CS 319 Project Design Report Iteration 1

Monopoly Sicilia

Group 2D

Asya Doğa Özer - 21803479

Oğuz Orhun Tüzgen - 21802313

Ufuk Palpas - 21702958

Uğur Utku Seyfeli - 21802239

Yiğit Harun - 21803241

Table of Contents

Introduction

Purpose of the system

Monopoly Sicilia is an augmented version of the classic board game Monopoly Classic. Since our design is digital and most of the tedious operations are handled by the computer, players get to enjoy the fun parts of the game and focus on becoming the richest person in the world. There are multiple new features to the game: a foreign exchange system where currencies react to purchases of players, the Mafia and the Police as NPCs which interact with both the players and each other, and power-ups which add a new layer of strategy to the game. The game is designed to be a responsive, user-friendly, and accessible party game where people gather around a computer and play their turns in order.

Design goals

Since we have limited time for developing our application, we value the extendibility, reusability of our design with well-documented code. Therefore, the addition of new features and debugging becomes much easier. We would like to make our design robust and reliable since our aim is to make people have fun playing the game without being frustrated with annoying bugs and unexpected system failures. We want our game to be accessible and easy to get into since Monopoly Sicilia is a party game. In addition, because we have multiple workload heavy classes in our schedule, we want the implementation to be light in terms of time cost.

1.2.1 Trade-Offs

Development Time vs Performance:

As you know, many of the developers indicate that Java is very efficient for game development. However, this does not mean that it requires less development time than other languages. It still requires so much effort to complete the project and we are aware that we have very limited time. Therefore, we decided to implement our project in JavaFX. This decreases the development time of the game incredibly.

Functionality vs Usability:

Monopoly is a game that is played by people of various ages and segments. Therefore, the monopoly game needs to have very basic and clear controls. Hence, we kept the interface as much as simple for the players. We decided to try making usability the primary factor of the user interface. Furthermore, “how to play screen” increases the usability of the game by explaining the game and controls simply.

Cost vs Portability:

Because of the fact that we have a limited time to complete the whole game. We preferred to continue with only desktop operating systems that can run java. Therefore, we cut down portability by not developing the game for mobile platforms.

Understandability vs Functionality:

Because of the fact, our game addresses a huge part of society, we emphasized the usability of the game. Thus, the understandability of the game is significant for us. To do this, we removed some considerably complex rules (such as auctions, etc.) that can be seen in the classic monopoly to make it more understandable and funny for everyone.

Functionality vs Robustness

Due to the structure of the monopoly game, we believe that if the implementation of the game can be designed successfully, we can maintain its robustness even if we increase

its functionality. Because in Monopoly our game there are no actions depending on the trial and error method. If each operation is coded faultless both can be provided.

1.2.2 Criteria

End-User Criteria

Usability:

Monopoly Sicilia is a game that addresses a large variety of player types thus we decided to go with a user-friendly user interface to make it more usable for each kind of player. To do that we decided to have functional but simple buttons to play the game so that people who did not read the “how to play” section can also learn the gameplay of the game easily. Therefore, players will understand many of the operations and features of the game by only combining their logic and button names.

Performance:

Performance is a primary factor for our game because if a single player waits too long for the turns of the bots or the operations he/she makes takes too long, it can spoil the fun of the game.

Maintenance Criteria

Extendibility:

The design of the game allows us to add new features in the future according to user feedback and our liking. We can easily add new power-ups, new power-up types, new tile types into our game in the future because of our design.

Modifiability:

Monopoly Sicilia game is a hot-seat and single-player game. Due to its structure, a multiplayer mode can also be added easily. Furthermore, because of its separated design structure, new

functionalities can be added easily to the game without causing errors in other subsystems. Therefore, our game is open to modify existing functionalities and adding new ones.

Reusability:

Some of our classes and most of our manager classes are independent of the game so that they can be used in different projects with little to no change.

Portability:

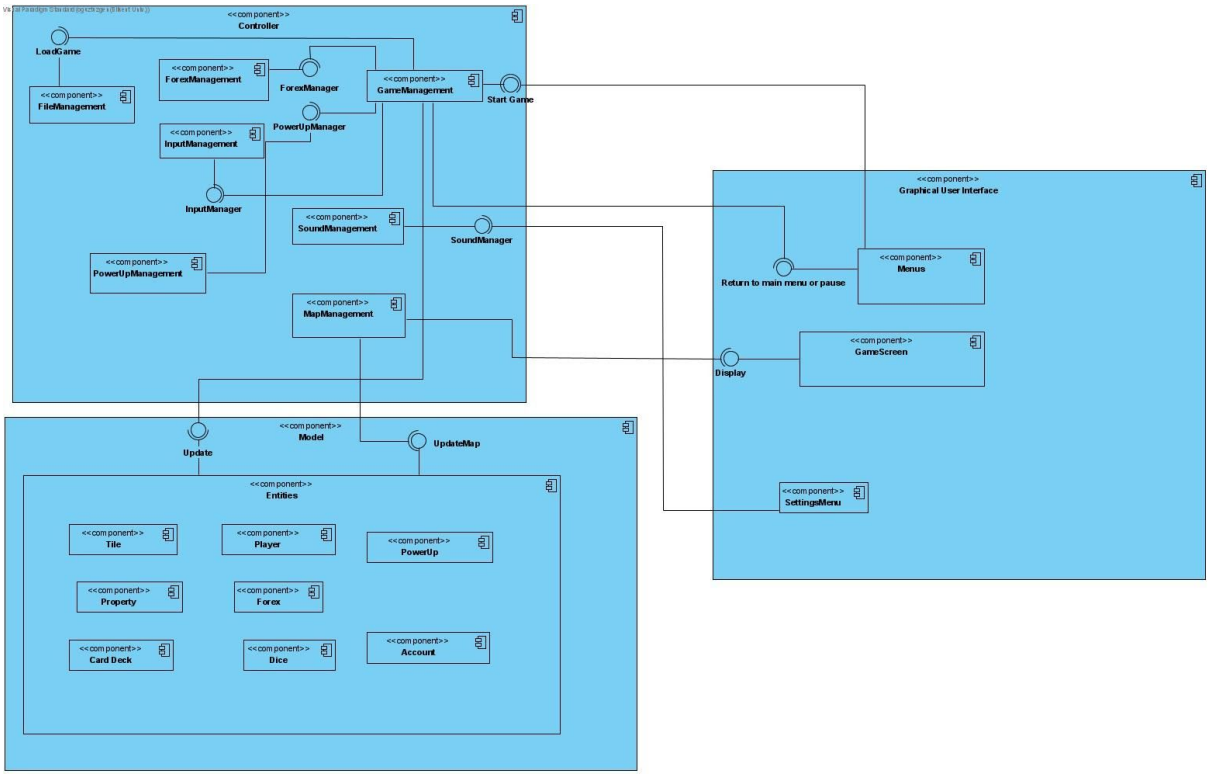
Since we have limited development time, we are not going to focus too much on porting our game to various platforms like mobile and gaming consoles. However, since we are using JavaFX, our game will hopefully be available on Linux, Mac, and Windows.

Performance Criteria

Response time: We would like our game to respond to boundary commands under one second. During gameplay, we would like to obtain a minimum of 30 frames per second render rate for responsiveness.

