Bilkent University

Department of Computer Engineering

CS 319 Term Project: Monopoly

Section 01

Group 1A

DESIGN REPORT

Atakan Dönmez
Elif Kurtay
Musa Ege Ünalan
Mustafa Göktan Güdükbay
Yusuf Ardahan Doğru

Instructor: Eray Tüzün

Teaching Assistant(s): Barış Ardıç, Emre Sülün, Elgun Jabrayilzade

Table of Contents

1. Introduction 1
   1.1. Purpose of the System 1
   1.2. Design Goals 1
2. High-level Software Architecture 2
   2.1. Subsystem Composition 2
   2.2. Hardware/Software Mapping 3
   2.3. Persistent Data Management 4
   2.4. Access Control and Security 4
   2.5. Control Flow 5
   2.6. Boundary Conditions 6
3. Subsystem Services 7
   3.1. Bank Management 7
   3.2. Board 9
   3.3. Game Entity 10
   3.4. Game 22
   3.5. UI Controller 23
4. Low-level Design 29
   4.1. Object Design Tradeoffs 29
   4.2. Final Object Design 29
   4.3. Packages 31
   4.4. Abstract Classes 31
5. References 32

Table of Figures

Figure 1. Subsystem Decomposition for the Monopoly game. 2
Figure 2 . UML Class Diagram..... 30

List of Tables

Table 1. Access matrix...5

1. Introduction

1.1. Purpose of the System

Monopoly is a board game where players buy and trade properties to play. Monopoly is normally played interactively on the board. There are many digital versions available for the Monopoly game. The purpose of this project is to create a digital version of Monopoly using virtual money, property, and financial operations. The proposed game will be played locally on one computer. It can be played multiplayer or single-player against computer-created players. There will be new features added to extend the enjoyability of the game. These features include each token having its own buff, putting a thief in the game to steal money, and so on.

1.2. Design Goals

Dependability

***Reliability*:** Constructing a game with a minimum number of mistakes and bugs is a crucial design goal. The operations should perform and produce expected results. There should not be any bugs that will affect players. Applying this design goal may require implementing several more functions to the game, and may lower readability. Besides, the game should not allow the players to input incorrect values such as putting words in places where number input is required. When adding a new map to the game the format of the file will be checked if it is adaptable to the game.

Maintenance

***Extensibility*:** The game should allow new functionalities, such as the addition of new tokens or flexible rules that change the dynamic of the original Monopoly game. The system is divided into parts so that each part can be extended easily because they will not have big effects on other components. JavaFX library provides a modifiable interface for the UI components. Besides, the purpose of this project is to allow UI components to change.

***Portability*:** Portability is significant for games. Our system will be implemented in Java so any computer that has the suitable Java Runtime Environment can be used to play the game.

End User

***Usability*:** The players should easily be able to understand how the game is played and they should not be confused by external factors. That is to say, the game should be user-friendly. Each operation will require simple user interaction. Each operation will be divided into subparts to help the user. For example, operations like auction, trade, and build will have more than one step to complete. This design goal was favored over others because the user experience is the most important factor when designing a game.

2. High-level Software Architecture

2.1. Subsystem Composition

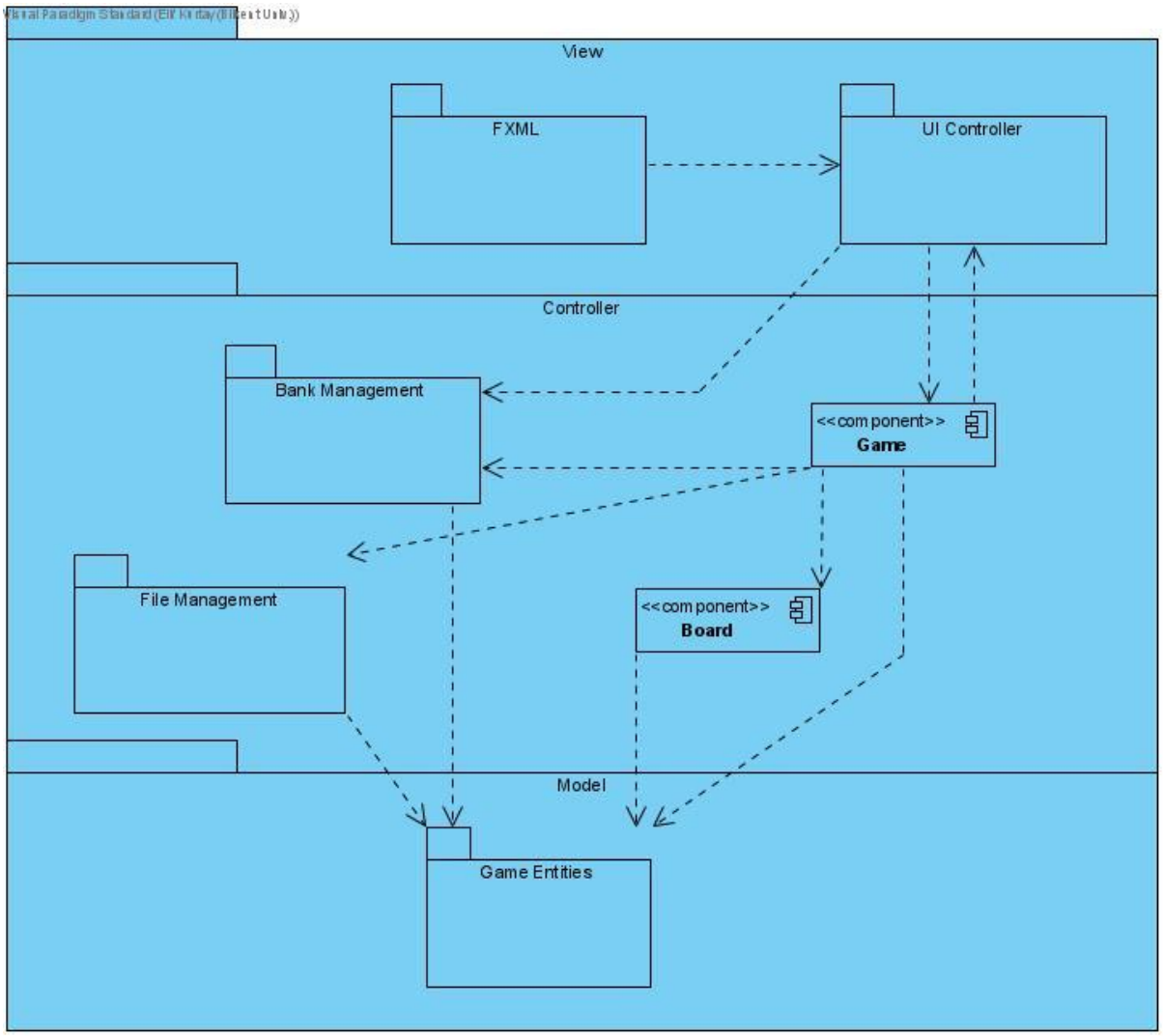


Figure 1. Subsystem Decomposition for the Monopoly game.

The system of the game is decomposed into smaller subsystems. The purpose of this decomposition is to reduce the complexity of the system. With this decomposition, it is easier to divide work, understand where the problem lies, and modify required subsystems. Figure 1 presents the subsystem decomposition for the monopoly game. We chose a three-layer architectural design in our system and decided to have a Model-View-Controller (MVC) design pattern. This system design pattern is suitable for our game software because the system has three distinct subsystems: a subsystem for the user interface, called *View*, a subsystem for game objects, called *Model*, and a subsystem for the game dynamics, functionality, and management called *Controller*. Dividing the system into these three subsystems increased coherence and lowered coupling of the system when compared to a system without decomposition. In this design pattern, packages and components that perform similar tasks that are related to each

other are in the same subsystems, and a change in one component usually affects the other packages or components inside the same subsystem.

The View subsystem consists of two packages named FXML and UI Controller. FXML includes the files for different screens for the user to interact with. UI Controller includes classes that manage the inputs from the user and the game controls. UI Controller forms a dependency with the Game class and Bank package in the Controller subsystem. In addition, Game class forms a dependency with UI Controller as well. Therefore, any interaction between View and Controller needs to be done in the UI Controller package. This provides a convenient and easy way when solving bugs related to user input, or user interface.

