Bilkent University

Department of Computer Engineering

Monopoly Project Group 2J

Bilopoly: Monopoly's Bilkent version

Design Report

Asım Güneş Üstünalp, Mohammed S. Yaseen, Turan Mert Duran, Radman Lotfiazar, Mohammad Elham Amin

Instructor: Eray Tüzün
Teaching Assistants: Barış Ardıç, Emre Sülün, and Elgun Jabrayilzade

Contents

1. Introduction

1.1. Purpose of the system

Bilopoly is a virtual version of Monopoly with some interesting variations in the rules and design. Its design is inspired from Bilkent University and most of the buildings are named based on that.

1.2. Design goals

1.2.1. Trade-Offs

Memory vs Performance

Game will be high performance when it runs and reflects users' actions in a smooth and fast way. Therefore, we are planning to use object oriented principles not only to increase the performance of the game but also to handle the memory in an efficient way. How and how much memory will be occupied by the game is really important because if they are not handled in a reasonable way will lead to serious problems in performance of the game. Therefore, memory and performance have direct relation with each other and object oriented principles let us manage these two in the best way.

Functionality vs User-friendliness

Functions in this game will be similar to the original version of Monopoly because most of the people are familiar to it. Furthermore, functions will be designed in a way that people are easily able to understand and memorize them. Therefore, this virtual version of Monopoly will be user-friendly as much as possible because functions are familiar for people and easily understandable. On the other hand, while introducing new functionality, we are careful to add them in a way that they will seem very intuitive to be easily understandable and memorizable. That can be achieved by having the new functionality language similar to the original

functionality as well as the placement of the buttons or choices convenient.

Cost vs Maintenance

We will try hard to use the object oriented principles in the best way. However, Due to our limited time, energy and knowledge, it is possible we as a group make some mistakes. One of the most important sectors influenced is maintenance. For instance, these mistakes might lead to problems in modifiability or extendability which are parts of maintenance. Moreover, for solving a mistake might have to change the higher level which indicates the lack of modifiability in design of the game. However, these problems are inevitable because of a lack of knowledge, time and energy which are a cost for a project.

1.2.2. Criteria

Maintainability

Maintainability includes four different parts such as extendibility, modifiability, reusability and portability. Design of Bilopoly will be based on object oriented programming which will help us to design the system in an extensible and modifiable way because each subsystem has hierarchical structure. Therefore, adding new features or changing them for enhancing the system is easily possible without changing the higher system. Furthermore, in the design of this system there are different subsystems such as GUI which are able to be used in different projects. Hence, these subsystems allow our project to be reusable as much as possible. As we decide to design this project in Java, our project will be portable because of the JVM.

User-Friendliness

As Monopoly is one of the most famous games around the world, this version of the game's interface will not have considerable

changes because most of the users are already familiar with it and they would not spend time to learn it. Furthermore, the icons, tokens and cards are designed and positioned in a way that users easily are able to understand and memorize them due to increasing the user-friendliness. In addition, “How to play” option includes all rules, icons and cards which are necessary for the player to know.

Performance

For performance of the system we try to decrease the time which users have to wait for the system to start and the result of their action. Therefore, the system will immediately react and process the inputs in a small amount of time. Moreover, in this system exceptions will be handled in the best way for decreasing the possibility of bugs and crash of the system.

2. High-level software architecture

2.1. Subsystem decomposition

It is important for a system that is being implemented by a number of developers and that is subject to change in the future to be decomposed into multiple subsystems. The decomposition’s goal is to introduce decoupling between the subsystems and cohesion in each of them. The decoupling between the subsystems will make it easier for the system to adapt to change since very few subsystems might need to change due to change in any of the other subsystems.

We have examined two high-level architecture solutions to organise the subsystems that are MVC and MVP. Our final decision was for the MVP (Model - View - Presenter) architecture pattern. Following are the reasons for not choosing the MVC (Model - View - Controller) pattern:

1. MVC architecture was originally introduced in smalltalk language where input used to come from sources other than the view. However in our

application the view could be an input source as well, which means some complications might occur.

2. In the most common way of implementing MVC, the model is clustered with the business logic as well as state storage. This means that the model will have more than one concern, which can be problematic.

3. In the most common implementation of MVC, the observer pattern is used between the model and the view, which causes the view to be dependent on the model. On the other hand the controller is dependent on the view for the input. This dependency cycle can be problematic.

Now these are the reasons for choosing MVP:

1. The flow of events is more natural and reduces the dependencies. The events initiated in the view, goes to the presenter, which fetches some data from the model, processes them, and updates the view.

2. The view is not dependent on the model, and both the view and the model are only dependent on the presenter.

The only change we introduced to the MVP is that the presenter was separated into two parts: Game Presenter and General Presenter.

- **Game Presenter** handles every event related to game science like board management, tokens movements, and trades.
- **Menus Presenter** handles all the logic that is not directly related to the game such as menus switching as well as handling local data, sound volume, and initiating a new game.

Other parts of the MVP that are Model and View (User Interface in our case) are handling the following:

- **Model** stores the current state of the game.
- **User Interface** transfers the events to the presenter as well as handling the views rendering as requested by the presenter.

