Bilkent University

Department of Computer Engineering

CS319 Term Project

Monopoly Bilkent Edition

Group 1B

Iteration 2 Design Report

Project 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 multiplayer 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 clearly as possible to not reduce this confusability of the 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 not to waste the users' time. We are planning to use a network library named *KryoNet* to achieve these fast response times.

2. High-Level Software Architecture

2.1. Subsystem Decomposition

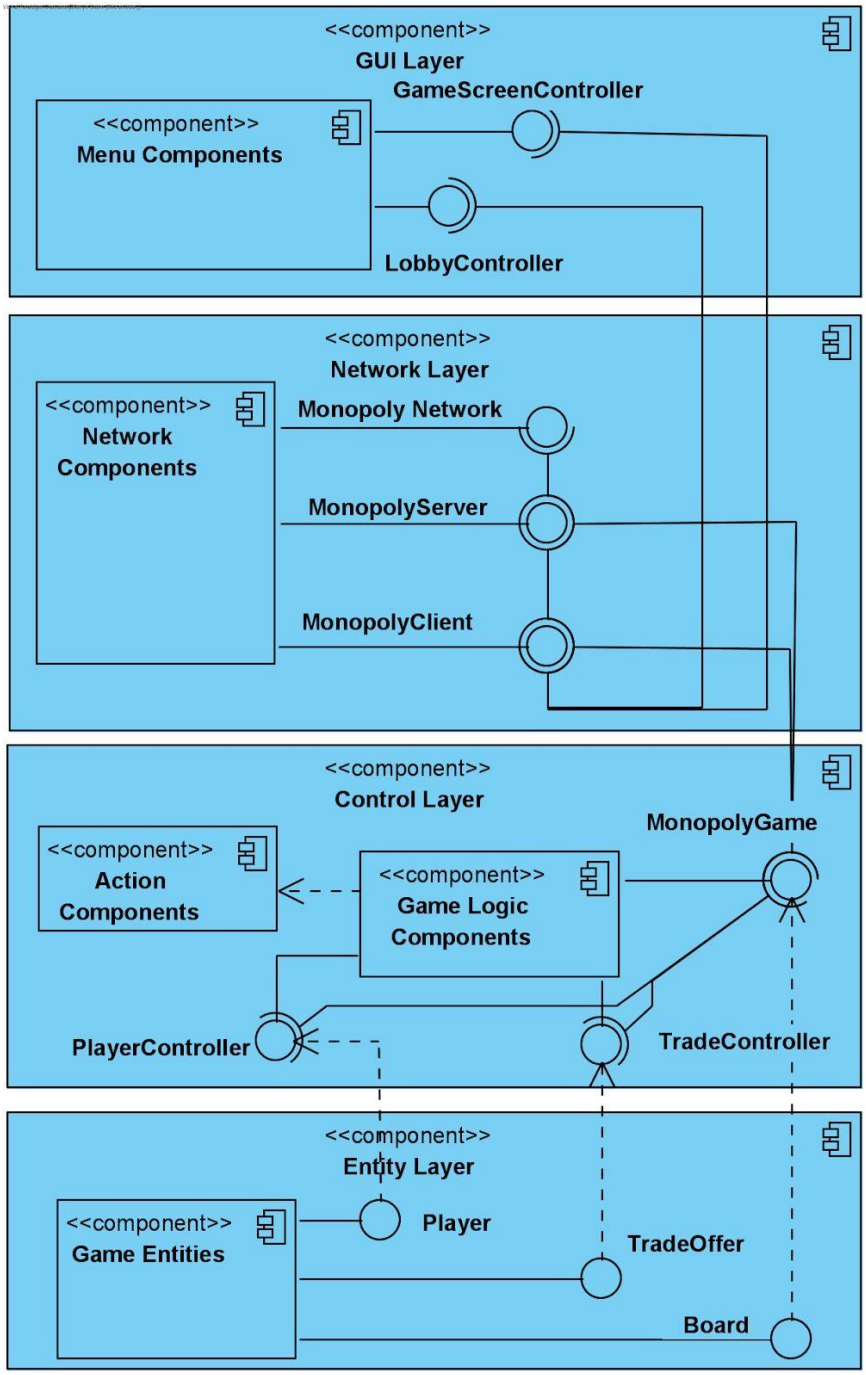

Figure 1 : Subsystem Decomposition of the game

We decided to decompose our system into four layers so that we not only address our design goals, but also have the flow going for the multiplayer functionality of the game. Such a decomposition was made so that we not only have an MVC like architecture (GUI for View, Controller for Controller, Entity for Model), but also separate services for multiplayer functionality from all other services.

Topmost subsystem is called the GUI Layer. It only interacts with the Network Layer and gets all the required information from that layer.

The layer below the GUI layer is called Network layer. It not only provides services for multiplayer functionality of the system, but also links the Control Layer with GUI Layer.

The one below the Network Layer is called the Control Layer, which communicates with Entity Layer and Network Layer.

The one in the bottom is called the Entity Subsystem, which are the game entities of the system.

2.2. Hardware / Software Mapping

Our system doesn't require an external hardware to connect to the PC. However, Java SE Runtime Environment 8 (JRE) is needed to be installed to this PC in order for our system to be working. Moreover, a stable internet connection is required as our system doesn't have a single player mode, it is only multiplayer.

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 or Join Game button and start playing the game. Players have two options to play multiplayer. In the first option, players need to connect into the same network or install a VPN application like ZeroTier, LogMeIn Hamachi. In the other option, the player who will create the game needs to forward the port 54555 with TCP on his/her router.

