CS 319 Project Design Report Iteration 2

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

1. Introduction

1.1. 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.

1.2. 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

1.2.1.1.Development Time vs Performance:

As you know, many of the developers indicate that Java is very time-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. For the sake of simplifying the design process and development time, 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 that decreases performance. In addition by applying the Singleton design pattern and again decreasing the development time, we created strong couplings between our entity manager classes and game manager which reduces performance.

1.2.1.2.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.

1.2.1.3.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.

1.2.1.4.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.

1.2.1.5.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

1.2.2.1.Criteria

1.2.2.1.1.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.

1.2.2.1.2.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.

1.2.2.2.Maintenance Criteria

1.2.2.2.1.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.

1.2.2.2.2Modifiability:

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.

1.2.2.2.3.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.

1.2.2.2.4.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.

1.2.2.3.Performance Criteria

1.2.2.3.1.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.

2. High-level system architecture

2.1. Subsystem decomposition

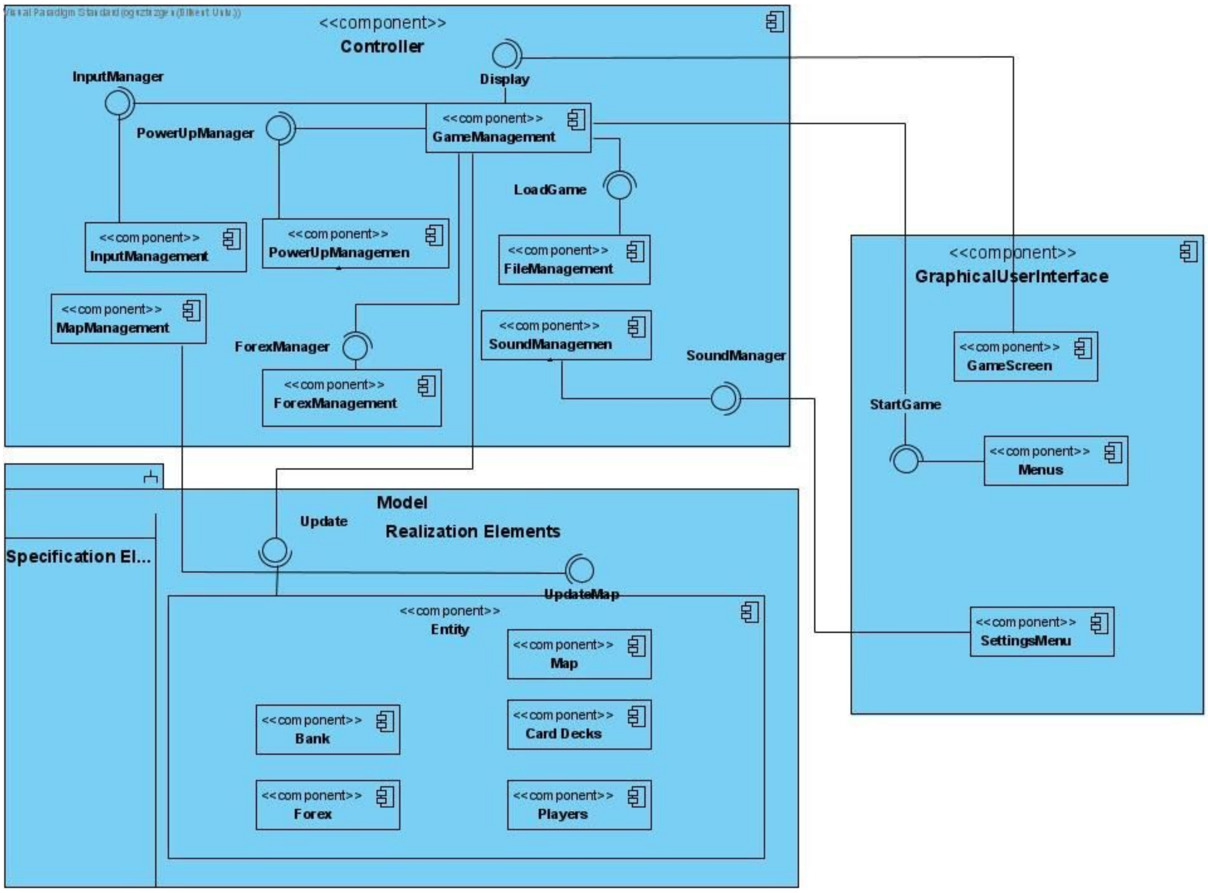

Figure 2.a. Component Diagram

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.