The Controller subsystem manages the game loop and controls game entities. It is divided into `Game` class, `Board` class, `Bank Management` package, and `File Management` package. The `Game` class has the game loop and it forms a dependency with all other subsystem components, UI Controller package and the Model subsystem. The `Board` class has board-related functions and controls. The Bank Management package includes different control mechanism classes for Bank, Auction, and Trade that are affected by the user interface. The `File Management` package includes classes to control the file-based database system for saving and loading data. All components, except for the `Game` component, only have a dependency with the Model subsystem and not with each other. By collecting all interactions in one component (`Game` class), the game becomes easily modifiable, easy to solve bugs, and extendable.

The Model subsystem includes a single package name GameEntities that has game objects such as Player, Property, and Space. These entities are affected by the components in the `Controller` subsystem. To have entities in one place adds to the writability and modifiability of the system.

This subsystem decomposition will be assisting the organization of the project and bring the project closer to our design goals.

2.2. Hardware/Software Mapping

The game will be implemented in Java and JavaFX libraries will be used. So a computer that has a Java Runtime Environment 8 or a newer version will be needed for execution. Other than the Java Runtime Environment there are no system requirements for the computer to be chosen. The game will run on the computer, so the hardware allocation will only consist of one device which is the computer itself. The game does not have any online dependency so an internet connection is not needed. Users will provide input through mouse and keyboard so those components are necessary for the game.

As the game does not include online multiplayer play, there will be no need for a server containing different players. That being said, there will not be a need for multiple nodes, meaning the game does not require functionalities being divided into nodes. It also implies that there will not be any communication between nodes as there will only be one node.

In summary, it can be said that the version of Monopoly will be executed on one hardware, which is a computer.

2.3. Persistent Data Management

As our game is played from one computer, there is limited persistent data. The persistent data includes game data. The game data consists of entity data such as but not limited to player information, map, property, cards, and so on. Maps will be stored in JavaScript Object Notation (JSON) format and the images that are needed will be stored in Portable Network Graphics (PNG) format.

The game data will be saved when the user saves through the save button or in exceptional conditions like the crash of the program. The game can be loaded later on to continue or view the scores of the players even if the game is finished. If the game is not finished, the loaded game has elements such as where the players are positioned, how much money they possess, and which properties they own. There will be a separate subsystem called File Management for data storage.

These data sets will be stored locally in different modifiable flat files. These data sets are quite small and have low information density. In general, flat files can lead to problems when there is concurrent access. However, there will not be a significant amount of concurrent accesses and no finer levels of detail in this software. There will be only one writer and one reader at an execution. Similarly, there is no far-reaching use of associations to retrieve data, nor there is the need to have associations between among objects. For these reasons, flat files were selected to store our persistent data as the other two types (relational databases and object-oriented databases) would only slow down the storage of data.

2.4. Access Control and Security

Table 1 provides the access matrix for the monopoly game. The only actor in our version of the Monopoly game is a player. A player can play a game, change the settings, load a previous game, or read the credits. However, the player is not able to change any attributes, like how much money players own, or their properties. It can be said that there is a minor limitation over access at that part.

The player can access UI controller classes through the interface. They can create a new game, load a game, change the settings, read the credits, or exit through the main menu. In the new game screen, users can add or remove players, choose or load a map, start the game or turn back to the main menu. On the game screen, they have access to all the operations required by the game and the system options like save, restart, and settings. In the settings menu, the user can go back to the main menu or the game screen. In the credits menu, the user has access to the `backButtonAction` method.

Objects Actors	**MainMenuController**	**NewGameMenuController**	**GameScreenController**
Player	newGameButtonAction() loadGameButtonAction() settingsButtonAction() creditsButtonAction() backButtonAction()	startButtonAction() backButtonAction() addPlayerButtonAction() removePlayerButtonAction() addMapButtonAction()	finishTurn() assetsButtonAction() redeemButtonAction() mortgageButtonAction() buildButtonAction() tradeButtonAction() saveButtonAcion() restartButtonAction() exitButtonAction()
Objects Actors	**LoadGameMenuController**	**SettingsMenuController**	**CreditsController**
Player	loadGameButtonAction() backButtonAction()	backButtonAction()	backAction()

Table 1. Access matrix

The system is not a multi-user system, therefore there is no need for the creation of accounts or a login system for authentication, as the game will automatically start after execution. There may be more than one player in one game, but since they all access the game from the same computer, the actor remains singular. Together with the fact that we will not be using a database, the game can be said to be secure in terms of the distribution of private user information and there won't be any need to make major efforts to secure access control.

Users can modify the contents of a saved file or find a way to cheat while playing the game. We are not taking any action towards unfavorable actions done by players because the game is an offline game.

2.5. Control Flow

When a game is started through the user interface a Game object will be initialized and the `gameLoop` method will be called. The game loop will wait for user input. The game is a turn-based game, therefore, every activity has a preceding and a next activity, until the game ends. The turn process consists of the acts that the player does before the turn advances to the other player. Accordingly, the turn processes are event-driven.

When a player provides input, such as the dice rolling or participating in the bidding, an event will be fired to advance to the next stage of the turn. Similar action will happen when a player finishes his turn. As there are not many concurrent events and more than one user interaction only happens through the act of engaging in an auction, there is no need for threads.

The control flow will be centralized. The dynamic behavior will be controlled by the Game object. We chose a centralized design because modifying the control structure will be easy in the future.

2.6. Boundary Conditions

***Configuration*:** There are two ways to configure this game. The first way is to create a new game through the new game screen. The users will add the players' names and choose a map. The second way is to configure by loading a previous game. The persistent data will be created upon the exit of a game. When the game is saved with the save game option, the game closes and all its persistent data will be saved into the flat files. For example, when the game is closed, the game object that has information on the players' positions, wealth, and possessions will be recorded in the file. A new game can be configured using that previous game. The settings will be created with default values when the game starts and will be updated with every change that the user makes for that execution. For any configuration of the game, the settings will be set to default values.

***Start-up*:** The game starts upon the running of the executable file. When the user creates a new game through the main menu, The game will be configured as described above. The image of the monopoly board and the players' UI are loaded at this point.

***Shutdown*:** The player can terminate the game by clicking the exit button in the main menu.

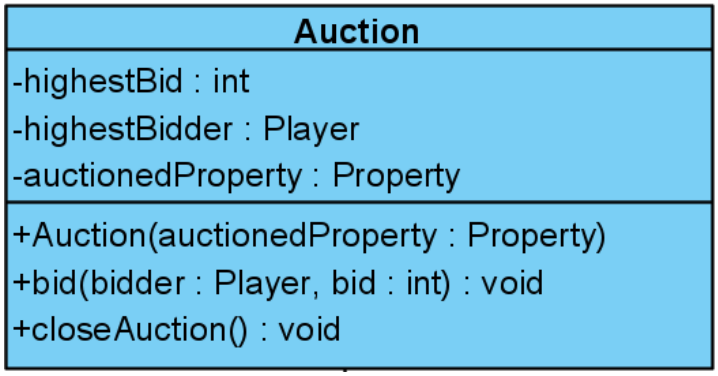
***Exception handling*:** The game will be automatically saved periodically after the completion of each three laps. In the case of a program crash, there may be small losses, however, in the worst case, a previous version of the game that starts from the last three laps completed will be stored in the files. These automatic saves will be available on the load game screen in case of a crash. During a normal termination of the game, if the user chooses not to save the game, automatically saved game data will be deleted.

Errors will be saved in a log file to keep track and permit future solutions of bugs.

3. Subsystem Services

3.1. Bank Management

Auction Class



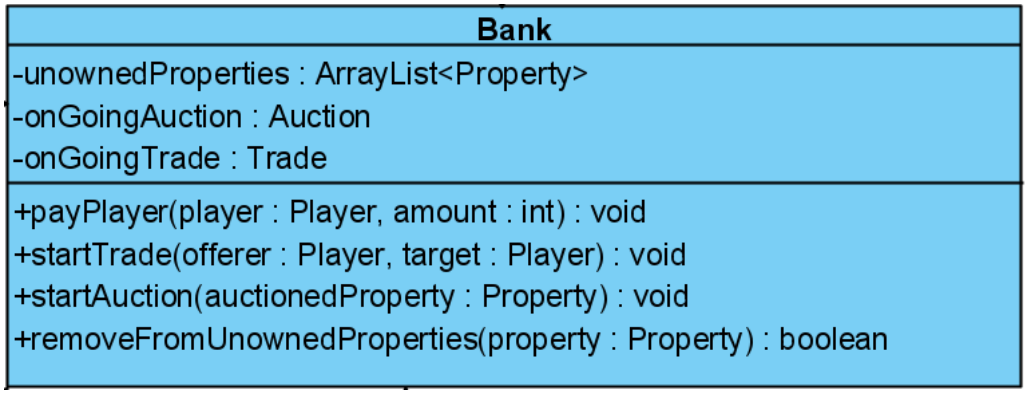
Attributes:

- private final Property auctionedProperty: This attribute is an instance of Property class that is auctioned.
- private int highestBid: This attribute is used to determine the winning amount in the auction.
- private Player highestBidder: This attribute is used to determine the winning player in the auction.