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

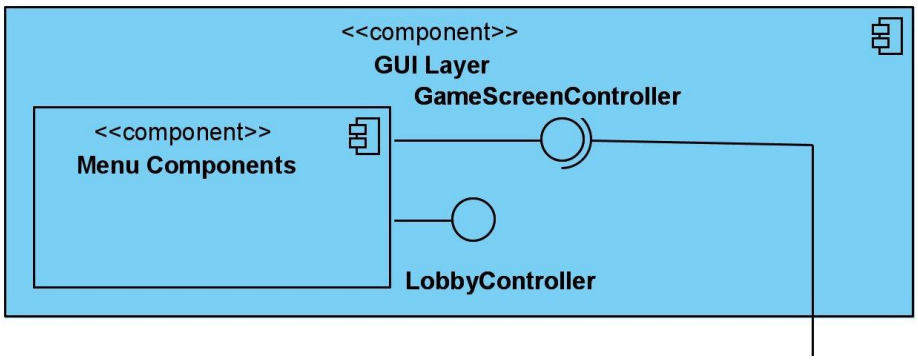

Figure 2 : GUI Layer Component from the component diagram

This layer stands for the GUI interfaces in the system, which navigates the user and displays the user interface. The component called *Menu Components* refers to all menu interfaces and files required for the GUI. The following interfaces are shown so that interaction between GUI and network is better depicted; Lobby Controller, GameScreenController. The rest of the interfaces (OptionsController, HowToPlayerController, MainMenuController) in GUI Layer are not shown for simplicity, as they don't interact with Network Layer.

GameScreenController controls the game screen and LobbyController controls the lobby screen. To do this, both communicate with the MonopolyClient interface of the Network Layer.

3.2. Network Layer

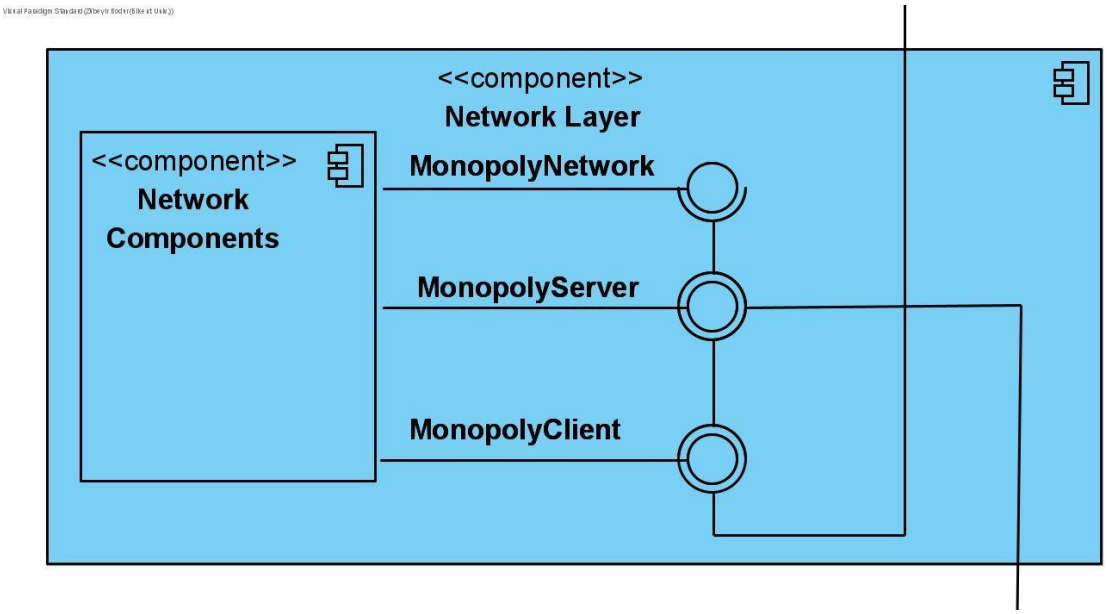

Figure 3 : Network Layer Component from the component diagram

This layer provides services for the multiplayer feature of our system using *KryoNet*. Components of this layer, called network components, are three interfaces called MonopolyNetwork, MonopolyServer and MonopolyClient.

MonopolyClient is the interface that communicates with its MonopolyGame interface to get the application data and sends it to the target views, to the specific GameScreenController and LobbyController.

MonopolyServer is the interface that communicates with all instances of MonopolyClient interface. It also communicates with the MonopolyNetwork interface.

3.3. Control Layer

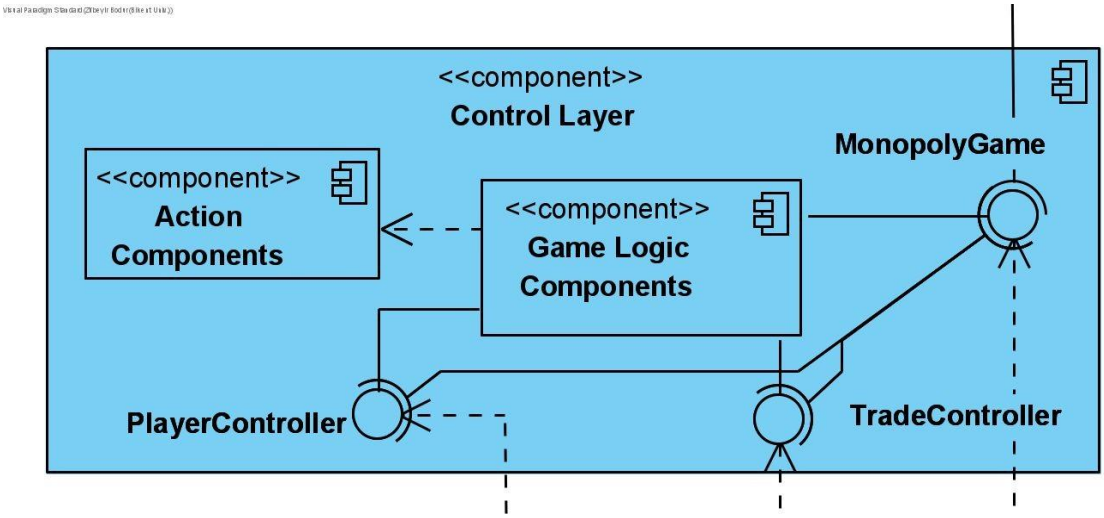

Figure 4 : Control Layer Component from the component diagram

The Control layer controls the gameplay. It has two components; Action and Game Logic. Action components help how Game Logic Components change during the game.

