CS 319 Object Oriented Software Engineering

Design Report (1st Iteration)

Instructor : Eray Tüzün

Group Members
Arnisa Fazla
Ege Türker
İbrahim Furkan Aygar
Hassam Abdullah
Oğulcan Pirim
Pınar Yücel

Fall 2020

1. Introduction

1.1 Purpose of the system

Monopoly is our implementation of the board game, Monopoly, as a desktop game that can be played online via separate laptops. The game experience will be enhanced by the addition of new features that were aforementioned in the analysis report, which will provide a more flexible and interactive game play.

1.2 Design goals

Our main goal is to provide an entertaining, fast and dependable software. These are the design goals we identified in more detail:

1.2.1 Performance

Server-client response time, such, that the time between a request from the client to the server and the UI to respond should be less than 1 second. In addition, It should be able to run on networks with a minimum bandwidth of 1 Mbps.

1.2.2 Dependability

Security and robustness is not a priority since we will not have a database to keep information and the software will not track private information. The only requirement is that, in case the users disconnect, they should reconnect back to the game in 15 seconds or they will be considered bankrupted and the game will continue without them. If they reconnect to the game within 15 seconds, the game will wait until their UI is synced with game logic; this should take a time of upto 30 seconds.

1.2.3 Cost

The game should be implemented within 1 month (development cost).

1.2.4 Maintenance

We divided the system into subsystems and subsystems into classes with well-defined responsibilities, aiming that maintaining and modifying the software for future needs will be easier. An architecture containing separate boundary, control and entity objects will be implemented, so working on different layers without affecting other layers will be possible later. Furthermore, all of the classes, methods and algorithms will be well documented and explained both in the documents and in the code.

1.2.5 End User Criteria

Target User Base: Upon observation of successful games available in the market, we can conclude that they share a common attribute.It the range of their user base.

Monopoly will implement the same feature as the popular Board Game, Monopoly, that is popular among all age groups.

Ease of Use: Most pc games nowadays use common controls depending on the genre to increase the familiarity and ease of use. Our game will be mostly a mouse pointer based implementation i.e selecting options/cards or rolling the dice like most board games available in the market. Since most users are already familiar with this design Monopoly will be easy to learn and play.

Lastly,the Overall design of the game should be straightforward and must avoid ambiguity. The help page should be easy to follow. All of the images used for components of the game and all the sound effects must be pertinent to their purpose. The game will be packaged as a jar file and must be loaded within 1 second.

1.3 Object design trade-offs

Performance vs. Portability: The implementation of our software will be done in Java using the Javafx library. We could have implemented this in another programming language such as C++ to maximize the performance but Java was chosen due to it being portable and running on all platforms i.e Windows, Linux, MacOS. As long as the platform has a JVM written for them.

User base vs Complexity: In our end user criteria, we detailed that the user base for Monopoly will be broad. Therefore, the game mechanics should be suitable for both the old and young user base. For example, keyboard shortcuts or unusual rules could have been implemented which would make it harder for the non tech-savvy older user base or the user base familiar with the standard rules of monopoly to play. Therefore, a decision was made to keep the game simple and the user base broader.

1.4 Definitions, acronyms and abbreviations

- MVC: Model View Controller
- Host/Server: A centralized computer in a computer network to which clients are connected.
- Client: A computer connected to the server and sends requests to the server to make changes in the model.

1.5 Overview

In high-level software architecture, we will present how our software will work and be implemented from a high-level perspective. We will explain the general structure of our software and how we will implement it in a way that satisfies our nonfunctional requirements. While in the low-level software architecture part; we will delve into deeper detail into all the aforementioned topics covered in the high-level architecture section.

2. High-level software architecture

High-level software architecture represents a way to look at a broader picture of components in regards to software. It includes subsystem decomposition, architectural style, hardware/software mapping, persistent data management, access control, global control flow and boundary conditions.

2.1. Subsystem decomposition

In order to deal with the complexity of the software, it was decomposed employing the Model-View-Controller design pattern, into three subsystems namely controller, user interface and game logic subsystems.

