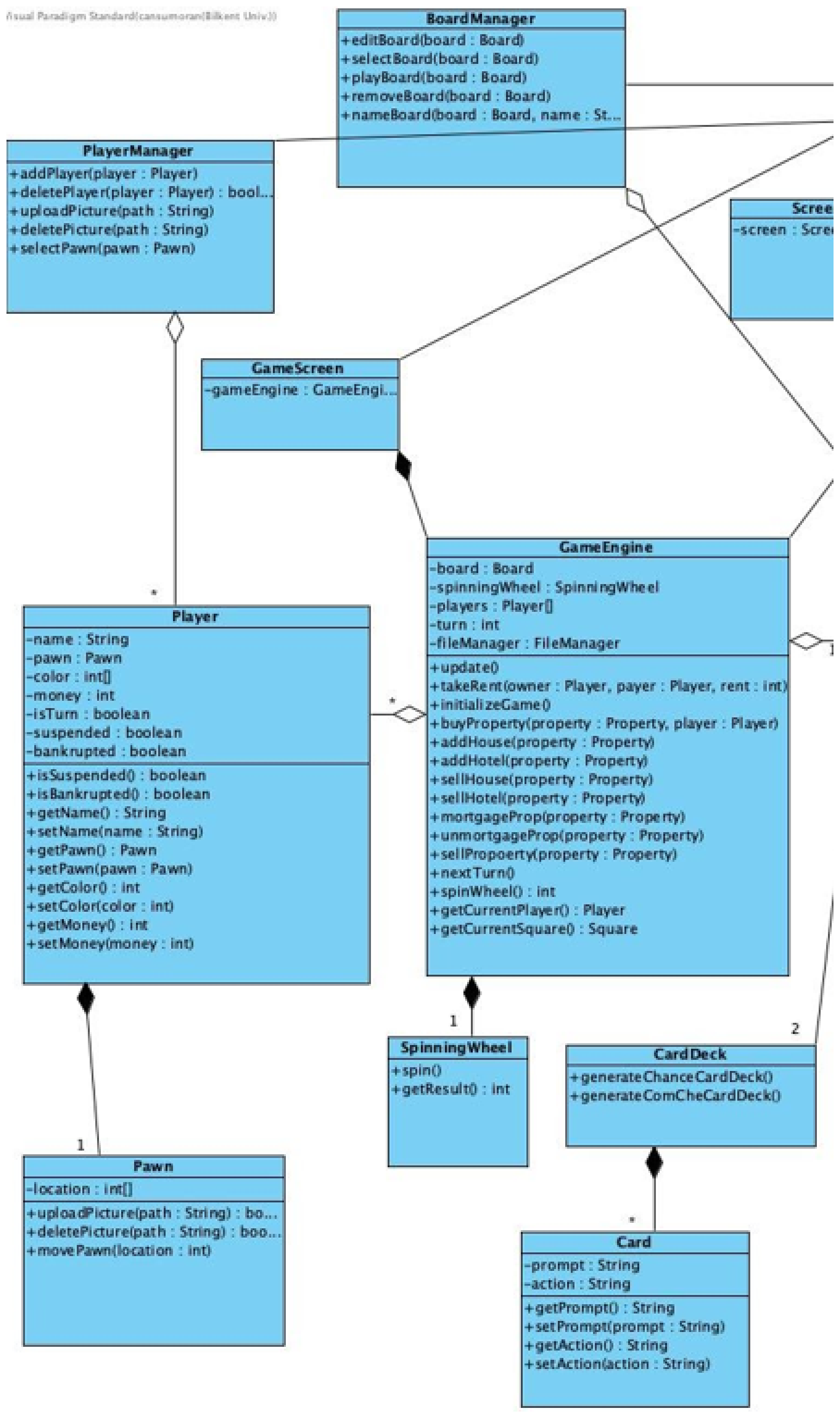
4.1 Object design trade-offs

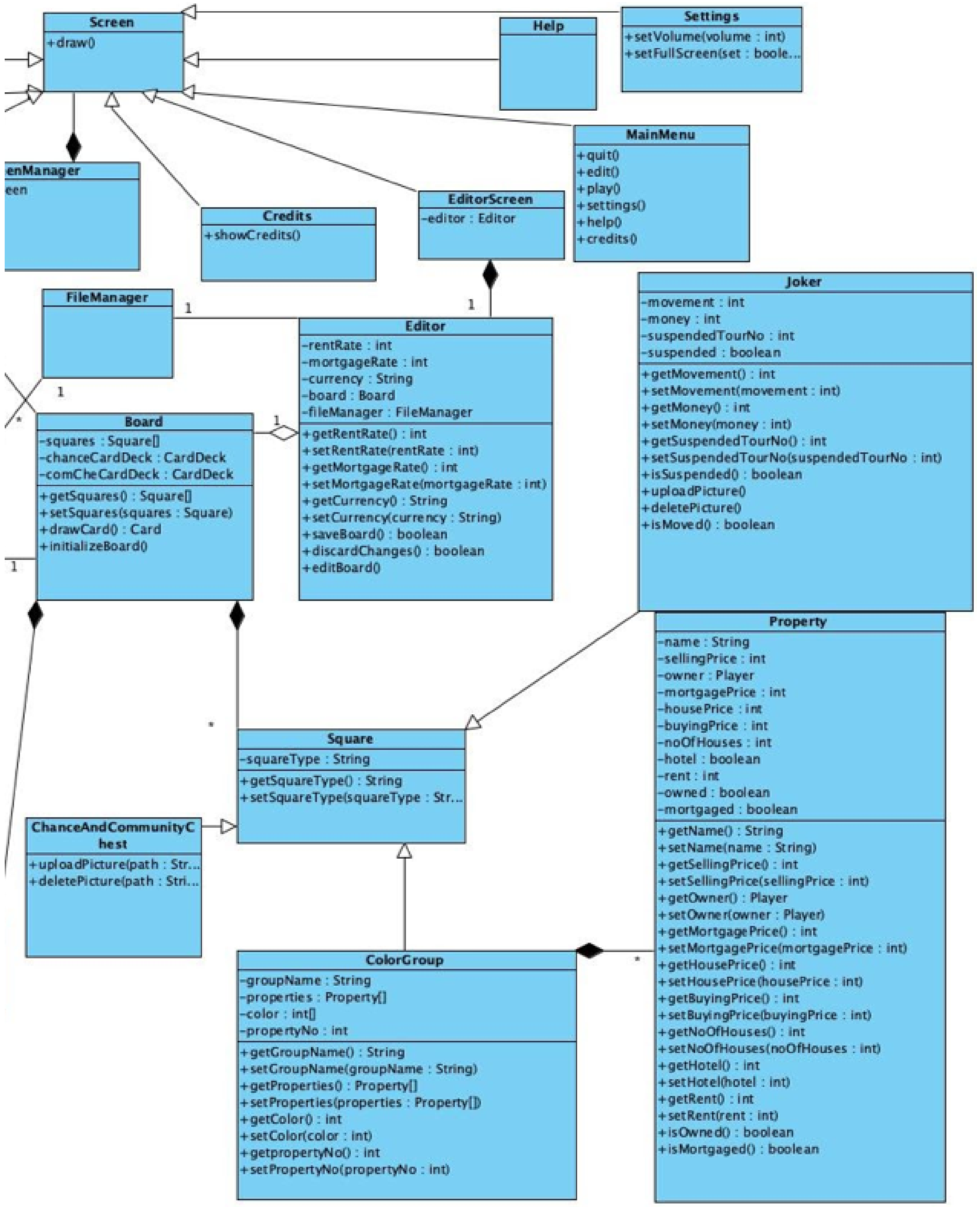
We have a sub-section called trade-offs (1.2.1) in which we discuss the same topic. Additionally, since we are still in the process of implementation, we have not made a final decision on which design patterns to use. We will be explaining the ones we make use of when we do the implementation.

4.2 Final object design



For readability purposes, we have divided the class diagram above into two pieces and shown those pieces individually below.


Visual Paradigm Standard(cansumoran(Bilkent Univ.))
BoardManager
+editBoard(board : Board)
+selectBoard(board : Board)
+playBoard(board : Board)
+removeBoard(board : Board)
+nameBoard(board : Board, name : St...
PlayerManager
+addPlayer(player : Player)
+deletePlayer(player : Player) : bool...
+uploadPicture(path : String)
+deletePicture(path : String)
+selectPawn(pawn : Pawn)
Scree
-screen : Scre
GameScreen
-gameEngine : GameEngi...
GameEngine
-board : Board
-spinningWheel : SpinningWheel
-players : Player[]
-turn : int
-fileManager : FileManager
+update()
