Bilkent University

Department of Computer Engineering

CS319 Term Project

Monopoly Bilkent Edition

Group 1B

Iteration 1 Design Report

Group Members:

Melike Fatma Aydoğan - 21704043

Ahmet Cemal Alıcıoğlu - 21801700

Zübeyir Bodur - 21702382

Beril Canbulan - 21602648

Mehmet Çalışkan - 21802356

Instructor: Eray Tüzün

Teaching Assistants: Barış Ardıç, Elgun Jabrayilzade, Emre Sülün

Table of Contents

1. Introduction

1.1. Purpose of the System

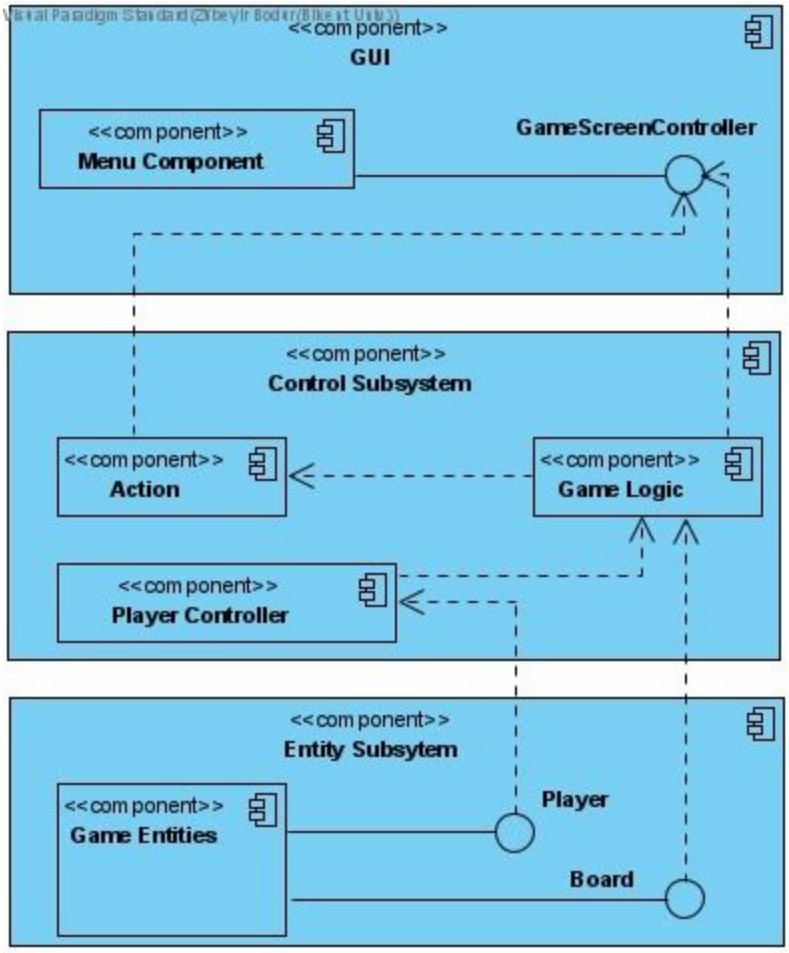
Monopoly is a board game where 2-6 players try to be the wealthiest person in the game through buying, selling, renting or even mortgaging. Purpose of our game, Monopoly Bilkent Edition, is to entertain its users and provide them with a simulation of the board game Monopoly™. With the aid of teams, trading and chat, the system also aims for interactive gameplay.

1.2. Design Goals

- **Usability**: Monopoly is a little bit complex because of its design and various components. We are planning to design our User Interface as clear as possible to not reduce this confusability of board version of the game. We will also add a section named “How to Play?” to guide players through our game.
- **Extendibility**: We are planning to add new features like Speed Die to the game and the game design lets us do this easily because of its clear design.
- **Modifiability**: Since an entity is used in numerous classes, modifiability is an important criteria. To achieve this modifiability, we stored our properties, tiles and cards in separate JSON files. By changing the content of these JSON files, we will be able to modify the game easily.
- **Portability**: Since the game needs to be played in various numbers of computers, the game needs to be portable. For this portability, we implemented the game with Java since JVM (Java Virtual Machine) is independent of the running system.
- **Performance**: Since it is a multiplayer game, the overall performance should be fast enough to not waste the users’ time. We are planning to use a network library named *KyroNet* to achieve these fast response times.

2. High-Level Software Architecture

2.1. Subsystem Decomposition



We decided to decompose our system into three subsystems (layers) so that we address our design goals. Topmost subsystem is called GUI. The one in the middle is called the Control Subsystem, which interacts with both subsystems. The one in the bottom is called the Entity Subsystem, which are game entities of the system.

2.2. Hardware / Software Mapping

Our system doesn't require an external hardware to connect to the PC. It only requires a keyboard, mouse and a monitor, which is found in a PC.

2.3. Persistent Data Management

We store JSON files of the saved games to the local storage of the server. We also have JSON files for properties, tiles and cards. We initialize our game board with these JSON files so modifying the attributes of our game is pretty easy. We also have image files for tokens and some tiles like Jail and Free Park.

2.4. Access Control and Security

Our system doesn't store private data such as passwords or any other personal data, it only asks users to enter a username so that they can differentiate a player from another during the game. Moreover, no verification was also required for users to play the game. So, a subsystem for access control or security was not necessary.

2.5. Boundary Conditions

2.5.1. Initialization

The program is executed using a jar file. After the user opens the jar file, the user may enter their username and press the Create New Game button and start playing the game.

2.5.2. Termination

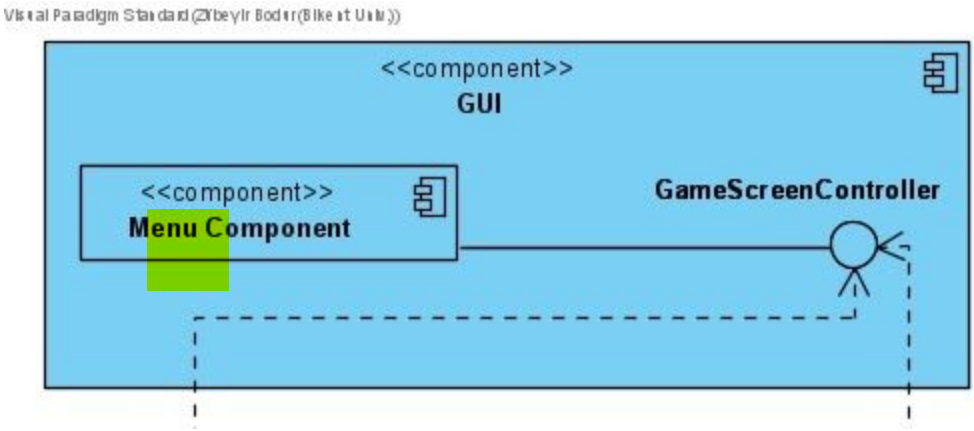
The program is closed when the user presses the X button on top right of the main screen. The game session is closed when the user presses the "Quit" button on top right of the game screen.

