# Group 4 Monopoly Project Report

**Submitted by:**

Le Nguyen Nguyen Thieng

Nguyen Tran Quoc Thai

Duong Minh Duy

# Chapter I – Introduction

Our project in the Object-Oriented Programming course is a complete application that utilizes both basic and advanced concepts of object-oriented programming. This project shows an extensive understanding of OOP's fundamental ideas in addition to adding further developed concepts to increase its overall sophistication. The only programming language we have utilized for this project is Java, which is an OOP-representative language. Aside from that, we created the project's GUI using Swing.

This report will offer an in-depth explanation of our project, outlining all its different aspects. Our report's organizational structure is outlined as follows:

Chapter I: Introduction

Chapter II: Rules and Gameplay

Chapter III: The details of the game

Chapter IV: UML class diagram

Chapter V: Evaluation

# Chapter II - Rules and gameplay

Monopoly is a classic board game renowned for its strategic gameplay and economic themes. The game involves the following rules and gameplay elements:

**1. Objective:**

- The primary goal is to bankrupt opponents by acquiring their properties and accumulating wealth.

**2. Game Setup:**

- Each player chooses a color and receives a set amount of money.

   - Properties are distributed evenly, and each player is allowed to purchase or sell properties in their initial rounds.

**3. Board Layout:**

- The board consists of a square layout with 40 spaces, including 32 properties (26 Cities, 4 Beaches, 2 Companies), 3 Chance spaces, a Go space, a Tax space, a World Tour space, a World Cup space, and a Lost Island space.

**4. Turn Sequence:**

- Players take turns in a clockwise direction.

   - A turn involves rolling two six-sided dice and moving the corresponding number of spaces.

**5. Property Ownership:**

- When a player lands on an unowned property, they have the option to purchase it.

   - A player must pay rent, which depends on the property value if they land on another player's property. Additionally, if they have enough money, a player will have the opportunity to purchase another person's property.

**6. Building Houses:**

- Players can enhance their properties by building houses.

   - Building costs increase with each subsequent house and are determined by the property group.

**7. Bankruptcy:**

- A player is declared bankrupt if they cannot pay debts, including rent or fines.

   - The remaining assets of a bankrupt player are usually distributed among the remaining players.

**8. Jail or Lost Island:**

- Players may land in jail by landing on the Lost Island space or get a Go to Jail Chance Card

   - Players can attempt to roll doubles to get out or after three rounds in jail or use the Get Out of Jail chance card to get out immediately.

**9. Chance Cards:**

- Players draw cards from these decks, which can provide various opportunities or challenges, such as advancing to specific spaces or paying fines.

**10. World Cup:**

- When player step on the World Cup square, they are allowed to set an event in their city. When a city has been set event (here is the World Cup event), the visit cost of that city will be ten times.

- On the board, only one city will be chosen to hold an event. For example, if player 1 has already held an event, and then player 2 steps into the event square and player 2 wants to hold an event, the event city of player 1 will be removed and player 2’s city can hold an event.

**11. Plane:**

- When a player steps on the World Tour square, the player can travel to one of these squares:

+ Their cities

+ Unowned city

-When players travel to these squares, they can buy or upgrade these cities if they have enough cash.

Free to use plane: 50k$.

**12. Winning the Game:**

- The last remaining solvent player, who has not gone bankrupt, is declared the winner.

- The player owns 4 beaches.

Monopoly's enduring popularity lies in its blend of strategy, luck, and negotiation, making it a timeless and engaging board game for players of all ages.

A screenshot of a game

Description automatically generated

**Monopoly Game Board**

# Chapter III – The detail of the game

* 1. Flow of main game:

First, the Monopoly class will call out a start menu for users to display the game Welcome, authors.

Second, when the user presses the Start Game button, the game will require users to input several players, the player’s name, and color for their representative label in the game.

Next, the main game GUI will display on the screen and the user can start to play by pressing the roll dice button. For each round, the game frame now will listen to the mouse click of the player on the roll dice button and randomly generate the dice value. The dice value will be assigned for the step of the player on the game board. All information or interactions will be stored and displayed in the text area on the left side of the screen. In each round, the player will be allowed to interact with the square and the game will continue to run until one of the winning conditions is reached.