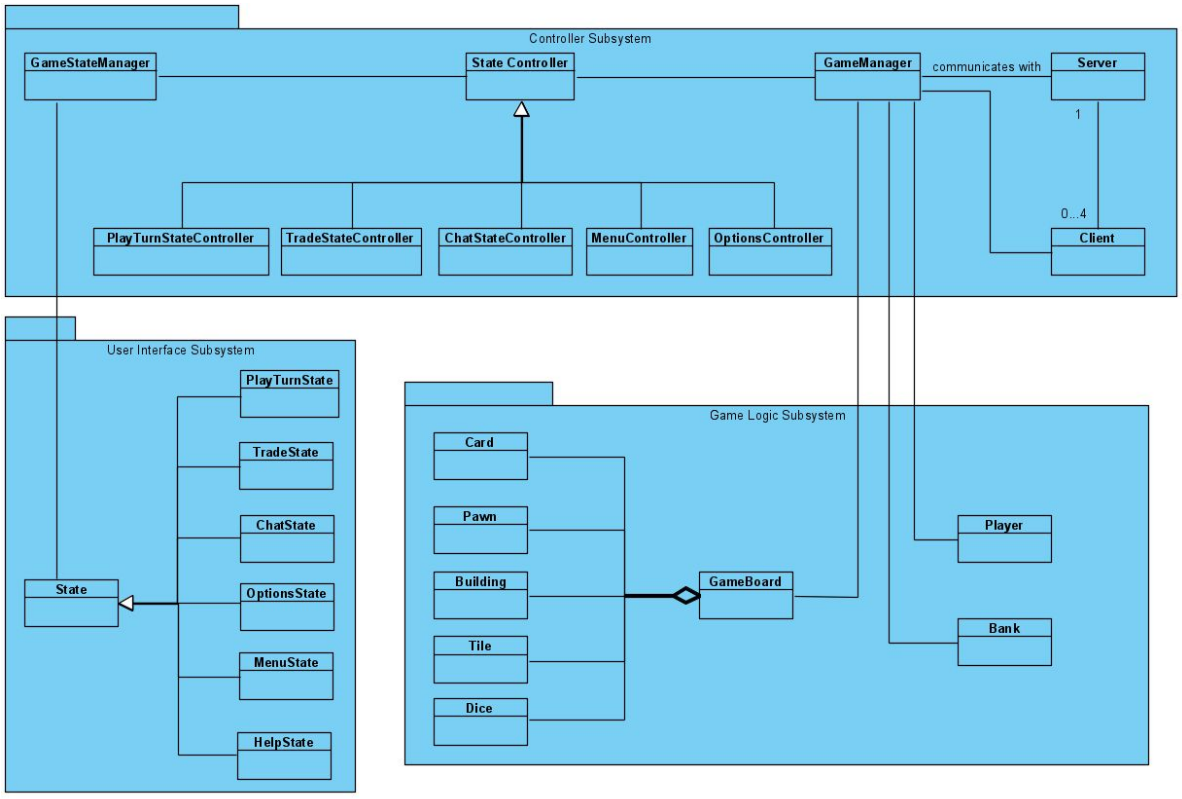

Figure 1: Subsystem decomposition diagram

2.2. Architectural style

We used MVC (Model-View-Control) architectural style. It consists of a subsystem for “view” that will only be responsible for reflecting the changes in the “model” to the screen and taking input from the user through a “UI” (user interface). There is a User Interface Subsystem to handle this. Lastly, we need a model which will have the basic game logic and all the data, which is assigned to the Game Logic Subsystem in our subsystem decomposition diagram.

2.3 Hardware/software mapping

Monopoly does not require advanced hardware. It has minimal animation, albeit, having lots of objects that are mostly processed under the hood and not visually shown. Therefore, the game is not processor intensive and can run flawlessly on a single processor. Similarly, there is no need for any extra memory storage since the data is stored server side. The game requires a mouse and keyboard as an external I/O device for the game control. The mouse will be used to click on buttons regarding options/settings etc and roll the dice as well, while the keyboard will be used to enter the players' name. This will be an online multiplayer game and will require connection to the internet. Therefore, the computer should have internet access. There is only one internal node in the System, namely the UserComputer. The UserComputer should have JDK installed and it interacts with the external node, HostComputer(Server), to update it's User Interface via the Controller. The HostComputer can be the UserComputer itself, or an external computer acting as a server. Furthermore, the UserComputer can also update its UI independently from the HostComputer(Server) when the user goes to settings, help or chat menu. In addition, Keyboard and Mouse I/O devices are also represented as external nodes. The components are associated with the subsystems: User Interface(View), Controller(Control) and Game Logic(Model). The Controller components provide game operations interface to the User Interface controller. The Game Logic component provides data operations and the Controller uses this interface. The architecture is represented by the diagram below.

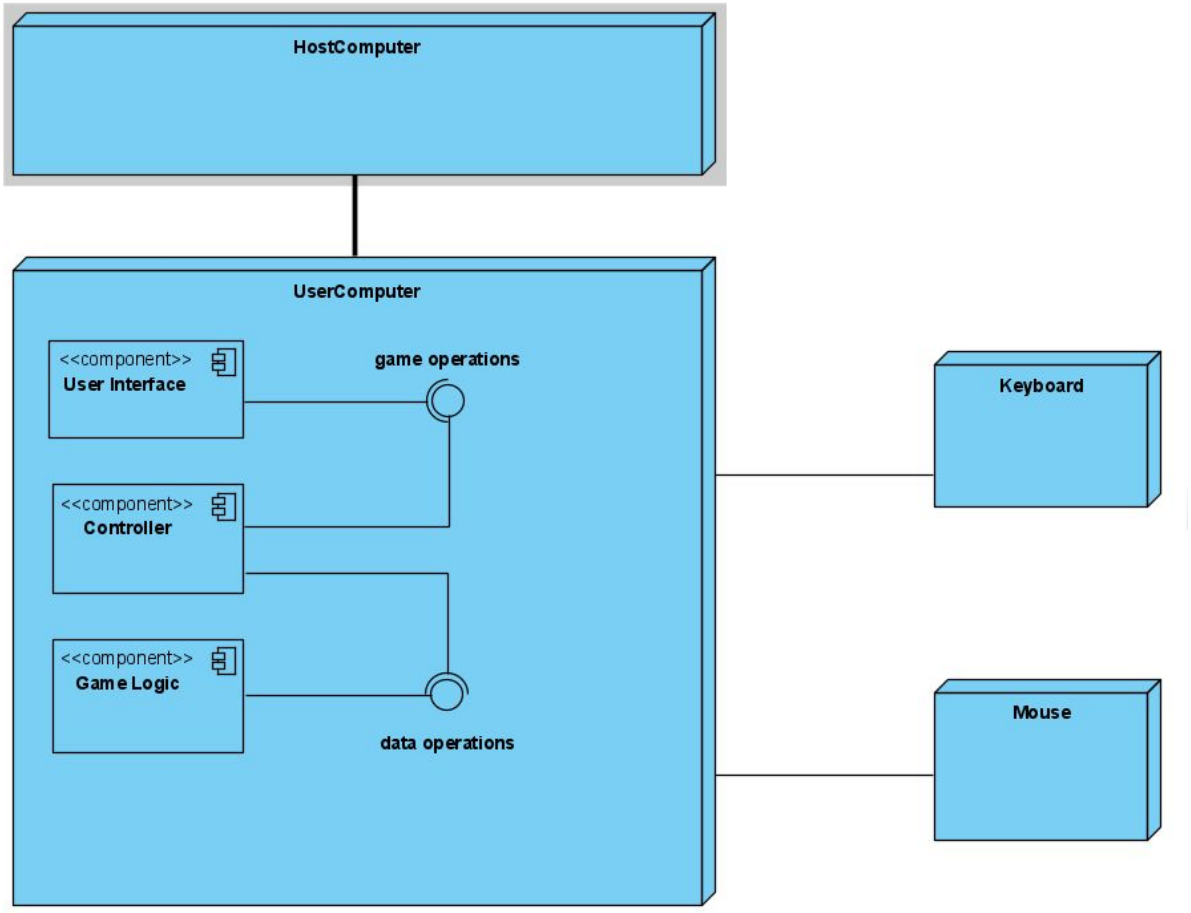

Figure 2: Hardware/software mapping

2.4 Persistent data management

We will store all related static data on the local hard drive of the user. The images of the board, cards and dice animations will be in jpg format; the animation files will be in gif format and the sounds effects will be mp3 files. Since we don't keep any data of the users and the game after the game ends we won't be implementing a database.

2.5 Access control and security

The game does not support users to login.The users can play the game and show the results when enough players have been reached.Thus, the game does not include access control, authentication.

2.6 Global control flow

In Monopoly , we have a main controller class, GameManager and it decides the flow events. Since Monopoly is a turn-based game and it depends on the mouse events,

an event driven control is chosen. Therefore, the system will have a simple structure that will be centralized within the main loop of the game.