2.5.3. Failure

The system doesn't have a mechanism that detects failures for failure to load. The system also doesn't detect the errors due to performance issues. In both cases, the user will lose their data without being saved.

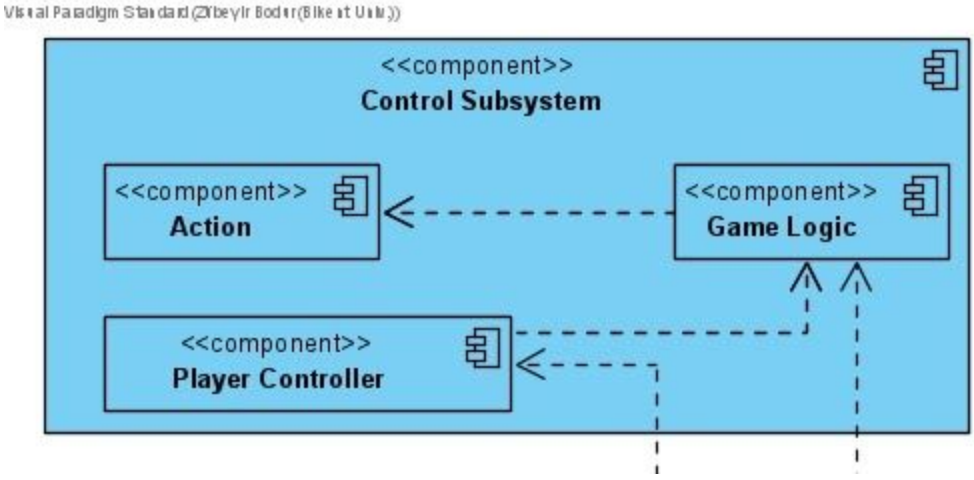
3. Subsystem Services

3.1. GUI Layer



This component stands for the GUI interfaces in the system, which navigates the user and displays the user interface. It has one subcomponent called Menu Component, which is every menu in the GUI component. It has an interface called GameScreenController, which interacts and communicates with the Control component.

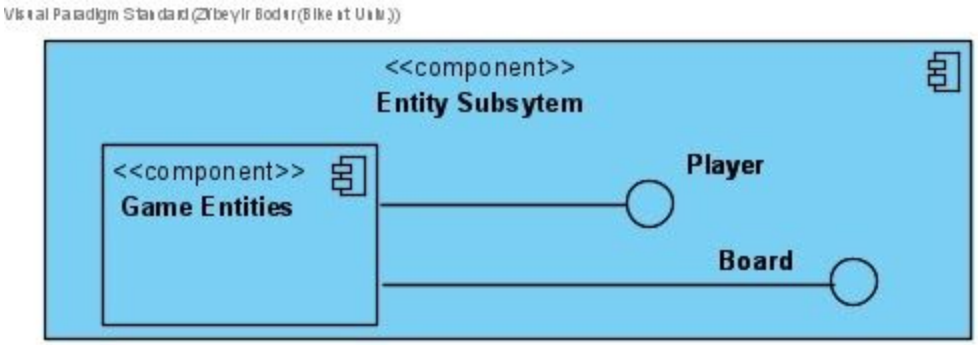
3.2. Control Layer



The Control component controls the gameplay. It has the following components, Action, Player Controller and Game Logic. Action component controls how the actions taken in the user interface will be done in the system. Player Controller

component controls the relation between the Game Logic component and Player interface in the Entity component. Finally, the Game Logic component controls the whole Entity component and it also depends on the GUI and action component.

3.3. Entity Layer



Entity components simply consist of game entities in the system. Board and Player interfaces communicate with the control components and are used by it.