Finally, the game will print out the winner and the total money that they earned during the game.

* 1. Class detail
* Class **Board**:
  + The purpose of this class is to represent a game board, which includes a timer, a collection of squares, a collection of players, and a collection of cities related to events.
* Class **Dice**:
  + Provide methods to initialize the dice images, set and get the dice result, and roll the dice.
  + **dice1 to dice6**: These are **BufferedImage** objects that store the images of the six faces of a dice (1 to 6).
  + The constructor initializes the **dice1 to dice6** fields by reading the corresponding image files from the **src/ImageIcon** directory using **ImageIO.read()**. If any of the files cannot be read, an **IOException** is caught, and an error message is printed to the console.
  + The **roll** method generates a random number between 1 and 6 using **Math.random()** and set diceResult to this value.
  + The method also updates the **diceLabel** parameter, which is a **JLabel** component, by setting its icon to the corresponding dice image based on the **diceResult** value.

A blue screen with white text

Description automatically generated

* Class **DicePanel**:
  + Create a custom panel for displaying a dice roll animation.
  + The **rollDice** method generates a random number between 1 and 6 and assigns it to the **diceValue** variable.

A blue screen with white text

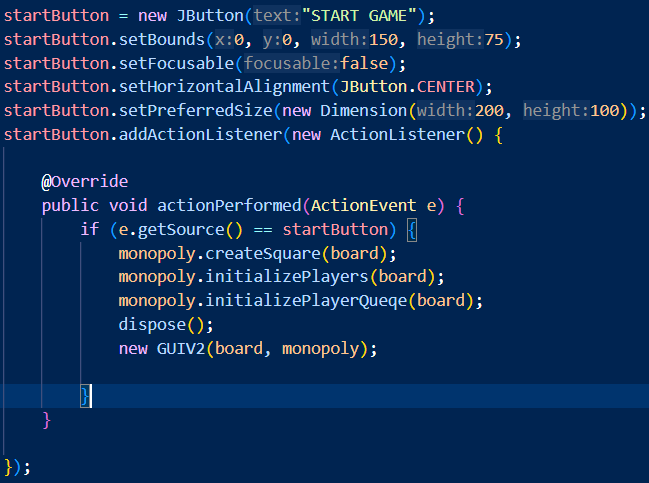
Description automatically generated

* + The **paintComponent** method is overridden to customize the painting of the panel. It is called whenever the panel needs to be redrawn. Here's what it does:
    - If **diceValue** is not 0, it:
      * Fills the panel with a black rectangle (background)
      * Draws a white rectangle around the panel (border)
      * Loops through the **diceValue** number of times and:
        + Fills a red rounded rectangle (dice) at the calculated x and y coordinates
        + Draws a black rounded rectangle (dice border) around the red dice

A computer code on a blue background

Description automatically generated

* + The **stopAnimation** method sets **diceValue** to 0 and calls **repaint** to redraw the panel. This effectively stops the animation and clears the panel.
* Class **ImgService**:
  + The class imports necessary packages and classes for handling images and graphical user interfaces.
* Class **StartMenuGUI**:
  + The UI is initialized with the Monopoly logo, a start button, and a message. When the start button is clicked, it triggers the game initialization and opens a new window with the game board and objects. The UI is also configured with a size, title, and icon image. It is set to be non-resizable and centered on the screen.
  + The UI consists of:
    - A **JButton** named **startButton** with the text "START GAME". When clicked, it initializes the game, creates squares on the board, and sets up players and their queues. It then disposes of the current window and opens a new window **GUIV2** with the board and game objects.



* + - A **JLabel** named **monopolyLabel** that displays the Monopoly logo and a message "This game is made by Group 4".
    - A **JPanel** named **contentPanel** that contains the **monopolyLabel** and **startButton** using a **BorderLayout** layout manager.