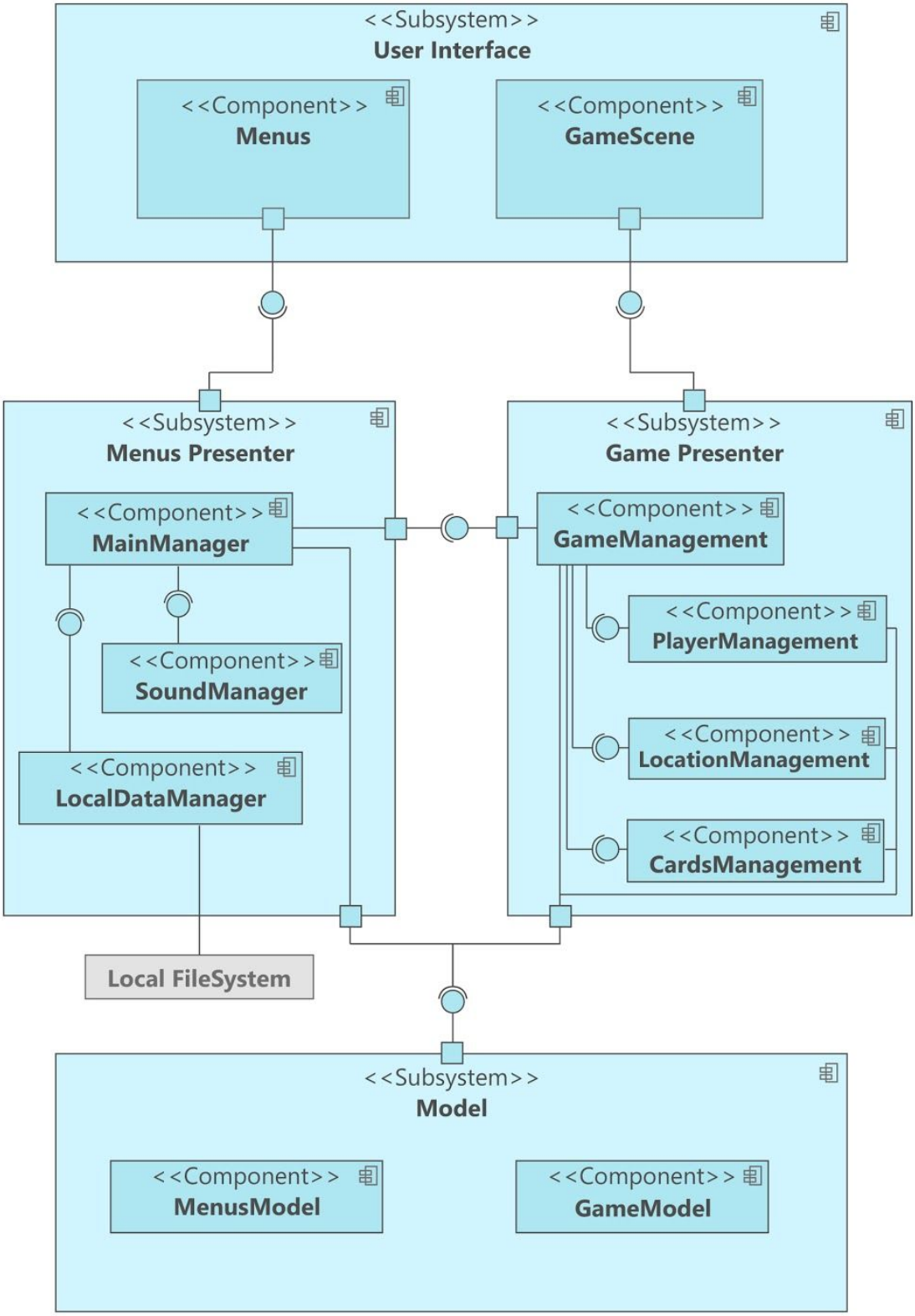


Figure (2.1): Subsystem Decomposition

2.2. Design Patterns

In terms of design patterns, we have decided to use the singleton design pattern for most of the presenter classes such as the MainManager, SoundManager, LocalDataManager, TurnManager, DiceManager, BoardManager, LocationManager, and CardsManager. Second, the facade design pattern is used to communicate between the subsystems such as between User Interface, GeneralPresenter, and Game Presenter. In our design, the only class that interacts with the UI is the BoardManager class in our Game Phase. In order to change the information seen on the screen the other classes need to communicate through BoardManager with the UI.

2.3. Hardware/software mapping

Since our game is going to be written in Java, the software side is Java code. Java code will require JRE (Java Runtime Environment.) The game can be run on any operating system, windows, Mac, or linux. Network connection is not necessary since the game will be entirely local.

2.4. Persistent data management

Bilopoly's persistent data include:

- The players' number and their colors, names, and tokens.
- The inventory of each player.
- The locations of the tokens.
- The current turn is to which player.
- The cards that each player is holding.
- The cards that are yet untaken.
- The sound volume.
- The match name

These data are going to be serialized and stored in a text format in a special directory in the files system.

2.5. Access control and security

Bilopoly is a local game. It can only be played using one computer; multiple players will play using only one computer and the game

alternates the turns between them. Game data will be stored in a local database and no data will be accessed through the network; therefore, there is no security risk of data loss or unauthorized access. On the other hand; the local database will be accessed using only one subsystem that is the LocalDataManagement.

2.6. Boundary conditions

- **Game initialization**

  In the case of a new game, the user clicks on the “new game” button then the system would start creating a new game. No two games can be initialized concurrently since after initializing the new game the main menu would disappear and the game scene would appear. On the other hand, in the case of loading a saved game, the player would click the “Load game” which will open a list of all the saved games for the player to choose from then press the “start game” button to continue playing. If no game was already saved, the player would see an empty list and the “start game” button would be disabled.

- **Game Exit**

  If the player is in the game scene, to exit, the player needs to press on the pause menu button. From there the player would press on the “main menu” button. If the player hasn’t already saved the game, using the pause menu option “save game,” a prompt will appear asking whether to save the game. When the player is out of the game scene and in the main menu, by pressing the “Quit” button, the player can exit the game; However while doing that they will be welcomed with a message saying “Are you sure you want to quit?”

- **Game failure**

  Failures can happen especially while doing the io operations. Therefore, specific mechanisms will be implemented to prevent game crash when facing a failure. Such mechanisms are the following:

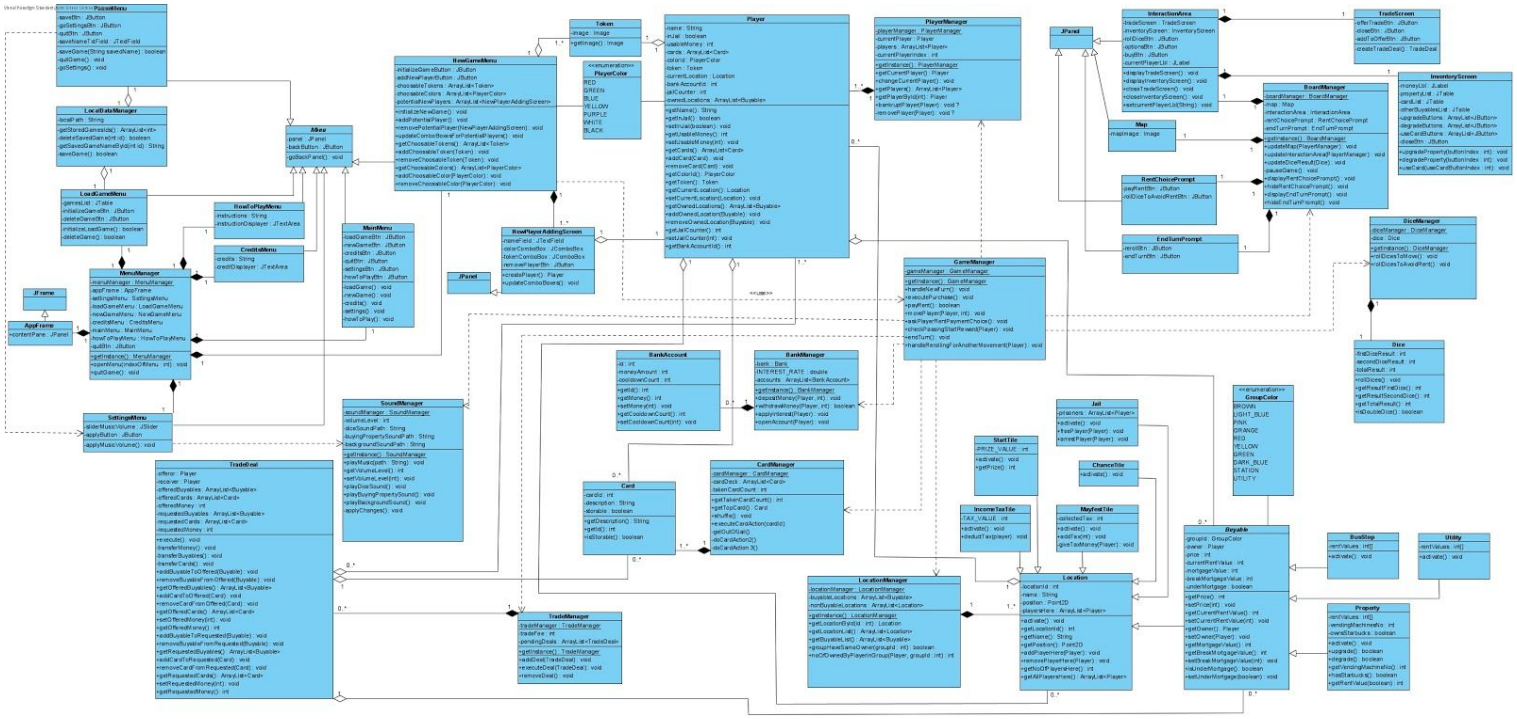
  - If sounds fail to load the user will be prompted with a message asking whether they want to continue without

sounds. Otherwise, they can choose to retry loading the sounds.