Game Logic Components control the whole Entity Layer, using MonopolyGame, PlayerController and TradeController interfaces. Those two simply control the interfaces that depend on them (Player and TradeOffer).

MonopolyGame interface is used by both MonopolyClient and MonopolyServer interfaces in the Network Layer. It also uses TradeController and PlayerController interfaces.

3.4. Entity Layer

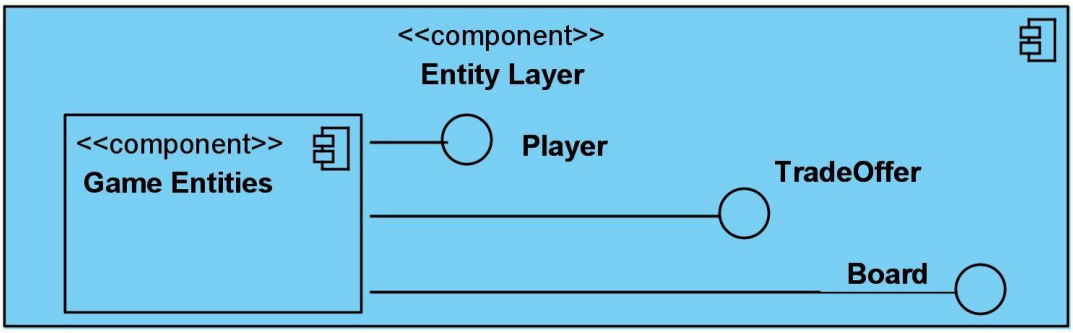

Figure 5 : Entity Layer Component from the component diagram

Entity components simply consist of game entities in the system. Board, Player and TradeOffer interfaces depend on the Control Layer components.

4. Low Level Design

4.1. Design Patterns

In this system, Singleton, Façade and Command 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 Network Layer, Control Layer and Entity Layer at the same time.

In addition, there are Singleton classes that are initiated once and used throughout the game, such as MonopolyClient, ActionLog and MonopolyServer classes.

Our Action classes use the command design pattern since they encapsulate all the data for performing an action.