A computer screen with text

Description automatically generated

* Class **GUIV2**:
  + Create game GUI.
  + The GUI consists of several components:
    - **Game Board**: A **JLayeredPane** named **boardPanel** that represents the Monopoly board. It contains several panels and components, including:
      * **North Panel**: A **JPanel** that represents the top edge of the board, containing squares 11-20.
      * **South Panel**: A **JPanel** that represents the bottom edge of the board, containing squares 30-39.
      * **East Panel**: A **JPanel** that represents the right edge of the board, containing squares 21-30.
      * **West Panel**: A **JPanel** that represents the left edge of the board, containing squares 1-10.

A computer code on a blue background

Description automatically generated

* + - **Player Panels**: Several **JPanel** instances that represent each player's information, including their name, cash, and color. These panels are added to the frame and updated dynamically.

A computer screen shot of text

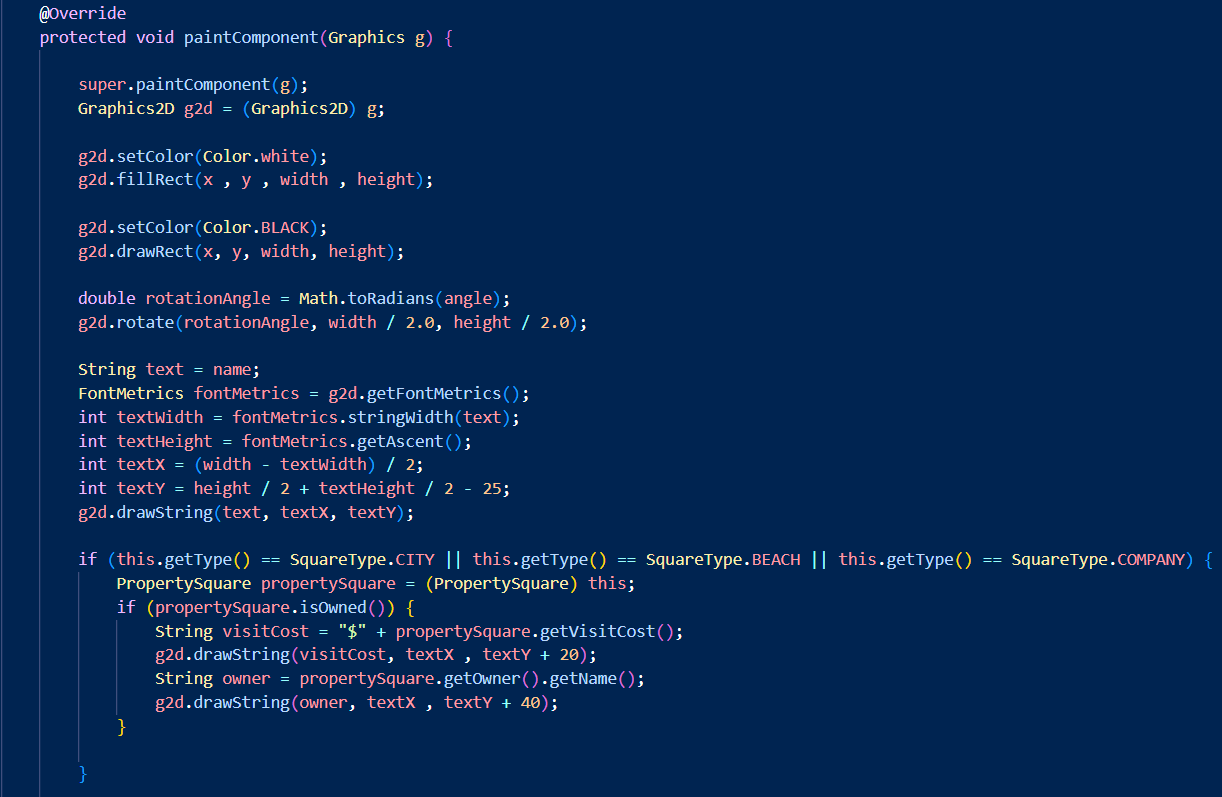
Description automatically generated

* + - **Dice Panel**: A **JPanel** that contains two **JLabel** instances that display the dice rolls.