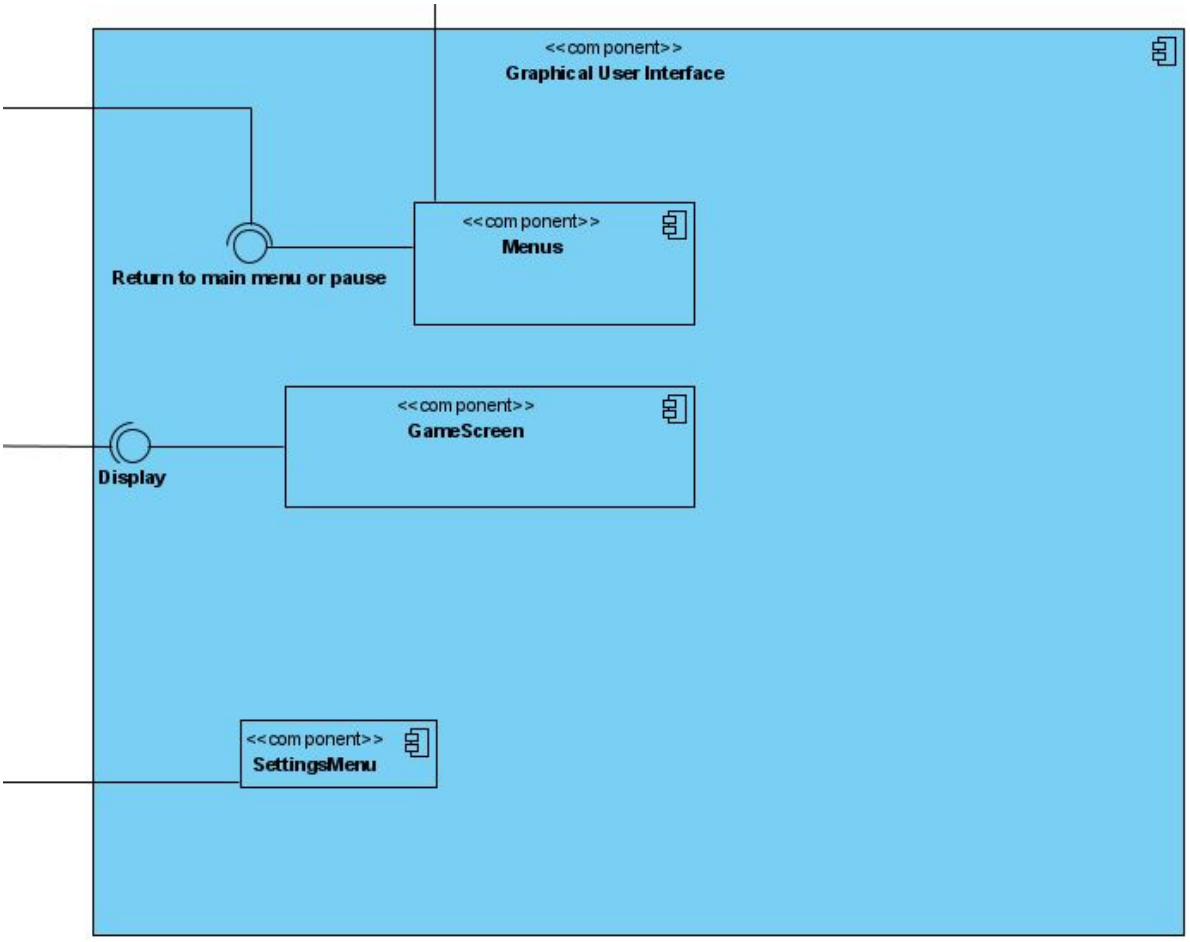
High-level system architecture

Subsystem decomposition



In our decomposition of the subsystems, we have decided to use the Model View Controller (MVC) design pattern to divide the system into subsystems by their functionality and see the relationships and dependencies. In our approach, we tried to increase the coherence of the system by reducing coupling them. This approach makes adding or modifying features a lot easier. We divided our system into three subsystems which correspond to model, view, and controller models respectively. These subsystems are called Model, Graphical User Interface, and Controller respectively. Model subsystem is composed of primitive game objects and updates these entities by the signals of the Control subsystem. Control subsystem manages the game states and interactions between the GUI and Model subsystems. This subsystem also manages the file system.

Graphical User interface Subsystem



GUI Subsystem is responsible for providing interfaces to the game.

It has 3 components that are Menu, GameScreen, SettingsMenu.

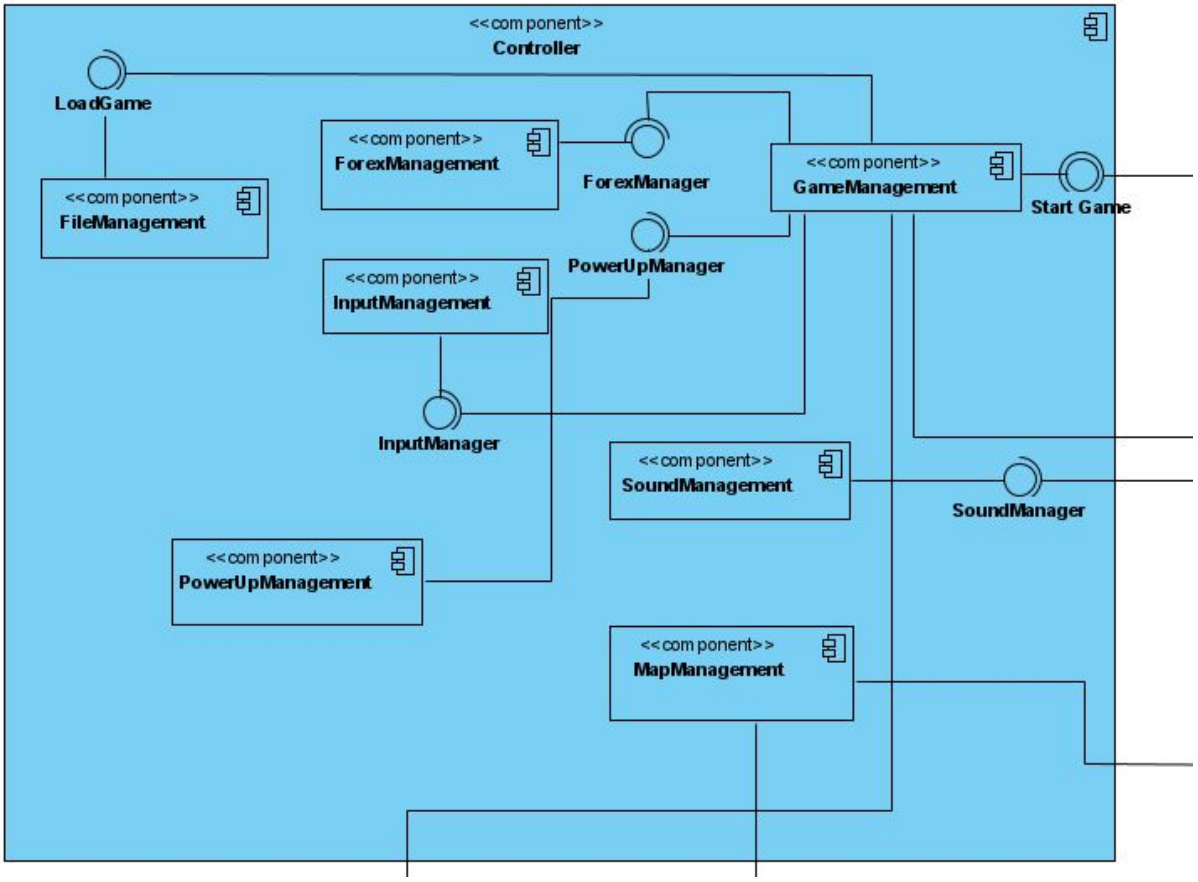
When the user starts the game a menu will be displayed. Menu has different sub-menus that have different roles such as settings menu or load screen and such. Menu can invoke and or instantiate other components like starting a game.

Settings Menu Component provides the user with the settings for the game that he can change according to his demands. This updates the sound manager, map manager, and game manager components from the controller subsystem.

Game Screen Component provides the view of the gameplay. It gets updated by the game manager and the map manager components from the controller subsystem.

As per the Façade Design, we made it so that the settings menu and the game screen can only be instantiated from the menu components

Controller Subsystem



Controller Subsystem is responsible for controlling the gameplay.

It has 7 components that are: Game Management, Forex Management, Input Management, Sound Management, Map Management, File Management, Powerup Management.

Powerup Management Component handles the powerups that are played, in effect and discarded. It updates the Game Manager Component. It is invoked by the Game Manager Component at the start of the game to track any possible power-ups that may be drawn.