2.1.1. Graphical User interface Subsystem

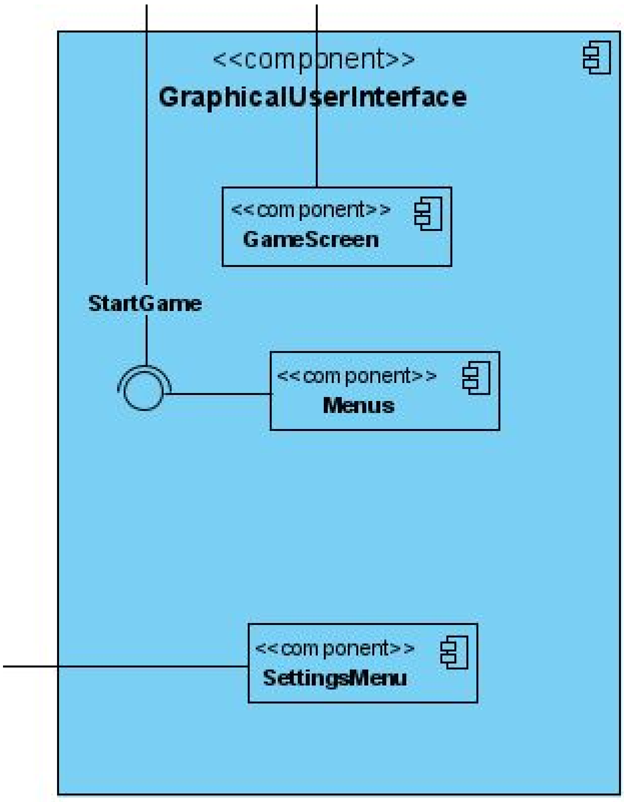

Figure 2.b. Graphical User Interface component

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

2.1.2. Controller Subsystem

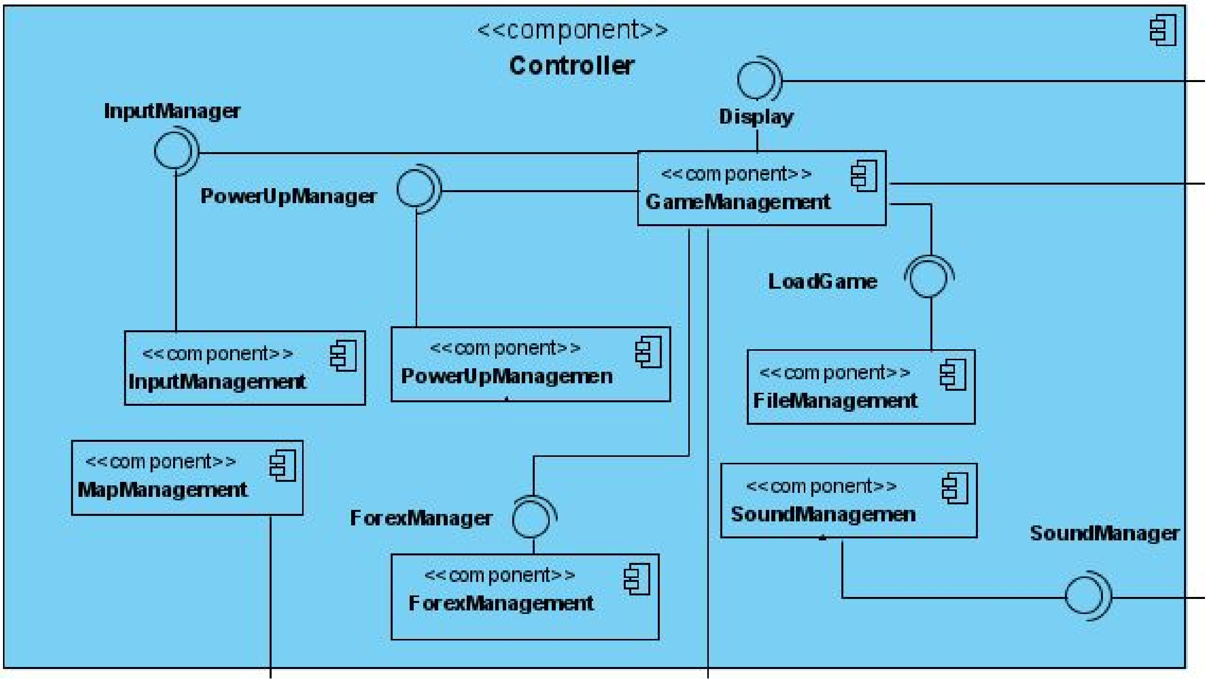

Figure 2.c. Controller component

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.

2.1.3. Model subsystem

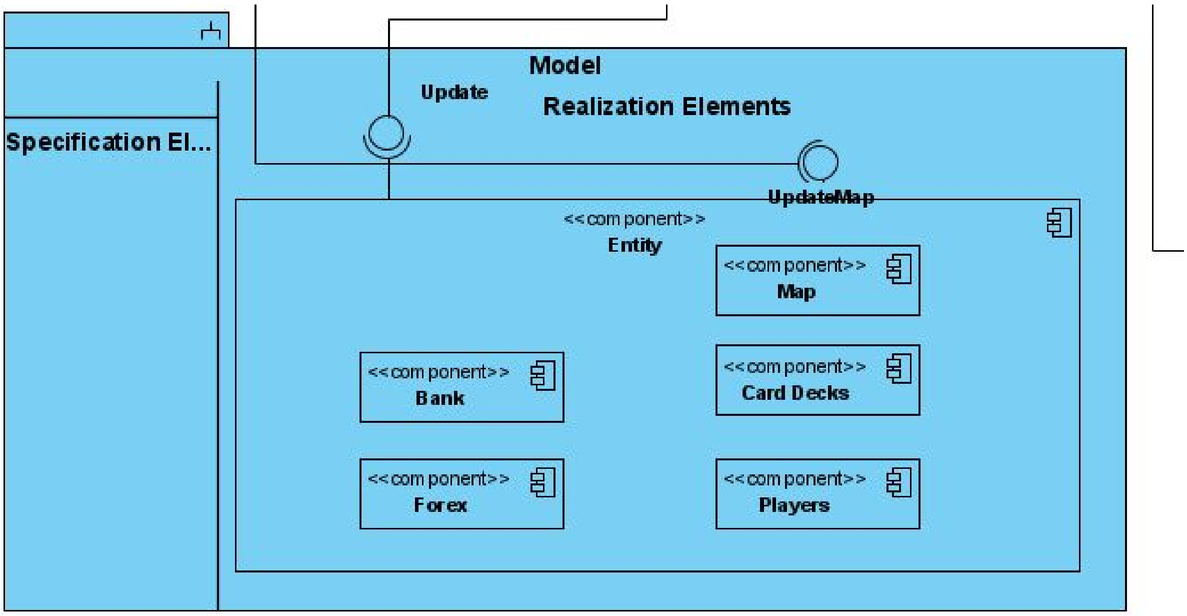

Figure 2.d. Model component

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, 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.

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.

2.2. 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.

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.

2.3. 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- the GameManager class's contents- in a text file in a subdirectory called local\. In addition, the user preferences (e.g. settings) will also be stored as a text file in a subdirectory called local\. The external images and sound effects will be stored inside a subdirectory called vendor\image\ and vendor\sound\ respectively. The images will have the PNG format and the sound effects will have the WAV format.

2.4. 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.

2.5. 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. 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 through the main menu. Similarly the game will ask the player if they really want to quit to prevent accidental quits. The last case is the system failures. These could be hardware failures and electric outages. To prevent this or to recover from this, our game is going to save on every round end this way it will be possible to continue the game if there was any fatal failure. After the failure player can re-open the game and load the saved game from the menu and continue to play the game.

3. Low-level design

3.1. 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.

3.1.1. 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. The Façade class is the GameManager class which serves as a front-facing interface which handles the game logic.

3.1.2. Singleton Design Pattern

To reduce problems we have used the singleton design pattern on our manager classes. There will be only one instance of the 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. Singleton classes are TradeManager, ForexManager, InputManager, FileManager, SoundManager, GameManager and PowerupManager.

3.1.3 Strategy Design Pattern

For the purposes of reusability and maintainability, and to prevent class implosions, we have used the Strategy design pattern on our Cards, Power-ups and Players. The subclasses of these were really similar to each other and they differed by only what they did. We extracted the common methods in these instances to create a family of algorithms which represent a specific behavior related to the context object. Thus, we have eased the process of adding different types of cards and power ups. However because of this, the number of classes we had for PowerUps and Cards were increased.