4.2. Final Object Design

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

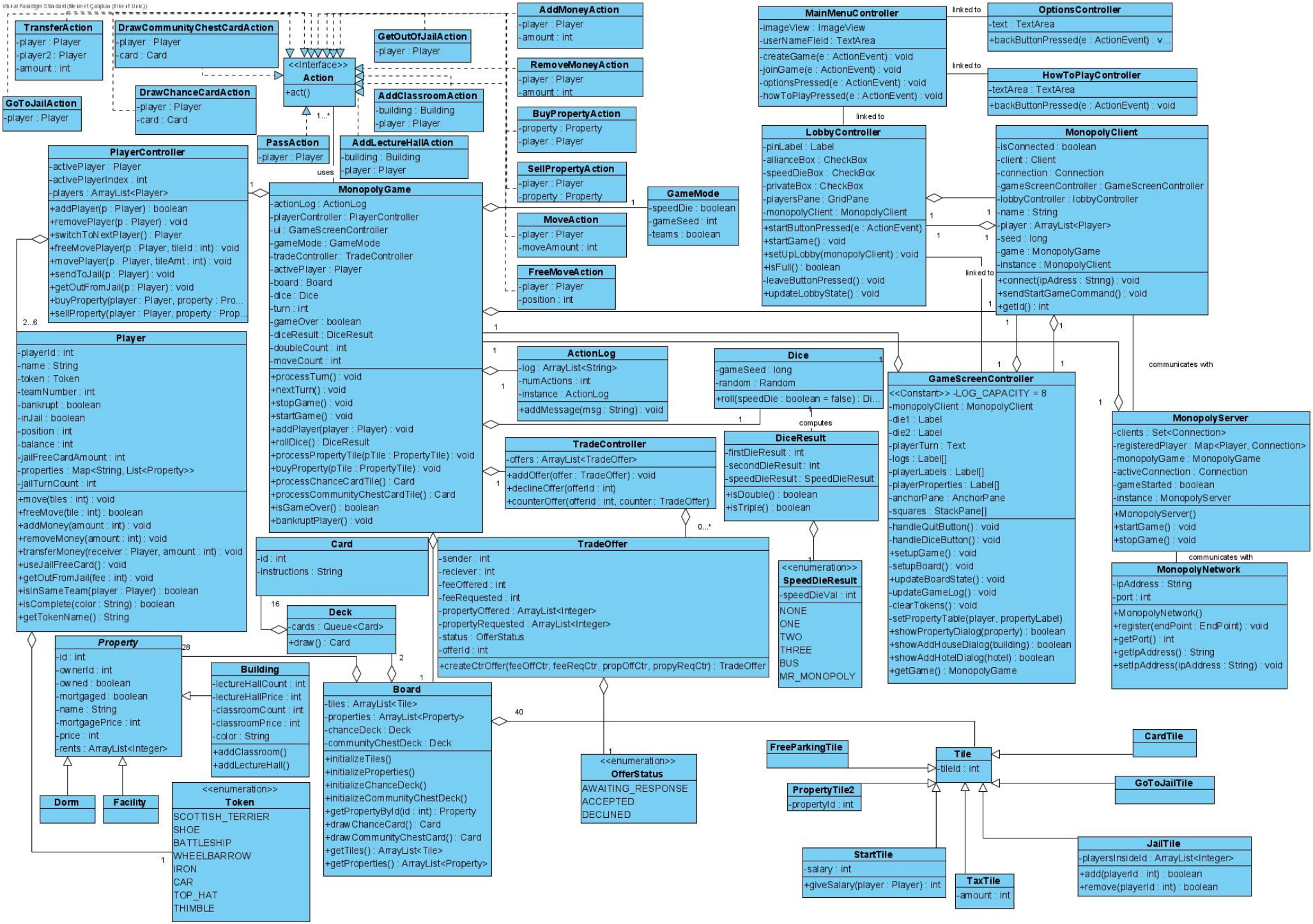

Figure 6: Class Diagram for the final object design

4.3. Layers

4.3.1. GUI Layer

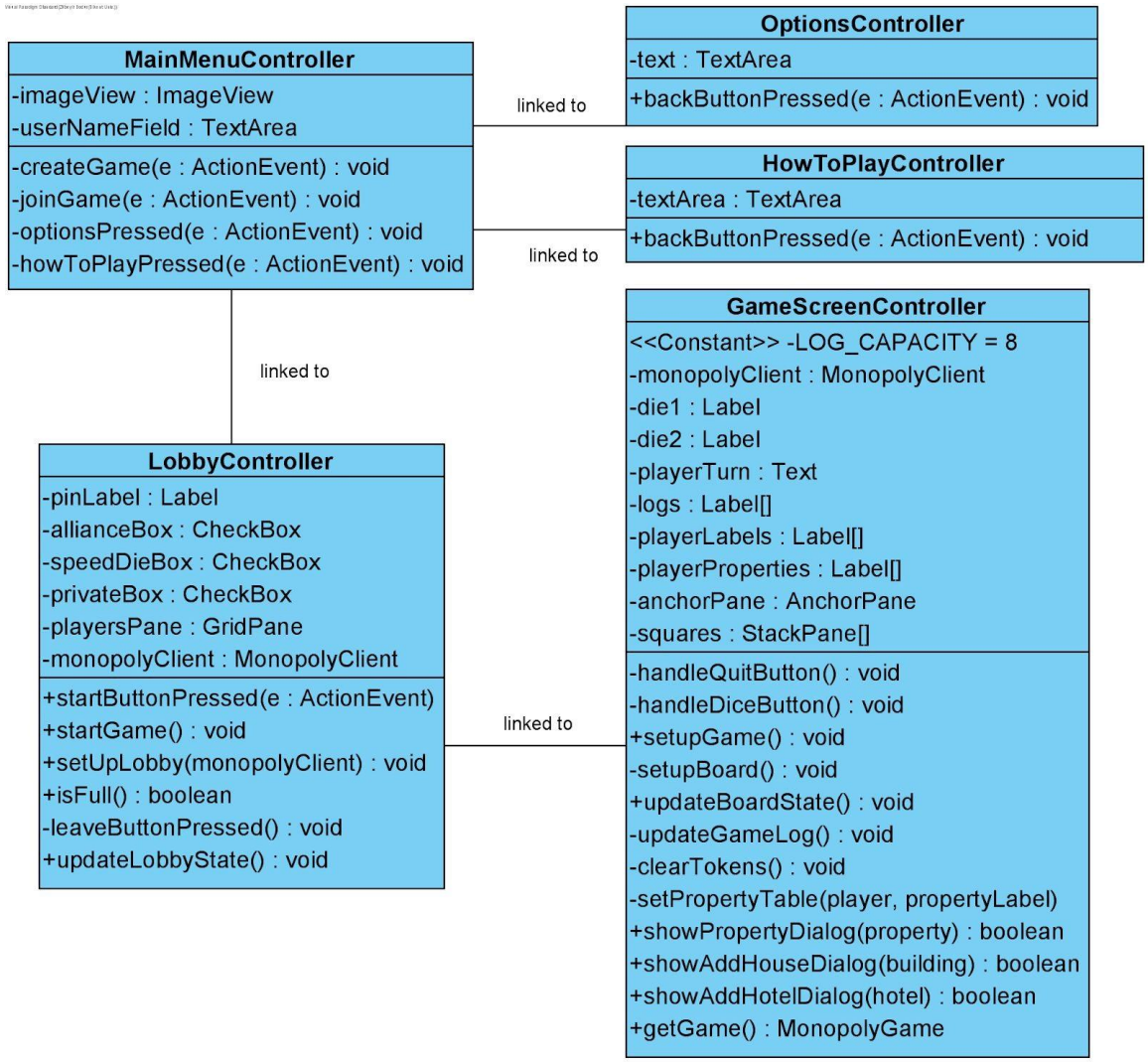

Figure 7 : Class diagram for GUI Layer

GUI Layer is the classes inside the gui package, which are the following: GameScreenController, MainMenuController, OptionsController, HowToPlayController and Lobby Controller. It communicates with the Network Layer.

MainMenuController keeps track of events that happen in the main menu (defined in the main_menu.fxml). MainMenuController is somewhat linked to other classes in this layer. If any button is pressed in the main menu, MainMenuController will open the suitable view for it (options, how to play, lobby screen etc.). This will cause the class that controls the menu to start listening to the actions happening in the new view.