File Management Component handles the save/load functions and loading in the assets that are needed for the game. It is updated by the Game Management Component while saving and it updates the Game Management while loading a previously saved game. It is called at the start of the game by the Game Manager Component to get the necessary assets from the file system such as pictures and music.

Sound Management handles the music that is playing and the sounds that should be played when certain triggers have happened. It is called by the Game Management Component at the start of the game to play the sounds. It is updated by the Game Management Component at the start to get the sound settings that have been done by the user at the settings menu.

Input Management handles the inputs from the user and it sends them to the Game Management Component. It updates the Game Management Component with the inputs from the user. It is called at the start of the application to get the inputs from the user.

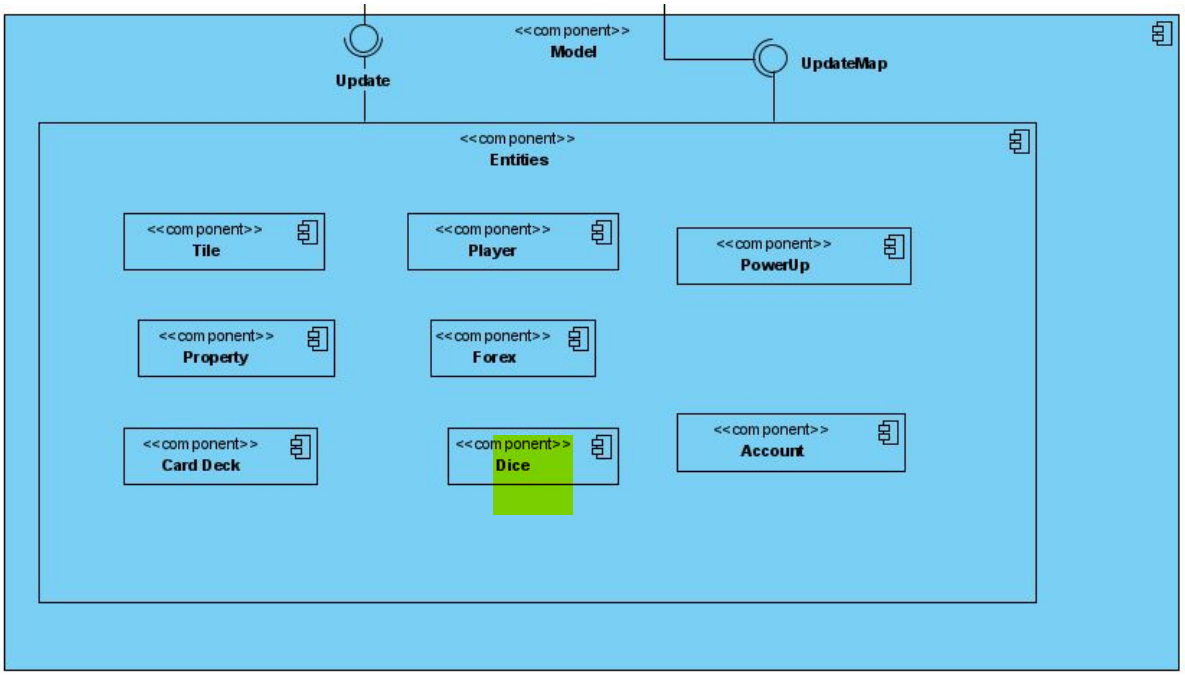
Forex Management handles the changes to the Forex system that is being done in the gameplay. It updates the Game Management Component with the changes to the Forex system. It is called by the Game Management Component at the start of the game to initialize the Forex values and it is updated with every turn in the game. It uses the singleton design to prevent unnecessary changes and double change requests to the Forex system

Map Management handles the creation and the changes made to the map. It updates the Game Management Component at the start of the game to provide the created map and it updates the Game Management Component every turn to give the changes made to the map. It is called at the start of the game by the Game Management Component and it continues

throughout the gameplay. It uses the singleton design to prevent errors while updating the changes and stop unnecessary changes through the gameplay.

Game Management handles the whole gameplay. Nearly every management component updates it and it calls every other management component. It is the main component that handles the gameplay. It updates the Game Screen Component in the GUI Subsystem. It also updates most of the manager components to maintain the gameplay. It updates them on every interaction. It is called the Menu Component when a game is started. It uses the singleton design because there should be only one Management that handles the gameplay. If it were to divide into multiple pieces there would be communication errors while updating the screen and other components.

Model subsystem



Model Subsystem is responsible for the entities. They are the objects of the game. This subsystem has a component called Entities that holds all the entities that are in the game. All game entities are updated by the Game Management Component from Controller Subsystem.

It has 7 components that are: PowerUp Component, Account Component, Dice Component, Player Component, Property Component, Tile Component, Card Deck Component.

PowerUp Component handles the power-ups and their effects.

Account Component handles the accounts of every player and their functions.

Dice component handles the dice rolls.

Player Component handles the information about the players and their actions.

Property Component handles the properties and the information about them.

Tile Component handles the tiles of the map and their functions.

Card Deck Component handles the chance and community chest cards of the game

All of these components are called at the start of the game by the Game Management Component from Controller Subsystem.

Hardware-software mapping

We will implement our game in Java language. We decided to use the JavaFX library in our game. Since the program is going to be written in Java language, Java Runtime Environment (JRE) will be required to run our game as a software requirement. Also in order to support the JavaFx library, JRE has to be at least at the 8th version since we are going to use JavaFX 8 on our project. The operating system does not matter in order to run our game, only the software requirement is JRE 8.

As a hardware requirement, our game only needs a computer with a mouse and a keyboard. Keyboard support isn't going to be complicated since Monopoly is an easy game that does not require any complicated mechanics. The keyboard is only going to be used when typing the player names or city names. For playing the game, there are going to be graphical buttons for rolling the dice, selling and buying properties, dealing with mafia, etc. All

of these buttons can be clicked with a mouse and they invoke the respective functionalities. Since our game is a very simple one, a computer that can display properly, which has a mouse, keyboard, and which has JRE 8 installed can run our game.

We are not going to use a database system in order to store saved game files. Those files would be saved inside folders. So storage is going to be physical and will not require an internet connection.

Persistent data management

In Monopoly Sicilia, we are not going to use database systems, cloud storage, or any other third-party services. The game will save the game state- everything which the game manager possesses- in a formatted manner in a subdirectory called local\game\. In addition, the user preferences (e.g. settings) will also be stored in a subdirectory called local\settings\. The external images and sound effects will be stored inside a subdirectory called vendor\image\ and vendor\sound\ respectively. The images will have the JPEG format and the sound effects will have the WAW format.

Access control and security

Since our game is not a game that uses the internet to make users play from different places, there will not be any access control or security measurement according to internet and database services. But there are going to be save files, which can be changed in order to modify money, house number, etc. In order to prevent this, we are going to make the naming inside the save files encrypted thus the average user cannot read it and change it.

Boundary conditions

Since we are using Java as our main development language, we would like to use the language's generating executable JAR file feature. The user can start the game by starting

this file with the .jar extension. A game of Monopoly Sicilia can be exited in four different ways. The first one is to exit the game using the menu buttons and exiting properly. In this case, we would like to ensure that the game saves the state to a subdirectory properly. The second possibility is that the player may use the quit button. In this case, we will ask the user if they really want to quit to prevent accidental quits and if the player confirms, the game will ensure that the game state is saved properly and terminates the process. The third case is when a player quits the game

Low-level design

Object design trade-offs

For the sake of simplifying the design process, we used managers for almost every entity. Therefore in order to perform an entity-related operation, we have to make a bunch of nested function calls which results in extra memory usage. In addition by applying the Singleton design pattern, we created strong couplings between our entity manager classes and game manager.

Decision and Design Patterns

In the low-level design, we have used some design patterns. These design patterns are going to have their own tradeoffs.