2.7 Boundary conditions

2.7.1 Initialization

Monopoly is provided as an executable .jar file. Hence, it does not require any installation and can be run directly on any computer that has JDK installed.

2.7.2 Termination

Users will be able to quit their gaming sessions any time they wish after pausing the game. There will be an exit game option present, which upon clicking will present the user a warning prompt asking if they really want to forfeit and quit the game. However, doing so in the case of only two players remaining, will terminate the game of the other player as well declaring him/her as the winner.

2.7.3 Failure

In case of a network failure in any players connections, the game will be paused for the other players as well. If connection is re-established within 15 seconds then the player connects back to the game and the game resumes. In the case of more than two players, if connection is not re-established, then the player forfeits and the game continues for the other players. In the case of exactly two players, if connection is not re-established, then the game session of both players will be terminated and the game will declare the player with the stable connection the winner.

If the server dies during the game, then all players will be returned to the Main Menu that is loaded at the execution of the .jar file.

Furthermore, the data might get skewed during the transfer because it is not hashed and cannot be verified on the client side. In that case, the client may receive bad data and may not be able to respond to the server even though there is a stable connection. This case will also be treated as a network failure and the players will be returned back to the Main menu.

3. Low Level Design

Low level design is constituted by specific descriptions of all classes in the subsystems along with the associations between them.

3.1. Controller Subsystem

Controller subsystem is responsible for managing the software according to the input it takes from the user.

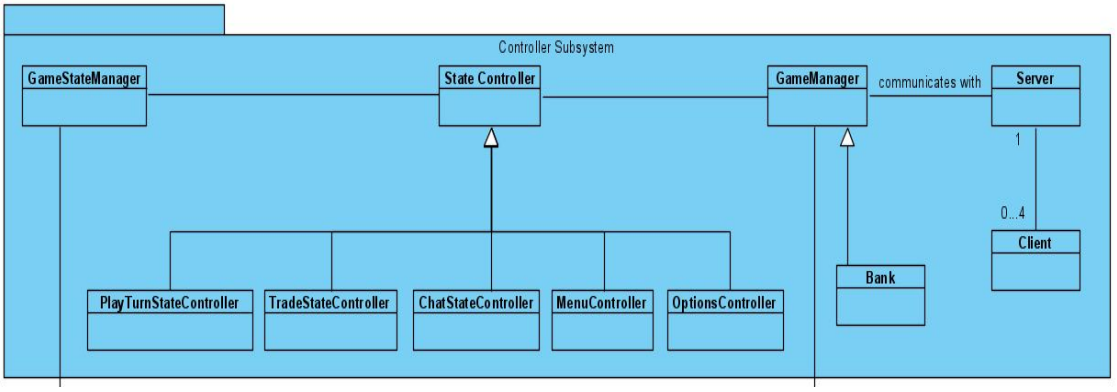

Figure 3: Controller Subsystem

3.2. Game Logic Subsystem

Game logic subsystem includes the entity objects of the software and corresponds to the model part of the MVC. We use Facade pattern on the GameBoard which defines a higher-level interface that makes the subsystem easier to use.

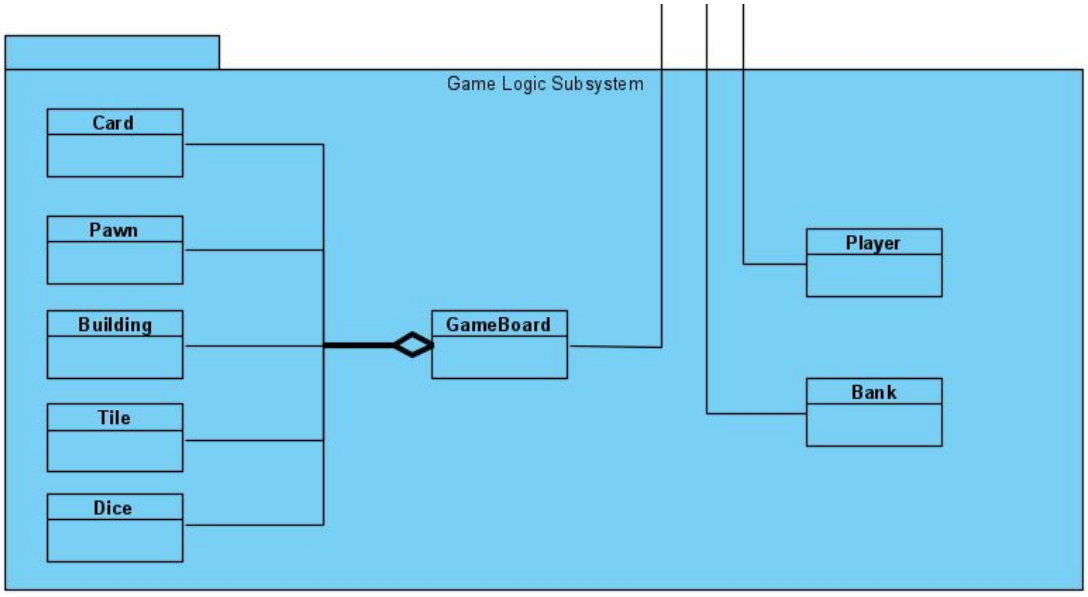

Figure 4: Game Logic Subsystem

3.3. User Interface Subsystem

User interface subsystem corresponds to the view of MVC. Screens change according to the user input and events according to the State pattern. All of the states

correspond to different screens and they all implement a common abstract class named State.