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).

LobbyController and GameScreenController not only keeps track of events that happen in the game/ lobby screen, but also informs the network layer through the instance of MonopolyClient. For example, if there are four players, there are going to be four MonopolyClients, each having different lobby screens and game screens, therefore different Lobby/GameScreenControllers. Therefore, it is necessary for those two classes to have MonopolyClient as an attribute.

4.3.2. Network Layer

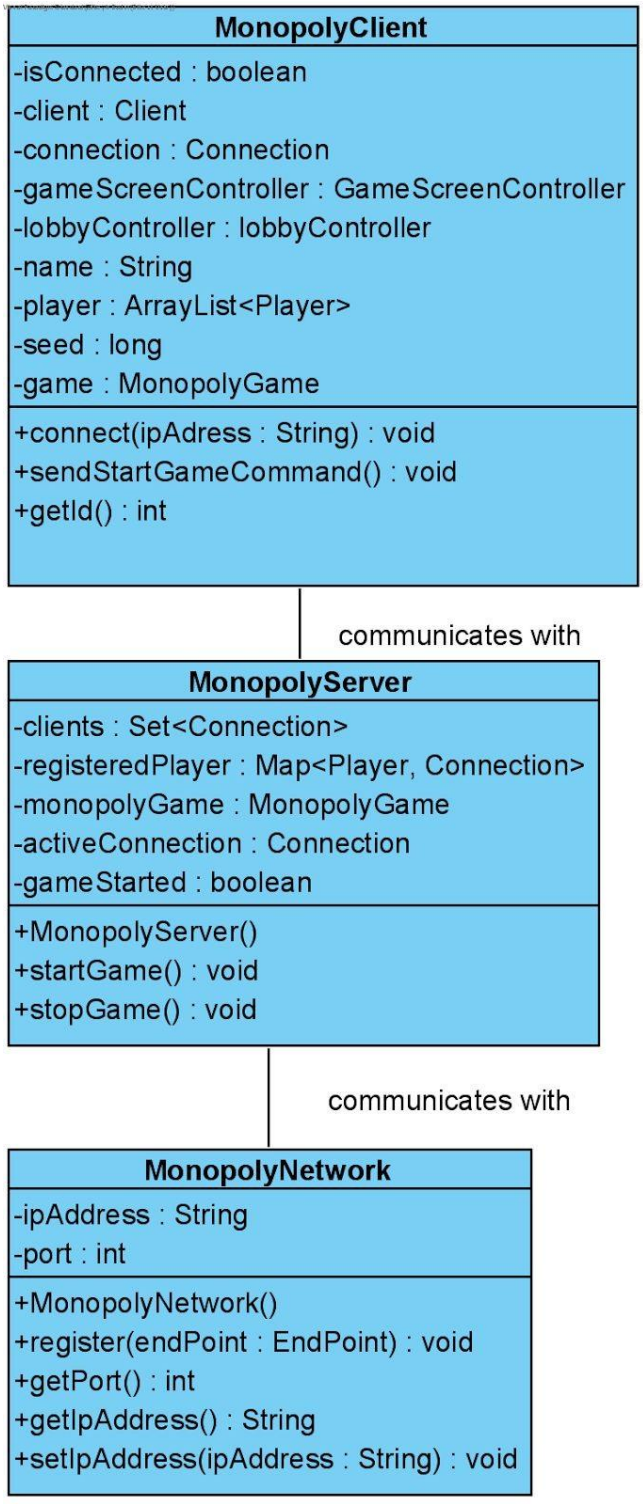

Figure 8 : Class diagram for Network Layer

Network Layer consists of network classes which provide multiplayer over LAN support. This layer communicates between the Control Layer and View Layer. It consists of three classes: MonopolyNetwork, MonopolyServer and MonopolyClient.

MonopolyServer class creates an instance of Server object by using the KryoNet library and creates a two-way communication between clients.

MonopolyNetwork class holds the required parameters for creating Server and Client objects such as Port and IP Address. It also registers game classes to transfer them by serializing/deserializing between clients and server.

MonopolyClient class creates an instance of Client object and connects to the server with the provided IP Address. This class also sends and receives messages from the server.

4.3.3. Control Layer

Control Layer is simply the control package, which communicates between the Entity Layer and Network Layer. It consists of three main parts: action interfaces, TradeController, PlayerController. Other two classes, ActionLog and GameMode also provide services for MonopolyGame.

ActionLog class holds the data for the previous actions happening throughout the game as string, so that the players can read what happened throughout the game.

Actions are used for good design so that an action taken inside a MonopolyGame is specific and concrete. A MonopolyGame will use one or more actions till the game ends.

TradeController and PlayerController classes not only provide services for MonopolyGame class, but also provide abstraction to how players in the game change and how trade operations are performed.

GameMode helps us abstract the combinations of a game mode in a single class. Those are teams and the speed die. It also has the game seed in it.

MonopolyGame is the façade class that provides services for the network layer. Its instance is both contained by all of the clients (MonopolyClient) and the server (MonopolyServer).