3.2. Final object design

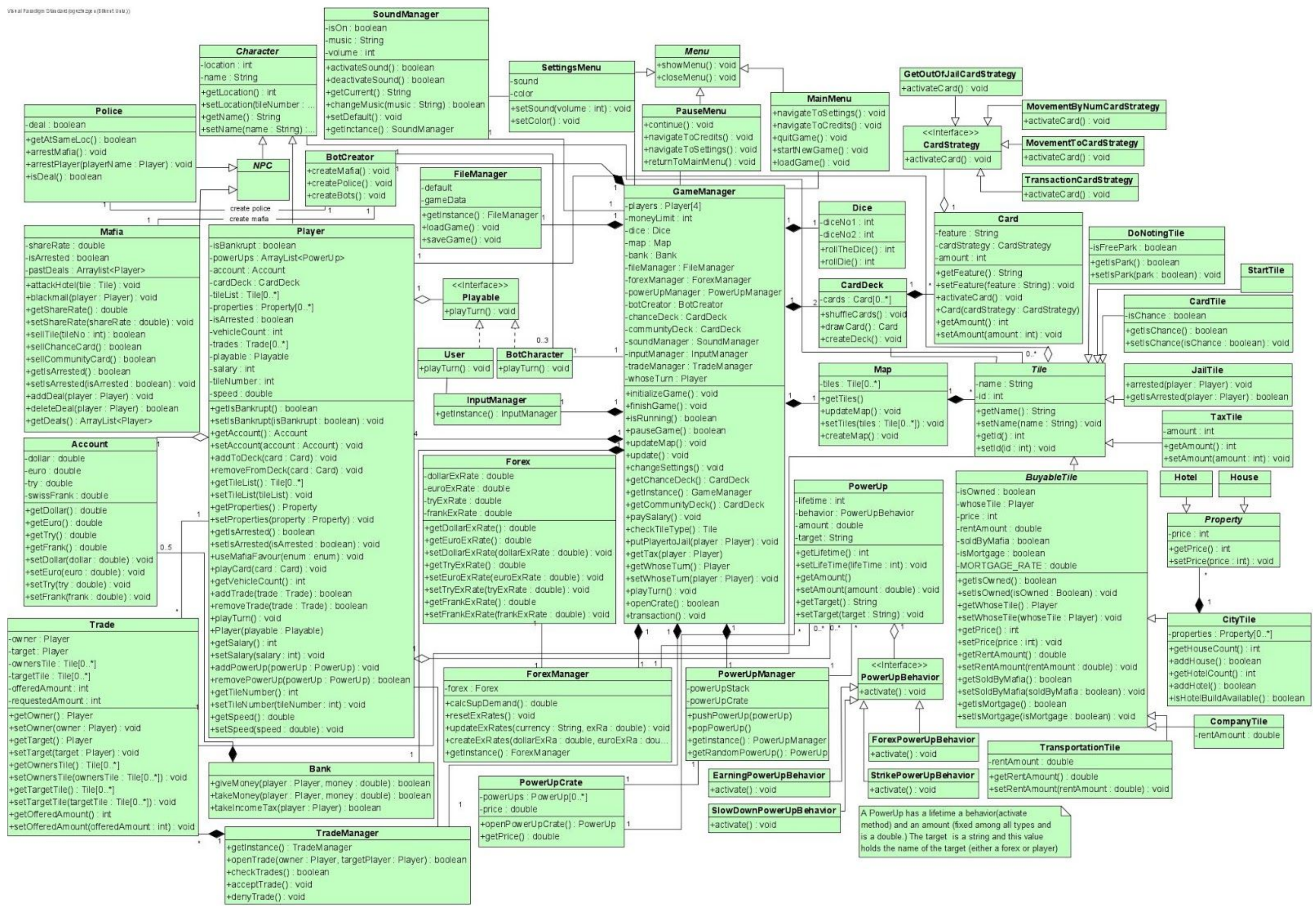
Police
-deal : boolean
+getAtSameLoc() : boolean
+arrestMafia() : void
+arrestPlayer(playerName : Player) : void
+isDeal() : boolean
Character
-location : int
-name : String
+getLocation() : int
+setLocation(tileNumber : ...
+getName() : String
+setName(name : String) : ...
NPC
SoundManager
-isOn : boolean
-music : String
-volume : int
+activateSound() : boolean
+deactivateSound() : boolean
+getCurrent() : String
+changeMusic(music : String) : boolean
+setDefault() : void
+getInctance() : SoundManager
BotCreator
+createMafia() : void
+createPolice() : void
+createBots() : void
create police
create mafia
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
-vehicleCount : int
-trades : Trade[0..*]
-playable : Playable
-salary : int
-tileNumber : int
-speed : double
+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
+playCard(card : Card) : void
+getVehicleCount() : int
+addTrade(trade : Trade) : boolean
+removeTrade(trade : Trade) : boolean
+playTurn() : void
+Player(playable : Playable)
+getSalary() : int
+setSalary(salary : int) : void
+addPowerUp(powerUp : PowerUp) : void
+removePowerUp(powerUp : PowerUp) : boolean
+getTileNumber() : int
+setTileNumber(tileNumber : int) : void
+getSpeed() : double
+setSpeed(speed : double) : 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
0..5
Trade
-owner : Player
-target : Player
-ownersTile : Tile[0..*]
-targetTile : Tile[0..*]
-offeredAmount : int
-requestedAmount : int
+getOwner() : Player
+setOwner(owner : Player) : void
+getTarget() : Player
+setTarget(target : Player) : void
+getOwnersTile() : Tile[0..*]
+setOwnersTile(ownersTile : Tile[0..*]) : void
+getTargetTile() : Tile[0..*]
+setTargetTile(targetTile : Tile[0..*]) : void
+getOfferedAmount() : int
+setOfferedAmount(offeredAmount : int) : void
Bank
+giveMoney(player : Player, money : double) : boolean
+takeMoney(player : Player, money : double) : boolean
+takeIncomeTax(player : Player) : boolean
TradeManager
+getInstance() : TradeManager
+openTrade(owner : Player, targetPlayer : Player) : boolean
+checkTrades() : boolean
+acceptTrade() : void
+denyTrade() : void
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
FileManager
-default
-gameData
+getInstance() : FileManager
+loadGame() : void
+saveGame() : void
<<Interface>>
Playable
+playTurn() : void
User
+playTurn() : void
BotCharacter
+playTurn() : void
0..3
InputManager
+getInstance() : InputManager
GameManager
-players : Player[4]
-moneyLimit : int
-dice : Dice
-map : Map
-bank : Bank
-fileManager : FileManager
-forexManager : ForexManager
-powerUpManager : PowerUpManager
-botCreator : BotCreator
-chanceDeck : CardDeck
-communityDeck : CardDeck
-soundManager : SoundManager
-inputManager : InputManager
-tradeManager : TradeManager
-whoseTurn : Player
+initializeGame() : void
+finishGame() : void
+isRunning() : boolean
+pauseGame() : boolean
+updateMap() : void
+update() : void
+changeSettings() : void
+getChanceDeck() : CardDeck
+getInstance() : GameManager
+getCommunityDeck() : CardDeck
+paySalary() : void
+checkTileType() : Tile
+putPlayertoJail(player : Player) : void
+getTax(player : Player)
+getWhoseTurn() : Player