4. Low Level Design

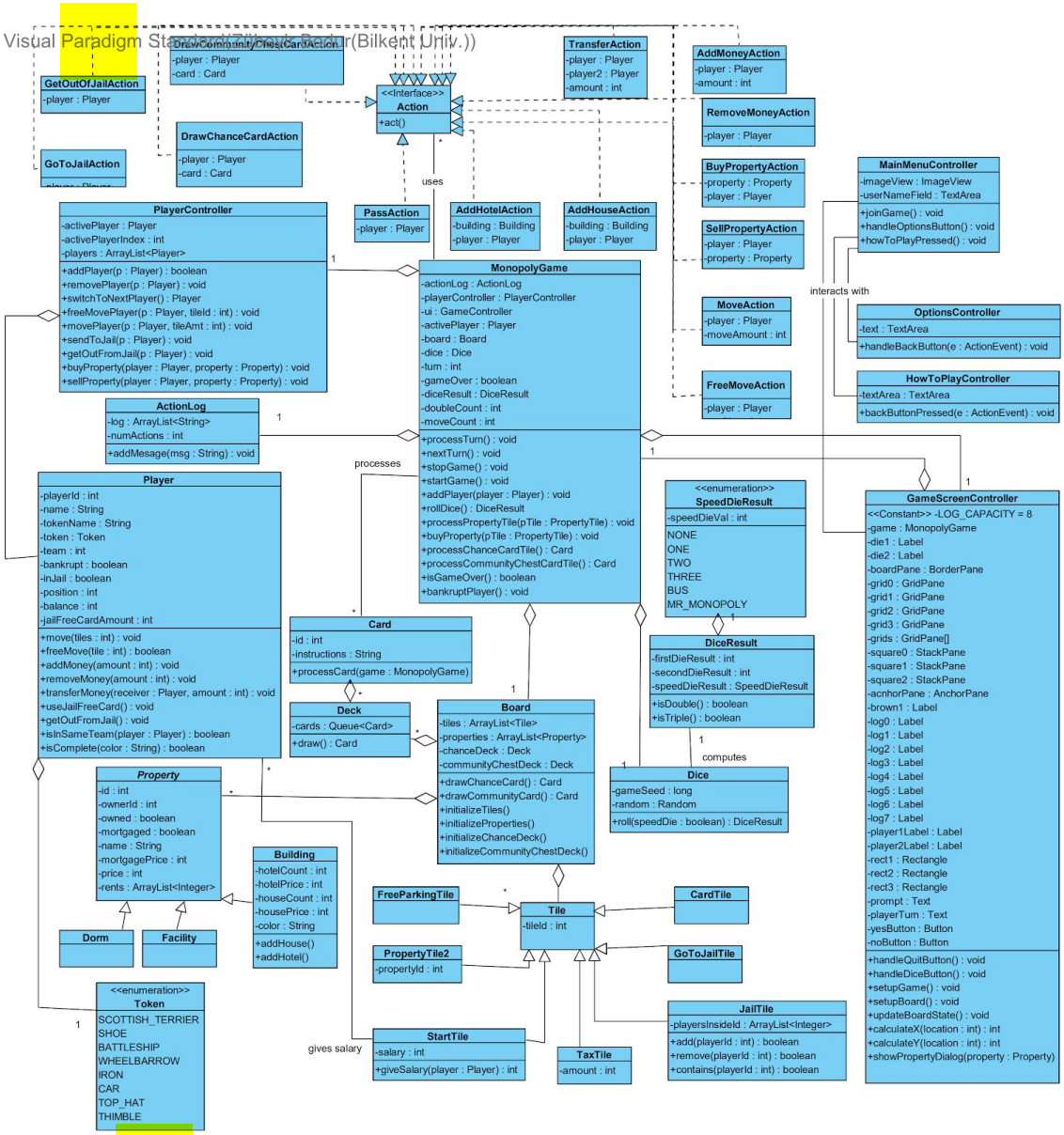
4.1. Design Patterns

In this system, MVC and Façade patterns are used during the design.

In the Control Layer, inside the Game Logic component, the MonopolyGame class is the Façade class. It provides services for Control layer, GUI Layer and Entity Layer at the same time.

The system is also composed in a way that it addresses the requirements of the MVC pattern: the Model objects are in the Entity Layer, View objects are in the GUI Layer and Control objects are in the Control Layer. Of those layers, the GUI layer doesn't directly connect with the Entity layer, meaning the user can't control the model directly, they have to use the GUI, which uses predefined controls in the Control Layer.

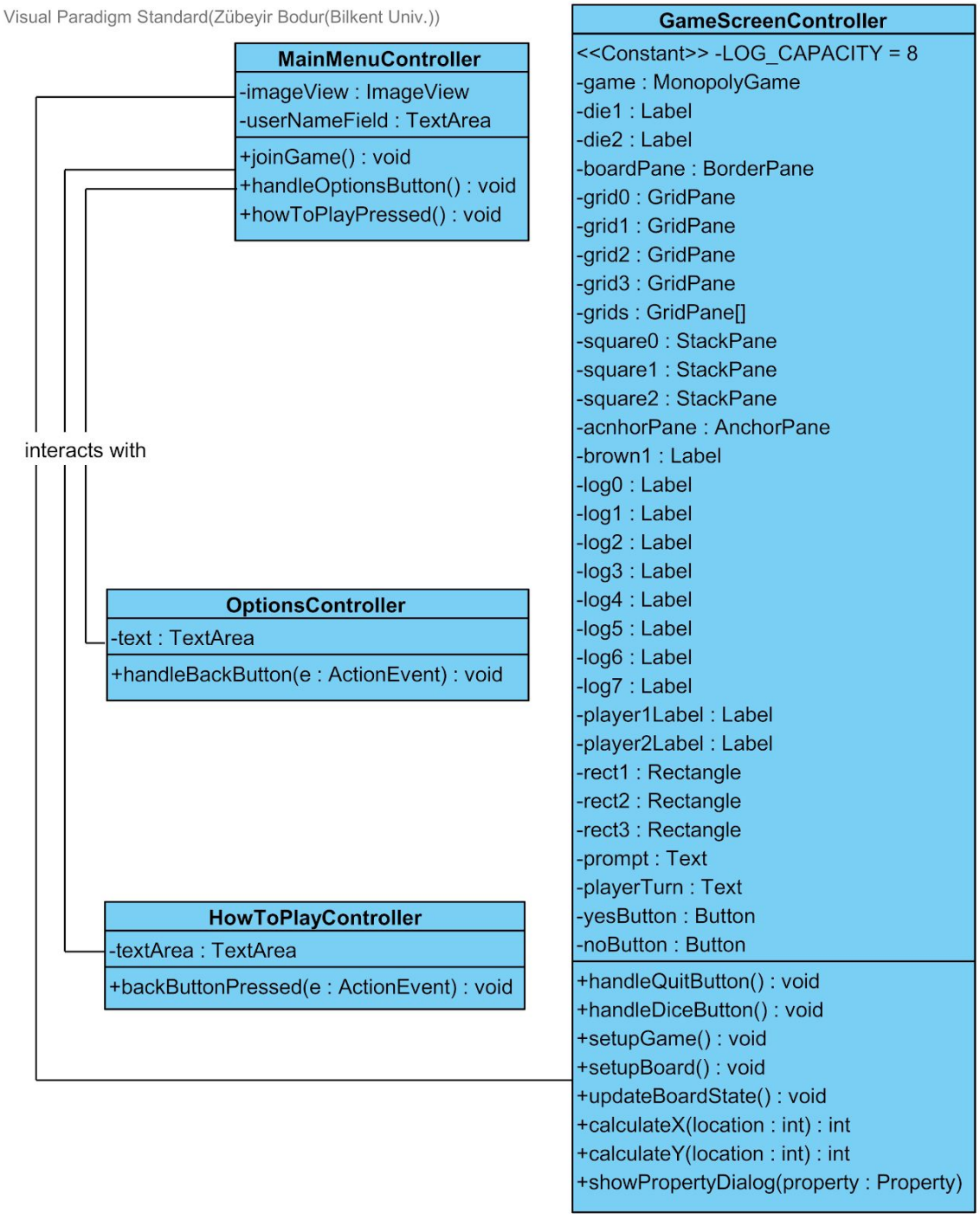
4.2. Final Object Design



In the following section, the object design is divided into three parts, called layers. Those layers match the ones defined in the subsystem services section of the report.

4.3. Layers

4.3.1. GUI Layer



GUI Layer is simply the gui package, which has the following classes: GameScreenController, MainMenuController, OptionsController, HowToPlayController.

