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

Introduction **3**
Purpose of the system 3
Design goals 3
Trade-Offs 3
Criteria 4
High-level software architecture **5**
Subsystem decomposition 5
Hardware/software mapping 9
Persistent data management 9
Access control and security 9
Boundary conditions 10
Low-level design **12**
Final object design 12
Packages 31
Class Interfaces 32
Glossary & references: **33**

1. Introduction

1.1. Purpose of the system

Bilopoly is a virtual version of Monopoly. However, it is different from the original version of Monopoly and some rules and buildings are changed with new ones. Its design is inspired from Bilkent university and most of the buildings are named based on that. It is designed in the way that bilkenters experience the most satisfaction, while students from other universities are able to entertain because of its atmosphere which allows players to feel like they are managing a university. The aim of the game is challenging players, while entertaining them and letting them experience the maximum joy from the virtual Bilkent version of Monopoly. Therefore, this game by offering a user friendly atmosphere and high performance engine and fast running system allow users to achieve their aim which is to survive in a capitalist university.

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 one of the systems 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.

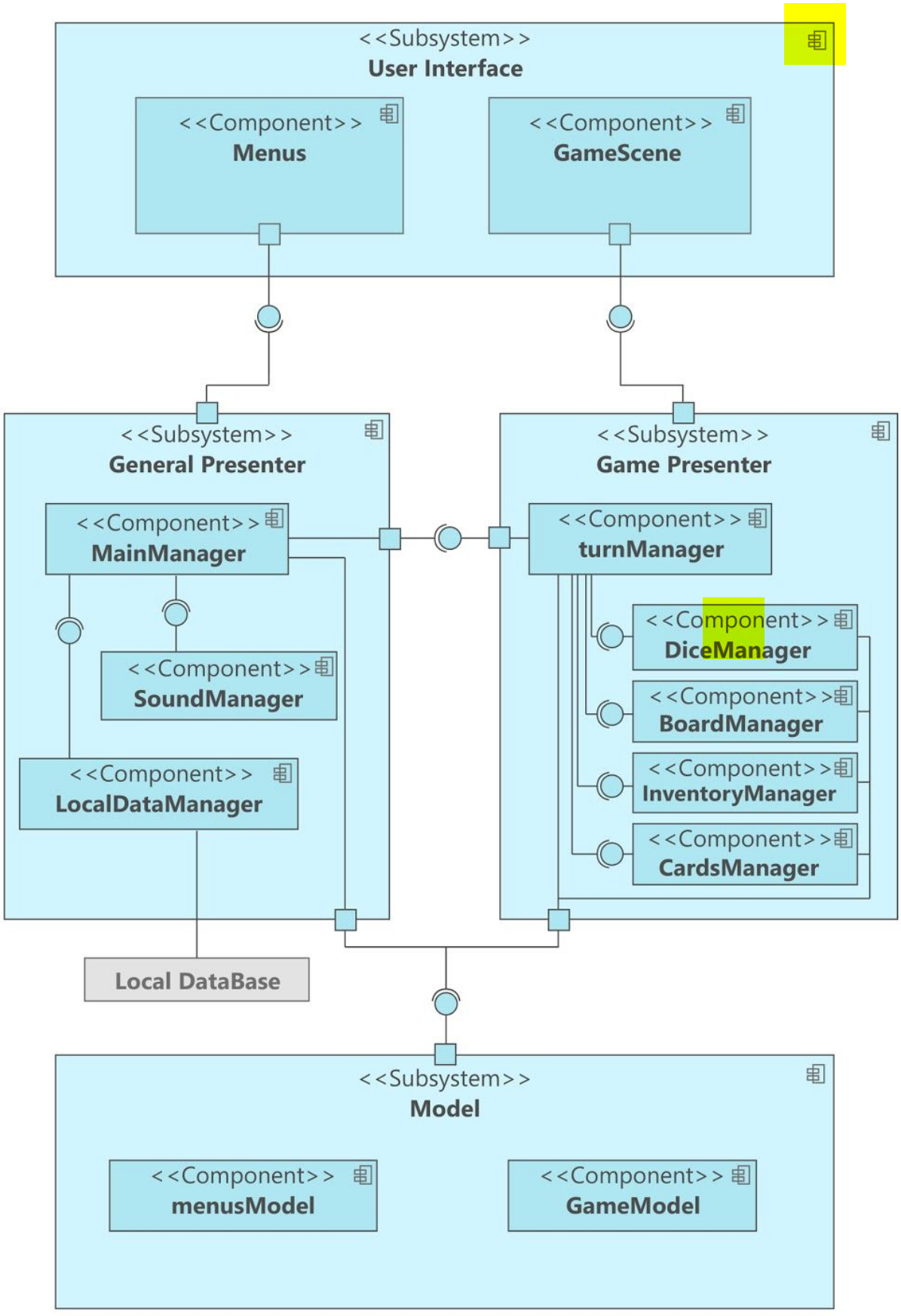
- **Game Presenter** handles every event related to game science like board management, tokens movements, and trades.
- **General Presenter** handles all the logic that is not directly related to the game such as menus switching as well as handling local data, sound volume, and initiating a new game.