+setWhoseTurn(player : Player) : void
+playTurn() : void
+openCrate() : boolean
+transaction() : void
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
ForexManager
-forex : Forex
+calcSupDemand() : double
+resetExRates() : void
+updateExRates(currency : String, exRa : double) : void
+createExRates(dollarExRa : double, euroExRa : dou...
+getInstance() : ForexManager
PowerUpCrate
-powerUps : PowerUp[0..*]
-price : double
+openPowerUpCrate() : PowerUp
+getPrice() : double
PowerUpManager
-powerUpStack
-powerUpCrate
+pushPowerUp(powerUp)
+popPowerUp()
+getInstance() : PowerUpManager
+getRandomPowerUp() : PowerUp
EarningPowerUpBehavior
+activate() : void
SlowDownPowerUpBehavior
+activate() : void
Dice
-diceNo1 : int
-diceNo2 : int
+rollTheDice() : int
+rollDie() : int
CardDeck
-cards : Card[0..*]
+shuffleCards() : void
+drawCard() : Card
+createDeck() : void
Map
-tiles : Tile[0..*]
+getTiles()
+updateMap() : void
+setTiles(tiles : Tile[0..*]) : void
+createMap() : void
PowerUp
-lifetime : int
-behavior : PowerUpBehavior
-amount : double
-target : String
+getLifetime() : int
+setLifeTime(lifeTime : int) : void
+getAmount()
+setAmount(amount : double) : void
+getTarget() : String
+setTarget(target : String) : void
<<Interface>>
PowerUpBehavior
+activate() : void
ForexPowerUpBehavior
+activate() : void
StrikePowerUpBehavior
+activate() : void
A PowerUp has a lifetime a behavior(activate method) and an amount (fixed among all types and is a double.) The target is a string and this value holds the name of the target (either a forex or player)
GetOutOfJailCardStrategy
+activateCard() : void
MovementByNumCardStrategy
+activateCard() : void
<<Interface>>
CardStrategy
+activateCard() : void
MovementToCardStrategy
+activateCard() : void
TransactionCardStrategy
+activateCard() : void
Card
-feature : String
-cardStrategy : CardStrategy
-amount : int
+getFeature() : String
+setFeature(feature : String) : void
+activateCard() : void
+Card(cardStrategy : CardStrategy)
+getAmount() : int
+setAmount(amount : int) : void
DoNotingTile
-isFreePark : boolean
+getIsPark() : boolean
+setIsPark(park : boolean) : void
StartTile
CardTile
-isChance : boolean
+getIsChance() : boolean
+setIsChance(isChance : boolean) : void
Tile
-name : String
-id : int
+getName() : String
+setName(name : String) : void
+getId() : int
+setId(id : int) : void
JailTile
+arrested(player : Player) : void
+getIsArrested(player : Player) : boolean
TaxTile
-amount : int
+getAmount() : int
+setAmount(amount : int) : void
BuyableTile
-isOwned : boolean
-whoseTile : Player
-price : int
-rentAmount : double
-soldByMafia : boolean
-isMortgage : boolean
-MORTGAGE_RATE : double
+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(soldByMafia : boolean) : void
+getIsMortgage() : boolean
+setIsMortgage(isMortgage : boolean) : void
Hotel
House
Property
-price : int
+getPrice() : int
+setPrice(price : int) : void
CityTile
-properties : Property[0..*]
+getHouseCount() : int
+addHouse() : boolean
+getHotelCount() : int
+addHotel() : boolean
+isHotelBuildAvailable() : boolean
CompanyTile
-rentAmount : double
TransportationTile
-rentAmount : double
+getRentAmount() : double
+setRentAmount(rentAmount : double) : void