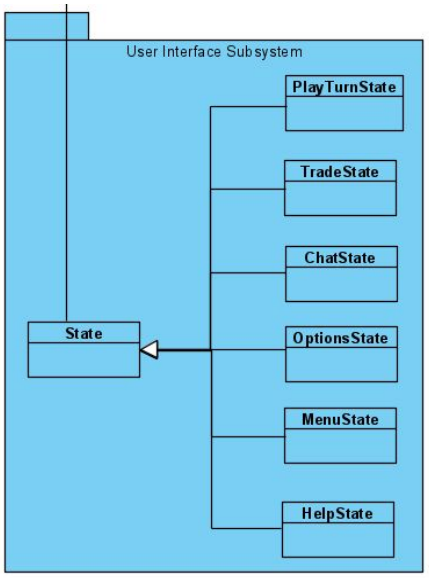

Figure 5: User Interface Subsystem

3.4. Final object design

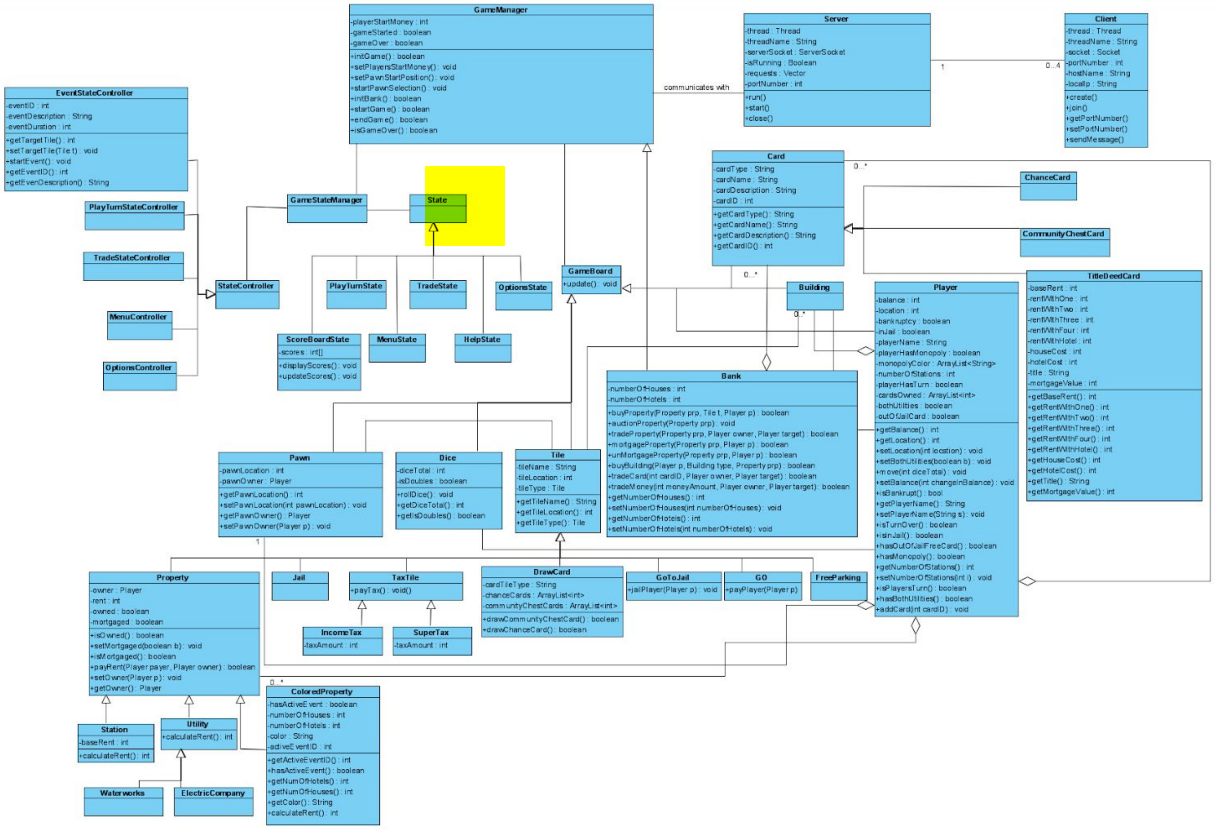

Figure 6: Final Class Diagram

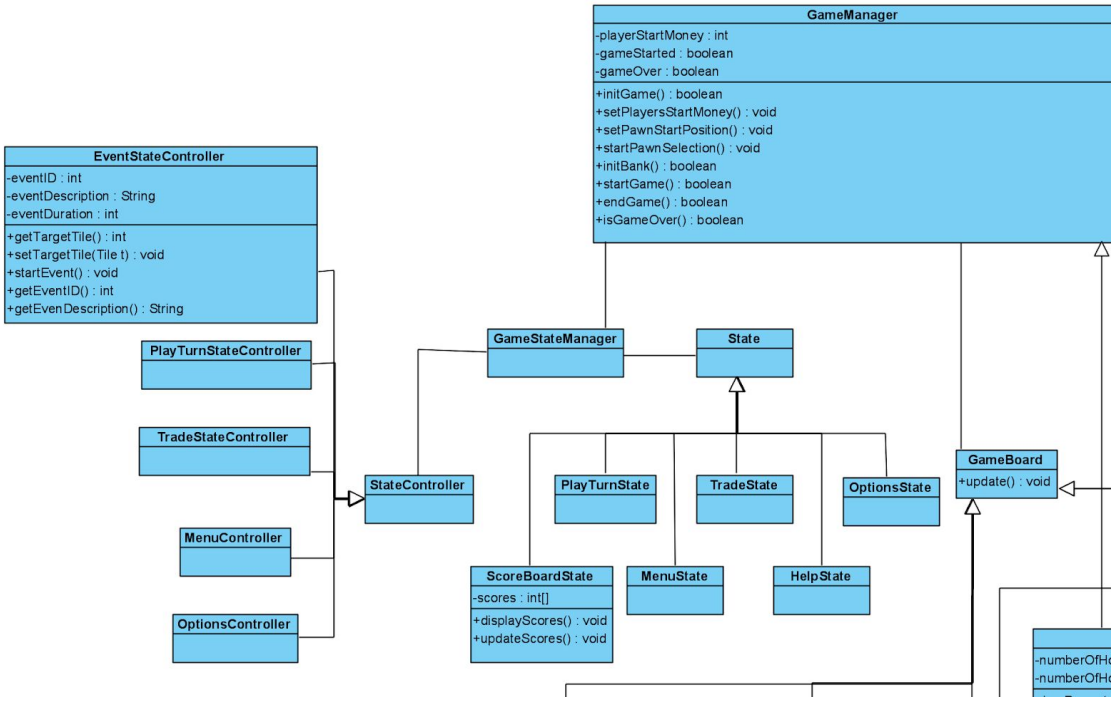

Figure 7: Final Class Diagram Zoomed-In Part 1

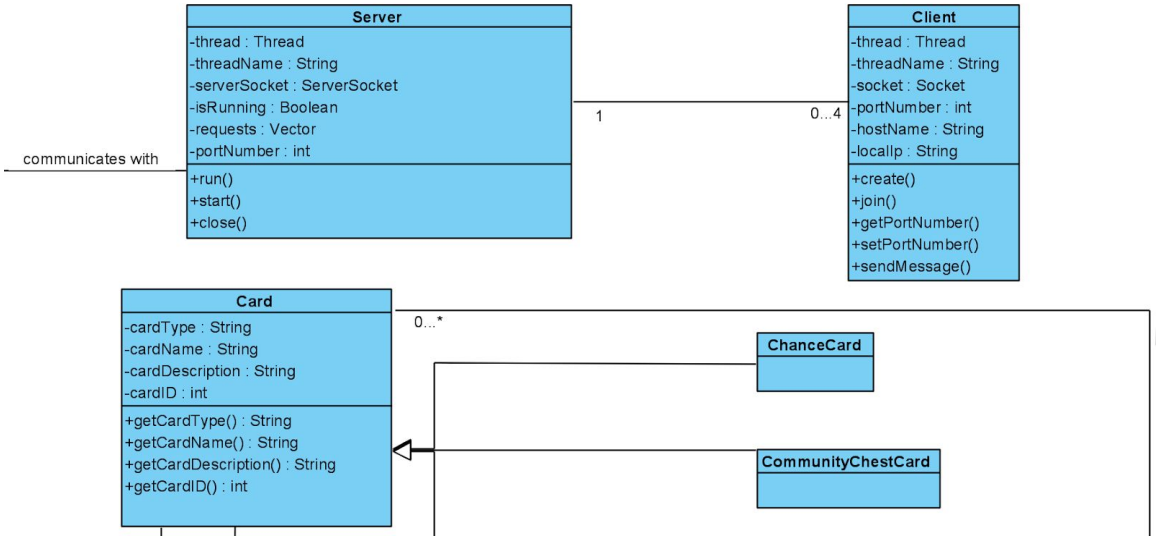

Figure 8: Final Class Diagram Zoomed-In Part 2

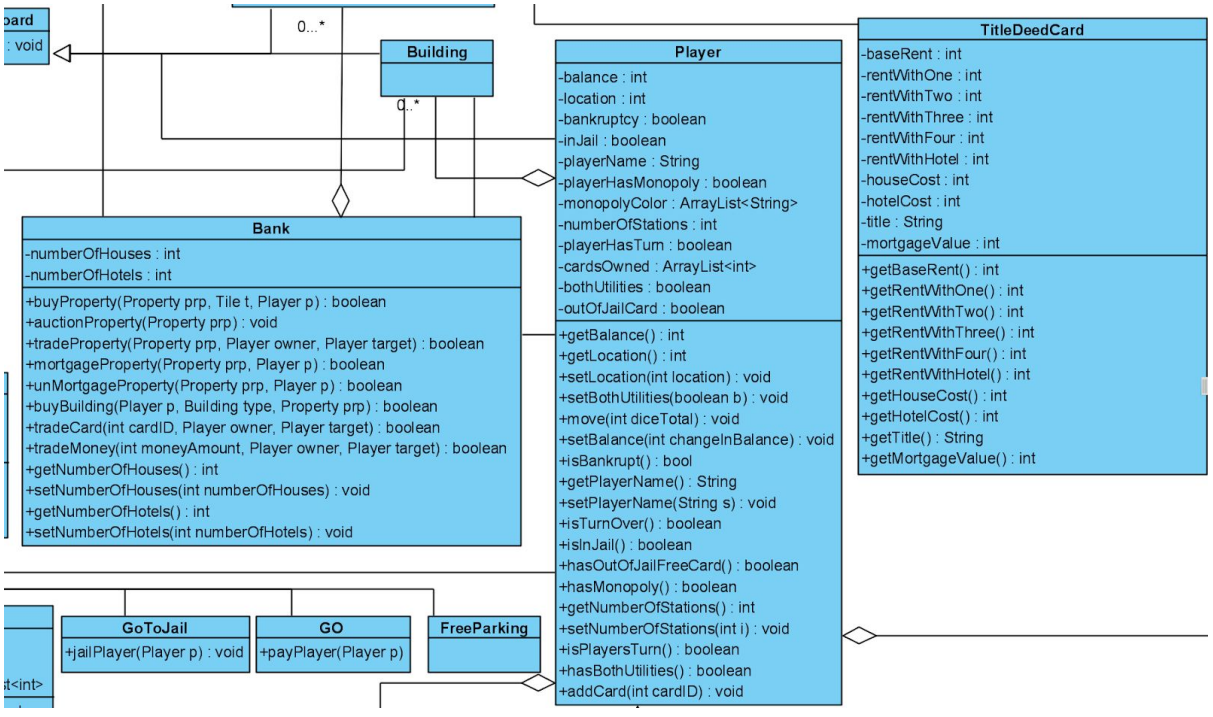

Figure 9: Final Class Diagram Zoomed-In Part 3

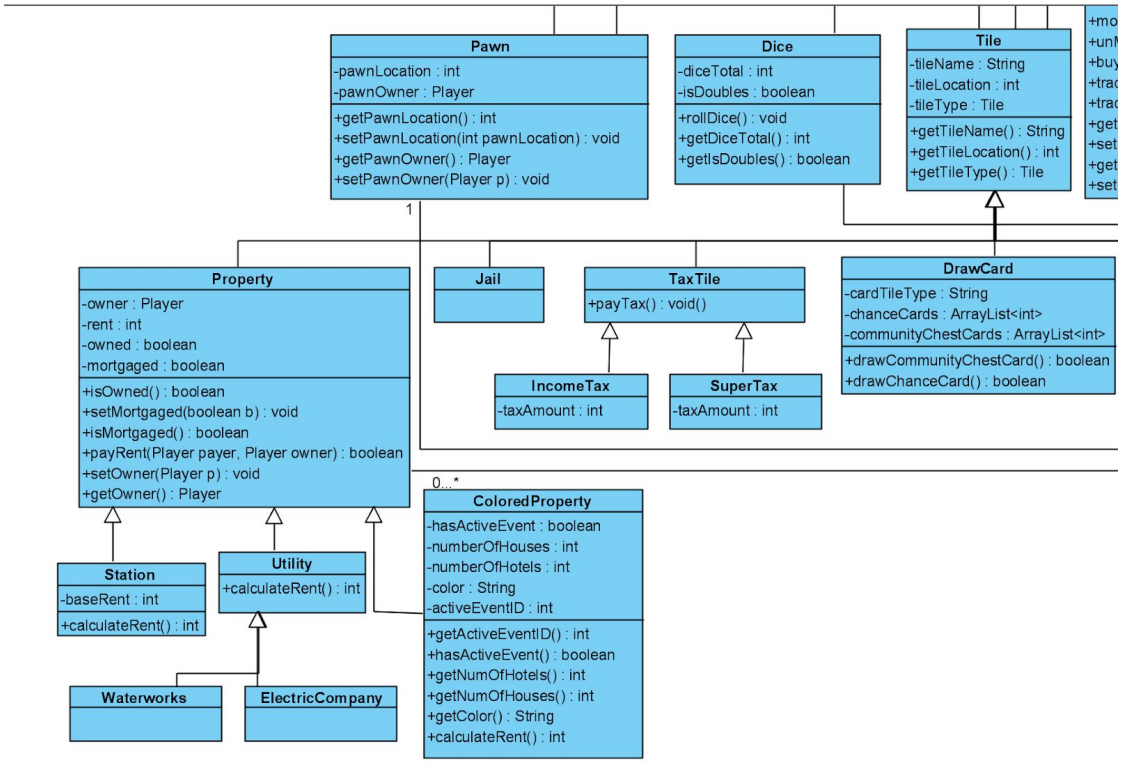

Figure 10: Final Class Diagram Zoomed-In Part 4

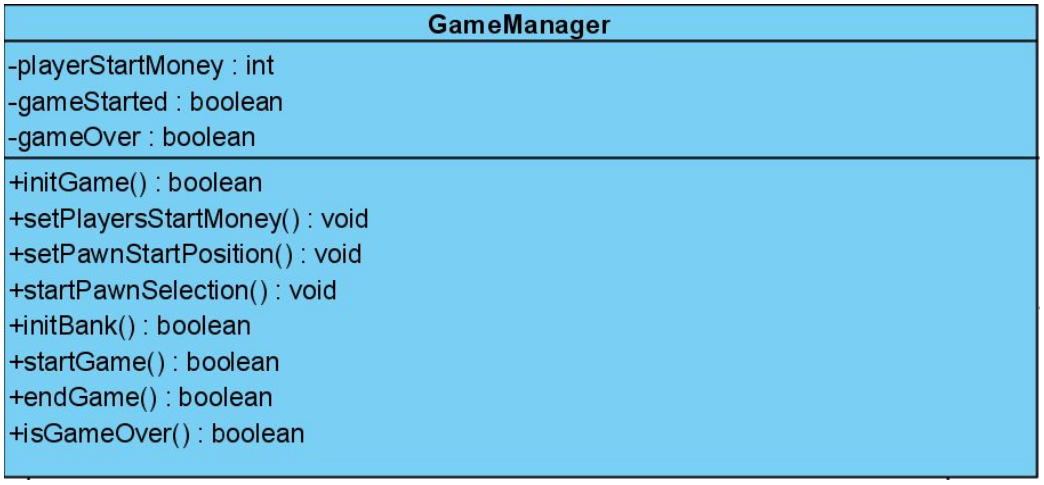

Figure 11: Subsystem decomposition diagram

Attributes:

private int playerStartMoney: Predetermined amount of money which all players will start the game with.

public boolean gameStarted: Set to `true` after the game starts. Otherwise always `false`. If `false`, prevents players from using in game functionalities.

public boolean gameOver: Checked by isGameOver() function every turn to see if the conditions for the game to be over are met. Is always `false` if the game over