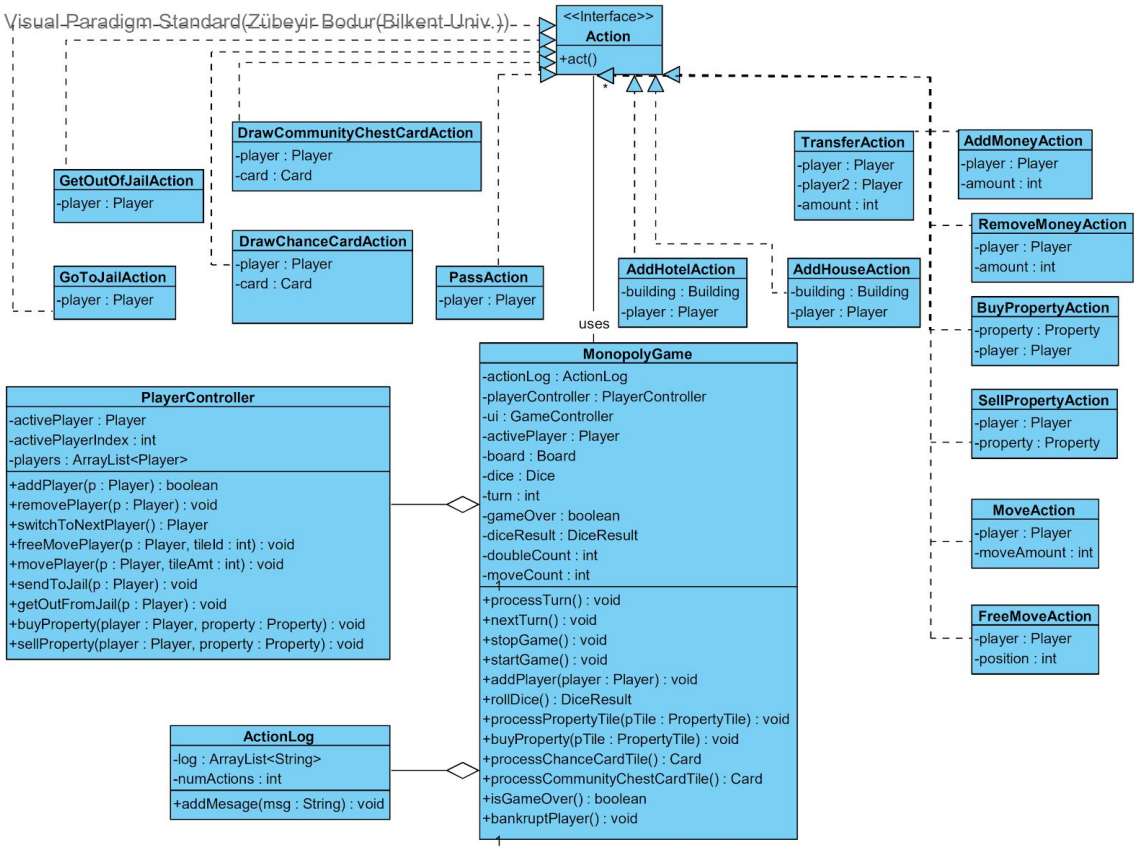
MainMenuController keeps track of events that happen in the main menu (defined in the main_menu.fxml).

OptionsController keeps track of events that happen in the options screen (defined in the options.fxml).

HowToPlayController keeps track of events that happen in the how to play screen (defined in the main_menu.fxml).

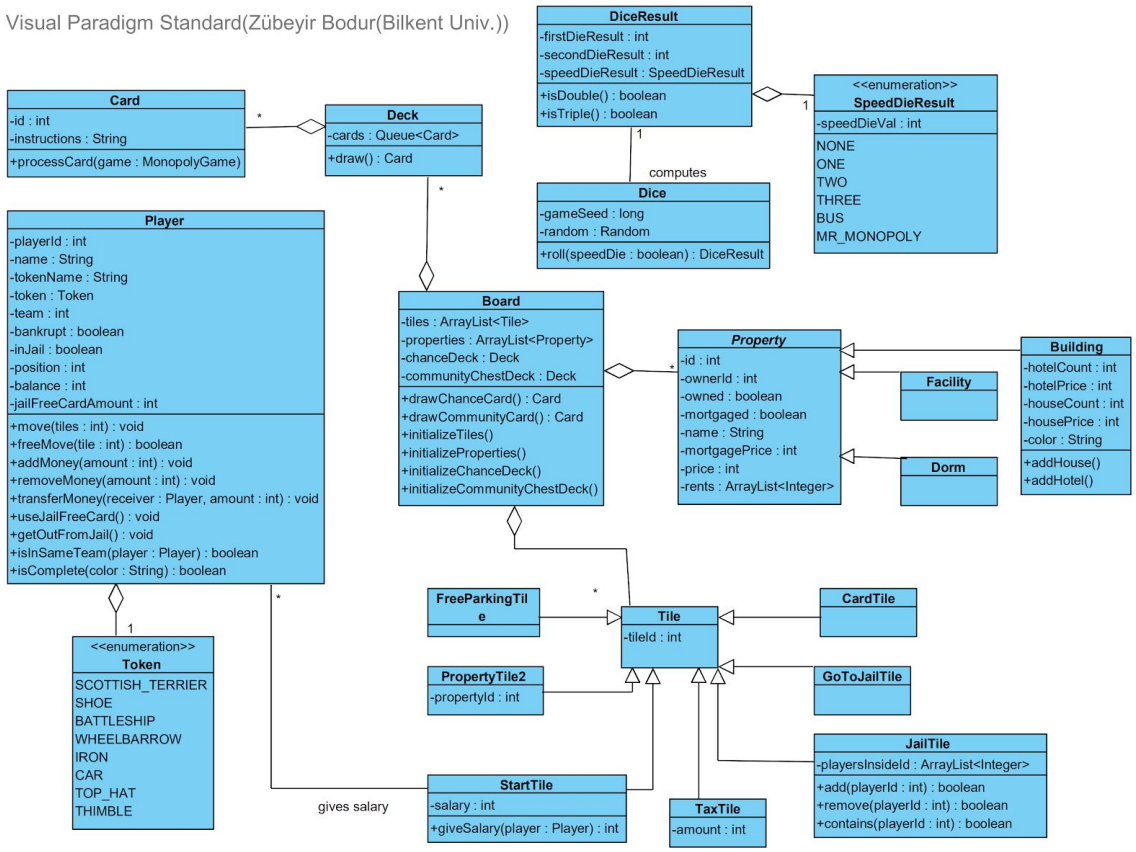
GameScreenController not only keeps track of events that happen in the game screen, but also informs the control objects of the system that those events happened. So, other than buttons and labels, it has the MonopolyGame class as an attribute, which is the instance of the game.