A computer screen shot of a program code

Description automatically generated

* + - **Roll Dice Button**: A **JButton** that triggers the dice roll and game logic when clicked.
    - **Text Area:** A **JTextArea** that displays the game log and output.
    - **Frame**: A **JFrame** instance that contains all the GUI components and is responsible for displaying the game.
  + **changePlayerPosition** method updates a player's position on the board.
  + **updatePlayerCash** method updates a player's cash amount.
  + **updatePlayerPosition** method updates a player's position on the board and repaints the GUI.
  + **removePlayerFromGUI** method removes a player from the GUI when they are eliminated from the game.
  + **getUserInputNumber** method prompts the user to enter a number and returns the input.
  + **getUserInputYesNo** method prompts the user to choose YES or NO and returns the input.
  + **setUserInput** method sets the user input string.
  + The **CustomOutputStream** class is a custom output stream that appends text to a **JTextArea** and scrolls to the bottom to always show the latest output.
* Class **Squares**:
  + Create basic attributes, method of each square.
  + Create simple GUI for each square with **paintComponent** method.



* Class **PropertySquare**:
  + Represent a property square and provides a template for creating different types of property squares, each with their own unique characteristics, such as property value, visit cost, and upgrade cost.
  + The constructor of the class initializes the Square with the type of **PROPERTY**.

A computer screen shot of a code

Description automatically generated

* + **resetAllAtribute** method is used to reset level, property value and upgrade cost to initial value.
* Class **SquareType**:
  + An enumeration (enum) is a value type that represents a set of named square constants.

A blue screen with red text

Description automatically generated

* Class **Start**:
  + The constructor of the class initializes the **Square** with the name " Start ", at position 1, and the type of **START**.

A blue screen with white text

Description automatically generated

* Class **Event**:
  + Create World Cup event square and handle the player’s city after event with **setWorldCup** method.
  + When a player step on this square, that player will have an option to choose a city by number from their properties, the player can check which city is valid in the text area by **printCity** method.
  + After that, the **isValidChoice** method will check if the input is a valid number, then check if that number is a city or not. If it is valid, it will set a world cup event on that city by increasing the visit cost and the event will remains until the next time a player step on this square.

A screen shot of a computer code

Description automatically generated

A computer screen shot of text

Description automatically generated

A blue screen with text on it

Description automatically generated

* Class **Plane**:
  + This class represents a plane square, provides methods for printing the list of available cities and moving the player to a chosen city.
  + Has an **ArrayList** of **City** objects called **availableCities**, which stores cities that the player can travel to.
  + The constructor of the class initializes the **Square** with the name "World Tour", at position 31, and the type of **PLANE**.

A blue screen with white text

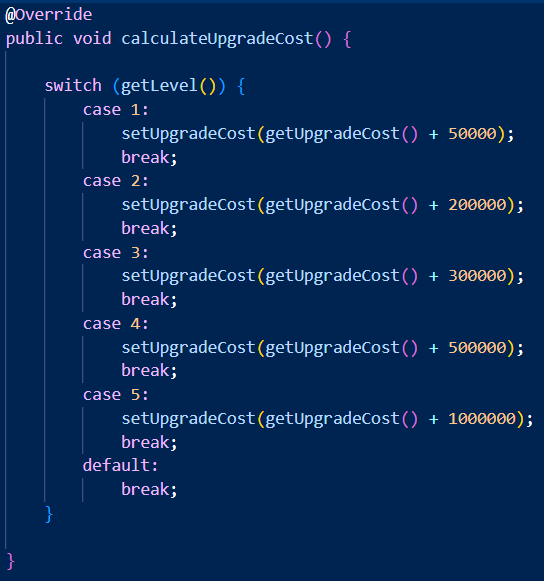
Description automatically generated

* + The **printLocation** method prints the list of cities that the player can travel to.
  + The **travel** method allows the player to choose a city from the **availableCities** list and move to that city. It first calls **printLocation** to display the list of available cities. It then gets user input and continues to ask for input until a valid number is entered. Once a valid number is entered, the method moves the player to the chosen city, and interact with the chosen city.