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.

In terms of design patterns, we have decided to use the singleton design pattern for the main parts of the system such as the mainManager, soundManager, localDataManager, turnManager, DiceManager, BoardManager, InventoryManager, and cardsManager. Second, the Decorator design pattern will be used for locations to upgrade or degrade them. Finally, the facade design pattern is used to communicate between the subsystems such as between User Interface, GeneralPresenter, Game Presenter, and Model.


<<Subsystem>>
User Interface
<<Component>>
Menus
<<Component>>
GameScene
<<Subsystem>>
General Presenter
<<Component>>
MainManager
<<Component>>
SoundManager
<<Component>>
LocalDataManager
<<Subsystem>>
Game Presenter
<<Component>>
turnManager
<<Component>>
DiceManager
<<Component>>
BoardManager
<<Component>>
InventoryManager
<<Component>>
CardsManager
Local DataBase
<<Subsystem>>
Model
<<Component>>
menusModel
<<Component>>
GameModel

2.2. 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. The game requires a PC computer as well as a keyboard and a mouse as hardware components. The keyboard is going to be used to input some textual information; on the other hand, the mouse is going to be used to input some click events on menu items or on certain locations in game science. Moreover, some sort of physical storage is needed to store the game's local database. These could be an HDD or an SSD. Nevertheless, a network connection is not necessary since the game will be entirely local.

2.3. 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.

2.4. 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.

2.5. 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**

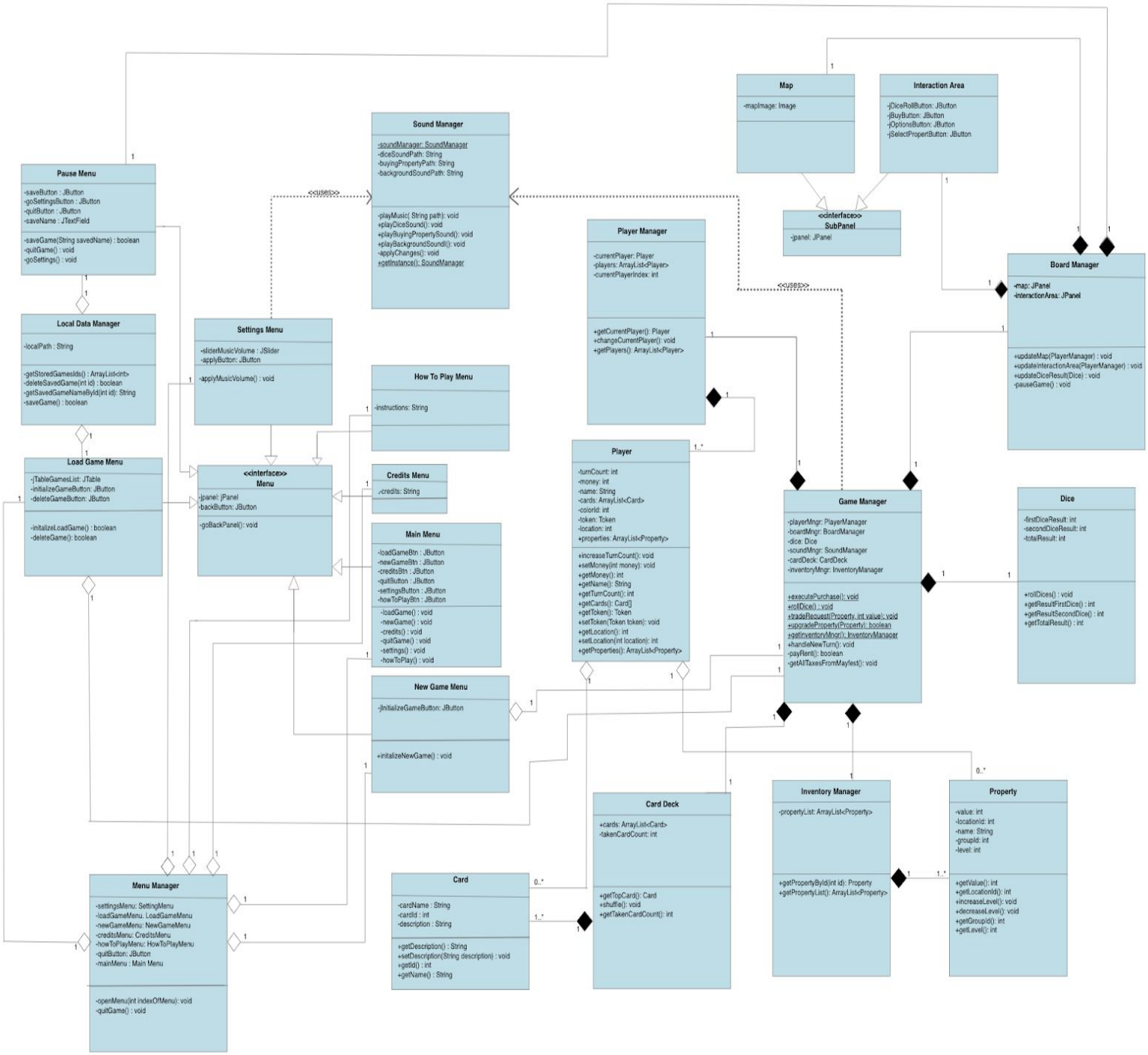
  Failures can happen especially while doing the io operations. Therefore, specific mechanisms will be implemented to prevent game crash when facing a failure. Such mechanisms are the following:

  - 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.

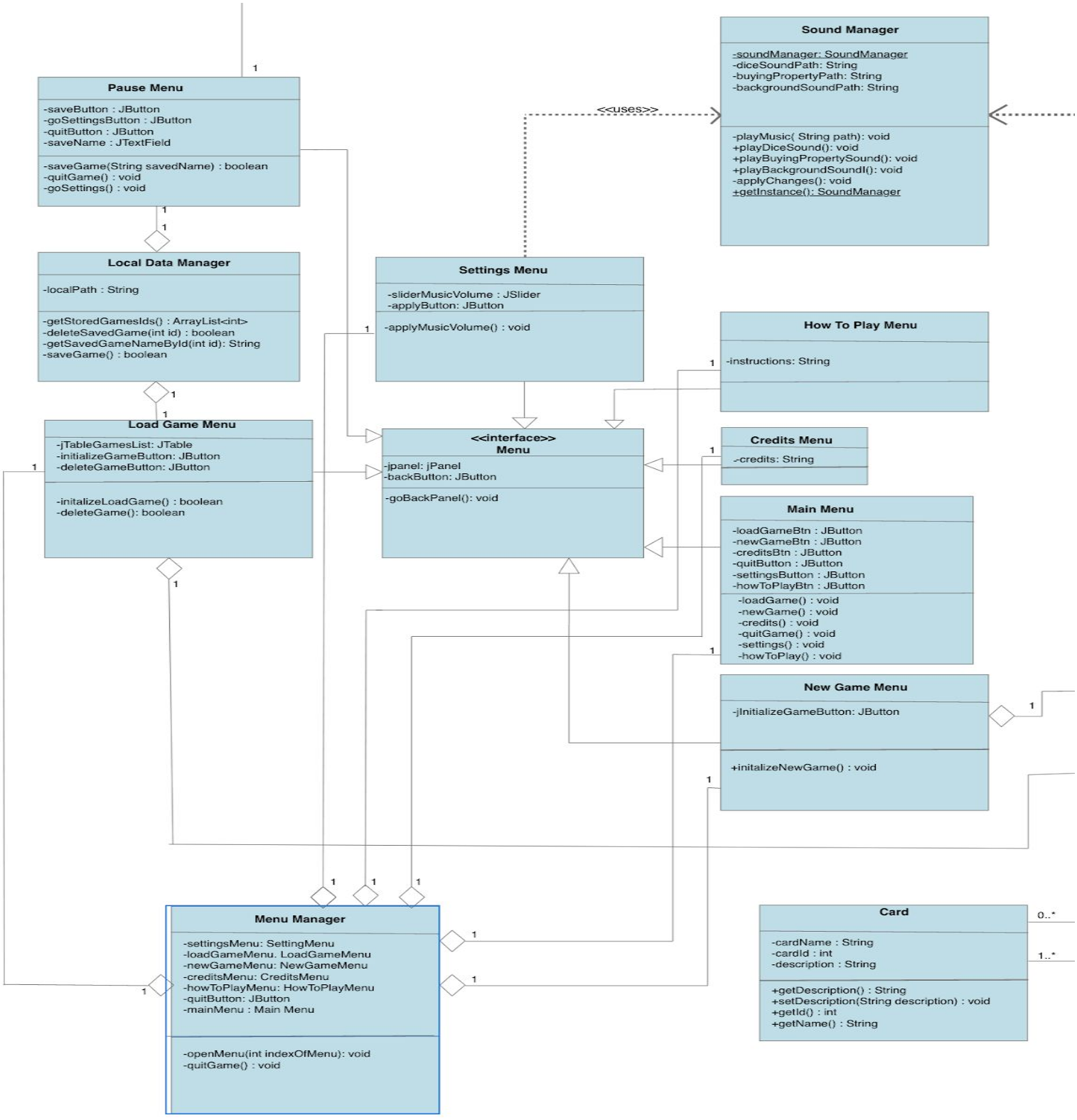
- If the game fails to load the previously saved sessions, the user will get an error message on the saved games list indicating that saved games could not be loaded.
- If the game fails to save the current game after the user request, a prompt will appear indicating the failure and asking the user's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