4.3.2. Control Layer



Control Layer is simply the control package (the game engine), which communicates between the Entity Layer and GUI Layer. It has another package called action, other class interfaces for the game engine, called MonopolyGame, PlayerController, and ActionLog class that holds the data for the previous actions happening throughout the game.

4.3.3. Entity Layer



Entity layer consists of three parts, one is Player entities, one is Board entities and other one is the Dice entities. Board entities are separated into three packages; card, property and tile.Dice entities are also gathered in one package called dice. In total, the Entity layer has four packages. For the Player entity, no package was necessary, we just put the enum Token, inside the Player class.

4.4. Packages

4.4.1. Internal Packages

entity Package

This package includes Board and Player class and card, dice, property and tile subpackages.

entity.card Package

Card package is used to represent Community Chest and Chance Decks.

entity.tile Package

This package has various classes to represent necessary tile types in the game board.

entity.dice Package

This package has necessary classes related to dice.

entity.property Package

This package includes classes that represent properties with different types.

entity.tile Package

The Monopoly board consists of tiles so that in this package different types of tiles exist.

control Package

This package consists of control objects in the MVC design pattern. It has the game logic, as MonopolyGame class, and controller classes such as PlayerController and action package.

control.action Package

This package includes the Action interface, which helps for implementing actions that occur in the game, such as buying a house/hotel, going to jail etc.

gui Package

This package has all the user interfaces possible used in the system. These are the main menu, game screen, options screen and how to play screen.

4.4.2. External Library Packages

4.4.2.1 java.util Package

We use the java.util package for ArrayList, Random, HashMap, Arrays and Objects classes. These classes are fundamental for our entities. Player, Board, Deck classes use ArrayList class to store multiple instances of an object. Dice class uses Random class to randomize the dice results. Player class uses the HashMap class to store properties of a player. It gives the property type as key and gets these types of properties of the player in an ArrayList as value. Deck class use Arrays class to convert ArrayList to Queue since Deck works like a queue. GameScreenController class uses Objects class for null check operation.

4.4.2.2 java.io Package

We use the java.io package for Reader, IOException and File classes. File and Reader classes are required for parsing the JSON files which hold our properties, tiles and cards. If the parsing operation is unsuccessful, the program throws IOException to notify the user.

4.4.2.3 java.nio Package

Java.nio package contains Files and Paths classes which is required for JSON parsing as mentioned above. We use these classes for parsing our JSON files which hold attributes of our game.

4.4.2.4 com.google.gson Package

GSON is a Java serialization/deserialization library made by Google for converting the objects into the JSON files and vice versa. We use the GSON package for parsing our JSON files. We use JsonDeserializationContext, JsonDeserializer, JsonElement, JsonParseException classes of the GSON package for this parsing operation.

4.5. Class Interfaces

Below is the detailed explanation for each class interface implemented so far.

4.5.1. GUI Layer Class Interfaces

4.5.1.1. GameScreenController

Attributes:

- **final int LOG_CAPACITY:** Determines the amount of action log messages that will be displayed on the game screen.
- **MonopolyGame game:** Determines the game controller that the user actions will be sent to.
- **Label die1:** Shows result of the first die.
- **Label die2:** Shows result of the second die.
- **BorderPane borderPane:** Border pane of the game screen, contains the GridPane's.
- **Grid grid0 → grid3:** Contains the grids which contain the tile squares for top, right, bottom and left of the screen.
- **GridPane[] grids:** Contains the grids above.
- **AnchorPane anchorPane:** Main pane of the game screen, contains all of the UI elements.
- **Label log0 → log7:** Contains one action log message for each label.
- **Label player1Label → player6Label:** Indicates the player's name, token and balance.
- **Text playerTurn:** Indicates which player has the turn and control.

Methods:

- **protected void handleQuitButton(ActionEvent)**: Listener for the quit button. Stops the game and returns to main menu.

- **protected void handleDiceButton(ActionEvent):** Listener for the dice button. Uses the rollDice function in the MonopolyGame class and updates the dice labels accordingly
- **public void setupGame(String):** Takes the username of the first player and sets up the game. Is designed to be used by MainMenuController for setting up the game before changing the scene.
- **private void setupBoard():** Sets up the tiles according to the data from the MonopolyGame object.
- **public void updateBoardState():** Updates action log, player balance, and locations of the tokens. Is public because it should be called whenever an Action is performed from the MonopolyGame class.
- **public boolean showPropertyDialog(Property):** Prompts the user if they want to buy / add house / add hotel to the property that they landed on. Returns the response of the user.

4.5.1.2. HowToPlayController

Attributes:

- **@FXML public TextArea textArea :** The TextArea object for the instructions given for the user. For now, it has the placeholder text “lorem ipsum”.

Methods:

- **protected void backButtonPressed(ActionEvent event) :** Describes the action to be taken when the back button is pressed in the how to play screen. If the button is pressed, this method is called and the user is navigated back to the main menu screen.

4.5.1.3. MainMenuController

Attributes:

- **@FXML ImageView imageView :** ImageView object that holds the attributes of the Mr. Monopoly image in the main menu screen.
- **@FXML TextArea textArea:** The TextArea object that informs the user that they will enter their username to the textbox in the main menu. There, it writes “Enter Username”.

Methods:

- **protected void joinGame(ActionEvent event) :** Describes the action to be taken when the join game button is pressed in the main menu. If the button is pressed, the game screen will be opened and what user entered to the textArea will be the user's username.
- **protected void handleOptionsButton(ActionEvent e) :** Describes the action to be taken when the options button is pressed in the main menu. If the button is pressed, the options screen will open. In the options screen, there is a back button and a text saying "Options are not yet implemented.".
- **protected void howToPlayPressed(ActionEvent event) :** Describes the action to be taken when the how to play button pressed in the main menu. If the button is pressed, the how to play screen will open.

4.5.1.4. OptionsController

Attributes:

- **@FXML Text text :** Says that this part is not implemented yet.

Methods:

- **protected void handleBackButton() :** This is a return button to the main menu controller since the game options are not implemented yet.

4.5.2. Control Layer Class Interfaces

4.5.2.1. Action

Methods:

- **public void act() :** Stands for realizing an action specified by the situation of the game. Those actions are adding a hotel to a player, buying a property etc.