Methods:

- void bid(@NotNull Player bidder, int bid): This method updates the winning bid if a new bid is higher than the previous highest one.
- boolean closeAuction(): This method ends the auction and the winner of the auction buys the property.

Bank Class



Attributes:

- private ArrayList<Property> unownedProperties: This attribute is an ArrayList of the instances of Property class that are not owned by players.
- private Auction onGoingAuction: This attribute is an instance of Auction class that is ongoing during the game or null if there is no ongoing auction.
- private Trade onGoingTrade: This attribute is an instance of Trade class that is ongoing during the game or null if there is no ongoing trade.

Methods:

- public void payPlayer(@NotNull Player player, int amount): This method pays a player a specified amount from the bank.
- public void startTrade(Player offerer, Player target): This method starts a trade between an offerer and a target player.
- public void startAuction(Property auctionedProperty): This method starts an auction for an unowned property.
- public boolean removeFromUnownedProperties(Property property): This method removes a property form the unowned properties list. Returns true if it successfully removes the property from the list.

Trade Class

Attributes:

- private final Player offerer: This attribute is an instance of Player class that is the offerer of the trade.
- private final Player target: This attribute is an instance of Player class that is the target of the trade.

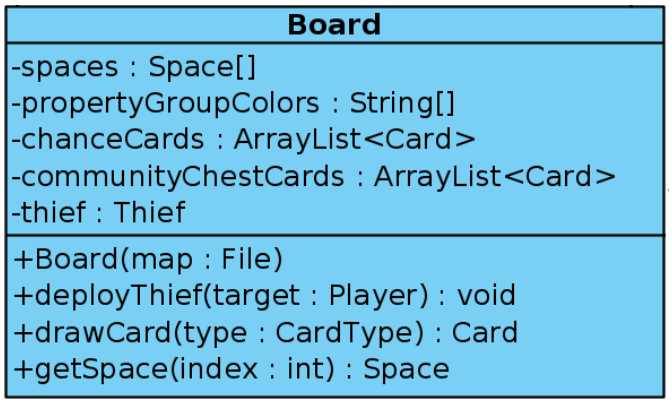
- private ArrayList<Property> offeredProperties: This attribute is an ArrayList of Property class that is offered as part of the trade.
- private int offeredMoney: This attribute is the amount of money offered as part of the trade.
- private ArrayList<Property> wantedProperties: This attribute is an ArrayList of Property class that is wanted as part of the trade.
- private int wantedMoney: This attribute is the amount of money wanted as part of the trade.
- private int offeredGOOJC: This attribute is the number of the Get Out Of Jail Cards offered as part of the trade.
- private int wantedGOOJC This attribute is the number of the Get Out Of Jail Cardswanted as part of the trade.
- private boolean accepted: This attribute is true if the offer is accepted.

Methods:

- void offer(ArrayList<Property> properties, int money, int goojc): This method is used to edit the offered part of the trade.
- void want(ArrayList<Property> properties, int money, int goojc): This method is used to edit the wanted part of the trade.
- void sendOffer(): This method sends the offer to the target.
- void acceptOffer(): This method accepts the offer.
- boolean closeTrade(): This method closes the trade. Returns true if the offer is accepted.

3.2. Board

Board Class



Attributes:

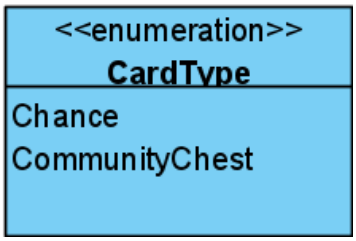
- private Space spaces[]: This attribute is an array of the Space class that is part of the board in the game. By default, it should hold 40 instances of Space.
- private String propertyGroupColors[]: This attribute is an array of the String class that denotes the property group colors.
- private ArrayList<Card> chanceCards: This attribute is an ArrayList of the Card class that makes up the Chance cards.
- private ArrayList<Card> communityChestCards: This attribute is an ArrayList of the Card class that makes up the Community Chest cards.
- private Thief thief: This attribute is an instance of the Thief class. It is initialized when a thief card is played to deploy a Thief.

Methods:

- public Space getSpace(int index): This method returns the space on the specified index.
- public void deployThief(Player target): This method deploys the Thief by initializing the thief property with a target.
- public Card drawCard(CardSpace.CardType type): This method draws and returns a card from the top of the deck of the specified card type. It then adds the card back to the bottom of the deck.

3.3. Game Entity

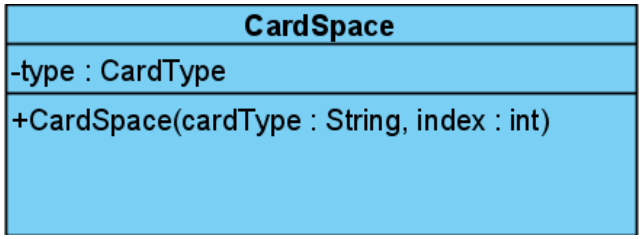
CardType Enum



Enumerations:

- CHANCE: Indicates the card's type is that of a Chance Card
- COMMUNITY_CHEST: Indicates the card's type is that of a Community Chest Card

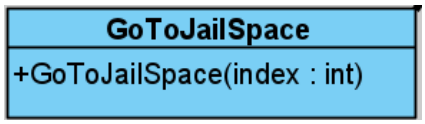
CardSpace Class



Attributes:

- private CardType type: This attribute is a CardType enumeration used to indicate if the card is a Chance Card or Community Chest Card

GoToJailSpace Class



Methods:

- public void sendToJail(Player player): This method sends the given Player to jail.

JailSpace Class

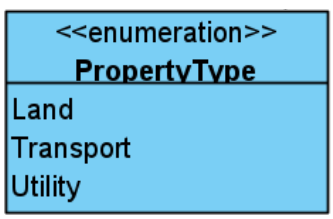
Attributes:

- private HashMap<Player, Integer> jailRecord: This attribute is a HashMap of Player and Integer classes which are the players in jail and the number of turns they spent in jail.

Methods:

- public void releasePlayer(Player player): This method releases the given player from the jail.
- public Property getAssociatedProperty(): This method gets the associated property.

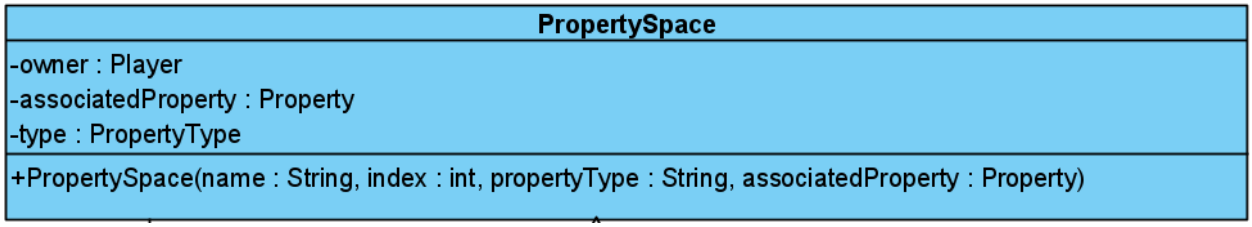
PropertyType Enum



Enumerations:

- LAND: Indicates the property's type is that of a Land Property.
- TRANSPORT: Indicates the property's type is that of a Transport Property.
- UTILITY: Indicates the property's type is that of a Utility Property.

PropertySpace Class



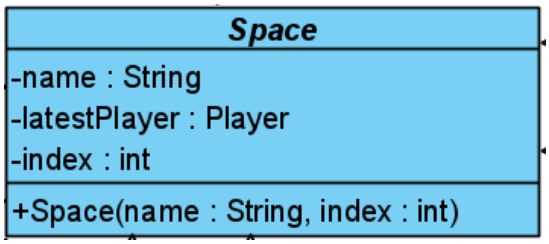
Attributes:

- private Player owner: This attribute is an instance of Player Class that owns the property.
- private int value: This attribute denotes the value of the property to be used in trades, mortgages or in auctions.
- private Property associatedProperty: This attribute is an instance of Property Class that is associated with this particular Space.
- private PropertyType type: This attribute is a PropertyType enumeration used to indicate what type of property is on the Space.