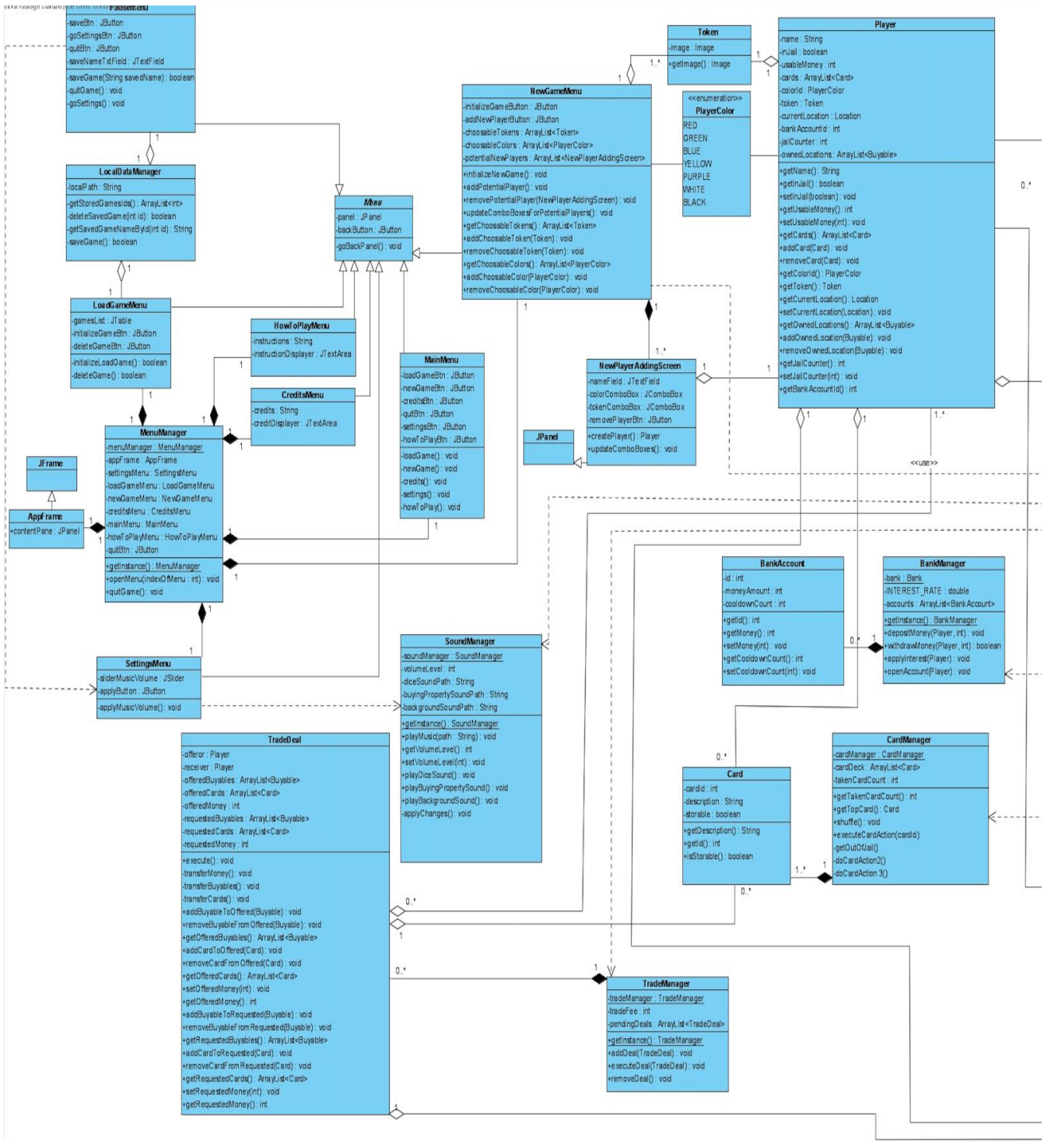
- If graphics fail to load, the game will not start and the user has to retry loading them.
- If the game fails to load the previously saved sessions, the user will get an error message on the saved games list indicating that saved games could not be loaded.
- If the game fails to save the current game after the user request, a prompt will appear indicating the failure and asking the user's option to retry saving or not.