4.5.2.2. AddHotelAction

Attributes:

- **Player player:** Determines which player adds the hotel.

- **Building building:** Determines which building will have the new hotel.

Constructors:

- **public AddHotelAction(Player player, Building building):** Action is created with the given data.

Methods:

- **public void act():** Performs the adding a hotel action to the specified building action.

4.5.2.3. AddHouseAction

Attributes:

- **Player player:** Determines which player adds the house.
- **Building building:** Determines which building will have the new house.

Constructors:

- **public AddHouseAction(Player player, Building building):** Action is created with the given data.

Methods:

- **public void act():** Performs the adding a house action to the specified building action.

4.5.2.4. AddMoneyAction

Attributes:

- **Player player:** Determines which player gets the money.
- **int amount:** Determines the amount of money that the player will get.

Constructors:

- **public AddMoneyAction(Player player, int amount):** Action is created with the given data

Methods:

- **public void act():** Performs the giving specified amount of money action to the player.

4.5.2.5. BuyPropertyAction

Attributes:

- **Player player:** Determines which player will buy the property.
- **Property property:** Determines the property that will be bought.

Constructors:

- **public BuyPropertyAction(Player player, Property property):** Action is created with the given data

Methods:

- **public void act():** Buys the property to the player.

4.5.2.6. DrawChanceCardAction

Attributes:

- **Player player:** Determines which player will draw the chance card.
- **Card card:** Determines the drawed card to register its contents.

Constructors:

- **public DrawChanceCard(Player player, Card card):** Action is created with the given data

Methods:

- **public void act():** Registers card and player information to the action log.

4.5.2.7. DrawCommunityChestCardAction

Attributes:

- **Player player:** Determines which player will draw the community chest card.
- **Card card:** Determines the drawed card to register its contents.

Constructors:

- **public DrawCommunityChestCard(Player player, Card card):** Action is created with the given data

Methods:

- **public void act():** Registers card and player information to the action log.

4.5.2.8. FreeMoveAction

Attributes:

- **Player player:** Determines which player will be moved.
- **int position**: Determines the position that player will be moved to.

Constructors:

- **public FreeMoveAction(Player player, int position):** Action is created with the given data

Methods:

- **public void act():** Moves the player into the specified position.

4.5.2.9. GetOutOfJailAction

Attributes:

- **Player player:** Determines which player will get out of jail.

Constructors:

- **public GetOutOfJailAction(Player player):** Action is created with the given data

Methods:

- **public void act():** Gets the player out of the jail.

4.5.2.10. GoToJailAction

Attributes:

- **Player player:** Determines which player will go to the jail.

Constructors:

- **public GoToJailAction(Player player):** Action is created with the given data

Methods:

- **public void act():** Puts the player into the jail.

4.5.2.11. MoveAction

Attributes:

- **Player player:** Determines which player will move.
- **int moveAmount:** Determines the move amount.

Constructors:

- **public MoveAction(Player player, int moveAmount):** Action is created with the given data

Methods:

- **public void act():** Moves the player by specified amount of squares.

4.5.2.12. PassAction

Attributes:

- **Player player:** Determines which player will get the money from passing the “Go!” tile.

Constructors:

- **public PassAction(Player player):** Action is created with the given data

Methods:

- **public void act():** Gives 20000$ to the specified player because of passing the “Go!” tile.

4.5.2.13. RemoveMoneyAction

Attributes:

- **Player player:** Determines which player will lose money
- **int amount:** Determines the amount of money that player will lose.

Constructors:

- **public RemoveMoneyAction(Player player, int amount):** Action is created with the given data

Methods:

- **public void act():** Removes the specified amount of money from the player.

4.5.2.14. SellPropertyAction

Attributes:

- **Player player:** Determines which player will sell the property.
- **Property property:** Determines which property will be sold.

Constructors:

- **public SellPropertyAction(Player player, Property property):** Action is created with the given data

Methods:

- **public void act():** Sells the specified property from the player.

4.5.2.15. TransferAction

Attributes:

- **Player player:** Determines which player will give the money.
- **Player player2:** Determines which player will take the money.
- **int amount:** Determines the amount of money that will be transferred.

Constructors:

- **public TransferMoneyAction(Player player, Player player2, int amount):** Action is created with the given data

Methods:

- **public void act():** Transfer the specified amount of money from a player to another player.

4.5.2.16. ActionLog

Attributes:

- **final int MAX_LOG:** Determines the maximum amount of messages that log will create.
- **ArrayList<String> log**: Holds the log messages.
- **int numActions:** Determines the number of actions that ActionLog processed.

Constructors:

- **public ActionLog():** Creates the ActionLog and initializes the attributes. It doesn't require any parameters since it is independent from the game logic.

Methods:

- **public void addMessage(String s):** Adds the specified message to the log and prints it to the system console.

4.5.2.17. MonopolyGame

Attributes:

- **Board board :** The instance of the Board class, that stores the tiles, properties, community and chance decks.
- **int turn :** Number of turns so far in the MonopolyGame. Each time next turn() is called, this attribute is incremented.
- **static ActionLog actionLog :** The instance of the ActionLog, that holds the previous actions that happened during the game.
- **PlayerController playerController :** The instance of the Controller class for Player, that manages how players are added to the game, stores the players and stores the id of the active player.
- **boolean gameStarted = false :** The boolean that tells if the game started, with the initial value false.
- **boolean gamePaused = false :** The boolean that tells if the game is paused, with the initial value false.

- **int doubleCount = 0 :** The count of consecutive double rolls that activePlayer in the playerController have made. Initial value is 0.
- **int moveCount = 0 :** The sum of dices in the DiceResult.
- **Dice dice :** The instance of the Dice class that will be “rolled” in the game
- **DiceResult diceResult :** The instance of DiceResult, that is the result of the roll of the current Dice in a turn.
- **GameScreenController ui :** The instance of the GameScreenController, where users will give orders to take actions in the game.

Constructors:

- **public MonopolyGame(ArrayList<Player> players, GameScreenController ui) :** Initializes a new game with given players and GameScreenController that uses it. In this way, there is a two-way connection between Control and GUI Layers.

Methods:

- **public void addPlayer(Player player) :** Adds a player to the playerController.
- **public void stopGame() :** Stops the game, simply sets the gameStarted to false.
- **public void startGame() :** Starts the game, simply sets the gameStarted to true.
- **public DiceResult rollDice() :** Rolls the dice attribute, simply calls dice.roll(false), as we don’t test speed die mode yet. Then, adds a message to the action log that tells the player the result of the dice. Also sets the moveCount.
- **public void processTurn() :** The turn is processed according to the current attributes of the MonopolyGame object.
- **public void nextTurn() :** If the result of the dice is not a double or there is consequently three double rolls, playerController switches to the next player. Else, turn is incremented.