Methods:

- public boolean buySpace(): This method is used to buy the space.

Space Abstract Class



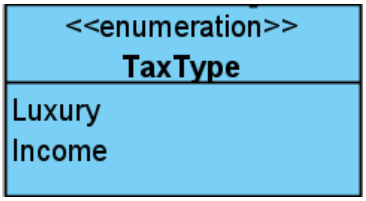
Attributes:

- private String name: This attribute denotes the name of the space.
- private Player latestPlayer: This attribute is an instance of the Player Class that last stepped on the space.
- private int index: This attribute denotes the index number of the space on the board.

Methods:

- public Property getAssociatedProperty():This method gets the associated property.

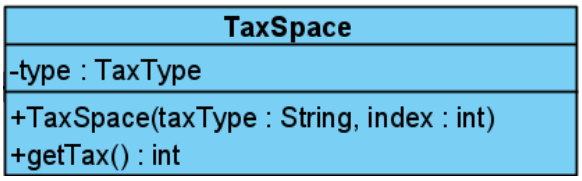
TaxType Enum



Enumerations:

- LUXURY: Indicates the tax's type is that of a Luxury Tax.
- INCOME: Indicates the tax's type is that of an Income Tax.

TaxSpace Class



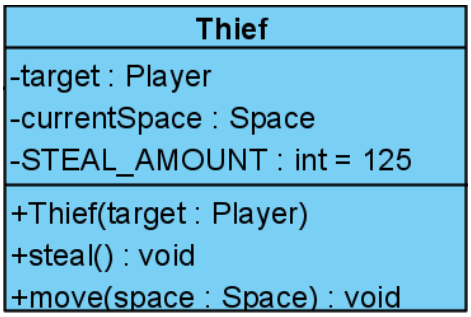
Attributes:

- private TaxType type: This attribute is a TaxType enumeration used to indicate what type of tax is applied to the Players that step on the Space.

Methods:

- public int getTax(): This method returns the base tax according to the TaxType of the Space.

Thief Class



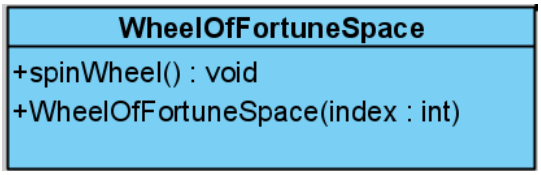
Attributes:

- private Player target: This attribute is an instance of the Player Class that is being targeted by the Thief.
- private Space currentSpace: This attribute is an instance of the Space Class that the Thief is standing on.
- private final static int AMOUNT: This attribute is the amount the thief steals from its target. The constant is set as 125.

Methods:

- public void steal(): This method steals money from the target. It is called when the thief catches up to its target.
- public void move(Space space): This method moves the Thief on the Board.

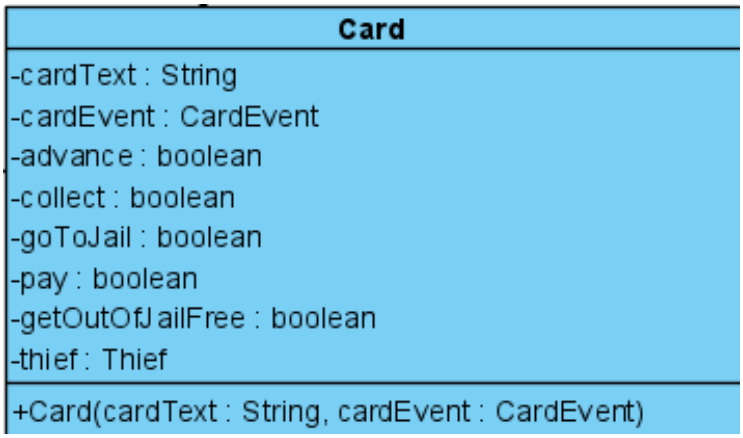
WheelOfFortuneSpace Class



Methods:

- public int spinWheel(): This method spins the wheel and generates a random amount to be used as the reward.

Card Class

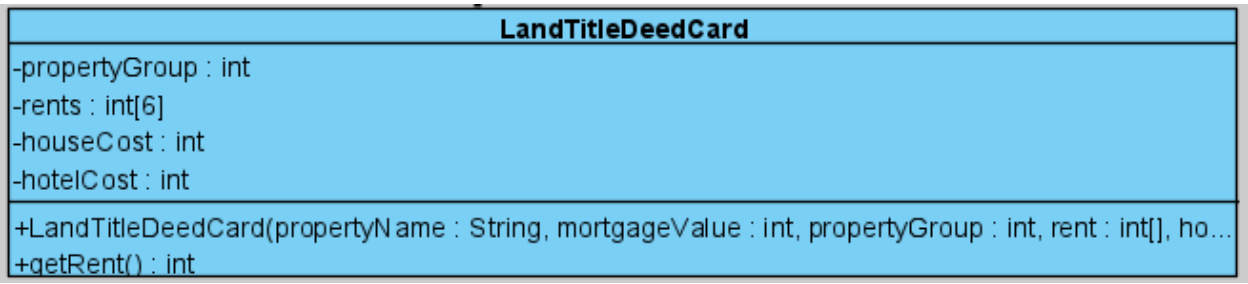


Attributes:

- private String cardText: This attribute denotes the text of the Card.
- private CardEvent cardEvent: This attribute is an instance of CardEvent class that denotes the event to occur when the card is opened.
- private boolean advance: This attribute denotes if the card causes an advance event to occur.
- private boolean collect: This attribute denotes if the card causes a collect event to occur.
- private boolean goToJail: This attribute denotes if the card causes go to jail event to occur.
- private boolean pay: This attribute denotes if the card causes a pay event to occur.

- private boolean getOutOfJailFree: This attribute denotes if the card causes receive get out of jail free event to occur.
- private boolean thief: This attribute denotes if the card causes a thief event to occur.

LandTitleDeedCard Class



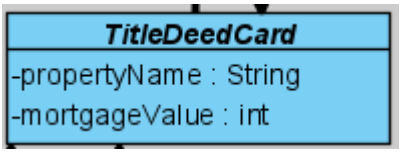
Attributes:

- private int propertyGroup: This attribute denotes which property group the land belongs to.
- private int[] rents: This attribute denotes the rent to be paid according to the amount of buildings the land has.
- private int houseCost: This attribute denotes how much each house building action costs.
- private int hotelCost: This attribute denotes how much hotel building action costs.

Methods:

- public int getRent(@NotNull Property property): This method calculates the rent to be paid.

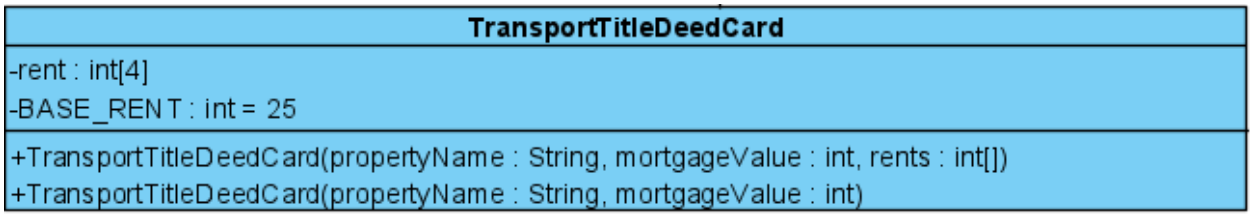
TitleDeedCard Class



Attributes:

- private String propertyName: This attribute denotes the name of the property.
- private int mortgageValue: This attribute denotes the property's mortgage value.

TransportTitleDeedCard Class



Attributes:

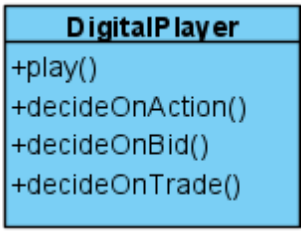
- private int[] rents: This attribute denotes the rent to be paid according to the amount of TransportTitleDeedCards a player owns. The rent cost doubles for each other TransportTitleDeedCards a player owns.
- private final int BASE_RENT: This attribute denotes the base rent to be paid if the player owns only a single TransportTitleDeedCard. The base value is determined to be 25.