conditions aren't met. Set to `true` by isGameOver() function when the conditions are met.

Methods:

public boolean initGame(): Initializes the game by calling the methods setPlayersStartMoney(), initBank(), setPawnStartPosition(), startPawnSelection(), with the given order. Also calls the initBoard() method from board class. Returns `true` if the game is successfully initialized.

public void setPlayersStartMoney(): Sets the start money for all Player objects to playerStartMoney.

public void setPawnStartPosition(): Initializes the start positions of the pawns.

public void startPawnSelection(): Starts the pawn selection phase of the game. Uses the pawn class's functions to set the owners of the pawns. The function works in a way that the first player to pick a pawn secures it.

public void initBank(): Initializes the numberOfHouses and numberOfHotels the bank owns. Sets all the properties owned attribute to `false`.

public boolean startGame(): After the initGame() method returns true, starts the game. Returns `true` if the game is started successfully.

public boolean endGame(): Calls the isGameOver() method every turn to check if it returns `true`. If it does, ends the game, calls displayScores() method of scoreboard class and announces the winner.

public boolean isGameOver(): Checks if the game is over at the end of every turn according to predefined game over conditions. Sets gameOver attribute to `true` and returns `true` if the conditions are met.

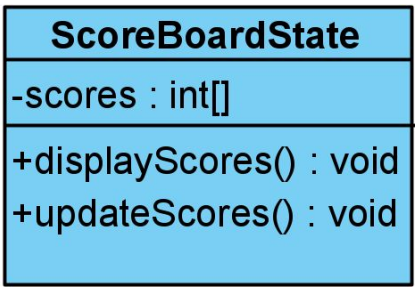

Figure 12: ScoreBoardState Class

Attributes:

private int[ ] scores: Keeps the scores of the players.

Methods:

public void displayScores(): Display the scores on the screen.

public void updateScores(): Updates the scores array.

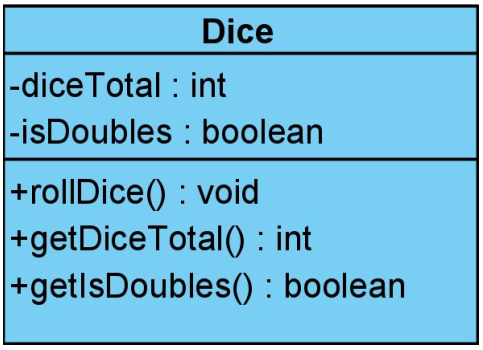

Figure 13: Dice Class

Attributes:

private int diceTotal: Stores the total of the current dice faces.

private boolean isDoubles: Stores if both dice faces have the same value or not in boolean form.

Methods:

public void rollDice(): Gets two random values between 1-6. Adds the two values and sets the diceTotal to the sum.

public int getDiceTotal(): Returns the diceTotal attribute.

public boolean getIsDoubles(): Returns the isDoubles attribute.