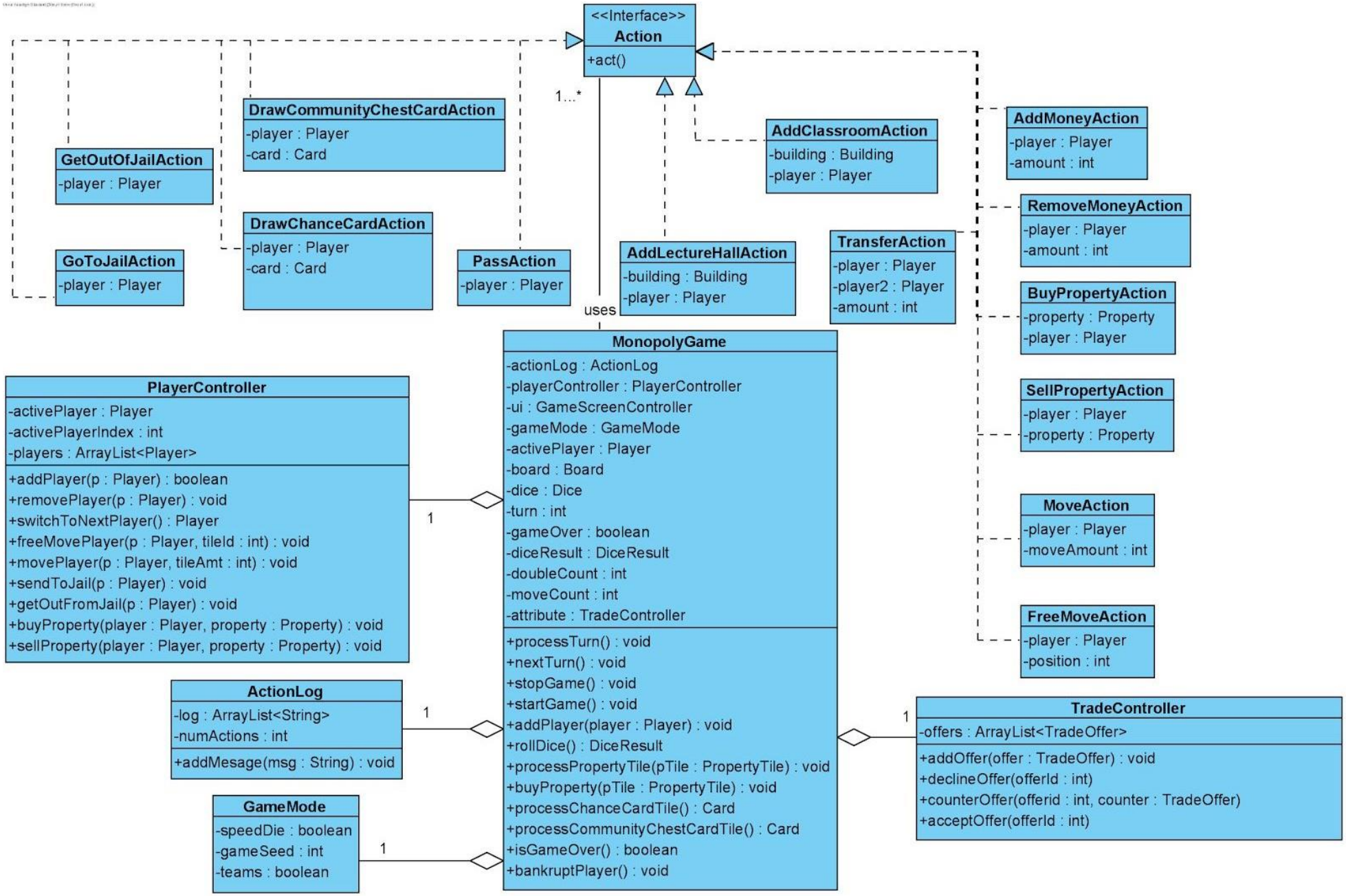

Figure 9: Class Diagram for the control layer

4.3.4. Entity Layer

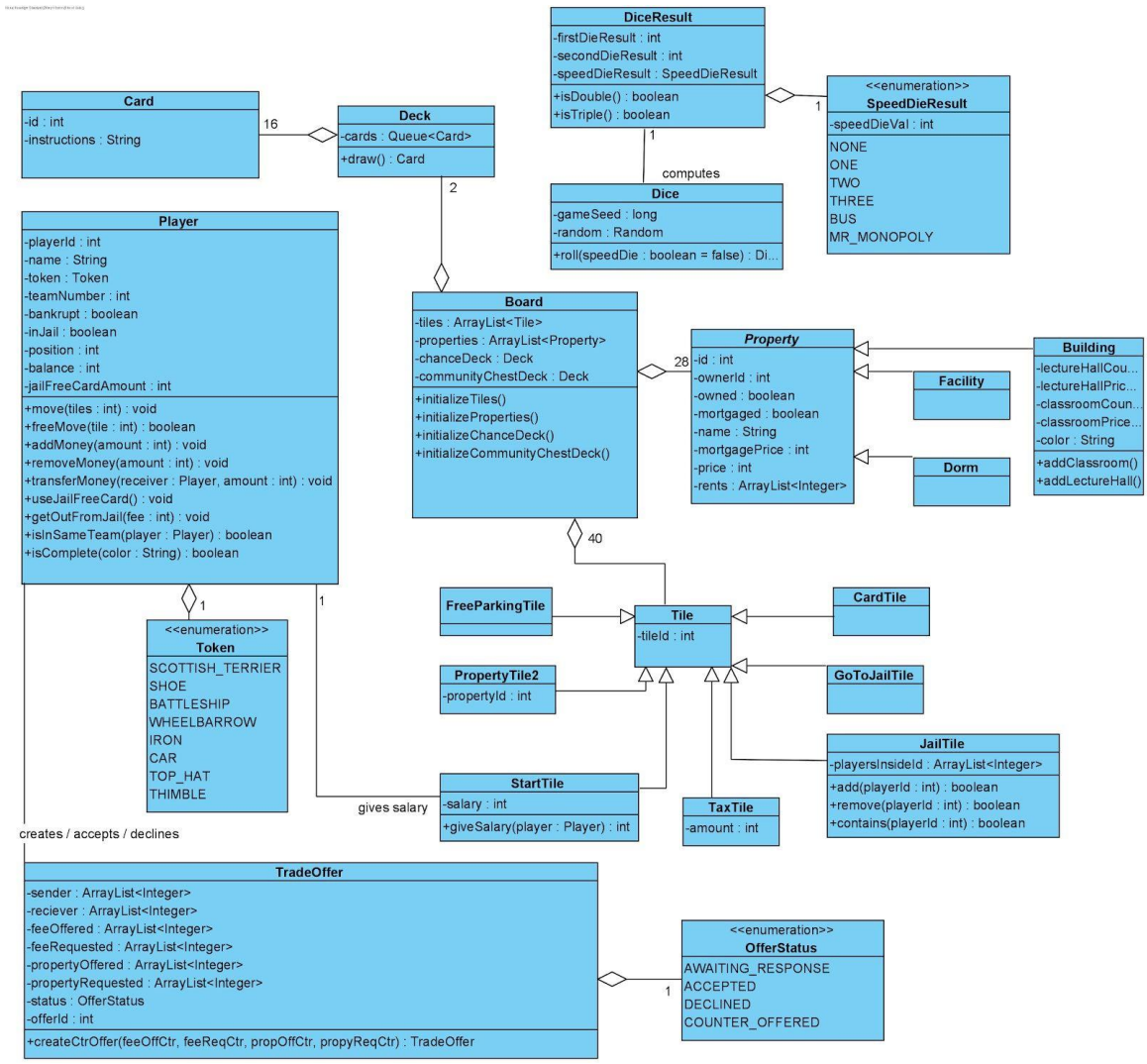

Figure 10 : Class diagram for Entity Layer

Entity layer consists of four parts, one is Player entities, one is Board entities, one is Dice entities and the other is Trade entities. Board entities are separated into three packages; card, property and tile. Dice entities are also gathered in one package called dice, and Trade entities are into a package called trade. In total, the Entity layer has five 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 consists of all interfaces and components in the Entity Layer, described in the subsystem decomposition. 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.

entity.trade Package

This package contains trade classes necessary for making a trade (except TradeController, as it is in the control package).

control Package

This package consists of all interfaces and components in the Control Layer, described in the subsystem decomposition. 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 classroom/lecture hall, going to jail etc.

network Package

This package consists of all interfaces and components in the Network Layer, described in the subsystem decomposition package. This package is required for multiplayer gameplay.

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

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 uses Arrays class to convert ArrayList to Queue since Deck works like a queue. GameScreenController class uses Objects class for null check operation.

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.

java.nio Package

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

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.

com.esotericsoftware.kryonet Package

KryoNet is a Java library that provides a clean and simple API for efficient TCP and UDP client/server network [1]. We use KryoNet for our network layer which contains MonopolyClient, MonopolyServer and MonopolyNetwork classes. With KryoNet, the server creates a communication between clients with TCP (Transmission Control Protocol).

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:

- **private final int LOG_CAPACITY:** Determines the amount of action log messages that will be displayed on the game screen.
- **MonopolyClient monopolyClient:** 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.
- **StackPane square0 → square39 :** Those stack panes contain the image of a tile.
- **StackPane[] squares:** Contains all of the 40 stack panes.