Façade Design Pattern

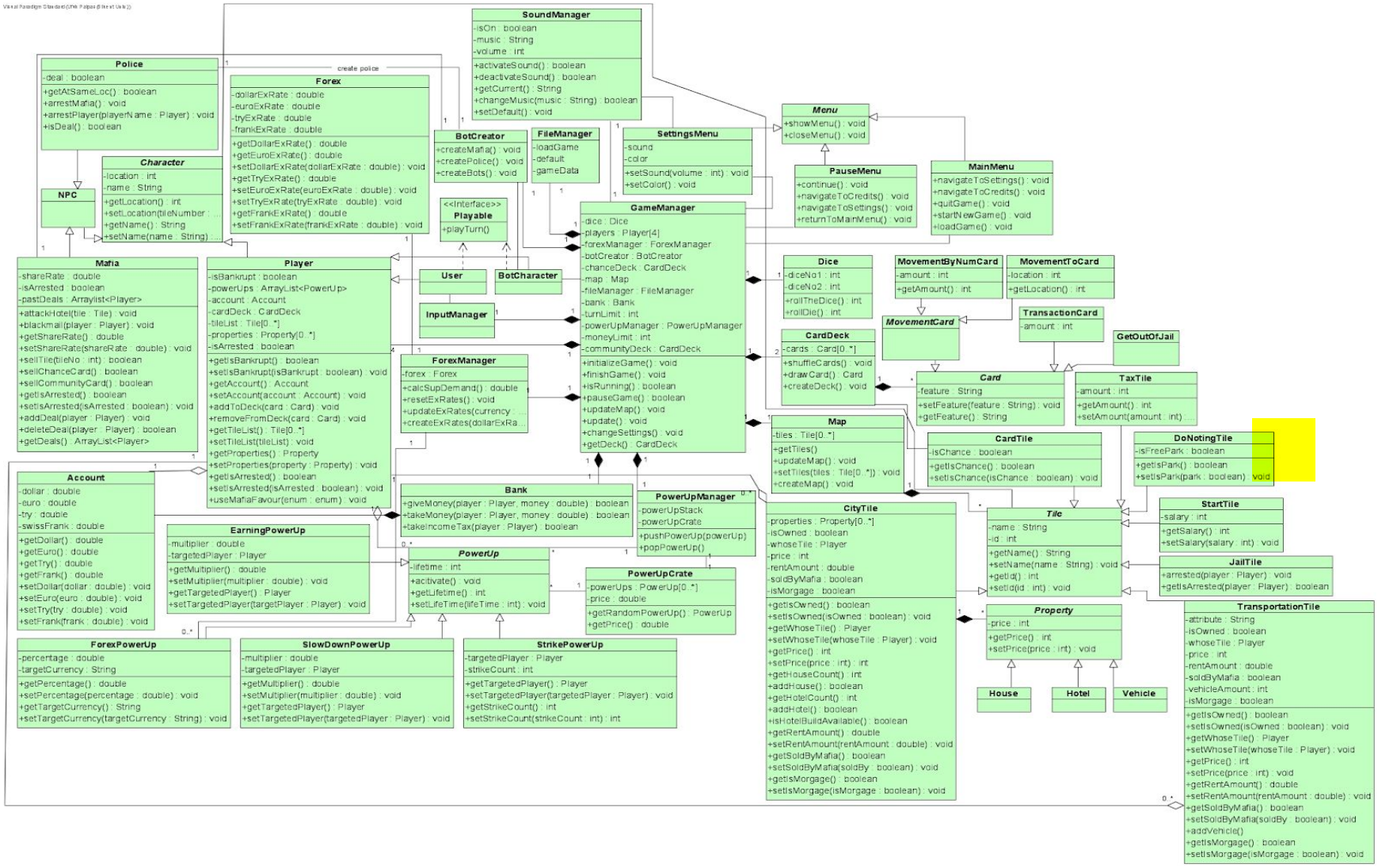
The Façade design pattern is a structural design pattern and we used it to reduce the complexity of the system. We created some interfaces to manage the existing objects by using the Façade design pattern. We used Façade design patterns to group some common methods. This generally appears in-game management. Façade design can be seen clearly

from the interfaces in the design. Although it requires some extra method calls and memory usage, it has enough advantages to implement the design.

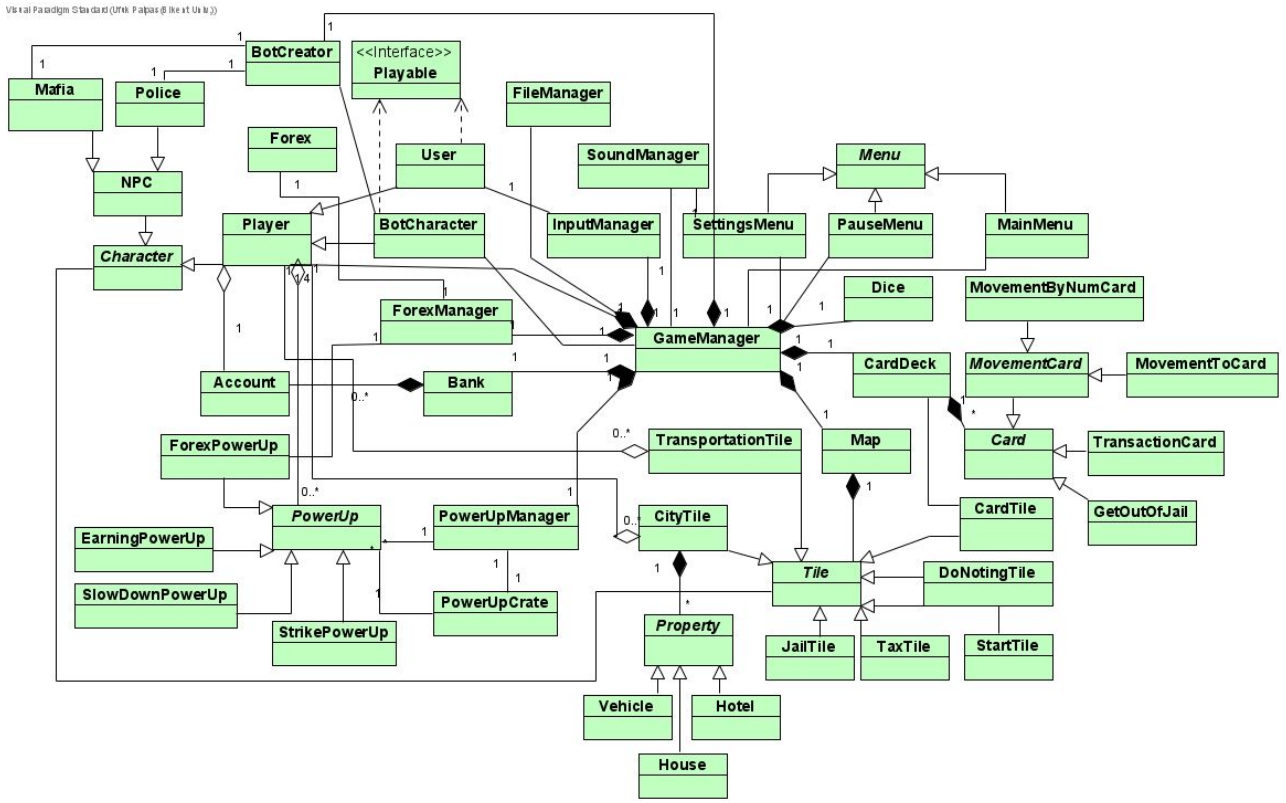
Singleton Design Pattern

To reduce problems we have used the singleton design pattern on our manager classes There will be only one game so if we had multiple game managers there would be possible errors while changing scenes or variables. Also, there is only one forex system so having multiple forex managers would prove meaningless to the point that it would cause complications. However, this design choice made testing harder and it made the game more reliant on one manager.