+takeRent(owner : Player, payer : Player, rent : int)
+initializeGame()
+buyProperty(property : Property, player : Player)
+addHouse(property : Property)
+addHotel(property : Property)
+sellHouse(property : Property)
+sellHotel(property : Property)
+mortgageProp(property : Property)
+unmortgageProp(property : Property)
+sellPropoerty(property : Property)
+nextTurn()
+spinWheel() : int
+getCurrentPlayer() : Player
+getCurrentSquare() : Square
*
Player
-name : String
-pawn : Pawn
-color : int[]
-money : int
-isTurn : boolean
-suspended : boolean
-bankrupted : boolean
+isSuspended() : boolean
+isBankrupted() : boolean
+getName() : String
+setName(name : String)
+getPawn() : Pawn
+setPawn(pawn : Pawn)
+getColor() : int
+setColor(color : int)
+getMoney() : int
+setMoney(money : int)
*
1
1
SpinningWheel
+spin()
+getResult() : int
2
CardDeck
+generateChanceCardDeck()
+generateComCheCardDeck()
1
Pawn
-location : int[]
+uploadPicture(path : String) : bo...
+deletePicture(path : String) : boo...
+movePawn(location : int)
*
Card
-prompt : String
-action : String
+getPrompt() : String
+setPrompt(prompt : String)
+getAction() : String
+setAction(action : String)