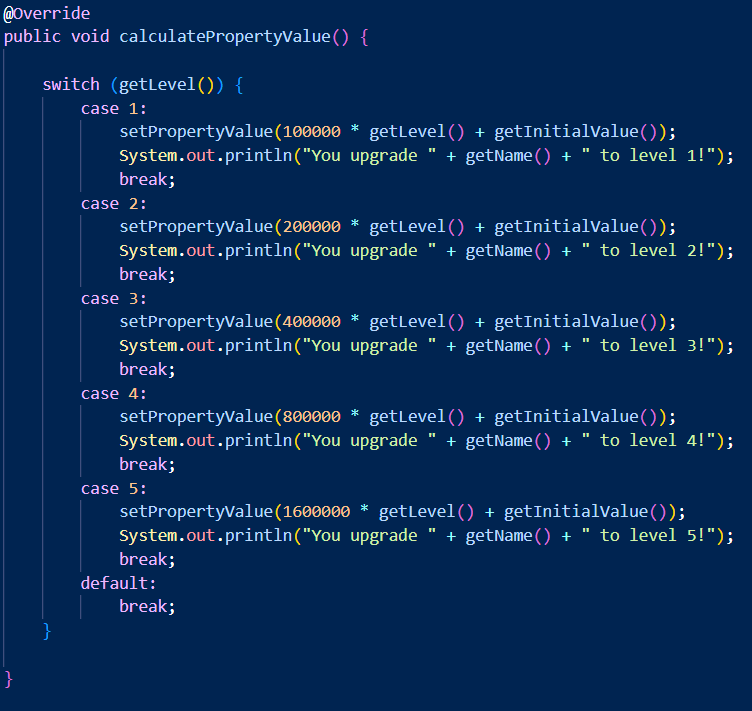
A computer screen shot of text

Description automatically generated

* Class **City**:
  + This class represents a city square, and it has methods to calculate its upgrade cost, property value, visit cost, and sell value based on its level and whether it's hosting a World Cup event or not.
  + The constructor of the class initializes the **Square** the type of **CITY**.
  + **calculateUpgradeCost** method to calculate the upgrade cost of the city based on its current level.



* + **calculatePropertyValue** method to calculate the upgrade cost of the city based on its current level.