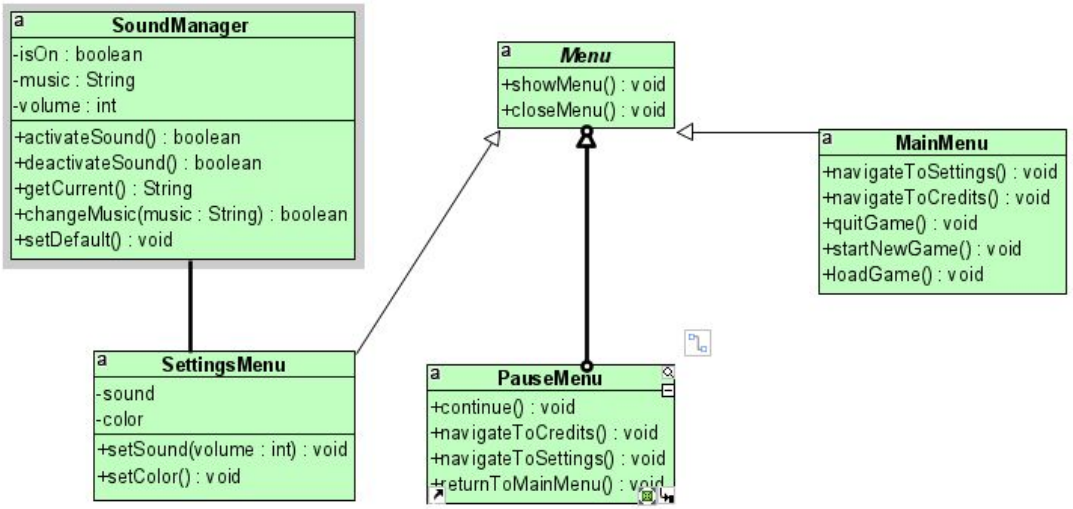
Final object design


SoundManager
-isOn : boolean
-music : String
-volume : int
+activateSound() : boolean
+deactivateSound() : boolean
+getCurrent() : String
+changeMusic(music : String) : boolean
+setDefault() : void
Police
-deal : boolean
+getAtSameLoc() : boolean
+arrestMafia() : void
+arrestPlayer(playerName : Player) : void
+isDeal() : boolean
create police
Forex
-dollarExRate : double
-euroExRate : double
-tryExRate : double
-frankExRate : double
+getDollarExRate() : double
+getEuroExRate() : double
+setDollarExRate(dollarExRate : double) : void
+getTryExRate() : double
+setEuroExRate(euroExRate : double) : void
+setTryExRate(tryExRate : double) : void
+getFrankExRate() : double
+setFrankExRate(frankExRate : double) : void
BotCreator
+createMafia() : void
+createPolice() : void
+createBots() : void
FileManager
-loadGame
-default
-gameData
SettingsMenu
-sound
-color
+setSound(volume : int) : void
+setColor() : void
Menu
+showMenu() : void
+closeMenu() : void
PauseMenu
+continue() : void
+navigateToCredits() : void
+navigateToSettings() : void
+returnToMainMenu() : void
MainMenu
+navigateToSettings() : void
+navigateToCredits() : void
+quitGame() : void
+startNewGame() : void
+loadGame() : void
Character
-location : int
-name : String
+getLocation() : int
+setLocation(tileNumber : ...
+getName() : String
+setName(name : String) ...
NPC
<<Interface>>
Playable
+playTurn()
GameManager
-dice : Dice
-players : Player[4]
-forexManager : ForexManager
-botCreator : BotCreator
-chanceDeck : CardDeck
-map : Map
-fileManager : FileManager
-bank : Bank
-turnLimit : int
-powerUpManager : PowerUpManager
-moneyLimit : int
-communityDeck : CardDeck
+initializeGame() : void
+finishGame() : void
+isRunning() : boolean
+pauseGame() : boolean
+updateMap() : void
+update() : void
+changeSettings() : void
+getDeck() : CardDeck
Dice
-diceNo1 : int
-diceNo2 : int
+rollTheDice() : int
+rollDie() : int
MovementByNumCard
-amount : int
+getAmount() : int
MovementToCard
-location : int
+getLocation() : int
MovementCard
TransactionCard
-amount : int
GetOutOfJail
CardDeck
-cards : Card[0..*]
+shuffleCards() : void
+drawCard() : Card
+createDeck() : void
Card
-feature : String
+setFeature(feature : String) : void
+getFeature() : String
TaxTile
-amount : int
+getAmount() : int
+setAmount(amount : int) ...
Mafia
-shareRate : double
-isArrested : boolean
-pastDeals : Arraylist<Player>
+attackHotel(tile : Tile) : void
+blackmail(player : Player) : void
+getShareRate() : double
+setShareRate(shareRate : double) : void
+sellTile(tileNo : int) : boolean
+sellChanceCard() : boolean
+sellCommunityCard() : boolean
+getIsArrested() : boolean
+setIsArrested(isArrested : boolean) : void
+addDeal(player : Player) : void
+deleteDeal(player : Player) : boolean
+getDeals() : ArrayList<Player>
Player
-isBankrupt : boolean
-powerUps : ArrayList<PowerUp>
-account : Account
-cardDeck : CardDeck
-tileList : Tile[0..*]
-properties : Property[0..*]
-isArrested : boolean
+getIsBankrupt() : boolean
+setIsBankrupt(isBankrupt : boolean) : void
+getAccount() : Account
+setAccount(account : Account) : void
+addToDeck(card : Card) : void
+removeFromDeck(card : Card) : void
+getTileList() : Tile[0..*]
+setTileList(tileList) : void
+getProperties() : Property
+setProperties(property : Property) : void
+getIsArrested() : boolean
+setIsArrested(isArrested : boolean) : void
+useMafiaFavour(enum : enum) : void
User
BotCharacter
InputManager
ForexManager
-forex : Forex
+calcSupDemand() : double
+resetExRates() : void
+updateExRates(currency ...
+createExRates(dollarExRa...
Map
-tiles : Tile[0..*]
+getTiles()
+updateMap() : void
+setTiles(tiles : Tile[0..*]) : void
+createMap() : void
CardTile
-isChance : boolean
+getIsChance() : boolean
+setIsChance(isChance : boolean) : void
DoNotingTile
-isFreePark : boolean
+getIsPark() : boolean
+setIsPark(park : boolean) : void
Account
-dollar : double
-euro : double
-try : double
-swissFrank : double
+getDollar() : double
+getEuro() : double
+getTry() : double
+getFrank() : double
+setDollar(dollar : double) : void
+setEuro(euro : double) : void
+setTry(try : double) : void
+setFrank(frank : double) : void
Bank
+giveMoney(player : Player, money : double) : boolean
+takeMoney(player : Player, money : double) : boolean
+takeIncomeTax(player : Player) : boolean
PowerUpManager
-powerUpStack
-powerUpCrate
+pushPowerUp(powerUp)
+popPowerUp()
CityTile
-properties : Property[0..*]
-isOwned : boolean
-whoseTile : Player
-price : int
-rentAmount : double
-soldByMafia : boolean
-isMorgage : boolean
+getIsOwned() : boolean
+setIsOwned(isOwned : boolean) : void
+getWhoseTile() : Player
+setWhoseTile(whoseTile : Player) : void
+getPrice() : int
+setPrice(price : int) : int
+getHouseCount() : int
+addHouse() : boolean
+getHotelCount() : int
+addHotel() : boolean
+isHotelBuildAvailable() : boolean
+getRentAmount() : double
+setRentAmount(rentAmount : double) : void
+getSoldByMafia() : boolean
+setSoldByMafia(soldBy : boolean) : void
+getIsMorgage() : boolean
+setIsMorgage(isMorgage : boolean) : void
Tile
-name : String
-id : int
+getName() : String
+setName(name : String) : void
+getId() : int
+setId(id : int) : void
StartTile
-salary : int
+getSalary() : int
+setSalary(salary : int) : void
JailTile
+arrested(player : Player) : void
+getIsArrested(player : Player) : boolean
Property
-price : int
+getPrice() : int
+setPrice(price : int) : void
House
Hotel
Vehicle
TransportationTile
-attribute : String
-isOwned : boolean
-whoseTile : Player
-price : int
-rentAmount : double
-soldByMafia : boolean
-vehicleAmount : int
-isMorgage : boolean
+getIsOwned() : boolean
+setIsOwned(isOwned : boolean) : void
+getWhoseTile() : Player
+setWhoseTile(whoseTile : Player) : void
+getPrice() : int
+setPrice(price : int) : void
+getRentAmount() : double
+setRentAmount(rentAmount : double) : void
+getSoldByMafia() : boolean
+setSoldByMafia(soldBy : boolean) : void
+addVehicle()
+getIsMorgage() : boolean
+setIsMorgage(isMorgage : boolean) : void
EarningPowerUp
-multiplier : double
-targetedPlayer : Player
+getMultiplier() : double
+setMultiplier(multiplier : double) : void
+getTargetedPlayer() : Player
+setTargetedPlayer(targetPlayer : Player) : void
PowerUp
-lifetime : int
+activate() : void
+getLifetime() : int
+setLifeTime(lifeTime : int) : void
PowerUpCrate
-powerUps : PowerUp[0..*]
-price : double
+getRandomPowerUp() : PowerUp
+getPrice() : double
ForexPowerUp
-percentage : double
-targetCurrency : String
+getPercentage() : double
+setPercentage(percentage : double) : void
+getTargetCurrency() : String
+setTargetCurrency(targetCurrency : String) : void
SlowDownPowerUp
-multiplier : double
-targetedPlayer : Player
+getMultiplier() : double
+setMultiplier(multiplier : double) : void
+getTargetedPlayer() : Player
+setTargetedPlayer(targetedPlayer : Player) : void
StrikePowerUp
-targetedPlayer : Player
-strikeCount : int
+getTargetedPlayer() : Player
+setTargetedPlayer(targetedPlayer : Player) : void
+getStrikeCount() : int
+setStrikeCount(strikeCount : int) : int