UtilityTitleDeedCard Class

Attributes:

- private int[] diceMultipliers: This attribute denotes the dice multipliers which will be used to calculate the rent to be paid.

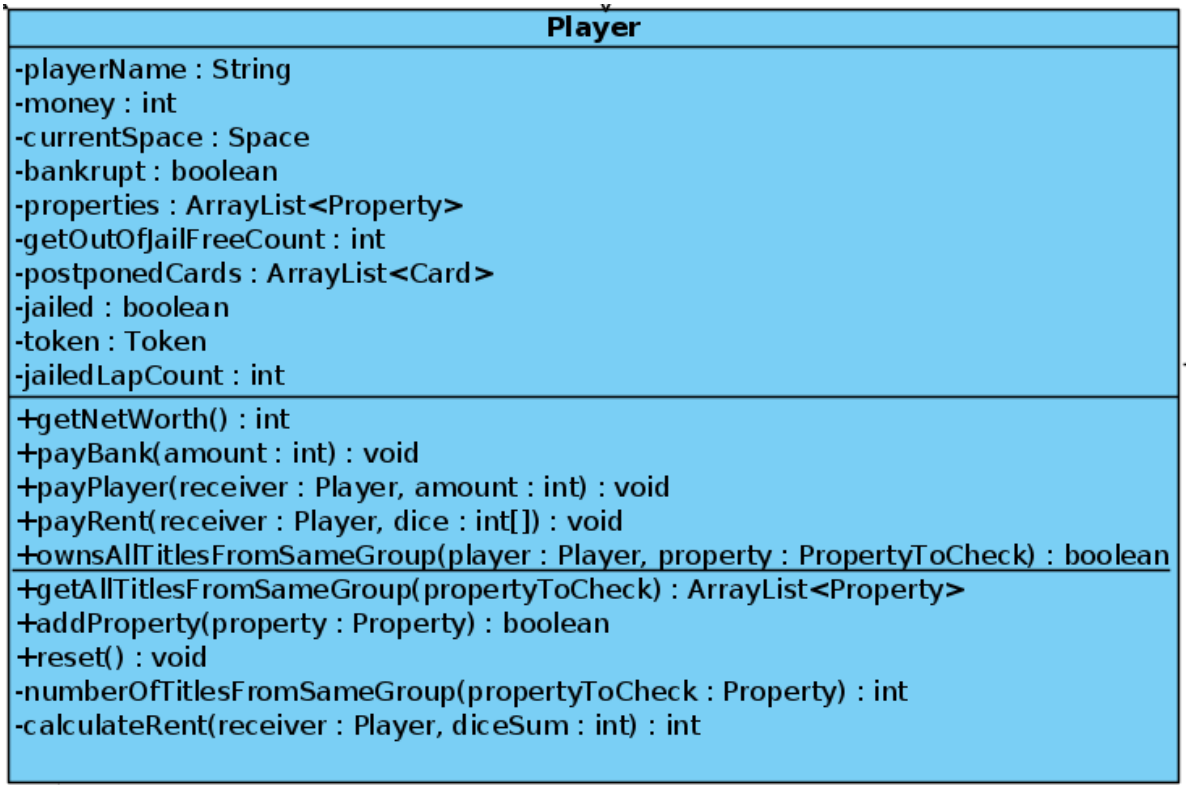
DigitalPlayer Class



Methods:

- public void play(): This method performs the actions of the DigitalPlayer.
- public void decideOnAction(): This method decide on the next action to be taken by the DigitalPlayer depending on the state of the game.
- public void decideOnBid(): This method assesses the auction and decides on a bid to be made by the DigitalPlayer.
- public void decideOnTrade(): This method assesses a trade offer and decides to whether accept the offer or reject it.

Player Class



Attributes:

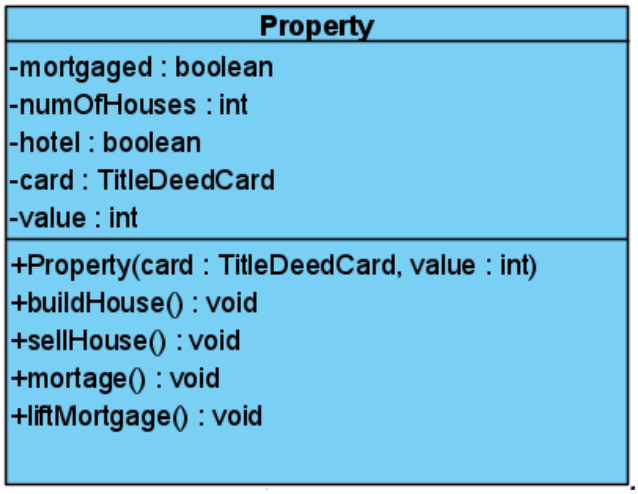
- private String playerName: This attribute denotes the name of the Player.
- private int money: This attribute denotes the amount of money the Player possesses.
- private Space currentSpace: This attribute is an instance of the Space class which is the current one the Player is standing on.
- private boolean bankrupt: This attribute denotes whether the Player has gone bankrupt or not.
- private ArrayList<Property> properties: This attribute is an ArrayList of the Property class which are the properties the Player owns:
- private int getOutOfJailFreeCount: This attribute denotes the amount of GetOutOfJailFreeCards the player owns.
- private ArrayList<Card> postponedCards: This attribute is an ArrayList of the Card class which are the cards the Player has drawn but not opened yet.
- private boolean jailed: This attribute denotes whether the player has been jailed or not.
- private Token token: This attribute is an instance of the Token class which is the Token that represents the Player.
- private int jailedLapCount: This attribute denotes the number of laps the Player has spent in jail.

Methods:

- public int getNetWorth(): This method calculates and reutrns the player's net worth.
- public void payRent(@NotNull Player receiver, int[] dice): This method is used to pay rent to another Player with regards to the dice roll.

- public void payPlayer(@NotNull Player receiver, int amount): This method is used to pay money to another player.
- private int calculateRent(Player receiver, int diceSum): This method is used to calculate the rent the Player has to pay with regards to the dice roll.
- public static boolean ownsAllTitlesFromSameGroup(Player player, Property propertyToCheck): This method returns whether the Player owns all of the tiles of the group that the propertToCheck is on.
- public ArrayList<Property> getAllTitlesFromSameGroup(Property propertyToCheck): This method returns an ArrayList of the Property class which are in the same group as the propertyToCheck..
- private int numberOfTitlesFromSameGroup(Property propertyToCheck): This method returns the amount of titles in the group of propertyToCheck.
- public boolean payBank(int amount): This method is used to make the player pay a certain amount to the Bank and it returns true if the action is successful.
- public boolean addProperty(Property property): This method adds the property to the Player's properties and returns true if the action is successful.
- public void reset(): This method is used to reset a Player's attributes.
- public ArrayList<Property> getProperties(): This method returns the ArrayList of Property class, which are owned by the Player.

Property Class



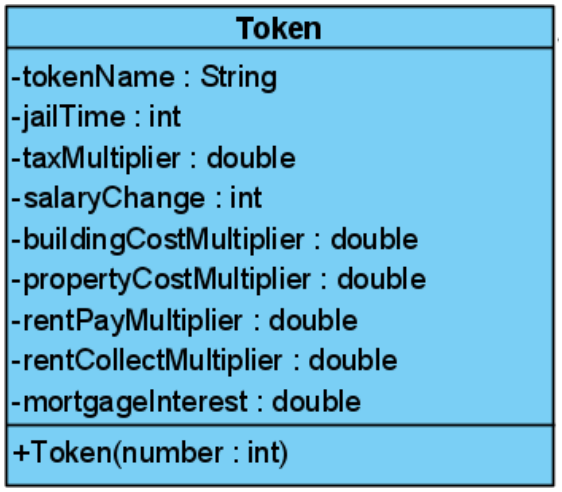
Attributes:

- private boolean mortgaged: This attribute denotes whether the Property is mortgaged or not.
- private int numOfHouses: This attribute denotes the number of houses on the Property.
- private boolean hotel: This attribute denotes whether there is a hotel on the Property or not.
- private TitleDeedCard card: This attribute is an instance of TitleDeedCard which is the card that is paired with the Property.
- private int value: This attribute denotes the value of the Property.

Methods:

- public void buildHouse(): This method is used to build a house on the Property.
- public void sellHouse (): This method is used to sell a house on the Property.
- public void mortgage (): This method is used to mortgage the Property.
- public void liftMortgage (): This method is used to lift the mortgage on the Property.