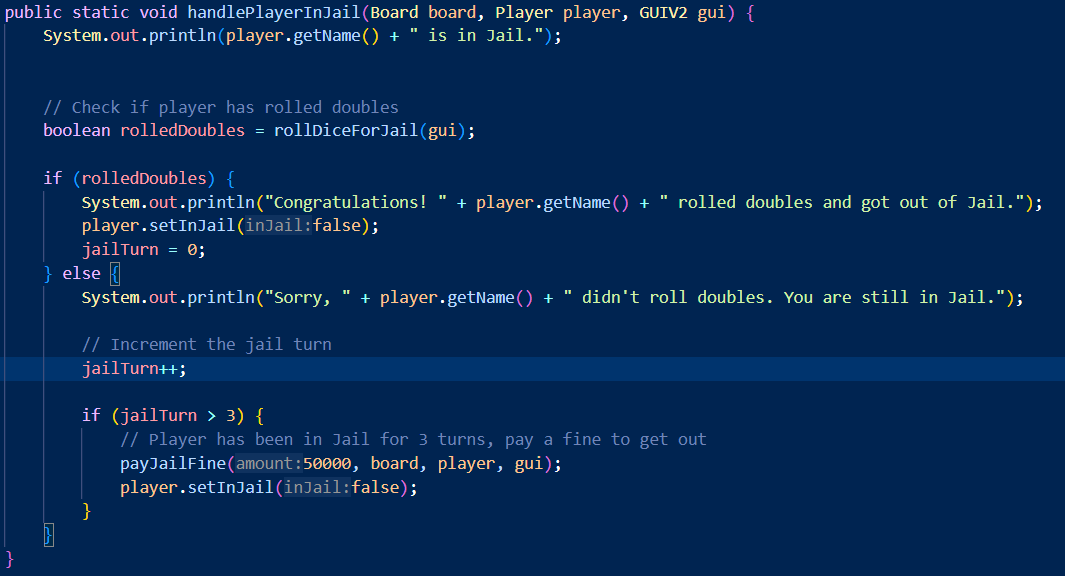


* + **calculateVisitCost** method to calculate the visit with condition. If the city is hosting a World Cup event, the visit cost is multiplied by 10. Otherwise, the visit cost is half of the property value.
  + **calculateSellValue** method to calculate the sell value by 70% of its property value.
* Class **Jail**:
  + Represents the jail square, and it has methods to handle the logic when a player lands on this square, including rolling a double to get out of jail and paying a fine if they can't get out after 3 turns.
  + The constructor of the class initializes the **Square** with the name " Jail ", at position 11, and the type of **JAIL**.

A blue screen with white text

Description automatically generated

* + **handlePlayerInJail** method first prints a message indicating that the player is in jail, then checks if the player has rolled doubles by calling the **rollDiceForJail** method.
  + If the player has rolled doubles, they are released from jail and the **jailTurn** counter is reset to 0.
  + If the player hasn't rolled doubles, they are still in jail and the **jailTurn** counter is incremented. If the player has been in jail for more than 3 turns, they are forced to pay a fine of 50,000 to get out of jail by calling **payJailFine**.



* Class **Beach**:
  + Represents a beach property in a board game, and it has methods to calculate its property value, visit cost, and sell value based on the number of beaches owned by the player.
  + The constructor of the class initializes the **Square** the type of **BEACH**.
  + **calculatePropertyValue** method calculates the property value of the beach based on the number of beaches owned by the player.

A computer screen shot of a code

Description automatically generated

* + **calculateVisitCost** method calculates the visit cost equal to the property value.
  + **calculateSellValue** method fixes the sell value to 200,000.
  + **calculateUpgradeCost** fixes the upgrade cost to 0, indicating that beaches cannot be upgraded.
* Class **Company**:
  + Basically, the same as beach class. The difference is **calculateVisitCost** method.
  + The constructor of the class initializes the **Square** the type of **COMPANY**.
  + **calculateVisitCost** method calculates the visit cost by multiplying the property value with a random number from 1 to 6.

A computer screen shot of numbers

Description automatically generated

* Class **Tax**:
  + Represents a tax square and it has methods to calculate the total tax amount based on the player's property and the game's progress.
  + The constructor of the class initializes the **Square** with the name " Tax ", at position 37, and the type of **TAX.**

A blue screen with white text

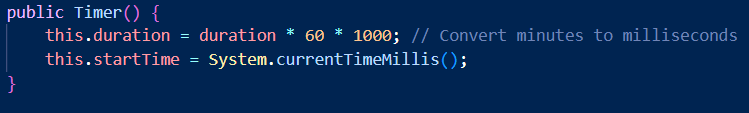
Description automatically generated

* + **calculateTotalTax** method calculates the total tax amount based on the player's cash and total property value. The tax calculation changes depending on the elapsed time:
    - If the elapsed time is 5 minutes or less, the tax is 1% of the player's cash and total property value.
    - If the elapsed time is between 5 and 10 minutes, the tax is 5% of the player's cash and total property value.
    - If the elapsed time is 10 minutes or more, the tax is 10% of the player's cash plus 15% of the player's total property value.

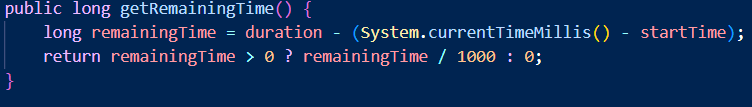
A screen shot of a computer code

Description automatically generated

* Class **Timer**:
  + Create a timer and calculate the time left.
  + The constructor takes no parameters and initializes the **startTime** to the current time in milliseconds, and the **duration** to a default value.



* + **getRemainingTime** method returns the remaining time by subtracting the play time from the total duration in seconds.



* + **getElapsedTime** method returns the elapsed time in seconds by subtracting the start time from the current time.