Figure 3.a. Final Object Design of Monopoly Sicilia

3.2.1 Abstraction of final object design

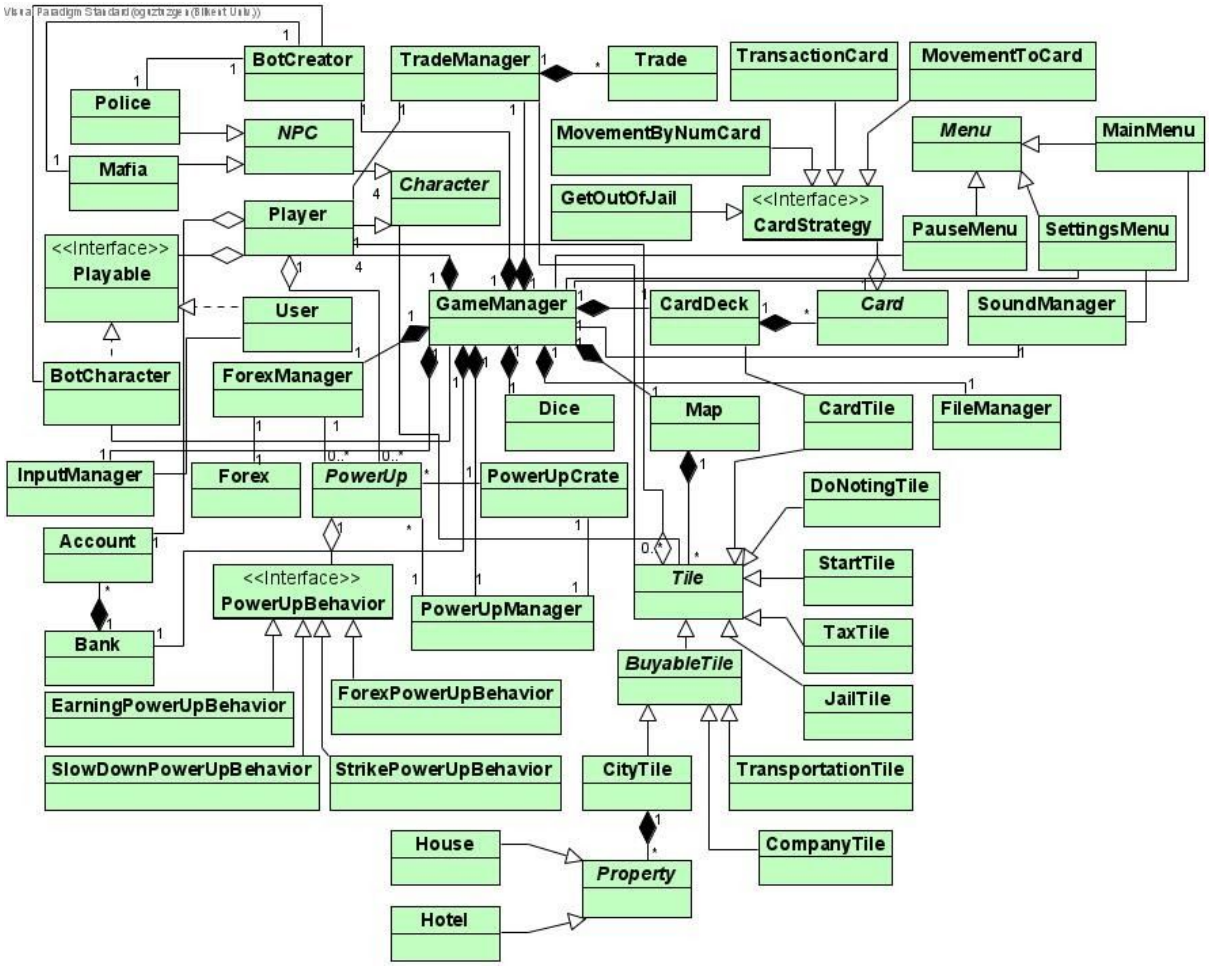

Figure 3.b. Abstraction of Final Object Design

3.2.2. Pieces

3.2.2.1.Piece 1

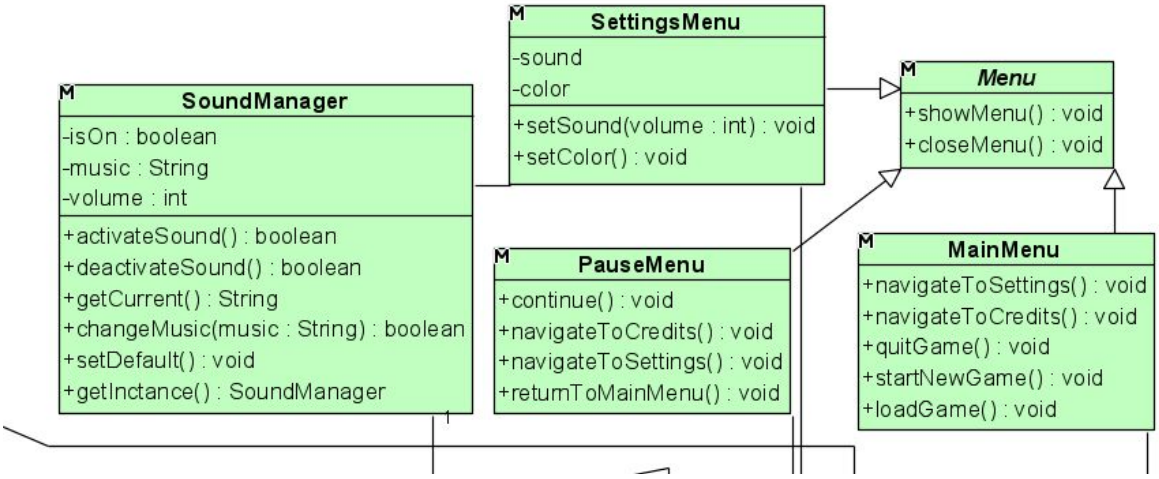
SettingsMenu
-sound
-color
+setSound(volume : int) : void
+setColor() : void
Menu
+showMenu() : void
+closeMenu() : void
SoundManager
-isOn : boolean
-music : String
-volume : int
+activateSound() : boolean
+deactivateSound() : boolean
+getCurrent() : String
+changeMusic(music : String) : boolean
+setDefault() : void
+getInctance() : SoundManager
PauseMenu
+continue() : void
+navigateToCredits() : void
+navigateToSettings() : void
+returnToMainMenu() : void
MainMenu
+navigateToSettings() : void
+navigateToCredits() : void
+quitGame() : void
+startNewGame() : void
+loadGame() : void
1