- **public void processPropertyTile(PropertyTile tile) :** The actions that a player can take when they land on a property tile is processed in this method.

- **public void buyProperty(PropertyTile tile) :** This method lets the active player buy the property located in said property tile.

- **public Card processChanceCardTile() :** This method processes the chance card tile that is at the top of the chance card. It is drawn first, and then the corresponding actions are taken.

- **public Card processCommunutyChestCardTile() :**This method processes the chance card tile that is at the top of the community chest card. It is drawn first, and then the corresponding actions are taken.

- **public boolean isGameOver() :** This method returns if the game is over or not. It simply checks if there are enough players that are not bankrupt.

- **public void bankruptPlayer(Player player) :** This method bankrupts a player, setting the bankrupt boolean of the player to true.

4.5.2.18. PlayerController

Attributes:

- **private ArrayList<Player> players :** List of players in the game
- **private Player activePlayer = null :** Instance of the active player, initially null.
- **private int activePlayerIndex = 0 :** Index of the active player in the players list.

Constructors:

- **public PlayerController(ArrayList<Player> players) :** Creates a PlayerController with a given list of players.
- **public PlayerController(Player Controller playerController) :** Copy constructor for the PlayerController

Methods:

- **public Player getById(int id) :** Gets the player object from the given playerId
- **public switchToNextPlayer() :** Switches the turn to the next player.
- **public boolean addPlayer(Player player) :** Adds a new player to the controller
- **public void removePlayer(Player player) :** Removes a player from the controller
- **public void getOutFromJail(Player player) :** Lets a player get out from the jail.
- **public void freeMovePlayer(Player player, int position) :** Lets a player make a free move to the given position, where the position is a tile id.
- **public void movePlayer(Player player, int moveAmount) :** Lets a player move a given amount.
- **public void buyProperty(Player player, Property property) :** Lets a player buy a given property.
- **public void sellProperty(Player player, Property property) :** Lets a player sell a given property.

4.5.3. Entity Layer Class Interfaces

4.5.3.1. Board

Attributes:

- **ArrayList<Tile> tiles :** Holds the tiles that it includes.
- **ArrayList<Property> properties :** Holds the properties that it includes.
- **Deck chanceDeck :** Deck of the chance cards.
- **Deck communityChestDeck :** Deck of the community chest cards.

Constructors:

- **public Board(ArrayList<Tile> tiles, ArrayList<Property> properties, Deck chanceDeck, Deck communityChestDeck) :** With the given data Board is created.
- **public Board( ) :** Initialize tiles, properties, card decks from the file.

- **public Board(Board savedBoard) :** Creates the board with copying the given savedBoard.

Methods:

- **public void initializeTiles(String filename) :** From the given filename, tiles are created.
- **public void initializeChanceCardDeck(String filename) :** From the given filename, chance card deck is created.
- **public void initializeCommunityChestDeck(String filename) :** From the given filename, community chest card deck is created.
- **public Property getPropertyById(int id) :** Return the property with the given id.
- **public Card drawCommunityChestCard () :** Returns the card that is in the top of the community chest card deck.
- **public Card drawChanceCard () :** Returns the card that is in the top of the chance card deck.

4.5.3.2. Player

Attributes:

- **public enum Token{ SCOTTISH_TERRIER, SHOE, BATTLESHIP, WHEELBARROW, IRON, CAR, TOP_HAT, THIMBLE } :** Specify the players' token.
- **private int playerId :** Id of the player.
- **private String name :** Name of the player.
- **private Token token :** Players token.
- **private int position :** The position of the player in the tail.
- **private boolean bankrupt :** The information that shows whether the player is in the jail or not.
- **private int doubleCounter :** The number of consecutive doubled dice that the player rolled within a turn.
- **private int jailFreeCardAmount :** The amount of the jail free card of the player has owned.
- **private int teamNumber :** Holds the information of the players team number.
- **private HashMap<String, ArrayList<Property>> properties :** Holds the properties of the player within a map.
- **private boolean inJail :** Holds the information of whether the player is in the jail or not.

Constructors:

- **public Player(int playerId, String name, Token token, int TeamNumber) :** Creates a player with given information.
- **Player(Player player) :** Creates a player with copying the given player.

Methods:

- **public void removeMoney(int amount) :** Remove the amount of money from the balance of the player.
- **public void transferMoney(Player player, int amount) :** Transfer the money between players.
- **public void addMoney(int amount) :** Add the amount of money to the balance of the player.
- **public boolean move(int tiles) :** Player is moved the number of tiles that is given. Return true if the player is passed the Start Tile otherwise return false.
- **public void freeMove (int tiles) :** Player is moved to the given tile.
- **public boolean useJailFreeCard() :** When the player is in jail and has at least one jail free card, the player will be released from the jail.
- **public boolean getOutFromJail (int fee) :** The player will be released by giving the fee.
- **public boolean isInSameTeam(Player player) :** return whether the two players are in the same team or not.
- **public boolean isComplete(Building building) :** It returns whether the player has each building that has the same color with the given building or not.

4.5.3.3. Card

Attributes:

- **int id :** to specify the cards
- **String instructions :** instructions of an card

Constructors:

- **public Card() :** Creates a card without specifying information.
- **Card(Card c) :** Creates a card with copying the given card
- **Card(int int, String instructions) :** Creates a card with given id and instructions.

Methods:

- **void processCard(monopolyGame:MonopolyGame) :** According to the id of the card, necessary functions of MonopolyGame are called to make actions.

4.5.3.4. Deck

Attributes:

- **Queue<Card> cards :** Deck class collects many Cards within a queue.

Constructors:

- **public Deck(String filename) :** Creates a deck from getting information of Cards from a file.
- **public Deck(Deck savedDeck) :** Creates a deck with copying the given deck

Methods:

- **Card draw() :** Returns the card at the front of the deck and puts it back to the end of the deck.

4.5.3.5. Dice

Attributes:

- **private final long gameSeed :** It is used to generate the random variable.
- **private final Random random :** It is created with gameSeed and used to generate dice results.

Constructors:

- **public Dice(long gameSeed) :** Creates a dice with a given game seed.
- **public Dice(Dice savedDice) :** Creates a dice with copying the given dice

Methods:

- **DiceResult roll(boolean isSpeedDie) :** Returns the dice result according to game type whether it is with speed die or not.