3. Low-level design

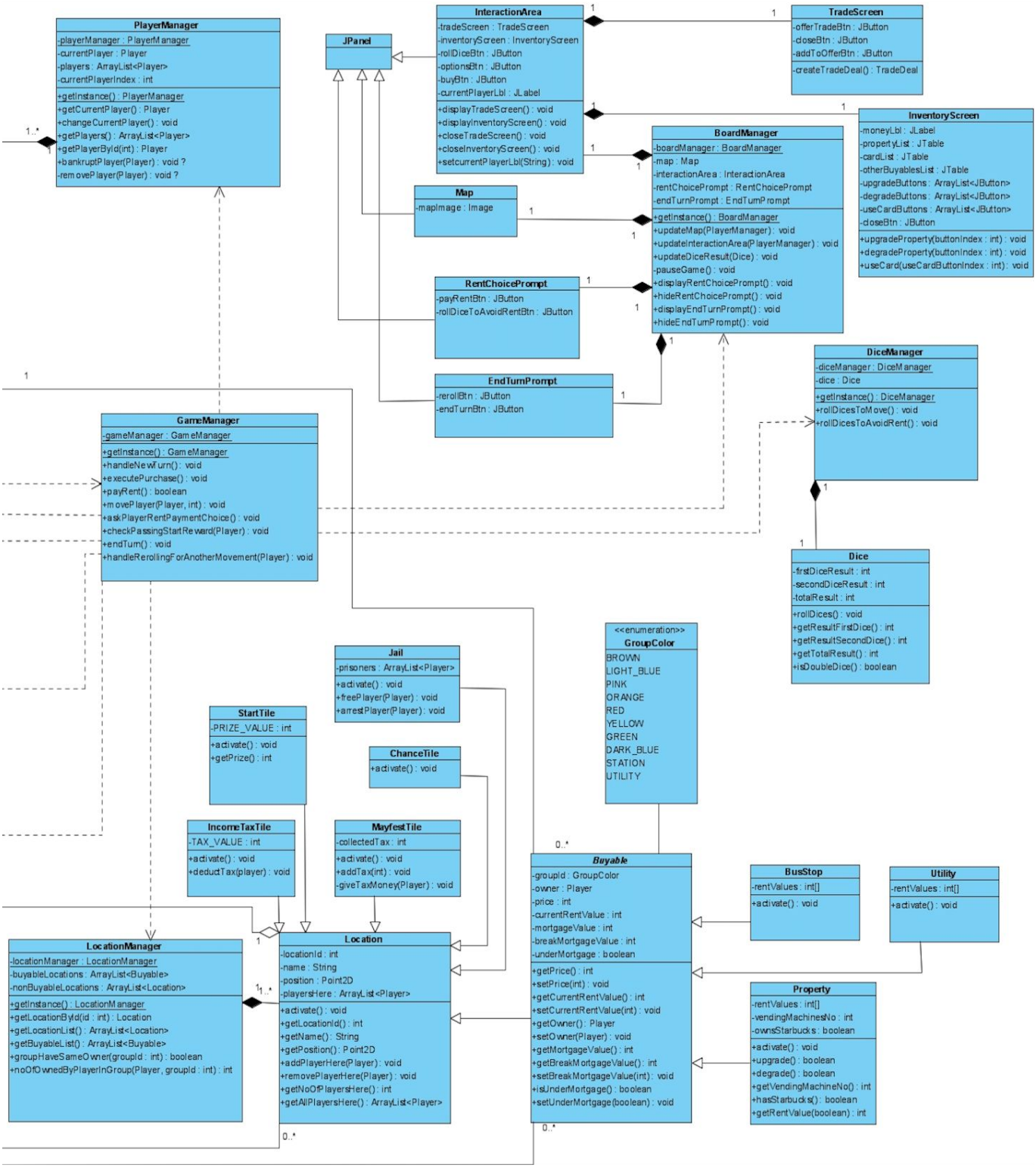
3.1. Final object design



Piece 1


PauseMenu
-saveBtn : JButton
-goSettingsBtn : JButton
-quitBtn : JButton
-saveNameTxtField : JTextField
-saveGame(String savedName) : boolean
-quitGame() : void
-goSettings() : void
LocalDataManager
-localPath : String
-getStoredGamesIds() : ArrayList<int>
-deleteSavedGame(int id) : boolean
-getSavedGameNameById(int id) : String
-saveGame() : boolean
Menu
-panel : JPanel
-backButton : JButton
-goBackPanel() : void
Token
-image : Image
+getImage() : Image
Player
-name : String
-inJail : boolean
-usableMoney : int
-cards : ArrayList<Card>
-colorId : PlayerColor
-token : Token
-currentLocation : Location
-bankAccountId : int
-jailCounter : int
-ownedLocations : ArrayList<Buyable>
+getName() : String
+getInJail() : boolean
+setInJail(boolean) : void
+getUsableMoney() : int
+setUsableMoney(int) : void
+getCards() : ArrayList<Card>
+addCard(Card) : void
+removeCard(Card) : void
+getColorId() : PlayerColor
+getToken() : Token
+getCurrentLocation() : Location
+setCurrentLocation(Location) : void
+getOwnedLocations() : ArrayList<Buyable>
+addOwnedLocation(Buyable) : void
+removeOwnedLocation(Buyable) : void
+getJailCounter() : int
+setJailCounter(int) : void
+getBankAccountId() : int
NewGameMenu
-initializeGameButton : JButton
-addNewPlayerButton : JButton
-choosableTokens : ArrayList<Token>
-choosableColors : ArrayList<PlayerColor>
-potentialNewPlayers : ArrayList<NewPlayerAddingScreen>
+initializeNewGame() : void
+addPotentialPlayer() : void
+removePotentialPlayer(NewPlayerAddingScreen) : void
+updateComboBoxesForPotentialPlayers() : void
+getChoosableTokens() : ArrayList<Token>
+addChoosableToken(Token) : void
+removeChoosableToken(Token) : void
+getChoosableColors() : ArrayList<PlayerColor>
+addChoosableColor(PlayerColor) : void
+removeChoosableColor(PlayerColor) : void
<<enumeration>>
PlayerColor
RED
GREEN
BLUE
YELLOW
PURPLE
WHITE
BLACK
LoadGameMenu
-gamesList : JTable
-initializeGameBtn : JButton
-deleteGameBtn : JButton
-initializeLoadGame() : boolean
-deleteGame() : boolean
HowToPlayMenu
-instructions : String
-instructionDisplayer : JTextArea
CreditsMenu
-credits : String
-creditDisplayer : JTextArea
MainMenu
-loadGameBtn : JButton
-newGameBtn : JButton
-creditsBtn : JButton
-quitBtn : JButton
-settingsBtn : JButton
-howToPlayBtn : JButton
-loadGame() : void
-newGame() : void
-credits() : void
-settings() : void
-howToPlay() : void
NewPlayerAddingScreen
-nameField : JTextField
-colorComboBox : JComboBox
-tokenComboBox : JComboBox
-removePlayerBtn : JButton
+createPlayer() : Player
+updateComboBoxes() : void
JPanel
MenuManager
-menuManager : MenuManager
-appFrame : AppFrame
-settingsMenu : SettingsMenu
-loadGameMenu : LoadGameMenu
-newGameMenu : NewGameMenu
-creditsMenu : CreditsMenu
-mainMenu : MainMenu
-howToPlayMenu : HowToPlayMenu
-quitBtn : JButton
+getInstance() : MenuManager
+openMenu(indexOfMenu : int) : void
+qutGame() : void
JFrame
AppFrame
+contentPane : JPanel
<<use>>
BankAccount
-id : int
-moneyAmount : int
-cooldownCount : int
+getId() : int
+getMoney() : int
+setMoney(int) : void
+getCooldownCount() : int
+setCooldownCount(int) : void
BankManager
-bank : Bank
-INTEREST_RATE : double
-accounts : ArrayList<BankAccount>
+getInstance() : BankManager
+depositMoney(Player, int) : void
+withdrawMoney(Player, int) : boolean
+applyInterest(Player) : void
+openAccount(Player) : void
SettingsMenu
-sliderMusicVolume : JSlider
-applyButton : JButton
-applyMusicVolume() : void
SoundManager
-soundManager : SoundManager
-volumeLevel : int
-diceSoundPath : String
-buyingPropertySoundPath : String
-backgroundSoundPath : String
+getInstance() : SoundManager
+playMusic(path : String) : void
+getVolumeLevel() : int
+setVolumeLevel(int) : void
+playDiceSound() : void
+playBuyingPropertySound() : void
+playBackgroundSound() : void
-applyChanges() : void
TradeDeal
-offeror : Player
-receiver : Player
-offeredBuyables : ArrayList<Buyable>
-offeredCards : ArrayList<Card>
-offeredMoney : int
-requestedBuyables : ArrayList<Buyable>
-requestedCards : ArrayList<Card>
-requestedMoney : int
+execute() : void
-transferMoney() : void
-transferBuyables() : void
-transferCards() : void
+addBuyableToOffered(Buyable) : void
+removeBuyableFromOffered(Buyable) : void
+getOfferedBuyables() : ArrayList<Buyable>
+addCardToOffered(Card) : void
+removeCardFromOffered(Card) : void
+getOfferedCards() : ArrayList<Card>
+setOfferedMoney(int) : void
+getOfferedMoney() : int
+addBuyableToRequested(Buyable) : void
+removeBuyableFromRequested(Buyable) : void
+getRequestedBuyables() : ArrayList<Buyable>
+addCardToRequested(Card) : void
+removeCardFromRequested(Card) : void
+getRequestedCards() : ArrayList<Card>
+setRequestedMoney(int) : void
+getRequestedMoney() : int
Card
-cardId : int
-description : String
-storable : boolean
+getDescription() : String
+getId() : int
+isStorable() : boolean
CardManager
-cardManager : CardManager
-cardDeck : ArrayList<Card>
-takenCardCount : int
+getTakenCardCount() : int
+getTopCard() : Card
+shuffle() : void
+executeCardAction(cardId)
-getOutOfJail()
-doCardAction2()
-doCardAction 3()
TradeManager
-tradeManager : TradeManager
-tradeFee : int
-pendingDeals : ArrayList<TradeDeal>
+getInstance() : TradeManager
+addDeal(TradeDeal) : void
+executeDeal(TradeDeal) : void
+removeDeal() : void