3. Low-level design

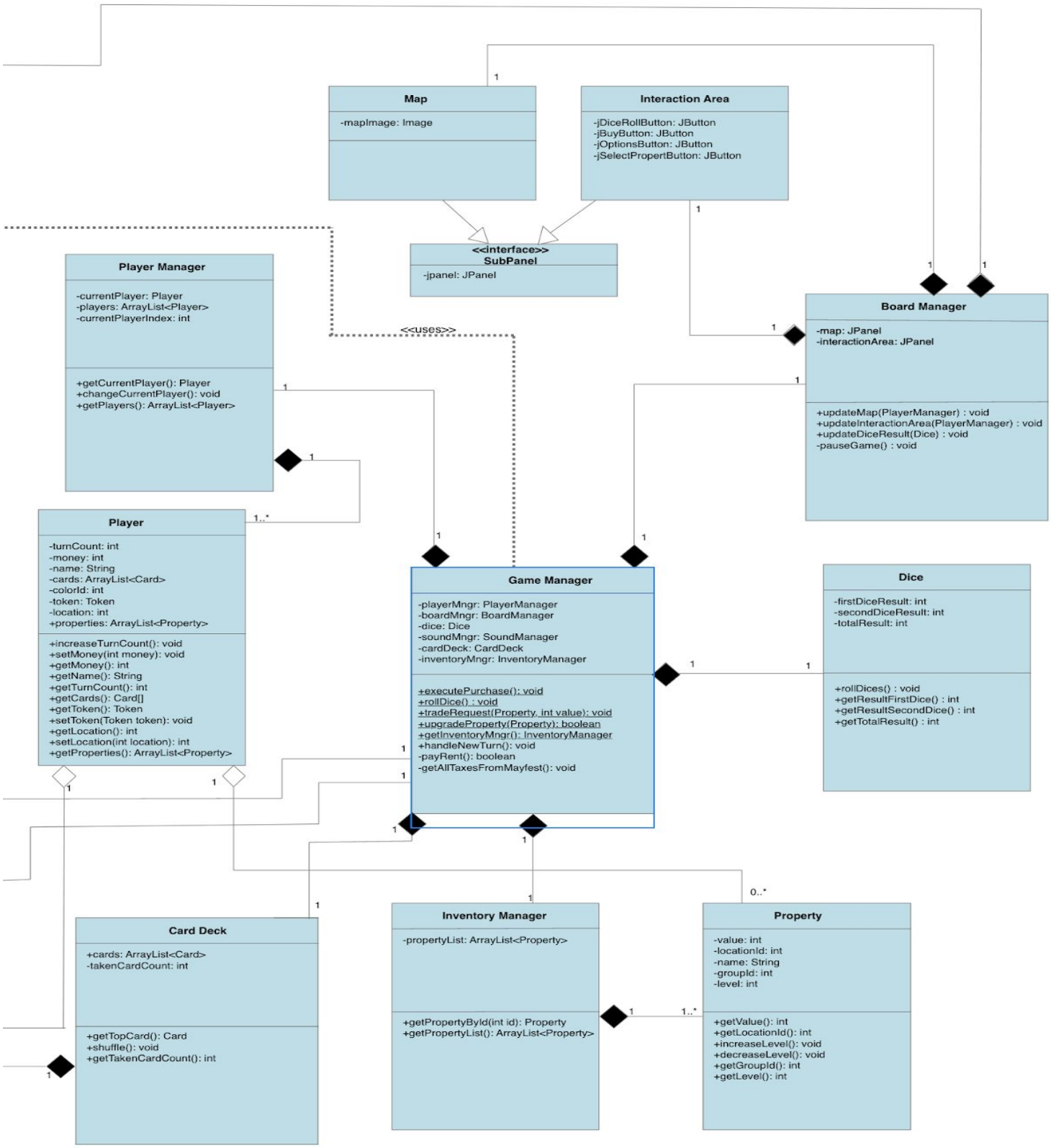
3.1. Final object design



Piece 1


Sound Manager
-soundManager: SoundManager
-diceSoundPath: String
-buyingPropertyPath: String
-backgroundSoundPath: String
-playMusic(String path): void
+playDiceSound(): void
+playBuyingPropertySound(): void
+playBackgroundSoundI(): void
-applyChanges(): void
+getInstance(): SoundManager
<<uses>>
Pause Menu
-saveButton : JButton
-goSettingsButton : JButton
-quitButton : JButton
-saveName : JTextField
-saveGame(String savedName) : boolean
-quitGame() : void
-goSettings() : void
Local Data Manager
-localPath : String
-getStoredGamesIds() : ArrayList<int>
-deleteSavedGame(int id) : boolean
-getSavedGameNameById(int id): String
-saveGame() : boolean
Settings Menu
-sliderMusicVolume : JSlider
-applyButton: JButton
-applyMusicVolume() : void
How To Play Menu
-instructions: String
Load Game Menu
-jTableGamesList: JTable
-initializeGameButton: JButton
-deleteGameButton: JButton
-initalizeLoadGame() : boolean
-deleteGame(): boolean
<<interface>>
Menu
-jpanel: jPanel
-backButton: JButton
-goBackPanel(): void
Credits Menu
-credits: String
Main Menu
-loadGameBtn : JButton
-newGameBtn : JButton
-creditsBtn : JButton
-quitButton : JButton
-settingsButton : JButton
-howToPlayBtn : JButton
-loadGame() : void
-newGame() : void
-credits() : void
-quitGame() : void
-settings() : void
-howToPlay() : void
New Game Menu
-jInitializeGameButton: JButton
+initalizeNewGame() : void
Menu Manager
-settingsMenu: SettingMenu
-loadGameMenu. LoadGameMenu
-newGameMenu: NewGameMenu
-creditsMenu: CreditsMenu
-howToPlayMenu: HowToPlayMenu
-quitButton: JButton
-mainMenu : Main Menu
-openMenu(int indexOfMenu): void
-quitGame() : void
Card
-cardName : String
-cardId : int
-description : String
+getDescription() : String
+setDescription(String description) : void
+getId() : int
+getName() : String
0..*
1..*

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 String cardName :** This attribute will be a string of storing name of the card.
- **private int cardId :** This attribute will be a id of each card.
- **private String description :** This attribute will store the description of each card.

Methods:

- **public String getDescription() :** This method will be called and return descriptions of card.
- **public void setDescription(String description) :** This method will be called for changing description of card.
- **public int getId() :** This method will be called and return id of the card.
- **public String getName() :** This method will be called and return the name of card.

Card Deck Class

Attributes:

- **public ArrayList<Card> cards :** This attribute will be storing cards of the game.
- **private int takenCardCount :** This attribute will be counting how many cards will be drawn.

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 int getTakenCardCount() :** This method will be called and return the taken card count as int.