Figure 3.c. The classes that include the menu details

3.2.2.2.Piece 2

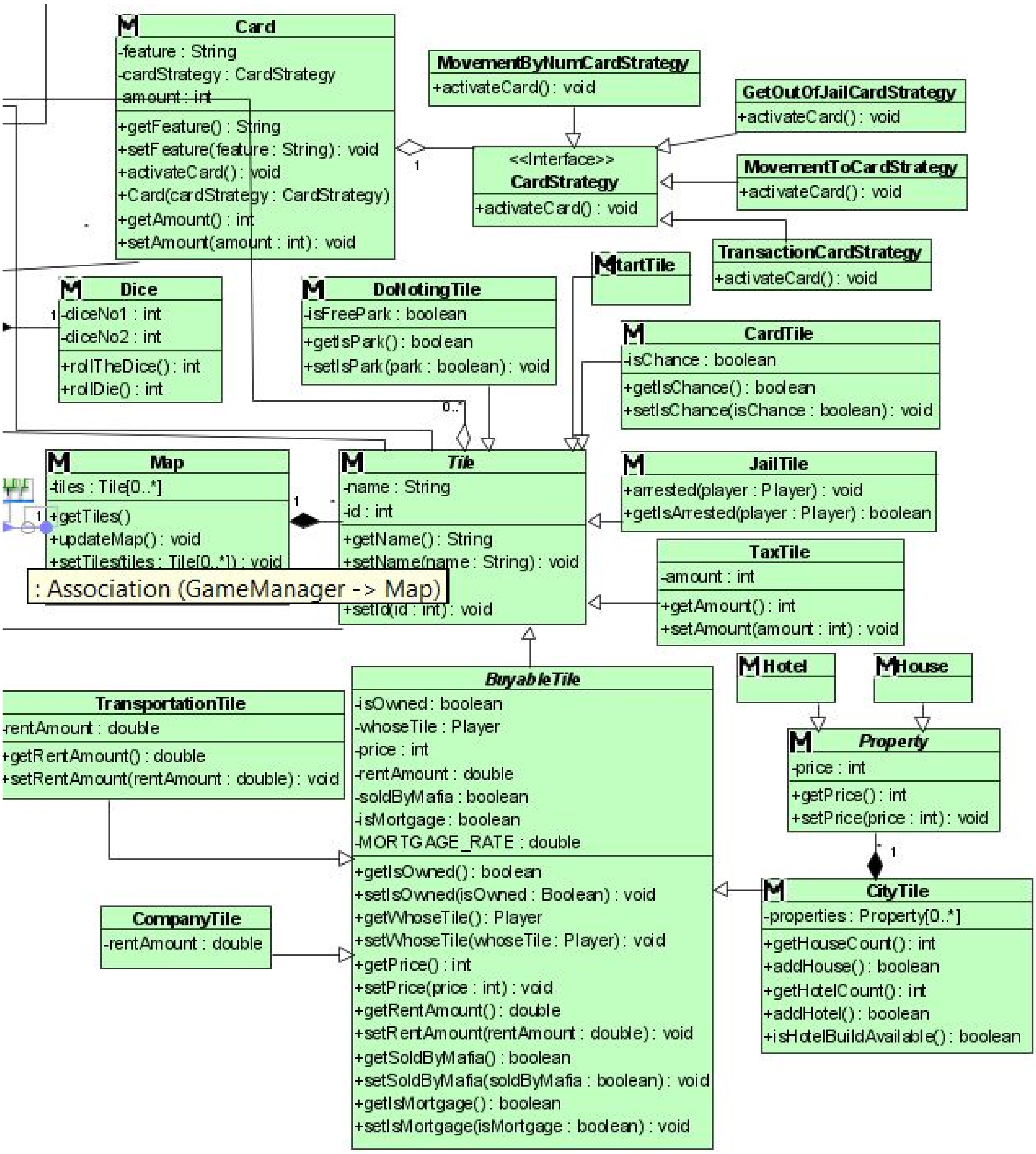

Figure 3.d. The piece that zooms tile related classes

3.2.2.3.Piece 3

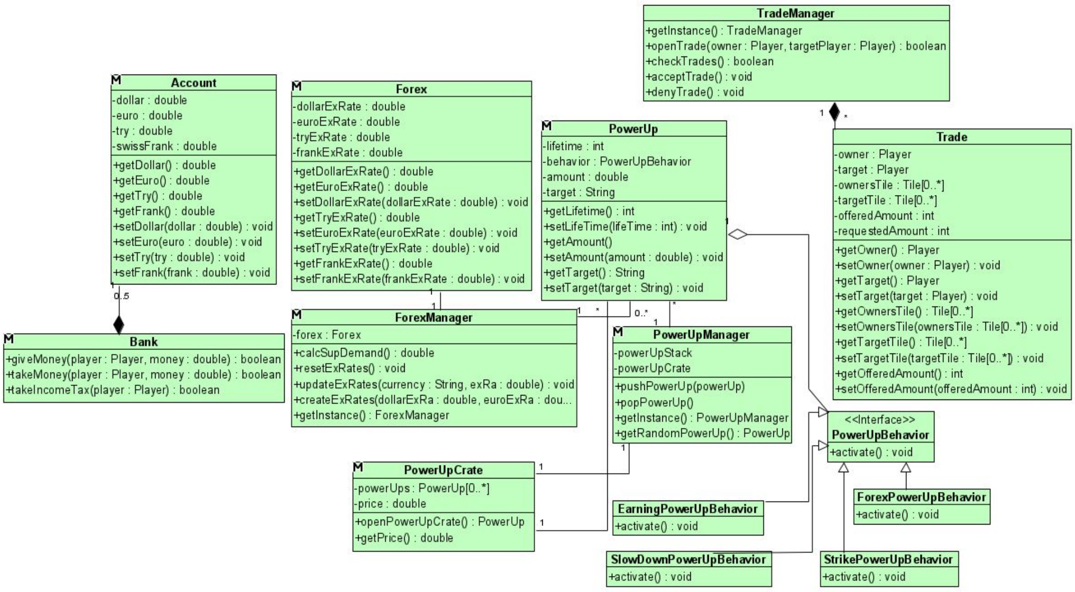
TradeManager
+getInstance() : TradeManager
+openTrade(owner : Player, targetPlayer : Player) : boolean
+checkTrades() : boolean
+acceptTrade() : void
+denyTrade() : void
Account
-dollar : double
-euro : double
-try : double
-swissFrank : double
+getDollar() : double
+getEuro() : double
+getTry() : double
+getFrank() : double
+setDollar(dollar : double) : void
+setEuro(euro : double) : void
+setTry(try : double) : void
+setFrank(frank : double) : void
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
PowerUp
-lifetime : int
-behavior : PowerUpBehavior
-amount : double
-target : String
+getLifetime() : int
+setLifeTime(lifeTime : int) : void
+getAmount()
+setAmount(amount : double) : void
+getTarget() : String
+setTarget(target : String) : void
Trade
-owner : Player
-target : Player
-ownersTile : Tile[0..*]
-targetTile : Tile[0..*]
-offeredAmount : int
-requestedAmount : int
+getOwner() : Player
+setOwner(owner : Player) : void
+getTarget() : Player
+setTarget(target : Player) : void
+getOwnersTile() : Tile[0..*]
+setOwnersTile(ownersTile : Tile[0..*]) : void
+getTargetTile() : Tile[0..*]
+setTargetTile(targetTile : Tile[0..*]) : void
+getOfferedAmount() : int
+setOfferedAmount(offeredAmount : int) : void
1
*
1
0..5
Bank
+giveMoney(player : Player, money : double) : boolean
+takeMoney(player : Player, money : double) : boolean
+takeIncomeTax(player : Player) : boolean
1
1
ForexManager
-forex : Forex
+calcSupDemand() : double
+resetExRates() : void
+updateExRates(currency : String, exRa : double) : void
+createExRates(dollarExRa : double, euroExRa : dou...
+getInstance() : ForexManager
1
*
0..*
*
1
PowerUpManager
-powerUpStack
-powerUpCrate
+pushPowerUp(powerUp)
+popPowerUp()
+getInstance() : PowerUpManager
+getRandomPowerUp() : PowerUp
1
<<Interface>>
PowerUpBehavior
+activate() : void
PowerUpCrate
-powerUps : PowerUp[0..*]
-price : double
+openPowerUpCrate() : PowerUp
+getPrice() : double
1
1
EarningPowerUpBehavior
+activate() : void
ForexPowerUpBehavior
+activate() : void
SlowDownPowerUpBehavior
+activate() : void
StrikePowerUpBehavior
+activate() : void