4.5.3.6. DiceResult

Attributes:

- **private final int firstDieResult :** It holds the value of the first die.
- **private final int secondDieResult :** It holds the value of the second die.
- **private final SpeedDieResult speedDieResult :** It holds the value of the speed die.

Constructors:

- **public DiceResult(int firstDieResult, int secondDieResult, int speedDieResult) :** Creates the dice result with speed die and two dice.
- **public DiceResult(int firstDieResult, int secondDieResult) :** Creates the dice result with two dice.

Methods:

- **boolean isDouble( ) :** Returns true if first and second dice are the same otherwise returns false.
- **boolean isTriple( ) :** Returns true if first, second and speed dice are the same otherwise returns false.

4.5.3.7. SpeedDieResult

Attributes:

- **int speedDieVal :** Holds the value of the SpeedDieResult enum as an integer. Those values are assigned so that a speed die wouldn't contribute to the sum of the dice if the outcome was Mr. Monopoly, Bus or the game mode didn't have a speed die (NONE). The values for each enums are the following:
  NONE : 0
  ONE : 1
  TWO : 2
  THREE : 3
  BUS : 0
  MR_MONOPOLY : 0

4.5.3.8. Property

Attributes:

- **String name :** Name of the property.
- **int id :** Id of the property.
- **int price :** Price of the property.
- **ArrayList<Integer> rent :** Lists of the rents of the property.
- **int mortgagePrice :** The mortgage price of the property.
- **boolean isMortgaged :** Holds whether the property is mortgaged or not.
- **boolean isOwned :** Holds whether the property is owned or not.
- **int ownerId :** If the property is owned, its owner's id is stored.

Constructors:

- **public Property(String name, int id, int price, ArrayList<Integer> rents, int mortgagePrice) :** According to the given values, the property is created.

Sub Class:

- **public static class CustomDeserializer implements JsonDeserializer<Property> :** It has a function that is public Property deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context) throws JsonParseException. In the deserialize function, the

property data from the JSON file is categorized into Dorm, Facility and Building classes.

4.5.3.9. Building

Attributes:

- **int housePrice :** price of the house for this building.
- **int hotelPrice :** price of the hotel for this building.
- **String color :** To classify buildings according to the colors.
- **int houseCount :** The number of houses builded within this building.
- **int hotelCount :** The number of hotels builded within this building.

Constructors:

- **public Building(String name, int id, int price, ArrayList<Integer> rents, int mortgagePrice, int housePrice, int hotelPrice, String Color) :** According to the given values, the building is created with 0 house and 0 hotel.

Methods:

- **public void addHouse() :** It increases the houseCount with one.
- **public void addHotel() :** It increases the hotelCount with one.

4.5.3.10. Dorm

Dorm class extends Property class and it has only one constructor that uses the parents classes constructor.

- **public Dorm(String name, int id, int price, ArrayList<Integer> rents, int mortgagePrice)**

4.5.3.11. Facility

Similar to Dorm class, Facility class has only one constructor.

- **public Facility(String name, int id, int price, ArrayList<Integer> rents, int mortgagePrice)**

4.5.3.12. Tile

Attributes:

- **private final int tileId :** To specify tiles tileId is used.

Constructors:

- **public Tile( ) :** A tile is created with id equals to 0.
- **public Tile(int tileId) :** with the given tileId a Tile is created.

Sub Class:

- **public static class CustomDeserializer implements JsonDeserializer<Tile> :** It has a function that is public Tile deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context) throws JsonParseException. In the deserialize function, the tile data from the JSON file is categorized into StartTile, PropertyTile, CardTile, TaxTile, JailTile, FreeParkingTile and GoToJailTile classes.

4.5.3.13. CardTile

Attributes:

- **public enum CardType{CHANCE_CARD, COMMUNITY_CHEST_CARD} :** To specify the card tiles according to its card type.
- **CardType cardType :** to represent card tiles type.

Constructors:

- **public CardTile(int tileId, cardType cardType) :** A card tile is created with given id and card type.
- **public Tile(CardTile savedTile) :** According to the given tile, new CardTile is created.

4.5.3.14. FreeParkingTile

Constructors:

- **public FreeParkingTile(int tileId) :** A free parking tile is created with the given id.
- **public FreeParkingTile(FreeParkingTile savedTile) :** According to the given tile, a new FreeParkingTile is created.

4.5.3.15. GoToJailTile

Constructors:

- **public GoToJailTile(int tileId) :** A go to jail tile is created with the given id.
- **public GoToJailTile(GoToJailTile savedTile) :** According to the given tile, a new GoToJailTile is created.

4.5.3.16. JailTile

Attributes:

- **private final ArrayList<Integer> playersInsideId :** The list for players id who are in the jail.

Constructors:

- **public JailTile(int tileId) :** A jail tile is created with the given id with an empty players list.
- **public JailTile(JailTile savedTile) :** According to the given tile, a new JailTile is created.

Methods:

- **public boolean add(int playerId) :** Adds a player id, whose gets into jail, into playersInsideId list. Return true, if addition is successful otherwise returns false.
- **public boolean remove(int playerId) :** Removes a player id, whose player was released from the jail, from the playersInsideId list. Return true, if removal is successful otherwise returns false.
- **public boolean contains(int playerId) :** Return true, if the given player id is in the list otherwise it returns false.

4.5.3.17. PropertyTile

Attributes:

- **private final int propertyId :** Id of the property in this tile.

Constructors:

- **public PropertyTile(PropertyTile propertyTile) :** According to the given tile, a new PropertyTile is created.
- **public PropertyTile(int tileId, int propertyId) :** A property tile is created with the given tile id and property id.

4.5.3.18. StartTile

Attributes:

- **private final int salary :** The amount that players will get when they cross this tile.

Constructors:

- **public StartTile(StartTile startTile) :** According to the given tile, a new StartTile is created.
- **public StartTile(int tileId, int salary) :** A start tile is created with the given tile id and salary.

Methods:

- **public void giveSalary(Player player) :** Add the salary to the given player.

4.5.3.19. TaxTile

Attributes:

- **private final int amount :** The amount that players will pay the tax when they land to this tile.

Constructors:

- **public TaxTile(TaxTile taxTile) :** According to the given tile, a new TaxTile is created.
- **public TaxTile(int tileId, int amount) :** A tax tile is created with the given tile id and amount of the tax.