- **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[] logs :** Contains those log labels above.
- **Label player0Label → player5Label:** Indicates the player's name, token and balance.
- **Label playerLabels[] :** Contains player labels mentioned above.
- **Label player0Properties→ player5Properties:** Indicates the player's property list.
- **Label playerProperties[] :** Contains property list labels mentioned above.
- **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 the 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. It 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 balances, and locations of the tokens. It Is public because it should be called whenever an Action is performed from the MonopolyGame class.
- **public void updateGameLog():** Updates the action log.
- **public void clearTokens() :** Deletes the tokens from the squares on the board.
- **public boolean showPropertyDialog(Property) :** Prompts the user if they want to buy the property that they landed on. It returns the response of the user.
- **public boolean showAddClassroomDialog() :** Prompts the user if they want to build a classroom to the building. It returns the user's response.
- **public boolean showAddLectureHallDialog() :** Prompts the user if they want to build a lecture hall to the building. It returns the user's response.
- **public void setPropertyTable() :** Updates the property table which contains the players' properties with sorting them by their type.

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.

4.5.1.4. LobbyController

Attributes:

- **private Label pinLabel :** Label for the game pin
- **private CheckBox allianceBox :** check box for the alliance mode
- **private CheckBox speedDieBox :** check box for the speed die mode
- **private CheckBox privateBox :** check box for the locking the game for not new comer
- **private GridPane playersPane :** grid pane for the list of players and their team choice, token choice
- **private MonopolyClient monopolyClient :** for the communication with the server

Methods:

- **public void startButtonPressed (ActionEvent e) :** detect the action of start button
- **public void startGame () :** starts the game
- **public void setUpLobby (MonopolyClient monopolyClient) :** set up the values in the lobby according to the server.
- **public boolean isFull () :** returns true if six players entered the game and no place for new player, otherwise return false.
- **private void leaveButtonPressed () :** detect the leave button.
- **public void updateLobbyState () :** update the lobby according to the server.