A blue screen with white text

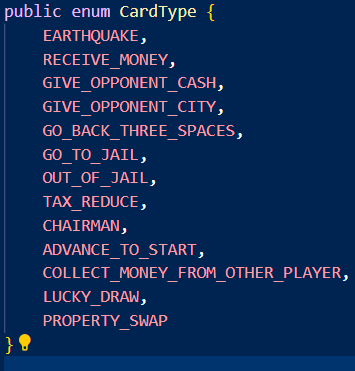
Description automatically generated

* + **start** method starts the timer. It creates a new thread that runs in the background and updates the remaining time every second. When 10 minutes have passed, it executes the **Runnable** action set by **setTenMinAction** method. When the timer reaches 0, it prints "Time is up!".

A computer screen shot of code

Description automatically generated

* Class **CardType**:
  + An enumeration (enum) is a value type that represents a set of named card constants.



* Class **Card**:
  + Represents a chance card and it has several methods that implement the effects of different cards.
  + **EarthQuakeChanceCard** method will destroy a random property owned by the player.
  + **ReceiveMoneyChanceCard** gives the player a certain amount of money.
  + **GiveOpponentCashChanceCard** allows the player to give a certain amount of money to another player.
  + **GiveOpponentCityChanceCard** allows the player to give a city to another player.
  + **GoBackThreeSpaceChanceCard** moves the player back three spaces on the board.
  + **GoToJailChanceCard** sends the player directly to jail.
  + **GoOutOfJailChanceCard** allows the player to get out of jail.
  + **ReduceTaxChanceCard** reduces the player's tax by 50%.
  + **ChairManChanceCard** allows the player to become the chairman and collect money from other players.
  + **AdvanceToStartChanceCard** moves the player to the start of the board and gives them a certain amount of money.
  + **CollectMoneyFromOtherPlayerCommunityCard** allows the player to collect money from other players.
  + **LuckyDrawChanceCard** gives the player a random amount of money.
  + **PropertySwapChanceCard** allows the player to swap one of their properties with another player's property.
* Class **Player**:
  + Represents a player and it has methods for player interacts with a square, implementing the specific effects of each square type, rolling the dice, calculating the total number of beach and company properties, and total property value.
  + **rollDice** method rolls the dice and returns the total value of the dice rolls.
  + **totalNumberOfBeach** returns method the total number of beach properties owned by the player.
  + **totalNumberOfCompanies** method returns the total number of company properties owned by the player.
  + **calculateTotalPropertyValue** method calculates the total property value of the player.
  + **interactWithSquare** method interacts with the square that the player is currently on.

A screen shot of a computer code

Description automatically generated

* + **handlePropertyInteraction** method handles the interaction with a property square.
  + **buyPropertyFromOtherPlayer** method buys a property from another player.
  + **sellPropertiesToCoverRent** method sells properties to cover the rent amount.
  + **buyProperty** method buys a property.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

* + **upgradeCity** method upgrades a city property.

A screen shot of a computer code

Description automatically generated

* + **handleTaxInteractionhandles** method handle the interaction with a tax square.
  + **handlePlaneInteraction** method handles the interaction with a plane square.
  + **handleJailInteraction** method handles the interaction with a jail square.
  + **handleEventInteraction** method handles the interaction with an event square.
  + **handleCardInteraction** method handles the interaction with a card.
* Class **Monopoly**:
  + The Monopoly class contains the main method, which creates a new Board object and a new Monopoly object, and then initializes the game by calling the **initializeGame** method.
  + The Monopoly class also contains several methods for initializing the game, running the game, and printing the game result:
    - The **runRound** method runs a round of the game by taking the turn of each player in the player queue.

A screen shot of a computer code

Description automatically generated

* + - The **takeTurn** method takes a turn of the game by rolling the dice, moving the player, and interacting with the square that the player lands on.

A computer screen shot of text

Description automatically generated

* + - The **initializePlayers** method initializes the players by prompting the user to enter the number of players and their names, and then creating a new Player object for each player.