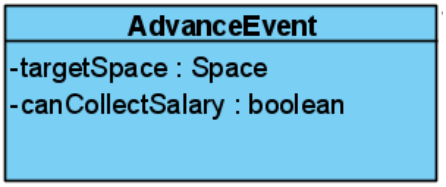
Token Class



Attributes:

- private String tokenName: This attribute denotes the name of the Token.
- private int jailtime: This attribute denotes the time in jail the owner of the Token has to spend.
- private double taxMultiplier: This attribute is used to determine how much tax the owner of the Token has to pay.
- private int salaryChange: This attribute is used to determine how much salary the owner of the Token receives.
- private double buildingCostMultiplier: This attribute is used to determine how much money the owner of the Token has to pay when building a house or a hotel.
- private double propertyCostMultiplier: This attribute is used to determine how much money the owner of the Token has to pay when purchasing a Property.
- private double rentPayMultiplier: This attribute is used to determine how much money the owner of the Token has to pay don rent.
- private double rentCollectMultiplier: This attribute is used to determine how much money the owner of the Token receives on rent.
- private double mortgageInterest: This attribute is used to determine how much money the owner of the Token has to pay when lifting a mortgage.

AdvanceEvent Class



Attributes:

- private Space targetSpace: This attribute is an instance of the Space class, which is the space the Player is to advance at.
- private boolean canCollectSalary: This attribute denotes if the Player can collect salary while advancing.

CardEvent Class

Attributes:

- private Player affectedPlayer: This attribute is an instance of the Player class which is the Player affected by the CardEvent.

CollectEvent Class

Attributes:

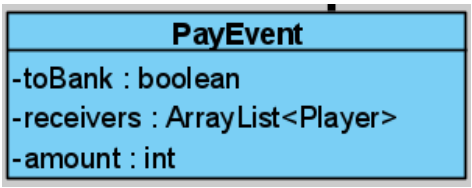
- private boolean fromBank: This attribute denotes whether the money is to be collected from the Bank or not.
- private ArrayList<Player> senders: This attribute is an ArrayList of Player class which are the ones that pay the Player the specified amount.
- private int amount: This attribute denotes the amount to be paid to the Player.

GoToJailEvent Class

Attributes:

- private boolean canCollectSalary: This attribute denotes whether the Player can collect a salary while advancing to the jail.

PayEvent Class



Attributes:

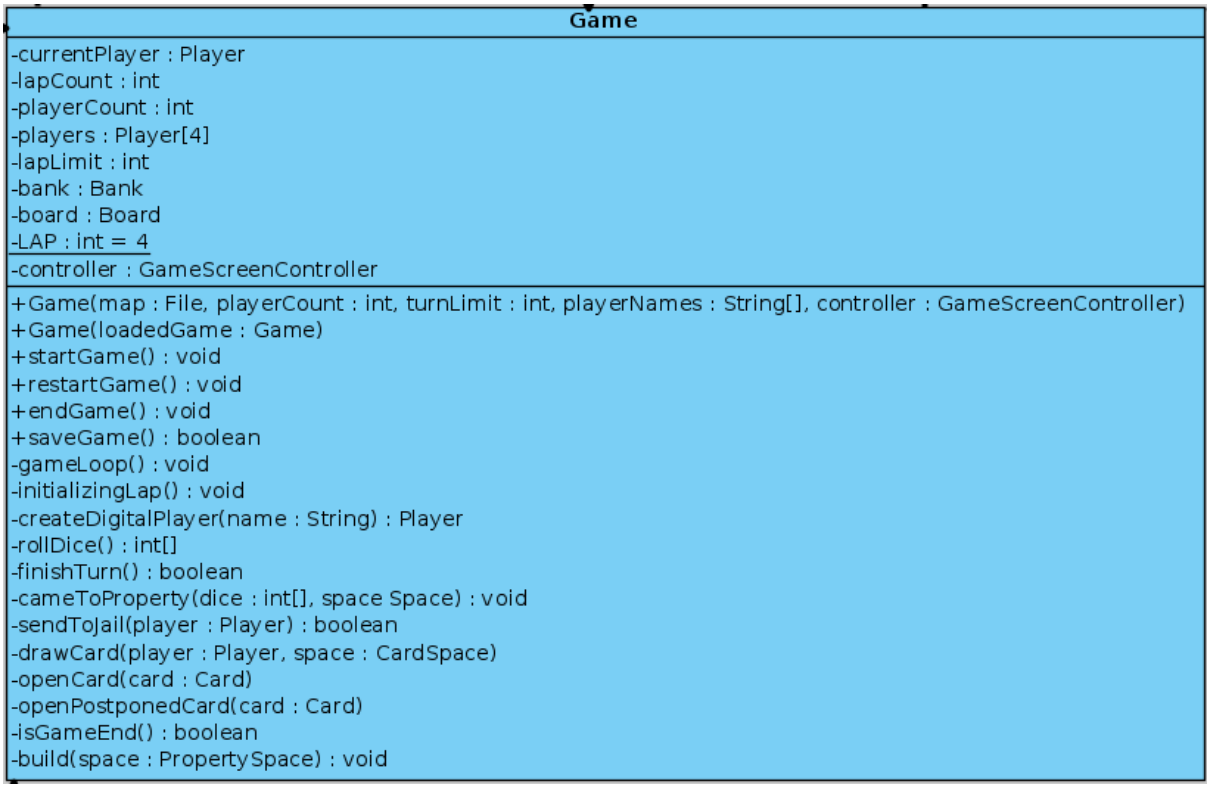
- private boolean toBank: This attribute denotes whether the Player pays the specified amount to the bank or not.
- private ArrayList<Player> receivers: This attribute is an ArrayList of Player class which are the ones that receive the specified amount.
- private int amount: This attribute is the amount the Player has to pay.

ReceiveGetOutOfJailEvent Class

ThiefEvent Class

3.4. Game

Game Class



Attributes:

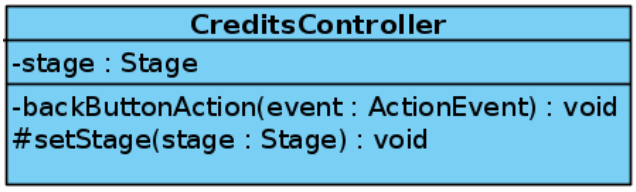
- private static final int LAP: This attribute denotes the number of turns needed to be played in a single lap. The constant's value is 4.
- private final Board board: This attribute denotes the Board the Game is played on.
- private final int lapLimit: This attribute denotes the lap limit for the game to end. It is -1 if there is no lap limit.
- private final int playerCount: This attribute denotes the number of players in the game.
- private Player[] players: This attribute is an array of the Player class which are the players of the game.
- private Player currentPlayer: This attribute is an instance of the Player class which is the one that has the active turn.
- private int lapCount: This attribute denotes how many laps have been performed in the game.
- private Bank bank: This attribute is an instance of the Bank class which is the one that holds and distributes money and properties.
- private GameScreenController controller: This attribute is an instance of the GameScreenController class which handles the user interface of the game screen.

Methods:

- public void startGame(): This method is used to start the Game.
- public void restartGame(): This method is used to restore the Game to its initial state and restart the game.
- public void endGame(): This method is used to end the game and display the results.
- public boolean saveGame(): This method is used to save the game state to a file to be able to continue later on.
- void gameLoop(): This method is used to start the gameplay loop and handle in-game events.
- private void initializingLap(): This method is used to perform the actions of the initializing lap. This lap requires a specific method as actions such as token selections and player initializations are only done in the initializing lap.
- private Player createDigitalPlayer(String name): This method is used to create a DigitalPlayer to fill the game with 4 Players. The created DigitalPlayer is returned.
- public int[] rollDice(String name): This method rolls and returns the dice.
- public void finishTurn(): This method is used to alert the user to finish their turn.
- private void cameToProperty(int[] dice, Space space): This method is used to decide which Space the Player landed on according to its dice roll.
- private void sendToJail(Player player): This method is used to send the player to jail.
- private void drawCard(Player player, CardSpace space): This method is used to draw a Card for the Player depending on the Space the Player landed on.
- private void openPostponedCard(Card card): This method is used to open a card that the Player postponed.
- private void openCard(Card card): This method is used to open a Card.
- private boolean isGameEnd(): This method returns whether the Game ended or not.