4.5.1.5. 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. Network Layer Class Interfaces

4.5.2.1. MonopolyClient (Singleton)

Attributes:

- **MonopolyClient instance:** Instance of the class since it is a Singleton class.
- **boolean isConnected :** Status of the connection with the server
- **Client client :** Holds the properties of the user in the server.
- **Connection connection :** Holds the connection properties of the user and server.
- **boolean gameStarted :** Status of the game whether started or not.
- **GameScreenController gameScreenController :** Holds the controller of the game screen to give data.
- **LobbyController lobbyController :** Holds the controller of the game lobby to give data.
- **String name :** Name of the client.
- **ArrayList<Player> player :** List of players.
- **long seed :** Seed of the Random in the Dice class.
- **MonopolyGame monopolyGame** : Game instance to transfer data.

Methods:

- **public void connect(String ipAddress) :** Makes the connection to the server with the given IP address.
- **public void sendStartGameCommand() :** Starts the game on the server.
- **public int getId() :** Returns the id of the player.

4.5.2.2. MonopolyServer (Singleton)

Attributes:

- **MonopolyServer instance:** Instance of the class since it is a Singleton class.
- **Set<Connection> clients** : Set of connections of the users.
- **Map<Player, Connection> registeredPlayer :** Mapping between Players to Connections
- **MonopolyGame monopolyGame :** Game to transfer data.
- **Connection activeConnection :** the give turns to a specific player, the connection is activated for only the necessary player.
- **boolean gameStarted :** status of the game

Methods:

- **public void startGame() :** Initializing the game on the server.
- **public void stopGame() :** Finishing the game on the server.

4.5.2.3. MonopolyNetwork

Attributes:

- **String ipAddress :** IP address of the network
- **int port :** port number

Methods:

- **public void register(EndPoint endPoint) :** Registration of the classes to the Kryo serialization class.

4.5.3. Control Layer Class Interfaces

4.5.3.1. Action

Methods:

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

4.5.3.2. AddClassroomAction

Attributes:

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

Constructors:

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

Methods:

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

4.5.3.3. AddLectureHallAction

Attributes:

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

Constructors:

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

Methods:

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

4.5.3.4. 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.

4.5.3.5. 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.

4.5.3.6. 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.

4.5.3.7. 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.

4.5.3.8. 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.

4.5.3.9. 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.

4.5.3.10. 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.

4.5.3.11. 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.

4.5.3.12. 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.

4.5.3.13. 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.

4.5.3.14. 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.

4.5.3.15. 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.

4.5.3.16. ActionLog (Singleton)

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.
- **ActionLog instance:** Holds the instance since it is a singleton class.

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.

4.5.3.17. MonopolyGame (Façade class)

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.
- **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.

4.5.3.18. Trade Controller

Attributes:

- **private ArrayList<TradeOffer> offers :** List of trade offers.

Methods:

- **public void addOffer(TradeOffer offer) :** new offer is added to the list
- **public void declineOffer(int offerId)** : declining the offer with given id
- **public void counterOffer(int offerId, TradeOffer counter)** : counter offer to the current offer with the given id
- **public void acceptOffer(int offerId)** : accepting the offer with given id

4.5.3.19. PlayerController

Attributes:

- **private ArrayList<Player> players :** List of players in the game

- **private Player activePlayer :** Instance of the active player.

- **private int activePlayerIndex :** 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.

4.5.4. Entity Layer Class Interfaces

4.5.4.1. 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, the 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.

4.5.4.2. TradeOffer

Attributes:

- **public enum OfferStatus{ AWAITING_RESPONSE, ACCEPTED, DECLINED} :** Specify the offer status.
- **private int sender :** playerId of the offer's sender
- **private int receiver :** playerId of the offer's receiver
- **private int feeOffered :** Offered fee for the trade
- **private int feeRequested :** Requested fee for the trade
- **private ArrayList<Integer> propertyOffered :** List of offered properties for trade

- **private ArrayList<Integer> propertyRequested :** List of requested properties for trade
- **private OfferStatus status :** status of the trade
- **private int offerId :** id of the trade offer

Methods:

- **public TradeOffer createCtrOffer(int feeOffCtr. int feeReqCtr, ArrayList<Integer> propOffCtr, ArrayList<Integer> propReqCtr) :** To the received offer, the player can create a new counter offer to change the deal.

4.5.4.3. 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.

4.5.4.4. 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.

4.5.4.5. 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.

4.5.4.6. 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.

4.5.4.7. 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.

4.5.4.8. 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

4.5.4.9. 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.

4.5.4.10. Building

Attributes:

- **int classroomPrice :** price of the classroom for this building.
- **int lectureHallPrice :** price of the lecture hall for this building.
- **String color :** To classify buildings according to the colors.

- **int classroomCount :** The number of classrooms builded within this building.
- **int lecturehallCount :** The number of lecture halls builded within this building.

Constructors:

- **public Building(String name, int id, int price, ArrayList<Integer> rents, int mortgagePrice, int classroomPrice, int lectureHallPrice, String Color) :** According to the given values, the building is created with 0 classroom and 0 lecture hall.

Methods:

- **public void addClassroom() :** It increases the classroomCount with one.
- **public void addLecturehall() :** It increases the lecturehallCount with one.

4.5.4.11. 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)**

4.5.4.12. Facility

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

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

4.5.4.13. 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 JsonParseExceptio*n. In the deserialize function, the tile data from the JSON file is categorized into StartTile, PropertyTile, CardTile, TaxTile, JailTile, FreeParkingTile and GoToJailTile classes.

4.5.4.14. 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, a new CardTile is created.

4.5.4.15. 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.

4.5.4.16. 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.

4.5.4.17. 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.

4.5.4.18. 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.

4.5.4.19. 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.

4.5.4.20. 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.

5. Improvement Summary

- Edited the class diagrams according to the design patterns that we applied.
- Added figure captions below pictures.
- Added Network Layer and its classes.
- Made the pictures more readable.

6. References

[1]. https://github.com/EsotericSoftware/kryonet