Piece 2



Class Interfaces

Menu Class

Attributes:

- private JPanel jpanel : This attribute will be the panel of menus.
- private JButton backButton : This attribute will be a button for going back on menus.

Methods:

- private void goBackPanel() : This method will be called and execute going back panel instructions when the user presses backButton.

Pause Menu Class

Attributes:

- **private JButton saveButton :** This attribute will be a button for saving game.
- **private JButton quitButton :** This attribute will be a button for quit game.
- **private JButton goSettingsButton :** This attribute will be a button for user to go settings menu from pause menu.
- **private JTextField saveName :** This attribute is for changing game name and saving it with this name to remember later.

Methods:

- **private boolean saveGame(String savedName) :** This method will be called and execute saving game instructions when the user presses saveButton.
- **private void quitGame() :** This method will be called and execute quit game instructions when the user presses quitButton.

- **private void goSettings() :** This method will be called and execute going settings from pause menu instructions when the user presses goSettingsButton.

Local Data Manager Class

Attributes:

- **private String localPath :** This attribute will be a string of location for where to save locally the game.

Methods:

- **private boolean deleteSavedGame(int id) :** This method will be called and execute deleting saved game by finding id instructions.
- **private ArrayList<int> getStoredGamesIds() :** This method will be called and execute getting all stored games in local data and returning int ids' of them instructions.
- **private String getSavedGameNameById(int id) :** This method will be called and execute finding the game by checking by id and returning its name instructions.
- **private boolean saveGame() :** This method will be called and execute saving game instructions.

Load Game Menu Class

Attributes:

- **private JTable jTableGamesList :** This attribute will be a table for showing all games on the load game menu.

- **private JButton initalizeGameButton :** This attribute will be a button for the user to press for initializing the selected game from the table.
- **private JButton deleteGameButton :** This attribute will be a button for the user to press for deleting the selected game from the table.

Methods:

- **private boolean initializeLoadGame()** : This method will be called and execute initializing selected saved game that is decided on table.
- **private boolean deleteGame() :** This method will be called and execute deleting selected saved game that is decided on the table.

Settings Menu Class

Attributes:

- **private JSlider sliderMusicVolume :** This attribute will be a slider for changing volume of the game from the settings menu.
- **private JButton applyButton :** This attribute will be a button for the user to press for applying changes that are made on settings menu.

Methods:

- **private void applyMusicVolume() :** This method will be called and execute changing music volume instructions.

Menu Manager Class

Attributes:

- **private SettingsMenu settingsMenu :** This attribute will be object of Settings Menu class to control it from menu manager class.
- **private LoadGameMenu loadGameMenu :** This attribute will be object of Load Game Menu class to control it from menu manager class.
- **private NewGameMenu newGameMenu :** This attribute will be object of New Game Menu class to control it from menu manager class.
- **private CreditsMenu creditsMenu :** This attribute will be object of Credits Menu class to control it from menu manager class.
- **private HowToPlayMenu howToPlayMenu :** This attribute will be object of How To Play Menu class to control it from menu manager class.
- **private MainMenu mainMenu :** This attribute will be object of Main Menu class to control it from menu manager class.
- **private JButton quitButton :** This attribute will be a button for the user to press for quit game.

Methods:

- **private void openMenu(int indexOfMenu) :** This method will be called and execute open menu of wanted panel instructions.
- **private void quitGame() :** This method will be called and execute quit game instructions when user presses quitButton.

Sound Manager Class

Attributes:

- **private static SoundManager soundManager :** This attribute will be constructor of Sound Manager class.
- **private String diceSoundPath :** This attribute will be string of music path of dice rolling.
- **private String buyingPropertyPath :** This attribute will be string of music path of buying property.
- **private String backgroundSoundPath :** This attribute will be string of music path of background sound.

Methods:

- **private void playMusic(String path) :** This method will be called and execute playing needed music instructions.
- **public void playDiceSound() :** This method will be called and execute playing rolling dice music instructions.
- **public void playBuyingPropertySound() :** This method will be called and execute playing buying property music instructions.
- **public void playBackgroundSound() :** This method will be called and execute playing background music instructions.
- **private void applyChanges() :** This method will be called and execute application of changes instructions.
- **public static SoundManager getInstance() :** This method will be called for returning sound manager of the game.

How To Play Menu Class

Attributes:

- **private String instructions :** This attribute will be a string of including how to play the game and rules.

Credits Menu Class

Attributes:

- **private String credits :** This attribute will be a string of including credits.

Main Menu Class

Attributes:

- **private JButton loadGameBtn :** This attribute will be a button of open load game.
- **private JButton newGameBtn :** This attribute will be a button of open new game.
- **private JButton creditsBtn :** This attribute will be a button of open credits.
- **private JButton quitBtn :** This attribute will be a button of quit.
- **private JButton settingsBtn :** This attribute will be a button of open settings of the game.
- **private JButton howToPlayBtn :** This attribute will be a button of open how to play.

Methods:

- **private void loadGame() :** This method will be called and execute open load game menu instructions.
- **private void newGame() :** This method will be called and execute open new game menu instructions.
- **private void credits() :** This method will be called and execute open credits menu instructions.
- **private void quitGame() :** This method will be called and execute quit game instructions.
- **private void settings() :** This method will be called and execute open settings menu of the game instructions.
- **private void howToPlay() :** This method will be called and execute open how to play game menu instructions.

New Game Menu Class

Attributes:

- **private JButton jInitializeGameButton :** This attribute will be a button of initializing new game.

Methods:

- **public void initializeNewGame() :** This method will be called and execute creating new game instructions.

Card Class

Attributes:

- **private int cardId :** This attribute will be a id of each card.
- **private String description :** This attribute will store the description of each card.
- **private boolean isStorable:** This attribute is used to indicate whether a card can be stored or its instructions need to be executed immediately.