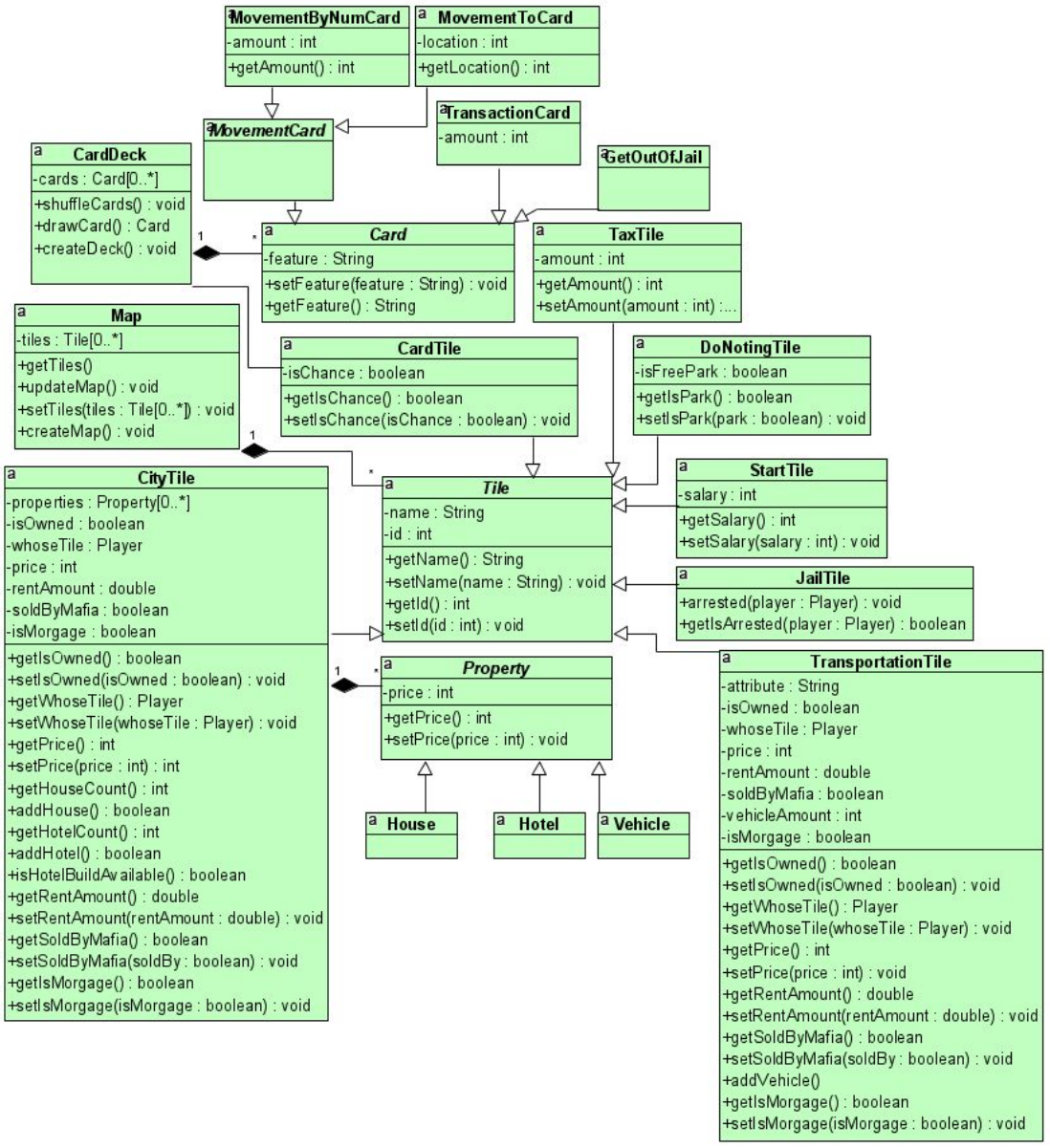
Abstraction of final object design



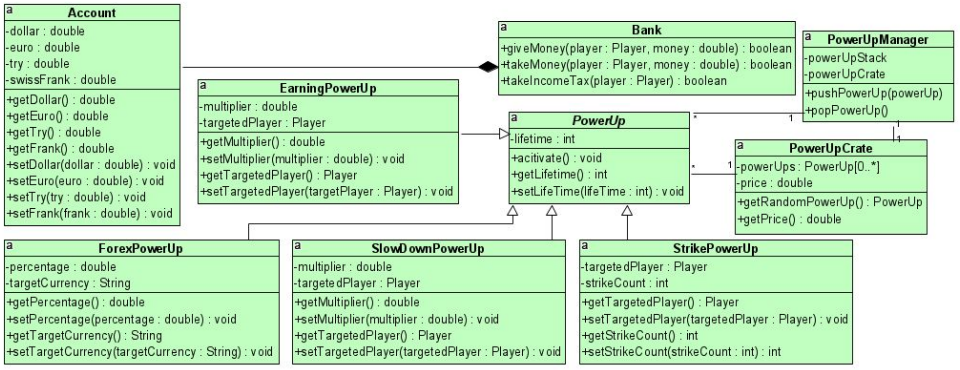
Piece 1



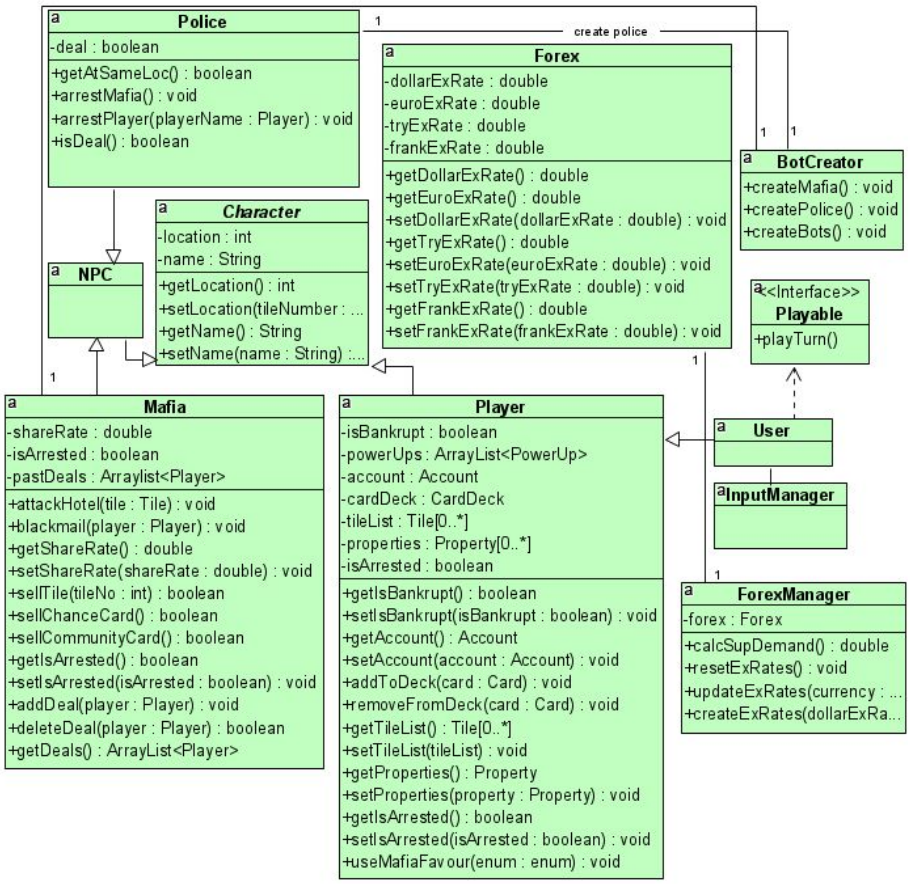
Piece 2


MovementByNumCard
-amount : int
+getAmount() : int
MovementToCard
-location : int
+getLocation() : int
MovementCard
TransactionCard
-amount : int
GetOutOfJail
CardDeck
-cards : Card[0..*]
+shuffleCards() : void
+drawCard() : Card
+createDeck() : void
Card
-feature : String
+setFeature(feature : String) : void
+getFeature() : String
TaxTile
-amount : int
+getAmount() : int
+setAmount(amount : int) :...
Map
-tiles : Tile[0..*]
+getTiles()
+updateMap() : void
+setTiles(tiles : Tile[0..*]) : void
+createMap() : void
CardTile
-isChance : boolean
+getIsChance() : boolean
+setIsChance(isChance : boolean) : void
DoNotingTile
-isFreePark : boolean
+getIsPark() : boolean
+setIsPark(park : boolean) : void
CityTile
-properties : Property[0..*]
-isOwned : boolean
-whoseTile : Player
-price : int
-rentAmount : double
-soldByMafia : boolean
-isMorgage : boolean
+getIsOwned() : boolean
+setIsOwned(isOwned : boolean) : void
+getWhoseTile() : Player
+setWhoseTile(whoseTile : Player) : void
+getPrice() : int
+setPrice(price : int) : int
+getHouseCount() : int
+addHouse() : boolean
+getHotelCount() : int
+addHotel() : boolean
+isHotelBuildAvailable() : boolean
+getRentAmount() : double
+setRentAmount(rentAmount : double) : void
+getSoldByMafia() : boolean
+setSoldByMafia(soldBy : boolean) : void
+getIsMorgage() : boolean
+setIsMorgage(isMorgage : boolean) : void
Tile
-name : String
-id : int
+getName() : String
+setName(name : String) : void
+getId() : int
+setId(id : int) : void
StartTile
-salary : int
+getSalary() : int
+setSalary(salary : int) : void
JailTile
+arrested(player : Player) : void
+getIsArrested(player : Player) : boolean
Property
-price : int
+getPrice() : int
+setPrice(price : int) : void
House
Hotel
Vehicle
TransportationTile
-attribute : String
-isOwned : boolean
-whoseTile : Player
-price : int
-rentAmount : double
-soldByMafia : boolean
-vehicleAmount : int
-isMorgage : boolean
+getIsOwned() : boolean
+setIsOwned(isOwned : boolean) : void
+getWhoseTile() : Player
+setWhoseTile(whoseTile : Player) : void
+getPrice() : int
+setPrice(price : int) : void
+getRentAmount() : double
+setRentAmount(rentAmount : double) : void
+getSoldByMafia() : boolean
+setSoldByMafia(soldBy : boolean) : void
+addVehicle()
+getIsMorgage() : boolean
+setIsMorgage(isMorgage : boolean) : void

Piece 3



Piece 4



Packages

Game Management Package

Includes classes which are responsible for game management, manages the interactions between different components and subsystems.

Entity Management Package

Includes classes which manage entities. E.g. ForexManager, PowerUpManager.

Entity Package

Includes classes which are game objects of the game and provides functions to interact with them

Graphical Interface Package

Includes the GUI classes like menus and pop-ups.

File Management Package

Includes packages which provide file management in the game.

Class Interfaces

GameManager:

- private Dice dice: An instance of the Dice object. Used to simulate dice roll.
- private Player[] players: An array of player instances.
- private ForexManager forexManager: A singleton instance of ForexManager object. Manages the Forex class.
- private BotCreator botCreator: A singleton instance of BotCreator object, creates the BOTs and the Mafia and the Police.