Figure 3.e. Forex, powerup, bank and trade related classes

3.2.2.4.Piece 4

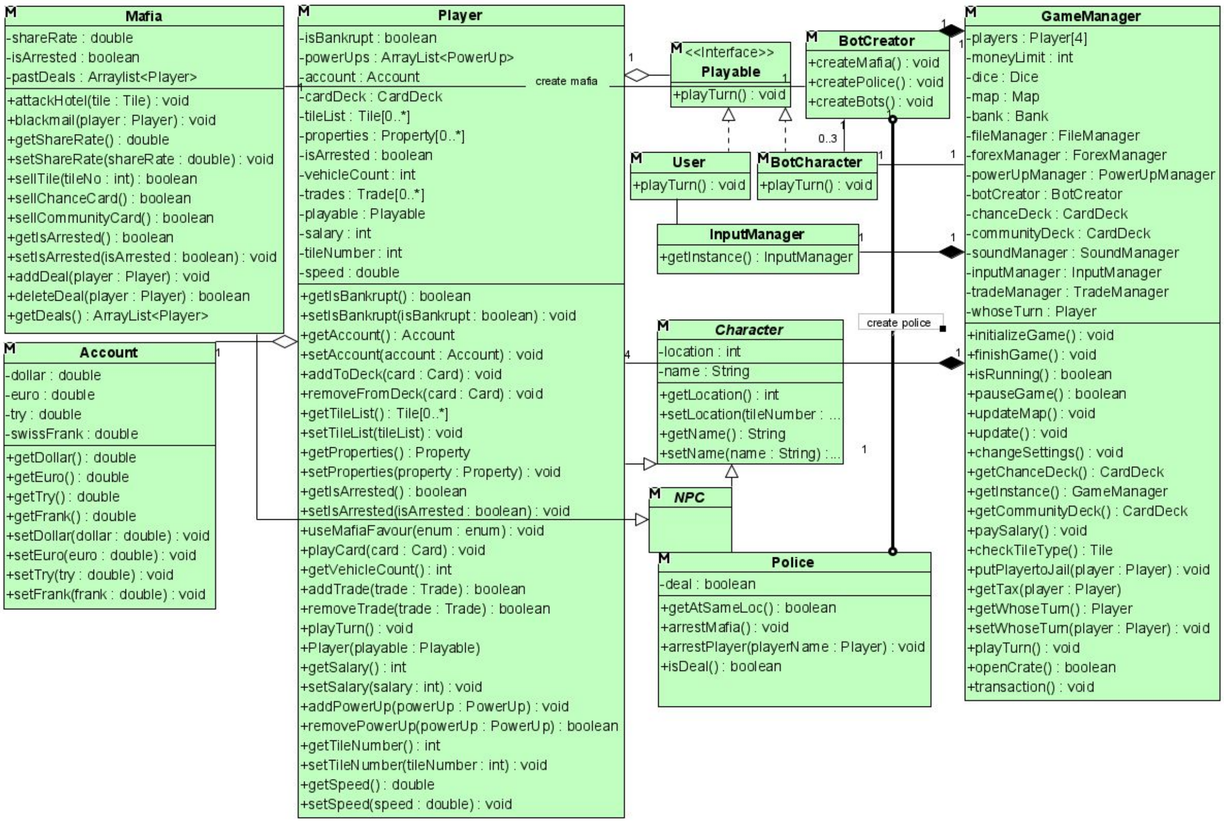
Mafia
-shareRate : double
-isArrested : boolean
-pastDeals : Arraylist<Player>
+attackHotel(tile : Tile) : void
+blackmail(player : Player) : void
+getShareRate() : double
+setShareRate(shareRate : double) : void
+sellTile(tileNo : int) : boolean
+sellChanceCard() : boolean
+sellCommunityCard() : boolean
+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
-vehicleCount : int
-trades : Trade[0..*]
-playable : Playable
-salary : int
-tileNumber : int
-speed : double
+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
+playCard(card : Card) : void
+getVehicleCount() : int
+addTrade(trade : Trade) : boolean
+removeTrade(trade : Trade) : boolean
+playTurn() : void
+Player(playable : Playable)
+getSalary() : int
+setSalary(salary : int) : void
+addPowerUp(powerUp : PowerUp) : void
+removePowerUp(powerUp : PowerUp) : boolean
+getTileNumber() : int
+setTileNumber(tileNumber : int) : void
+getSpeed() : double
+setSpeed(speed : double) : void
create mafia
1
<<Interface>>
Playable
+playTurn() : void
1
BotCreator
+createMafia() : void
+createPolice() : void
+createBots() : void
1
0..3
1
1
GameManager
-players : Player[4]
-moneyLimit : int
-dice : Dice
-map : Map
-bank : Bank
-fileManager : FileManager
-forexManager : ForexManager
-powerUpManager : PowerUpManager
-botCreator : BotCreator
-chanceDeck : CardDeck
-communityDeck : CardDeck
-soundManager : SoundManager
-inputManager : InputManager
-tradeManager : TradeManager
-whoseTurn : Player
+initializeGame() : void
+finishGame() : void
+isRunning() : boolean
+pauseGame() : boolean
+updateMap() : void
+update() : void
+changeSettings() : void
+getChanceDeck() : CardDeck
+getInstance() : GameManager
+getCommunityDeck() : CardDeck
+paySalary() : void
+checkTileType() : Tile
+putPlayertoJail(player : Player) : void
+getTax(player : Player)
+getWhoseTurn() : Player
+setWhoseTurn(player : Player) : void
+playTurn() : void
+openCrate() : boolean
+transaction() : void
User
+playTurn() : void
BotCharacter
+playTurn() : void
1
InputManager
+getInstance() : InputManager
1
1
create police
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
1
4
1
Character
-location : int
-name : String
+getLocation() : int
+setLocation(tileNumber : ...
+getName() : String
+setName(name : String) : ...
1
NPC
Police
-deal : boolean
+getAtSameLoc() : boolean
+arrestMafia() : void
+arrestPlayer(playerName : Player) : void
+isDeal() : boolean

Figure 3.f.Classes related to characters and players

3.2.3. Packages

3.2.3.1. Game Management Package

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

3.2.3.2. Entity Management Package

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

3.2.3.3. Entity Package

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

3.2.3.4. Graphical Interface Package

Includes the GUI classes like menus and pop-ups.

3.2.3.5. File Management Package

Includes packages which provide file management in the game.

3.2.4. 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(int): 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 String music: Integer value to control the sound of the music.
- private int volume: Integer value to control the sound level.
- public boolean activateSound(): Method to activate the muted sound.
- public boolean 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
- Public SoundManager getInstance(): Returns the current sound manager object.

FileManager:

- public void loadGame: Method to load the old game file.
- public void saveGame(): Method to save the game.
- Public FileManager getInstance(): Returns the current file manager object.