Methods:

- **public String getDescription() :** This method will be called and return descriptions of card.
- **public int getId() :** This method will be called and return id of the card.
- **public boolean isStorable() :** This method will be called and return whether card a storable one or not.

CardManager

Attributes:

- **Public CardManager cardManager:** Since card manager is a Singleton class it holds an object of itself.
- **ArrayList<Card> cards :** This attribute will be storing cards of the game.
- **private int takenCardCount :** This attribute will be counting how many cards are drawn from the deck.

Methods:

- **public Card getTopCard() :** This method will be called and returns top card on card list.
- **public void shuffle() :** This method will be called and shuffles card orders.
- **public void executeCardAction(cardID) :** This method will execute the instructions of the card.
- **private void getOutOfJail() :** This method will be called to execute instructions of jail the card.
- **Private void doCardAction2() :** This method will called to execute card 2's (will be defined later) instructions
- **Private void doCardAction2() :** This method will called to execute card 3's (to be defined later) instructions
- **public int getTakenCardCount() :** This method will be called and return the taken card count as int.

BankAccount

Attributes:

- **private int id:** This attribute will be a id of each card.
- **private int money:** This attribute will store the description of each card.
- **private int coolDownCount:** Number of the rounds after the money is deposited.

Methods:

- **public String getDescription() :** This method will be called and return descriptions of card.
- **public int getId() :** This method will be called and return id of the card.

- **public int getMoney() :** Returns the amount of money in the account.
- **public voidsetMoney(amount) :** Sets the account money to the given amount.
- **public int getCoolDownCount() :** Returns the number of rounds since the money was deposited to the account.
- **public int getCoolDownCount(int) :** Sets the number of rounds since money was deposited.

BankManager

Attributes:

- **public BankManager bankManager :** Since bank manager is a Singleton class it holds an object of itself.
- **private double INTEREST_RATE:** Constant interest value that will be applied on the accounts in the bank.
- **private ArrayList<BankAccount> accounts:** Players' accounts in the bank.

Methods:

- **public BankManager getInstance() :** Returns an instance of the BankManager class.
- **public void depositMoney(Player, amount) :** Deposits the amount given to the given player's account.
- **public boolean withdrawMoney(Player, amount) :** Withdraws the specified amount from the given player's account.

- **private void applyInterest(Player) :** Apply the interest rate on the given player's account.
- **Private void openAccount(Player) :** Creates a new account for the given player.

TradeDeal

Attributes:

- **private Player offerer:** The player who offered the deal.
- **private Player receiver:** The player who will receive the offer.
- **private ArrayList<Buyables> offeredBuyables:** List of the Buyable objects that are offered.
- **private ArrayList<Card> offeredCards:** List of the Cards that are offered.
- **private int offeredMoney:** Amount of the offered money.
- **private ArrayList<Buyables> requestedBuyables:** List of the Buyable objects that are asked for.
- **private ArrayList<Card> offeredCards:** List of the Cards that are asked for.
- **private int requestedMoney:** Amount of the requested money.

Methods:

- **public void execute() :** Executes the TradeDeal.
- **public void transferMoney() :** Handles money transfers between players.
- **public void transferBuyables() :** Handles exchanging the Buyables in deal among the players.

- **public void transferCards() :** Handles transferring cards included in the deal.
- **public void addBuyableToOffered(Buyable) :** Adds the given Buyable to the list of offered Buyables.
- **public void removeBuyableFromOffered(Buyable) :** Removes the given Buyable from the list of offered Buyables.
- **public ArrayList<Buyable> getOfferedBuyables() :** Returns the list of offered Buyables in the deal.
- **public void addCardToOffered(Card) :** Adds the given Card to the list of offered Buyables.
- **public void removeCardFromOffered(Card) :** Removes the given Card from the list of offered Buyables.
- **public ArrayList<Card> getOfferedCards() :** Returns the list of offered Cards in the deal.
- **public void setOfferedMoney(amount) :** Sets the offered money to the given amount.
- **public int getOfferedMoney() :** Returns the amount of money offered.
- **public void addBuyableToRequested(Buyable) :** Adds the given Buyable to the list of requested Buyables.
- **public void removeBuyableFromRequesetd(Buyable) :** Removes the given Buyable from the list of requestedBuyables.
- **public ArrayList<Buyable> getOfferedRequested() :** Returns the list of request Buyables in the deal.
- **public void addCardToRequested(Card) :** Adds the given Card to the list of requested Buyables.

- **public void removeCardFromOffered(Card) :** Removes the given Card from the list of requested Buyables.
- **public ArrayList<Card> getRequestedCards() :** Returns the list of requested Cards in the deal.
- **public void setRequestedMoney(amount) :** Sets the requested money to the given amount.
- **public void getRequestedMoney() :** Returns the amount o the money requested.

TradeManager

Attributes:

- **public TradeManager tradeManager :** Since TradeManager is a Singleton class it holds an object of itself.
- **private int tradeFee:** The fee that will be applied on the trades
- **private ArrayList<TradeDeal> pendingDeals:** List of the currently pending deals.

Methods:

- **public BankManager getInstance() :** Returns an instance of the TradeManager class.
- **public void addDeal(TradeDeal) :** Adds the given deal to the list of pending deals.
- **public void executeDeal(TradeDeal) :** Executes the deal given.

- **private void removeDeal(TradeDeal) :** Removes a deal from pending deals after its execution.
- **Private void openAccount(Player) :** Creates a new account for the given player.

Player Class

Attributes:

- **private String name :** This attribute will be storing the name of the player.
- **Private boolean inJail :** This attribute will be storing whether a player is in jail or not.
- **private ArrayList<Card> cards :** This attribute will be storing all cards that the user holds.
- **private Playercolor colorId :** This attribute will be storing the color id of the player.
- **private Token token :** This attribute will be storing which token the player has.
- **private int usableMoney :** This attribute will be storing the money which a player has for during the game .
- **private BankAccount bankAccount :** The account of the player in the Bank.
- **private int jailCounter:** The number of turns that the player has been in the jail.
- **public ArrayList<Location> ownedLocations:** The list of locations that this player owns.

Methods:

- **public String getName() :** This method will be called and return the name of the player.
- **public boolean getInJail() :** This method will be called for returning boolean which indicates whether the player is in jail or not.
- **public boolean setInJail(boolean) :** This method will be called for setting a player in the jail.
- **public int getUsableMoney() :** This method will be called and return the amount of money that user has.
- **public int setUsableMoney() :** This method will be called for setting the amount of money for plate which he/she is able to spend during the game.
- **Public boolean setInJail(boolean) :** Sets the jail status of the player.
- **public ArrayList<Card> getCards() :** This method will be called and return the cards that the user holds.
- **public void removeCard(Card) :** Removes the given card from the player's cards list.
- **public Token getToken() :** This method will be called and return the taken token by player.
- **public PlayerColor getColorId() :** Returns the color id of the player.
- **public Location getCurrentLocation() :** This method will be called and return location where the player is.