- private CardDeck chanceDeck: The card deck of chance cards.

- private CardDeck communityDeck: The card deck of community cards.
- private Map map: The Map singleton instance. Manages the Monopoly board.
- private FileManager fileManager: The FileManager Singleton instance. Manages load game, save game and file retrieval operations.
- private Bank bank: Singleton instance of object Bank. Bank manages Account objects for each Player.
- private long int turnLimit: Maximum turn limit. Given by the user on game initialization. The game ends when this many turns are played.
- private PowerUpManager powerUpManager: Singleton instance of PowerUpManager object. Manages the PowerUp and PowerUpCrate.
- private long int moneyLimit: Maximum money limit. Given by the user on game initialization. The game ends when a player reaches this amount.
- public void initializeGame(): This method initializes the properties of GameManager.
- public void finishGame(): Checks the end conditions for the game and if the game is completed shows the winner and returns to the MainMenu.
- public boolean isRunning(): Returns true if the game is still on.
- public void pauseGame(): Pauses the game and calls the PauseMenu.
- public void updateMap(): Calls the updateMap method on the Map instance. Which in turn updates the map visual in the game screen
- public void update(): The game loop.
- public void changeSettings(): Updates the settings.

Menu:

- public void showMenu(): method to display the menu.
- public void closeMenu(): method to close the menu.

SettingsMenu:

- private int sound: Integer level of the sound.

- private boolean color: Colorblind mode is active if true.
- public void setSound(): Method to set sound level.
- public void setColor(): Method to change color scheme.

PauseMenu:

- public void continue(): Method to continue to the game.
- public void navigateToCredits(): Method to display credits.
- public void navigateToSettings(): Method to display settings.
- public void returnToMainMenu(): Method to return to the main menu.

MainMenu:

- public void navigateToSettings(): Method to display settings.
- public void navigateToCredits(): Method to display credits.
- public void quitGame(): Method to quit the game.
- public void startNewGame(): Method to start a new game.
- public void loadGame(): Method to load the old game file.

SoundManager:

- private boolean isOn: Boolean value to show if the sound is on.
- private int music: Integer value to control the sound of the music.
- private int volume: Integer value to control the sound level.
- public void activateSound(): Method to activate the muted sound.
- public void deactivateSound(): Method to mute the sound.
- public String getCurrent(): Gets the current music
- public void changeMusic(String): Changes the music
- public void setDefault(String): Sets the default music

FileManager:

- public void loadGame(): Method to load the old game file.
- public void saveGame(): Method to save the game.

InputManager:

Dice:

- public int rollDie(): Method to roll one die.
- Public int rollDice(): Method to roll the dice.

CardDeck:

- private ArrayList<Card> cards: The card array.
- public void shuffleCards(): Shuffles the card deck.
- public Card drawCard(): Draws the top card.
- public void createDeck(): Method to create a deck.

Card:

- private String feature: The name of the card which represents it's feature. The description of the card.