InputManager:

- public InputManager getInstance(): Returns the current input manager object.

Dice:

- Private int diceNo1: Value that represents first die value.
- Private int diceNo2: Value that represents first die value.
- 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.
- private CardStrategy cardStrategy: Chooses the card strategy
- private int amount: The amount for that card's effect.
- public String getFeature(): Returns the card’s feature
- Public void setFeature(String): Sets the card’s feature
- Public void activateCard(): Method to activate the card’s effect
- Public int getAmount(): Returns the card’s amount

- Public void setAmount(int): Sets the card’s amount

CardStrategy:

- Public void activateCard(): Method to activate the card’s effect

MovementByNumCardStrategy:

- Public void activateCard(): Method to activate the card’s effect

MovementToCardStrategy:

- Public void activateCard(): Method to activate the card’s effect

TransactionCardStrategy:

- Public void activateCard(): Method to activate the card’s effect

GetOutOfJailStrategy:

- Public void activateCard(): Method to activate the card’s effect

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.
- Public void setTile(Tile[0..*]): Sets the map’s tiles.

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:

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.

BuyableTile:

- Private boolean isOwned: Variable to show that the tile is owned or not
- Private Player whoseTile: Variable to show who is the owner
- Private int price: Price of the tile
- Private double rentAmount: Price of the rent when a player steps on it who is not the owner
- Private boolean soldByMafia: Variable to show if the tile was bought with the help of mafia
- Private boolean isMortgage: Variable to show if the tile is mortgaged
- Private double MORTGAGE_RATE: Constant to hold mortgage rate

TransportationTile:

- private int rentAmount: Variable that shows the amount of rent this tile demands

CompanyTile:

- private int rentAmount: Variable that shows the amount of rent this tile demands

CityTile:

- private ArrayList<Property> properties : A list that stores the properties on a city tile.

- Public int getHouseCount(): Returns the amount of houses on this tile
- Public boolean addHouse(): Adds a house to the tile if possible.
- Public int getHotelCount(): Returns the amount of hotels on this tile
- Public boolean addHotel(): Adds a hotel to the tile if possible
- Public boolean isHotelBuildAvailable(): Checks if the conditions to build a hotel is present

Property:

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

House:

Hotel:

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.
- Public PowerUpManager getInstance(): Returns the current powerup manager object.
- Public PowerUp getRandomPowerUp(): Returns a random powerup.

PowerUpCrate:

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

PowerUp:

- private int lifetime: Integer value which represents the powerups lifetime in terms of rounds.
- private PowerUpBehavior behavior: Encapsulates the activated behavior of the PowerUp.
- Private double amount: Has the amount of the effect the PowerUp has. This value is used behavior specific.
- Private String target: The variable that shows the targeted player of this power-up

PowerUpBehaviour:

- Public void activate(): Activates the PowerUp. Is implemented in other PowerUpBehavior subclasses.

EarningPowerUpBehaviour:

- Public void activate(): Activates the power-up

SlowDownPowerUpBehaviour:

- Public void activate(): Slows down the target player by the amount specified in the PowerUp class.

StrikePowerUpBehaviour:

- Public void activate(): Strikes the target player by the amount specified in the PowerUp class.

ForexPowerUpBehaviour:

- Public void activate(): Manipulates the target Forex by the amount specified in the PowerUp class.

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:

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:

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

Playable:

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

BotCharacter:

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

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.
- private int vehicleCount: The count of transportation tiles which a Player owns.
- private ArrayList<Trade> trades: The list of unresolved trades which involve the Player.
- private Playable playable: The play turn strategy which holds the function of playTurn().
- private int salary: The salary of the player.
- private int tileNumber: The number of the tile where the player is at.
- private double speed: The speed of the player.
- 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.
- public void playCard(Card card): This method plays the card specified in the CardDeck of the player.
- public boolean addTrade(Trade trade): This method adds a new trade associated with the Player. Returns true if the trade offer is legal, false if it is not.
- public boolean removeTrade(Trade trade): Removes a trade associated with the player. Returns the success of the removal in boolean.
- public void playTurn(): Plays the turn for the Player.
- public void addPowerUp(PowerUp powerUp): Adds the specified PowerUp to the powerUps of the Player.
- public void removePowerUp(PowerUp powerUp): Removes the specified PowerUp from the powerUps of the Player.

Trade:

- private Player owner: The player who wants to create a trade.
- private Player target: The player who the owner wants to trade with.
- private ArrayList<Tile> ownersTile: The ArrayList that stores the tiles of the owner that he wants to give.
- private ArrayList<Tile> targetTile: The ArrayList that stores the tiles of the target that the owner wants.
- private int offeredAmount: The amount of money, in Turkish liras, that the owner offers to the target player.
- private int requestedAmount: The amount of money, in Turkish liras, that the owner wants from the target player.

TradeManager:

- public TradeManager getInstance(): Gets the Singleton instance of the TradeManager class.
- public boolean openTrade(Player owner, Player targetPlayer): Opens a trade between two players. Returns if the process is successful, true if it is not.
- public boolean checkTrades(): A helper function for openTrade(), checks the eligibility of the trade.
- public void acceptTrade(): Accepts a trade offer and resolves it.
- public void denyTrade(): Denies and resolves a trade offer.

4. Improvement Summary

In our second iteration of the design stage of our project, we made several changes according to the feedback we received. First thing we changed was the class diagram. We implemented the strategy design pattern to our PowerUp, Card, User and Bot classes. In addition to that we applied inheritance to our Tile class and created a new abstract class called BuyableTile. Also we added several interface's to our class diagram to implement the strategy pattern. Other than that we made slight changes to some other class in order to make our design more sensible and easier to implement. Other than the class diagram we changed the component diagram. After the change the Dice component was removed from components. Since we added additional strategy design patterns to our class diagram, the low level design part of our report also changed accordingly. Also, the classes which the design patterns Singleton and Façade implemented are shown implicitly in the second iteration of the report. Lastly we moved the Object Design Trade-offs which was at the design patterns section to the Trade-offs section to organize all trade-offs in one section.

An overview of the improvement summary:

- Class diagram:
    - The strategy design pattern applied to PowerUps system, Card system, User and AI system.
    - Inheritance applied to tiles.
    - Some classes changed.
- Component diagram:
    - Dice is no longer a component.
    - Changed some of the components to resemble the more important parts.
- Low Level design:
    - Added the Strategy design pattern.
    - Mentioned the Facade, Singleton and Strategy classes.
- Moved the contents of Object Design Trade-offs to the Trade-offs section of our Design Goals section.

5. Glossary & references

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