- **public void setCurrentLocation(int location) :** This method will be called and change the location where the player is.
- **public ArrayList<Buyable> getOwnedLocations() :** Returns the list of locations owned by this player.
- **public Location addNewLocation(Buyable) :** Adds the given location to the list of player's owned locations.
- **public void removeOwnedLocation(Buyable) :** Removes the given location from the players owned locations list.
- **public int getJailCounter() :** Returns the number of turns the player has been in the jail.
- **public int getJailCounter() :** Returns the number of turns the player has been in the jail.
- **public void setJailCounter(int) :** Sets the number of turns the player has been in the jail.
- **public BankAccount getBankAccount() :** Returns the bank account of the player.

PlayerManager Class

Attributes:

- **private PlayerManager playerManager:** An instance of this Singleton class.
- **private ArrayList<Player> players :** This attribute will be storing the players of the game.
- **private int currentPlayer:** The player whose turn is now playing the turn.

Methods:

- **public Player getCurrentPlayer() :** This method will be called for returning which player plays the game now.
- **public ArrayList<Player> getPlayers() :** This method will be called for returning the list of players that plays the game.
- **public Player getPlayerByID(id) :** Returns player with the given id.
- **public void bankruptPlayer(Player) :** Sets the bankrupt status of the player and removes them from the game.
- **private void removePlayer(Player) :** Removes the given player form the game.
- **public void changeCurrentPlayer() :** This method will be called and execute changing turn instructions.

Map Class

Attributes:

- **private Image mapImage :** This attribute will be storing the board image of the game.

InteractionArea Class

Attributes:

- **private JButton jDiceRollButton :** This attribute will be storing the button that is used for rolling dice in the game.
- **private JButton jBuyButton :** This attribute will be storing the button that is used for buying property in the game.
- **private JButton jOptionsButton :** This attribute will be storing the button that is used for going options the game.
- **private JButton jSelectpropertyButton :** This attribute will be storing the button that is used choosing property in the game.

SubPanel Interface

Attributes:

- **private JPanel jpanel :** This attribute will be storing the panel of the game scene.

GameManager Class

Attributes:

- **private static GameManager gameManager :** This attribute will be a constructor of GameManager class and store a static object of it.

Methods:

- **public static GameManager getInstance() :** This method will return the static object of GameManager.
- **public void handleNewTurn() :** This method will be called and executes going for new turn instructions.

- **public void executePurchase() :** This method will be called for buying a location.
- **public boolean payRent() :** This method will be called for paying the rent of location he/she stands.
- **public void movePlayer(Player player, int num) :** This method will be called in order to change the location of the player based on his/her dice.
- **public void askPlayerRentPaymentChoice() :** This method will be called in order to ask in which way a player wants to pay the rent.
- **public void checkPassingStartReward(Player player) :** This method will be called in order to check whether a player passed the start point or not.
- **public void endTurn() :** This method will be called for ending a turn and starting it from the beginning .
- **public void handleRerollingForAnotherMovement(Player player) :** This method will be called for rerolling dice after the player pays the money.

InventoryScreen Class

Attributes:

- **private JLable moneyLbl :** This attribute will be a JLebel for showing the money on the screen.
- **private JTable propertyList :** This attribute will be a JTable for showing a list of property on the screen.
- **private JTable cardList :** This attribute will be a lebel for showing the money on the screen.

Methods:

- **public Property getPropertyById(int id) :** This method will be called for returning the desired id with given int.
- **public ArrayList<Property> getPropertyList() :** This method will be called for returning all the properties in the game.

DiceManger Class

Attributes:

- **private static DiceManger diceManager :** This attribute will be a constructor of DiceManager class and store a static object of it.
- **private Dice dice :** This attribute will be a constructor of Dice class and store a static object of it.

Methods:

- **public static DiceManager getInstance() :** This method will return the static object of DiceManaher.
- **public void rollDiceToMove() :** This method will be called for rolling dice in order to move.
- **public void rollDiceToAvoidRent() :** This method will be called for rolling dice in order to avoid paying rent.

Dice Class

Attributes:

- **private int firstDiceResult :** This attribute will be storing first result of dice.

- **private int secondDiceResult :** This attribute will be storing second result of dice.
- **private int totalResult :** This attribute will be storing total of first and second dice results.

Methods:

- **public void rollDices() :** This method will be called for rolling dices.
- **public int getResultFirstDice() :** This method will be called for returning first dice result.
- **public int getResultSecondDice() :** This method will be called for returning second dice result.
- **public int getTotalResult() :**This method will be called for returning total of dice results.
- **public boolean isDoubleDice() :**This method will be called for finding out whether dice is double or not.

BoardManager Class

Attributes:

- **private static BoardManager boardManager:** This attribute will be a constructor of BoardManager class and store a static object of it.
- **private InteractionArea interactionArea :** This attribute will be a constructor of InteractionArea class and store a static object of it.
- **private Map map :** This attribute will be a constructor of Map class and store a static object of it.

- **private RentChoicePrompt rentChoicePrompt :** This attribute will be a constructor of rentChoicePrompt class and store a static object of it.

Methods:

- **public static BoardManager getInstance() :** This method will return the static object of BoardManager.
- **public void updateMap(PlayerManager) :** This method will be called for updating the Map.
- **public void updateInteractionArea(Dice) :** This method will be called for updating the interaction area on the screen.
- **private void pauseGame() :**This method will be called for executing the pause game instructions.
- **private void updateDiceResualt(Dice) :**This method will be called for updating dice results on the board.
- **private void displayRentChoicePrompt() :**This method will be called for displaying rent choice.
- **private void hideRentChoicePrompt() :**This method will be called for hiding rent choice.

Location Manager Class

Attributes:

- **private static LocationManager locationManager :** This attribute will be a constructor of LocationManager class and creates a static object of LocationManager.

- **private ArrayList<Buyable> buyableLocations :** This attribute will be storing all places where players are able to buy.
- **private ArrayList<Location>nonBuyableLocation :** This attribute will be storing all places where players cannot buy.

Methods:

- **public static LocationManager getInstance():** This method will return the static object of LocationManager.
- **public ArrayList<Buyable> getBuyableList() :** This method will be called for returning all places which players are able to buy in the game.
- **public ArrayList<Location> getLocationList() :** This method will be called for returning all places which players cannot buy in the game.
- **public boolean groupHvaeSameOwner(int groupId) :**This method will be called for understanding whether or not a group of buyable locations have the same owner or not .
- **public int noOfOwnedByPlayerInGroup(Player player, int groupId) :** This method will be called for getting the number of places which is owned by a player in a specific group.

Location Class

Attributes:

- **private int locationId :** This attribute will be storing the location id of the location on the board.
- **private String name :** This attribute will be storing the name of the location

- **private Point2D position :** This attribute will be storing the position of the location on the board.
- **private ArrayList<Player> playersHere :** This attribute will be storing the position of players on the board in an arraylist.