3.5. UI Controller

CreditsController Class



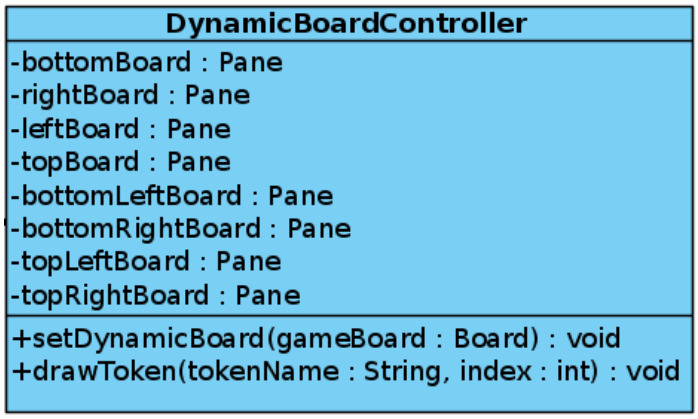
Attributes:

- private Stage stage: Holds the current JavaFX stage.

Methods:

- private void backAction(ActionEvent actionEvent): This method is used to return to the previous page.
- protected void setStage(Stage stage): Sets the stage variable to the given stage.

DynamicBoardController Class



Attributes:

- private Pane bottomBoard, rightBoard, leftBoard, topBoard, bottomLeftBoard, bottomRightBoard, topLeftBoard, topRightBoard: All of these attributes correspond to the parts of the dynamic Monopoly board in the game screen.

Methods:

- public void setDynamicBoard(Board gameBoard): Draws the game board with the Space information it gets from the Board object. This allows us to draw different game boards.
- public void drawToken(String tokenName, int index): Draws the token corresponding to the given tokenName on the given index of the board. (0 == First Space, 39 == Last Space).

GameScreenController Class

Attributes:

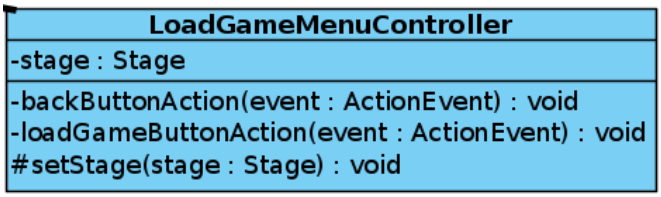
- Private DynamicBoardController dynamicBoardController: Allows the GameScreenController to draw on the DynamicBoard through the methods of DynamicBoardcontroller.

Methods:

- public void setBoard(Board board): Passes the board object to the dynamicBoardController's setDynamicBoard method for the map to be drawn to the screen.
- public int[] rollDice(String name): Shows the user the dice roll prompt and returns the rolled dice values in an array.
- public int chooseToken(String name): Shows the user the token choice prompt and returns the integer value that corresponds to the chosen token.
- public void drawToken(String tokenName,int index): Draws the token given with the tokenName property at the given index by calling the dynamicBoardController's drawToken method with these values.
- public void showMessage(String message): Shows the player a prompt with the given message and a button to close the prompt.
- public boolean buyProperty(PropertySpace property): Shows the player a prompt asking if they want to buy the given property or if they are willing to let it be auctioned among all players.
- public void finishTurn() : Shows the player a prompt that asks if they want to finish their turn.
- public boolean postponeCard(): Shows the player a prompt asking if they want to open the card they have drawn now or postpone it to open later.
- private void exitButtonAction(ActionEvent event): Shows the player a prompt asking if they want to save their game when they presses the "Exit" button in the game screen, saves if the response is positive, then exits from the game that is being played to the main menu.
- private void settingsButtonAction(ActionEvent event): Shows the player the settings menu when the player presses the "Settings" button in the game screen.
- private void restartButtonAction(ActionEvent event): Shows the player a prompt asking if they want to save their game when they press the "Restart" button in the game screen, saves if the response is positive, then restarts the game with the same players.
- private void saveButtonAction(ActionEvent event): Saves the game when the player presses the "Save" button in the game screen.
- private void tradeButtonAction(ActionEvent event): Opens the trading menu and handles the trade, when the player presses the "Trade" button in the game screen.
- private void buildButtonAction(ActionEvent event): Opens the building menu and handles the building action, when the player presses the "Build" button in the game screen.
- private void mortgageButtonAction(ActionEvent event): Opens the mortgage menu and handles the mortgage action, when the player presses the "Mortgage" button in the game screen.

- private void mortgageButtonAction(ActionEvent event): Opens the redeem menu and handles the redeem action, when the player presses the “Redeem” button in the game screen.
- private void playerAssetsButtonAction(ActionEvent actionEvent): Opens the player assets menu and shows the player their net worth, money, title deed cards, get out of jail free cards and their buildings.

LoadGameMenuController Class



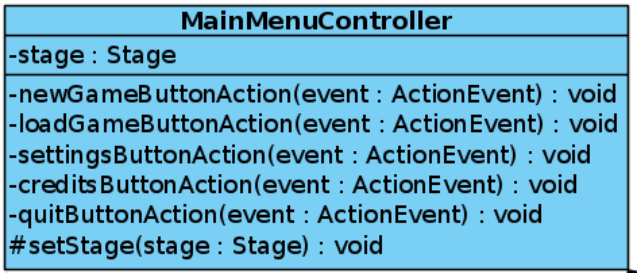
Attributes:

- Private Stage stage: Holds the current JavaFX stage

Methods:

- private void backButtonAction(ActionEvent event): This method is used to return to the previous page.
- private void loadGameButtonAction(ActionEvent event): Loads the given game file and initializes the Game and GameScreenController objects with the loaded data.
- protected void setStage(Stage stage): Sets the stage variable to the given stage.

MainMenuController Class



Attributes:

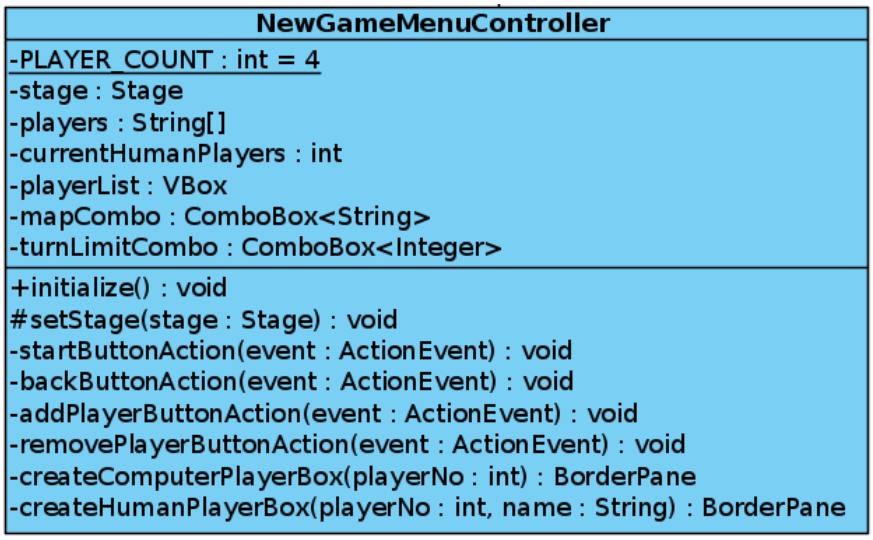
- private Stage stage: Holds the current JavaFX stage

Methods:

- private void newGameButtonAction(ActionEvent event): Navigates to the New Game menu screen.
- private void loadGameButtonAction(ActionEvent event): Navigates to the Load Game menu screen.

- private void settingsButtonAction(ActionEvent event): Navigates to the Settings menu screen.
- private void creditsButtonAction(ActionEvent event): Navigates to the Credits menu screen.
- private void quitButtonAction(ActionEvent event): Quits the application.
- protected void setStage(Stage stage): Sets the stage variable to the given stage.

NewGameMenuController Class



Attributes:

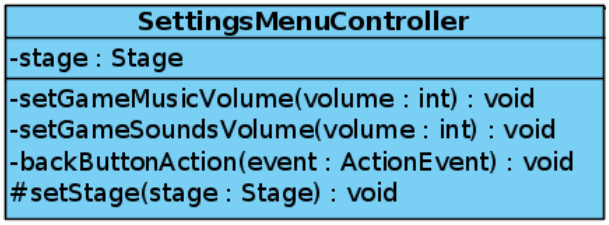
- Private static final int PLAYER_COUNT = 4: The maximum player count.
- Private Stage stage: Holds the current JavaFX stage
- Private String[] players: Holds the names of the current human players.
- Private Vbox playerList: The VBox(JavaFX component) that holds the player cards.
- Private ComboBox<String> mapCombo: The ComboBox(JavaFX component) that the user interacts with to choose the map file.
- Private ComboBox<Integer> turnLimitCombo: The ComboBox(JavaFX component) that the user interacts with to choose the turn limit.