A screen shot of a computer code

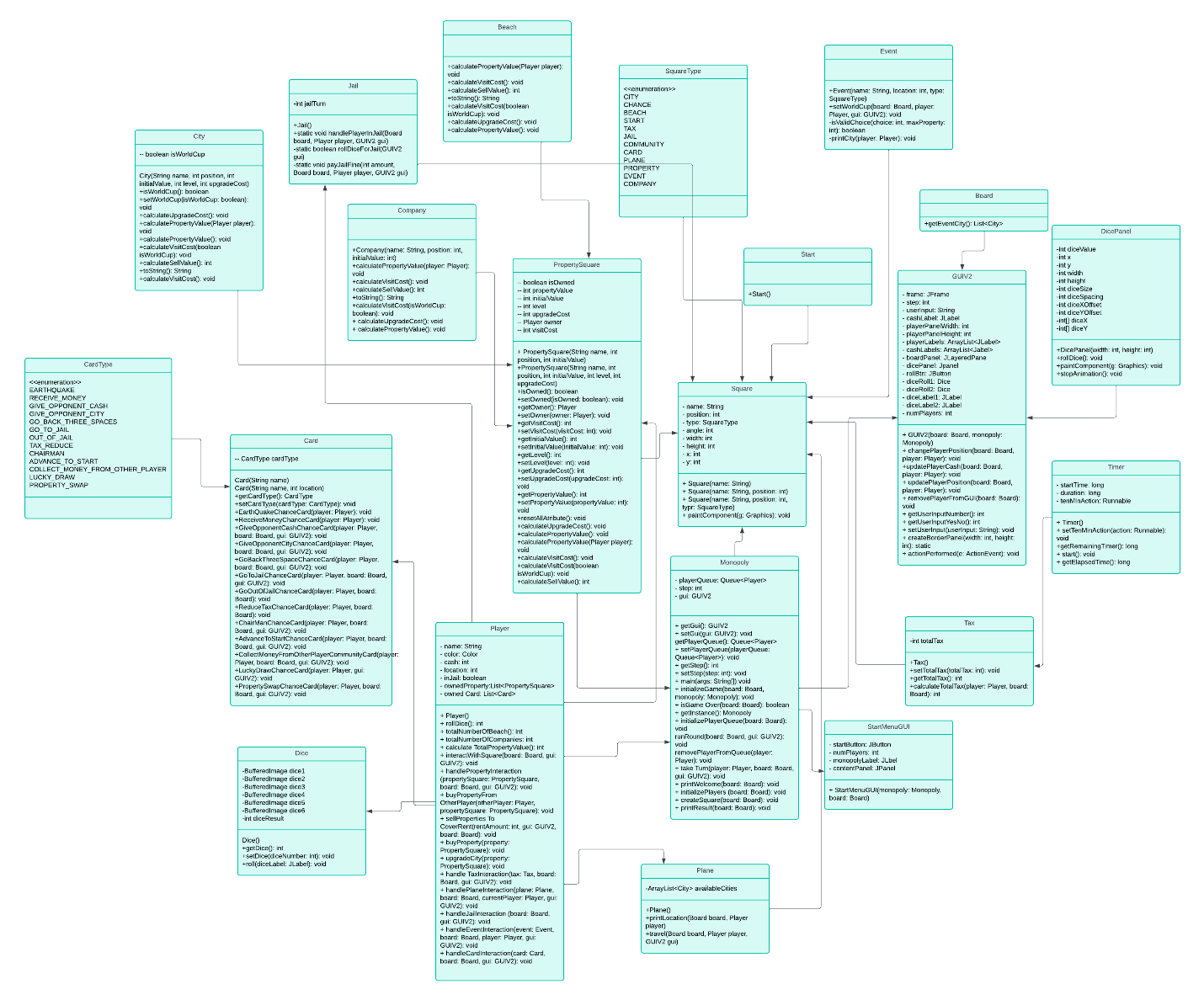
Description automatically generated

A blue screen with text on it

Description automatically generated

* + - The **createSquare** method creates the squares on the board, such as Start, City, Beach, Jail, Event, and Company.
    - The **printResult** method prints the game result by displaying the final balance of each player. To control game structure.

# Chapter IV - UML Class Diagram



# Chapter V – Evaluation

* This game is designed and built most simply and flexibly that can improve freely.
* Can improve a lot of features in the future (more chance cards, more events, change the location or properties’ cost in the game).
* Can improve better GUI.