Methods:

- **public void active():** This will be called when a player lands on a location.
- **public int getLocationId() :** This method will be called for returning location id of each location on the board.
- **public String getName() :** This method will be called for returning the name of each location on the board.
- **public Point2D getPosition() :**This method will be called for returning the position of each location on the board .
- **public void addPlayerHere(Player player) :** This method will be called for adding a player into a location.
- **public void removePlayerHere(Player player):** This method will be called for removing a player from a location.
- **public int getNoOfPlayersHere():** This method will be called for returning the number of players in a location.
- **public Arraylist<Player> getAllplayersHere():** This method will be called for returning all players who are in a location.

Buyable Class

Attributes:

- **private GroupColor groupId :** This attribute will be storing the id of each group color.
- **private Player owner :** This attribute will be storing the owner of each location which players are able to buy.
- **private int price :** This attribute will be storing the price of the location which players are able to buy.
- **private int currentRentValue :** This attribute will be storing the rent of the location which players have to pay at that moment.
- **private int mortgageValue:** This attribute will be storing the mortgage value of the locations which players are able to buy.
- **private int mortgageValue:** This attribute will be storing the break mortgage value which player has to pay for the locations.
- **private boolean mortgageValue:** This attribute will be storing whether or not a location is under mortgage.

Methods:

- **public int getPrice():** This method will be called for returning the price of location which players are able to buy.
- **public void setPrice(int price) :** This method will be called for setting the price of location which players are able to buy.
- **public int getCurrentRentValue() :** This method will be called for returning the rent of the location at that moment.
- **public void setCurrentrentValue() :**This method will be called for setting the rent for a location which players are able to buy .
- **public Player getOwner() :** This method will be called for returning the owner of a location if it has.

- **public void setOwner(Player player):** This method will be called when a player buies a place and he/she is the owner of that location.
- **public int getMortgage():** This method will be called for returning the mortgage value of a location which has mortgage value.
- **public int getBreakMortgage():** This method will be called for returning the price of the break mortgage of a location.
- **public int setBreakMortgage():** This method will be called for setting the break mortgage value for a location.
- **public boolean isUnderMortgage():** This method will be called for finding out a location is mortgaged or not.
- **public boolean setUnderMortgage():** This method will be called for changing the condition of location from mortgage to unmortgage or vice versa.

Property Class

Attributes:

- **private int[] rentValues:** This attribute will be storing values of all properties in an integer array.

- **private int vendingMachineNo :** This attribute will be storing the number of vending machines that exist in a property .
- **private boolean ownsStarbucks :** This attribute will be storing whether or not a property has Starbucks or not.

Methods:

- **public void active():** This method will be called when a player land in a property.
- **public boolean upgrade() :** This method will be called for upgrading a property which includes an increase in its rent, add vending machines or Starbucks and..
- **public boolean degrade() :** This method will be called for degrading a property which includes an decrease in its rent, remove vending machines or Starbucks and.. .
- **public int getVendingMachineNo(int groupId) :**This method will be called for returning the number of vending machines in a property.
- **public boolean hasStarbucks(Player player, int groupId) :** This method will be called for finding out whether or not a location has Starbuck.
- **public int getRentValue(boolean) :** This method will be called for returning the rent value of a property if it has an owner.

IncomeTaxTitle Class

Attributes:

- **private int TAX_VALUE :** This attribute will be storing the fixed amount of money which players have to pay for each trading.

Methods:

- **public void active():** this method will be activated when players want to trade anything between each other.
- **public void deductTax(player):** this method will be called for deducting tax from a player.

StartTitle Class

Attributes:

- **private int PRIZE_VALUE :** This attribute will be storing the fixed amount of money which is paid to players when they pass the start point for three times.

Methods:

- **public void active():** this method will be activated when players pass the start point.
- **public int getPrize() :** This method will be called for returning the amount of money which players give when they pass the start point for three times.

Jail Class

Attributes:

- **Private ArrayList<Player> prisoners :** This attribute will be storing the players who are in the jail.

Methods:

- **public void active():** this method will be activated when players are in the jail.
- **public void freePlayer(Player player) :** This method will be called when a player must be free from jail.
- **public void arrestPlayer(Player player) :** This method will be called when a player must be in the jail.

Jail Class

Attributes:

- **Private ArrayList<Player> prisoners :** This attribute will be storing the players who are in the jail.

Methods:

- **public void active():** this method will be activated when players are in the jail.
- **public void freePlayer(Player player) :** This method will be called when a player must be free from jail.
- **public void arrestPlayer(Player player) :** This method will be called when a player must be in the jail.

ChanceTitle Class

Methods:

- **public void active():** this method will be activated when players must draw a chance card.

MayfestTitle Class

Attributes:

- **Private int collectedTax :** This attribute will be storing taxes.

Methods:

- **public void active():** this method will be activated when players are in the Mayfest.
- **public void addTax(int) :** This method will be called for adding the amount of taxes to the collectedTax attribute.
- **public void giveTaxMoney(Player player) :** This method will be called for giving the tax money to a player who stands in Mayfest.

BusStop Class

Attributes:

- **Private int[] rentValues :** This attribute will store all rent value of bus stop locations in an array list

Methods:

- **public void active():** this method will be activated when players are in the BussStop locations.

BusStop Class

Attributes:

- **Private int[] rentValues :** This attribute will store all rent value of Utility locations in an array list

Methods:

- **public void active():** this method will be activated when players are in the Utility locations.

3.2. Packages

3.3.1 Java.util

This package contains some of the data types, such as ArrayList, that are used inside the classes. ArrayList is used in most of the classes to hold player objects, properties, cards and etc. The IO package will be used for saving and loading the game and related information to the system file. Another subpackage, Random, is used for generating random dice values.

3.3.2. Javax.swing

This package provides classes and interfaces and components needed for rendering UI objects.

3.3.2. Java.awt

This package provides a set of classes to handle painting graphics, images and colors.

3.3.2. Java.awt.event

This package defines a set of classes and interfaces to handle events.

3.3.2. Java.awt.event

This package defines a set of classes and interfaces to handle the various types of events fired by awt components.

3.3.2. Java.awt.graphics

This package defines a set of classes and interfaces that allow the application to draw on the components.

3.3.2. Java.awt.ActionListener

This package provides a set of classes that are responsible for handling user interactions.

3.3. Class Interfaces

3.4.1. MouseListener

This interface will be responsible for listening and responding to the mouse activities fired by the user. It is extensively used in the menus and game board to track mouse events and respond accordingly.

3.4.2. KeyListener

This interface will be responsible for listening and responding to the keyboard commands and activities initiated by the user. It is extensively used by the menus and game board to receive keyboard events and respond accordingly.

3.4.3. ActionListener

This interface will be responsible for listening to the actions that are happening in the game and invoke the corresponding event handling mechanism.

3.4.4. Serializable

This interface will be used to serialize a game object. It will be responsible for converting the game object to byte stream so that it can be stored in the file system.

Glossary & references:

[1]. https://www.youtube.com/watch?v=Nsjsiz2A9mg&t=3107s Access Date: 29.11.2020

[2]. https://www.youtube.com/watch?v=qzTeyxIW_ow Access Date: 29.11.2020

[3]. https://martinfowler.com/eaaDev/uiArchs.html Access Date: 29.11.2020

[4]. https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter#Java Access Date: 29.11.2020