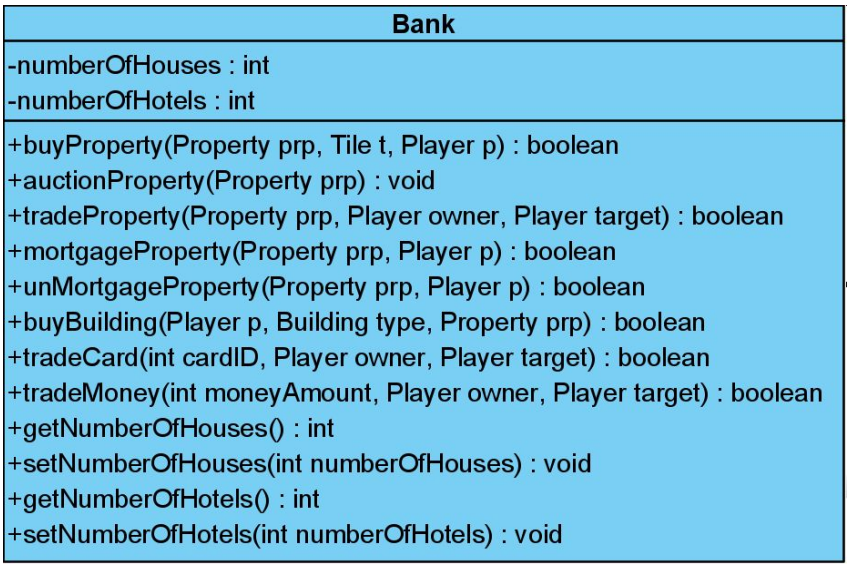

Figure 14: Bank Class

Attributes:

private int numberOfHouses: Current number of houses the bank owns.

private int numberOfHotels: Current number of hotels the bank owns.

Methods:

public boolean buyProperty(Property prp, Tile t, Player p): This method initiates the buying operation for the given property. It takes a Property object and a Player object and checks if the tile player is on matches the location of the property. Returns `false` if they don't match. It also returns `false` if the player doesn't have sufficient money or if the property is already owned and aborts the buying operation. If all the conditions for the sale are met, sets the owner of the property using the setOwner() method of the property class and updates the balance of the player using the setBalance() method of the player class. Returns `true` after the buying operation is done.

public void auctionProperty(Property prp): This method initiates the auction for the given property. It's called when a player lands on an ownerless property and doesn't want to buy it.

public boolean tradeProperty(Property prp, Player owner, Player target): This method takes two Player objects, one being the owner of the given property and one being the target Player who will receive the property. Receiving player can choose to reply with tradeProperty(), tradeCard() or tradeMoney() methods. Returns `true` if the trade is accepted and successfully executed. Returns `false` if not.

public boolean tradeCard(int cardID, Player owner, Player target): This method takes two Player objects, one being the owner of the given card and one being the

target Player who will receive the card. Receiving player can choose to reply with tradeProperty(), tradeCard() or tradeMoney() methods. Returns `true` if the trade is accepted and successfully executed. Returns `false` if not.

public boolean tradeMoney(int moneyAmount, Player owner, Player target): This method takes two Player objects, one being the owner of the money and one being the target Player who will receive the money. Receiving player can choose to reply with tradeProperty(), tradeCard() or tradeMoney() methods. Returns `true` if the trade is accepted and successfully executed. Returns `false` if not.

public boolean mortgageProperty(Property prp, Player p): This method takes the mortgageValue of the given property using getMortgageValue() of the associated TitleDeedCard class. Returns `false` if the property is already mortgaged or numberOfHouses > 0 or numberOfHotels > 0. If not, it uses setBalance(mortgageValue) method of Player class to add the mortgage money to Player's balance. It sets mortgaged attribute of Property class to `true` using setMortgaged() method. Returns `true` after the mortgage operation is done.

public boolean unMortgageProperty(Property prp, Player p): This method takes the mortgageValue of the given property using getMortgageValue() of the associated TitleDeedCard class. Returns `false` if the property is not mortgaged. If it is mortgaged, uses the setBalance(-1.1*(mortgageValue)) to take the money from the player. It sets mortgaged attribute of Property class to `false` using setMortgaged() method. Returns `true` after the unmortgage operation is done.

public boolean buyBuilding(Player p, Building type, Property prp): This method takes a Player object, a Building object to clarify the type of building, and a Property object. Decrements the numberOfHotels or numberOfHouses from the Bank class depending on the selected Building type and increments the numberOfHotels or numberOfHouses from the ColoredProperty class. Uses setBalance() function to take money from the player. Returns `true` if successful. Returns `false` if for any reason the buying operation can't occur, in cases like maximum number of houses/hotels is reached or bank doesn't have any houses/hotels left.

public int getNumberOfHouses(): Returns the numberOfHouses attribute of Bank class.

public void setNumberOfHouses(int numberOfHouses): Sets the numberOfHouses attribute of Bank class.

public int getNumberOfHotels(): Returns the numberOfHotels attribute of Bank class.

public void setNumberOfHotels(int numberOfHotels): Sets the numberOfHotels attribute of Bank class.

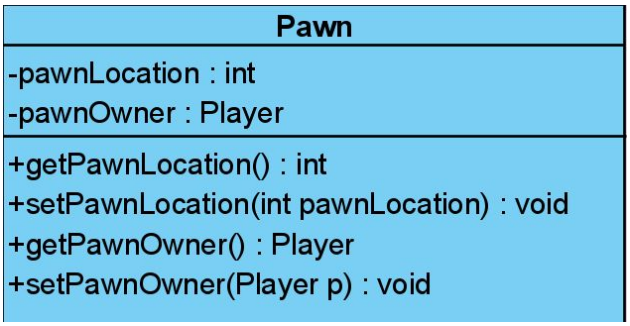

Figure 15: Pawn Class

Attributes:

private int pawnLocation: Stores the location of the pawn on the board as an integer.

private Player pawnOwner: Stores the owner of the pawn as a Player object.

Methods:

public int getPawnLocation(): Returns the pawnLocation attribute of Pawn class.

public void setPawnLocation(int PawnLocation): Sets the pawnLocation attribute. Used for updating the location of Pawn objects after dice rolls.

public Player getPawnOwner(): Returns the pawnOwner attribute of Pawn class.

public void setPawnOwner(Player p): Sets the pawnOwner attribute of Pawn class as a Player object. Used during the pawn selection phase of the game.

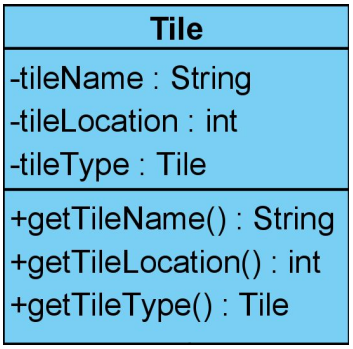

Figure 16:Tile Class

Attributes:

private String tileName: Title of the tile stored as a String.

private int tileLocation: Locations of tiles are numbered from 1-40. tileLocation holds the number of the given tile as an integer.

private Tile tileType: Type of the Tile object can be Property, Jail, TaxTile, DrawCard, GoToJail, GO, FreeParking.