Screen
+draw()
Help
Settings
+setVolume(volume : int)
+setFullScreen(set : boole...
enManager
een
MainMenu
+quit()
+edit()
+play()
+settings()
+help()
+credits()
Credits
+showCredits()
EditorScreen
-editor : Editor
Joker
-movement : int
-money : int
-suspendedTourNo : int
-suspended : boolean
+getMovement() : int
+setMovement(movement : int)
+getMoney() : int
+setMoney(money : int)
+getSuspendedTourNo() : int
+setSuspendedTourNo(suspendedTourNo : int)
+isSuspended() : boolean
+uploadPicture()
+deletePicture()
+isMoved() : boolean
FileManager
1
1
Editor
-rentRate : int
-mortgageRate : int
-currency : String
-board : Board
-fileManager : FileManager
+getRentRate() : int
+setRentRate(rentRate : int)
+getMortgageRate() : int
+setMortgageRate(mortgageRate : int)
+getCurrency() : String
+setCurrency(currency : String)
+saveBoard() : boolean
+discardChanges() : boolean
+editBoard()
1
*
1
Board
-squares : Square[]
-chanceCardDeck : CardDeck
-comCheCardDeck : CardDeck
+getSquares() : Square[]
+setSquares(squares : Square)
+drawCard() : Card
+initializeBoard()
1
Property
-name : String
-sellingPrice : int
-owner : Player
-mortgagePrice : int
-housePrice : int
-buyingPrice : int
-noOfHouses : int
-hotel : boolean
-rent : int
-owned : boolean
-mortgaged : boolean
+getName() : String
+setName(name : String)
+getSellingPrice() : int
+setSellingPrice(sellingPrice : int)
+getOwner() : Player
+setOwner(owner : Player)
+getMortgagePrice() : int
+setMortgagePrice(mortgagePrice : int)
+getHousePrice() : int
+setHousePrice(housePrice : int)
+getBuyingPrice() : int
+setBuyingPrice(buyingPrice : int)
+getNoOfHouses() : int
+setNoOfHouses(noOfHouses : int)
+getHotel() : int
+setHotel(hotel : int)
+getRent() : int
+setRent(rent : int)
+isOwned() : boolean
+isMortgaged() : boolean
*
Square
-squareType : String
+getSquareType() : String
+setSquareType(squareType : Str...
ChanceAndCommunityChest
+uploadPicture(path : Str...
+deletePicture(path : Stri...
ColorGroup
-groupName : String
-properties : Property[]
-color : int[]
-propertyNo : int
+getGroupName() : String
+setGroupName(groupName : String)
+getProperties() : Property[]
+setProperties(properties : Property[])
+getColor() : int
+setColor(color : int)
+getpropertyNo() : int
+setPropertyNo(propertyNo : int)
*

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.

4.4 Class Interfaces

4.4.1 Player Class

Attributes:

- **private String name:** player' name
- **private Pawn pawn:** playes' pawn
- **private ArrayList<Integer> color**: the color associated with the player
- **private int money**: the amount of money that the player has
- **private boolean isTurn**: boolean to indicate whether it is the player's turn or not
- **private boolean suspended**: boolean to indicate if the player is suspended
- **private boolean bankrupted**: boolean to indicate if the player is bankrupt

Methods:

- **boolean isSuspended**: Method that returns "suspended" attribute
- **boolean isBankrupted()**: Method that returns "bankrupted" attribute
- **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
- **int getColor()**: Method that returns "color" attribute
- **void setColor(int 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

4.4.2 Pawn Class

Attributes:

- **private ArrayList<Integer> location**: Attribute that stores the locations of active pawns in the game

Methods:

- **boolean uploadPicture(String path)**: Method that sets the picture of the pawn
- **boolean deletePicture(String path):** Method that deletes the picture of the pawn
- **void movePawn(int location)**: Method that moves to the pawn to the location which is specified in the parameters.

4.4.3 PlayerManager Class

Methods:

- **boolean addPlayer(Player player)**: Method that adds a player to the "players" ArrayList in the GameEngine class. This method will be used at the beginning of the game where players are added.
- **boolean deletePlayer(Player player)**: Method that deletes the player specified in the parameters from the players ArrayList in the GameEngine.
- **boolean uploadPicture(String path)**: Method that uploads a picture of a player from the local memory.
- **boolean deletePicture(String path):** Method that deletes the picture of the player.
- **void selectPawn(Pawn pawn):** Method to select the pawn of the player.