MovementCard:

MovementByNumCard:

- private int amount: The amount of tiles to move.

MovementToCard:

- private int location: Integer value which represents which location to go.

TransactionCard:

- private int amount: The integer amount of the card's transaction, positive or negative.

GetOutOfJail:

Map:

- private ArrayList<Tile> tiles : an array of tiles present in the map
- public void updateMap() : Method that updates the locations of the players and the states of each tile.
- public void createMap(): Method that creates the map from scratch.

Tile:

- private String name: the name of the tile
- private int id: the position of the tile in terms of integer value, starting from start tile which is 0.

CardTile:

- private boolean isChance: the type of the card tile, whether it is a change tile or a community tile.

DoNothingTile:

- private boolean isFreePark: the type of the do nothing tile, whether it is a free park tile or a visit jail tile.

StartTile:

- private int salary: The amount of salary that is given in the start tile in Turkish Liras.

JailTile:

- public void arrested(Player player): Arrest the Player who lands on the tile.
- public boolean getIsArrested(Player player): Gets if the player is arrested.

TaxTile:

- private int amount: The amount of tax the player has to play when they land on this tile.

TransportationTile:

- private String attribute: String variable which stores the attribute's name.
- private boolean isOwned: Boolean value which shows if the tile is owned or not.
- private Player whoseTile: Variable which shows which player's tile is that tile.
- private int price: Integer value which represents the price of the tile in Turkish Liras.
- private int rentAmount: Variable that shows the amount of rent this tile demands
- private String soldBy: String value which shows who sold this tile.
- private int numOfVehicle: Integer value which represents the number of vehicles in the tile.

CityTile:

- private ArrayList<Property> properties : A list that stores the properties on a city tile.
- private boolean isOwned : true if the city tile is owned.
- private Player whoseTile : the player who owns the tile.
- private int price: the price of the city tile in Turkish Liras.
- private int rentAmount: the rent of the city tile in Turkish Liras.
- private bool soldByMafia: Indicates if the tile is sold by the Mafia or not. Mafia takes their share of the rents if they sell the tile.
- public boolean isHotelBuildAvailable(): Method that returns true if 3 houses are built in the city tile.
- private boolean isMortgage: true if the property is under mortgage.

Property:

- private int price: the amount needed to pay for buying a property in Turkish Liras.

PowerUpManager:

- private ArrayList<PowerUp> powerUpStack: Stack to store PowerUp objects.
- private PowerUpCrate powerUpCrate: Instance of PowerUpCrate.
- public void pushPowerUp(PowerUp powerUp): Method to push the given power up to the top of the powerUpStack.
- public PowerUp popPowerUp(): Method to get the latest power up from powerUpStack.

PowerUpCrate:

- private ArrayList<PowerUp> powerUps: List of power up objects which are inside the crate.
- private int price: Integer value of the price of the power up crate.
- public PowerUp getRandomPowerUp(): method to draw random power up from the crate.

PowerUp:

- private int lifetime: Integer value which represents the powerups lifetime in terms of rounds.
- public void activate(): Method to activate the power up.

EarningPowerUp:

- private double multiplier: The player's earning multiplier.
- private Player targetedPlayer: The player to multiply the earning

SlowDownPowerUp:

- private double multiplier: The slowdown multiplier. Which slows down the dice movement of the player.
- private Player targetedPlayer: Player value which stores which player is targeted to slo

StrikePowerUp:

- private int strikeCount: The amount of tiles the targeted player has to go back.
- private Player targetedPlayer: The player to be striked backwards.

ForexPowerUp:

- private double percentage: the rate of the currency change when powerup is used
- private String targetCurrency: the currency whose percentage will be changed when powerup is used

Bank:

- public boolean giveMoney(Player player, int money): Method to give money amount of Turkis liras to the player’s account.
- public boolean takeMoney(Player player, int money): Method to take money amount of Turkish Liras from the player’s account.
- public boolean takeIncomeTax(Player player): Method to take the income tax in Turkish Liras from the player’s account.

Account:

- private int dollar: Integer value which stores the amount of dollars in it.
- private int euro: Integer value which stores the amount of euros in it.
- private int try: Integer value which stores the amount of Turkis Liras in it.
- private int swissFrank: Integer value which stores the amount of franks in it.

Forex:

- private double tryExRate : is a constant and its value is 1.
- private double dollarExRate : the rate of dollar according to Turkish Lira.
- private double euroExRate : the rate of euro according to Turkish Lira.
- private double frankExRate : the rate of frank according to Turkish Lira.

ForexManager:

- private Forex forex : a forex object that stores the rates of the 4 currencies.
- public calcSupDemand() : Method to calculate the rates of the 3 currencies according to Turkish Lira, using the exchange behaviour of the players.
- public resetExRates() : Method to reset the 3 currencies to 1.
- public updateExRates() : Method to update the exchange rates
- public createExRates() : Method to initialize exchange rates

BotCreator:

- public void createMafia(): Method to create a mafia bot.
- public void createPolice(): Method to create a police bot.
- public void createBots(): Method to create the bots.

NPC:

- private int location: Integer value which represents where the NPC is on the map.
- private string name: String value which represents NPC's name.
- public int getLocation(): Method which returns NPC's location.
- public void setLocation(int tileNumber): Method which sets the location of the NPC.
- public String getName(): Method which returns the name of the NPC.

- public void setName(String name): Method to set NPC'S name.

Mafia:

- private double shareRate: The constant share the Mafia takes from any earning of the Player.
- private boolean isArrested: Boolean value which shows if the mafia is arrested.
- private ArrayList<Player> pastDeals: ArrayList which shows which Players did a deal with the mafia.
- public void attackHotel(): Destroys the hotel that is in the same tile as mafia
- public void blackMail(Player player): Method to blackmail given player.
- public double getShareRate(): Method to get the share rate.
- public void setShareRate(double shareRate): Method to set the share rate.
- public boolean sellTile(int tileNo): Method to sell the given tile, if sell is successful it returns true.
- public boolean sellChanceCard(): Method to sell a chance card, if selling is successful it returns true.
- public boolean sellComunityCard(): Method to sell a community card, if selling is successful it returns true.
- public void addDeal(Player player): Method to add the player to past deals.
- public boolean deleteDeal(Player player):: Method to pop the player from past deals.
- public ArrayList<Player>: Method the get deals.

Police:

- private boolean deal: Boolean value which represents if any Player had a deal with the Mafia in the past five turns.
- public boolean getAtSameLoc(): Method to check whether it is at the same location with the Mafia.
- public void arrestMafia(): Method to arrest the Mafia.

- public void arrestPlayer(Player playerName): Method to arrest the given player.
- public boolean isDeal(): Method that returns true if mafia has dealt with players.

Character:

- private Tile location: The tile where the Character is at on the map.
- private string name: String value which represents Character's name.
- public Tile getLocation(): Method which returns Character's location.
- public void setLocation(Tile tile): Method which sets the location of the Character.
- public String getName(): Method which returns the name of the Character.
- public void setName(String name): Method to set Character's name.

User:

Playable:

- public void playTurn() : Method to play the turn.

BotCharacter:

Player:

- private boolean isBankrupt : is true if the player is bankrupt
- private ArrayList<PowerUp> powerups : the list that stores the powerups the player owns
- private Account account: the account of the user
- private CardDeck cardDeck: Instance of a CardDeck.
- private ArrayList<Tile> tileList: List to store all the tiles which the user has.

  private ArrayList<Property> properties: List to store all the tiles which the user has.
- private boolean isArrested: Boolean value which shows if the Player is arrested or not.
- public void addToDeck(Card card): Method to add the given card into the Player's deck.
- public boolean removeFromDeck(Card card): method to remove the given card from the deck if the card exists. If removal is successful, the method returns true.

- public void useMafiaFavour(): Method to use mafia favor.

5. Glossary & references

Hasbro. (n.d.). Monopoly Classic Rulebook. Retrieved November 29, 2020, from https://www.hasbro.com/common/instruct/00009.pdf