Methods:

- public void initialize(): Creates the initial player boxes for the computer players, loads the map files from the maps folder, adds the maps to the mapCombo component and adds the turn limit options to the turnLimitCombo component.
- protected void setStage(Stage stage): Sets the stage variable to the given stage.
- private void startButtonAction(ActionEvent event): Initializes the GameScreenController and Game objects when the user presses the start button in the new game screen.
- private void backButtonAction(ActionEvent event): This method is used to return to the previous page.

- private void addPlayerButton(ActionEvent event): If there are fewer human players than the maximum number of players prompts the user to enter a player name, constructing a player box for that player and adding that player to the players list.
- private void removePlayerButton(ActionEvent event): Removes the player box, on which this method was invoked, and the player from the players list.
- private BorderPane createComputerPlayerBox(int playerNo): Creates a player box for the computer player with the given playerNo.
- private BorderPane createHumanPlayerBox(int playerNo, String name): Creates a player box for the human player with the given playerNo and name.

SettingsMenuController Class



Attributes:

- private Stage stage: Holds the current JavaFX stage

Methods:

- private void setGameSoundsVolume(int volume): Sets the game sound effects volume level to the given int.
- private void setGameMusicVolume(int volume): Sets the game music volume level to the given int.
- private void backButtonAction(ActionEvent event): This method is used to return to the previous page.
- protected void setStage(Stage stage): Sets the stage variable to the given stage.

4. Low-level Design

4.1. Object Design Tradeoffs

***Usability vs. Functionality*:** Designs that aim functionally provide more operations to the user and designs that aim usability provide a simpler interface that is easy to learn. We prioritised usability over functionality because we believe a player should focus more on the game without trying to understand what each function does. In addition, if the interface is complicated and unusable both the new and old functionalities would not be utilized to their full extent anyways reducing the overall quality of the game.

***Reliability vs. Delivery Time*:** A code that is more reliable and handles each case separately will cause the length of the code to increase and cause a difference in the delivery time. However, experiencing crashes, game data losses would provide a bad experience for any user. Therefore, we believe it is important to make a reliable system, even though delivery time would increase.

Extensibility vs. Delivery Time: An extensible code separates the system into parts so that the amount of work that should be done increases and this causes a difference in the delivery time. We aim to make the game extensible by following Object Oriented Programming (OOP) principles so that we can make fixes and improvements on the game even after the initial delivery.

4.2. Final Object Design

Figure 2 shows the UML Class diagram for the Monopoly game. We designed our classes to be compatible with the Model-View-Controller architectural pattern. We also obey the Object-oriented software principles such as abstraction, inheritance, and encapsulation.

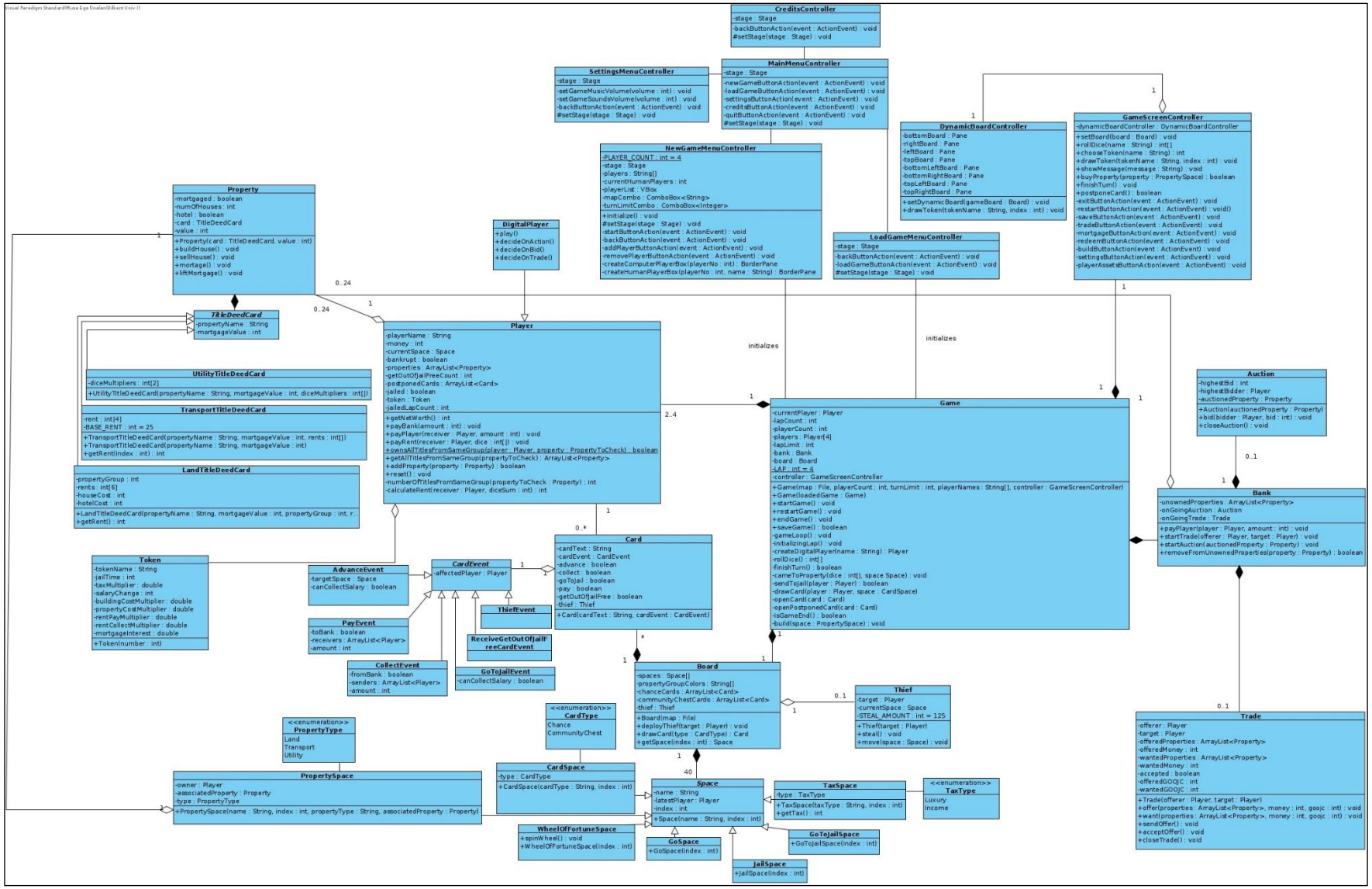


Figure 2 . UML Class Diagram

4.3. Packages

1. java.util

We used java.util classes such as ArrayList, Optional, and Scanner.

2. java.io.File

We used File class for reading and writing from text files.

3. java.fx

We used java.fx package classes that are listed below for event-driven programming, user interface components, such as scene controls, the set of classes for loading and displaying images, and so on.

- javafx.application.Platform
- javafx.event.ActionEvent
- javafx.fxml.FXML
- javafx.scene.control
- javafx.scene.image.Image
- javafx.stage.Modality
- javafx.stage.StageStyle
- javafx.event.ActionEvent
- javafx.fxml.FXML
- javafx.fxml.FXMLLoader
- javafx.scene.Parent
- javafx.scene.Scene
- javafx.stage.Stage

4. lombok

To make getters and setters for the attributes in classes automatically.

4.4. Abstract Classes

CardEvent:

This class will be used to produce card events. There will be several card events that will inherit from this class.

Space:

There are several types of spaces such as CardSpace, TaxSpace, WheelOfFortuneSpace, GoSpace, and so on. These classes will inherit from the Space class.

5. References

[1] *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

[2] Introducing JSON (JavaScript Object Notation), *https://www.json.org/json-en.html*, Accessed on November 29, 2020.

[3] OpenJavaFX Client Application Platform, *https://openjfx.io/,* Accessed on November 29, 2020.

[4] Oracle, Java Runtime Environment, https://www.oracle.com/java/technologies/javase-jre8-downloads.html, Accessed on November 29, 2020.

[5] "Monopoly Rules." Hasbro, *www.hasbro.com/common/instruct/00009.pdf*. Accessed on November 29, 2020.

[6] Monopoly | Ubisoft (US), *www.ubisoft.com*, Accessed on November 29, 2020.