Player Class

Attributes:

- **private int turnCount :** This attribute will be counting how many turns will be played by the player.
- **private int money :** This attribute will be storing how much money the player has.
- **private String name :** This attribute will be storing the name of the player.
- **private ArrayList<Card> cards :** This attribute will be storing all cards that the user holds.
- **private int 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 location :** This attribute will be storing which location id the user on.
- **public ArrayList<Property> properties :** This attribute will be storing which properties the user has.

Methods:

- **public void increaseTurnCount() :** This method will be called for increasing turn count of the user.
- **public void setMoney(int money) :** This method will be called for changing the money that user has.
- **public int getMoney() :** This method will be called and return the amount of money that user has.
- **public String getName() :** This method will be called and return the name of the player.
- **public int getTurnCount() :** This method will be called and return the turn count that user passed.
- **public ArrayList<Card> getCards() :** This method will be called and return the cards that the user holds.
- **public Token getToken() :** This method will be called and return the taken token by player.
- **public void setToken(Token token) :** This method will be called and set the token of the player.
- **public int getLocation() :** This method will be called and return location where the player is.
- **public void setLocation(int location) :** This method will be called and change the location where the player is.
- **public ArrayList<Property> getProperties() :** This method will be called and returns property list that user has.

Player Manager Class

Attributes:

- **private Player currentPlayer :** This attribute will be showing which player is playing the game now.
- **private ArrayList<Player> players :** This attribute will be storing the players of the game.
- **private int currentPlayerIndex :** This attribute will be storing which index in the player list is playing.

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 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.

Interaction Area 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.

Game Manager Class

Attributes:

- **private PlayerManager playerMngr :** This attribute will be used for managing player issues.
- **private BoardManager boardMngr :** This attribute will be used for managing board issues.

- **private SoundManager soundMngr:** This attribute will be used for managing sound issues.
- **private InventoryManager inventoryMngr :** This attribute will be used for managing inventory issues.
- **private CardDeck cardDeck :** This attribute will be used for managing card issues.
- **private Dice dice :** The attribute will be used for rolling dice.

Methods:

- **public static void executePurchase() :** This method will be called for executing purchase property instructions.
- **public static void rollDice() :** This method will be called for rolling dice.
- **public static void tradeRequest(Property prop, int value) :** This method will be called for trading requests of the users with given money.
- **public static boolean upgradeProperty(Property prop) :** This method will be called for upgrading level of the property.
- **public static InventoryManager getInventoryMngr() :** This method will be returning the inventory manager.
- **public void handleNewTurn() :** This method will be called and executes going for new turn instructions.
- **public boolean payRent() :** This method will be called and executes paying rent to other user where the player on instructions.
- **public void getAllTaxesFromMayfest() :** This method will be called and used for giving all gathered taxes to the player who drawed gather all taxes chance card.

Inventory Manager Class

Attributes:

- **private ArrayList<Property> propertyList :** This attribute will be storing all properties in the game.

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.

Property Class

Attributes:

- **private int value :** This attribute will be storing purchase money of the property.
- **private int locationId :** This attribute will be storing all location number where it is located in the map.
- **private int groupId :** This attribute will be storing group type of property.
- **private int level :** This attribute will be storing level (how many vending machine stores) of the property.
- **private String name :** This attribute will be storing the name of the property.

Methods:

- **public int getValue() :** This method will be called for returning value of the property.
- **public int getLocationId() :** This method will be called for returning location id of the property.
- **public void increaseLevel() :** This method will be called for executing increasing level of instructions.
- **public void decreaseLevel() :** This method will be called for executing decreasing level of instructions.
- **public int getGroupId() :** This method will be called for returning the group type of the property.
- **public int getLevel() :** This method will be called for returning the level of the property.

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.

Board Manager Class

Attributes:

- **private JPanel map :** This attribute will be map of the game.
- **private JPanel interactionArea :** This attribute will be storing interaction area of the game.

Methods:

- **public void updateMap(PlayerManager) :** This method will be called for changing things on the map according to the players' informations.
- **public void updateInteractionArea(PlayerManager) :** This method will be called for changing things on the interaction area according to the players' information.
- **public void updateDiceResult(Dice) :** This method will be called for updating dice results on the screen.
- **private void pauseGame() :**This method will be called for executing the pause game instructions.

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