4.4.4 GameEngine Class

Attributes:

- **private Board board**: The game board.
- **private SpinningWheel spinningWheel:** The spinning wheel of the game which will replace the "dice roll" feature of the monopoly board game.
- **private ArrayList<Player> players**: Players of the game.
- **private int turn**: Integer number which indicates it is which player's turn.
- **private FileManager fileManager :** File manager that is used to read board files.

Methods:

- **void update():** Method that updates the game whenever an action is performed.
- **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) money 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 sellProperty(Property property)**: Method for selling a property.
- **void nextTurn()** : Method for moving to the next player's turn.
- **int spinWheel()**: Method which spins the spinning wheel and returns an integer value in order to move the current player's pawn.
- **Player getCurrentPlayer()** : Method which returns the current player who has the turn to play.
- **Square getCurrentSquare() :** Method which returns the square that the current player's pawn is currently landed on.

4.4.5 SpinningWheel Class

Methods:

- **void spin()**: This method initiates the spinning action on the spinning wheel.
- **int getResult():** This method returns the result on the spinning wheel.

4.4.6 CardDeck Class

Methods:

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

4.4.7 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.

4.4.8 Board Class

Attributes:

- **private ArrayList <Square> squares:** This attribute is an arraylist 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.

Methods:

- **ArrayList<Square> getSquares()**: This method returns the arraylist of the squares that are included on the board.
- **void setSquares(Square squares):** This method sets the arraylist that contains the squares included on the board.
- **Card drawCard():** This method draws a card from a card deck and returns it.
- **void initializeBoard()**: This method initializes the board by preparing the card decks and board for the new game.

4.4.8 Editor 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.
- **private FileManager fileManager :** File manager that is used to read board files.

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 int sellingPrice:** This attribute is the selling price of 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 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 owned:** This attribute shows if the property is owned.
- **private boolean mortgaged:** 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.
- **int getSellingPrice():** This method returns the selling price of the property.
- **void setSellingPrice(int sellingPrice):** This method sets the selling price of the property to the value on parameter.
- **Player getOwned():** This method returns the Player which owns the property.
- void setOwned(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.

4.4.10 Square Class

Attributes:

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

Methods:

- **String getSquareType()**: This method returns the square type.
- **void setSquareType(String squareType)**: This method sets the square type to the type given as the parameter.

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 ArrayList<Integer> color**: This attribute represents the list of RGB values that represent the color of the color group.
- **private int propertyNo**: This attribute represents the number of properties belonging to 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.
- **ArrayList<Property> getProperties():** This method returns the property list which belong to the
- **void setProperties(ArrayList<Property> properties)**: This method updates the property list.
- **ArrayList<Integer> getColor():** This method returns the integer arraylist representing the color of the group.
- **void setColor(ArrayList<Integer> color):** This method changes the color of the group to the value given as the parameter.
- **int getPropertyNo:** This method returns the number of properties which belong to the color group.

- **void setPropertyNo(int propertyNo):** This method changes the number of properties to the value given as the parameter.

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

4.4.14 GameScreen Class

Attributes:

- **private GameEngine gameEngine**: This attribute is the game engine 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.

4.4.16 Screen Class

Methods:

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

4.4.17 Credits Class

Methods:

- **void showCredits():** This method is invoked when the credits page is open and it shows the credits.

4.4.18 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.

4.4.19 Help Class

4.4.20 EditorScreen Class

Attributes:

- **private Editor editor :** This attribute stores an instance of the Editor.

4.4.21 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.

4.4.22 Settings Class

Methods:

- **void setVolumeSize(int volume):** This method sets the volume using the integer represented by the slider bar related to the volume.
- **void setFullScreen(boolean set):** This method switches the game into fullscreen mode if it is not FullScreen and exits fullscreen if the game is in FullScreen mode according to its parameter set.

4.4.23 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 suspendedTourNo :** The amount of tours to be suspended that the Joker Square requires if a player's pawn lands on it.
- **private boolean suspended :** Boolean which is used to indicate whether that Joker square requires suspencion (like jail).

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 getSuspendedTourNo() :**Method which returns the amount of tours to suspend the players who land on the Joker square.
- **void setSuspendedTourNo(int suspendedTourNo):** Method which sets the amount of tours to suspend the players who land on the Joker square.
- **boolean isSuspended()**: Method which returns the boolean about whether the square is a suspencion square(like jail) or not.
- **boolean uploadPicture()** : Method which is used to upload a picture as a background to the joker square.
- **boolean deletePicture()**: Method which is used to delete the background picture of the joker place.
- **boolean isMoved()** : Method which returns a boolean indicating whether the Joker square requires any movement.