Methods:

public String getTileName(): Returns the tileName attribute of Tile class.

public int getTileLocation(): Returns the tileLocation attribute of Tile class.

public int getTileType(): Returns the tileType attribute of Tile class.

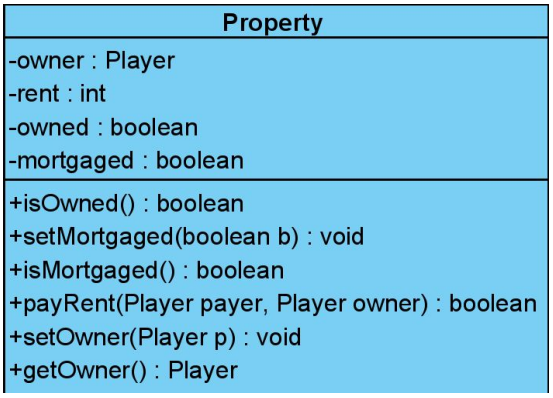

Figure 17: Property Class

Attributes:

private Player owner: The owner of the property stored as a Player object.

private int rent: Rent value of the property stored as an integer. It's used in payRent() function.

private boolean owned: Boolean value indicating if the property is owned or not.

private boolean mortgaged: Boolean value indicating if the property is mortgaged or not.

Methods:

public boolean isOwned(): Returns the owned value of Property class.

public void setMortgaged(boolean b): Sets the mortgaged value to the boolean value input.

public boolean isMortgaged(): Returns the mortgaged value of Property class.

public boolean payRent(Player payer, Player owner): Calls the calculateRent() method from its child classes RailRoad, Utility, ColoredProperty. Calls the setBalance() methods for payer and owner with the calculated rent. Returns `true` if

the transaction is made successfully. Returns `false` without making the transaction in cases like bankruptcy.

public void setOwner(Player p): Sets the owner attribute of Property class.

public Player getOwner(): Returns the owner attribute of Property class.

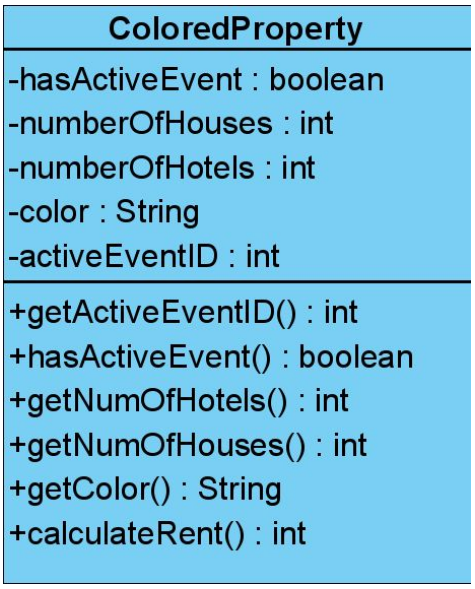

Figure 18: ColoredProperty Class

Attributes:

private boolean hasActiveEvent: Boolean variable that indicates if there is an active event present on the ColoredProperty or not.

private int numberOfHouses: Number of houses present on a colored property.

private int: numberOfHotels: Number of hotels present on a colored property.

private String color: A string that indicates the color group of the property.

private int activeEventID: The integer value that indicates which event is active on the property. If there isn't any it's -1 by default.

Methods:

public int getActiveEventID(): Returns the activeEventID attribute.

public boolean hasActiveEvent(): Returns the hasActiveEvent attribute.

public int getNumOfHotels(): Returns the numberOfHotels attribute.

public int getNumOfHouses(): Returns the numberOfHouses attribute.

public String getColor(): Returns the color attribute.

public int calculateRent(): Takes the rent value from TitleDeedCard class according to the numberOfHotels or numberOfHouses attribute of the ColoredProperty class and returns it.

Figure 19: Utility Superclass and its subclasses ElectricCompany and Waterworks Classes

Methods:

public int calculateRent(): Calculates the rent according to the current dice roll of the landing player and how many utilities the owner has. Dice roll is taken from getDiceTotal() method of Dice class. Then hasBothUtilities() method of Player class is executed. If it returns `true`, rent is returned as getDiceTotal()*10, if it returns `false`, rent is returned as getDiceTotal()*4.

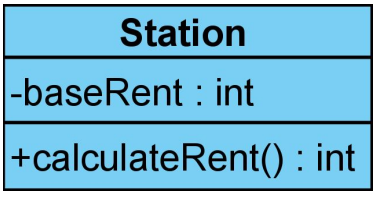

Figure 20: Station Class

Attributes:

private int baseRent: Predetermined base rent value of the station. Used in calculateRent() method.

Methods:

public int calculateRent(): Calculates the rent according to the number of stations the owner has and the base rent of the station. Number of stations is taken from getNumberOfStations() method of Player class. Then it's multiplied with base rent and the result is returned.

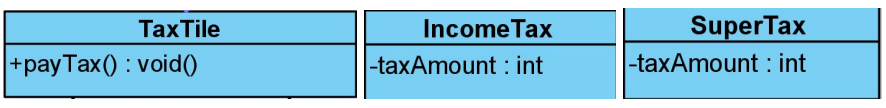

Figure 21: TaxTile Superclass and its subclasses IncomeTax and SuperTax Classes

Attributes:

private int taxAmount: Predetermined tax value for the TaxTile. SuperTax and IncomeTax classes have different values for taxAmount.

Methods:

public void payTax(): Uses setBalance() method of Player class to update the balance of the player that landed on the TaxTile by reducing taxAmount. setBalance (-taxAmount).

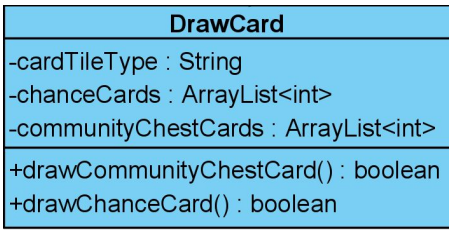

Figure22: DrawCard Class

Attributes:

private DrawCard cardTileType: This attribute can be either “DrawChanceCard” or “DrawCommunityChestCard” and it determines which method will be executed.

private ArrayList<int> chanceCards: This ArrayList holds the ID’s of the chance cards.

private ArrayList<int> communityChestCards: This ArrayList holds the ID’s of the chance cards.

Methods:

public void drawCommunityChestCard(): Takes the last element of the communityChestCards ArrayList and removes it. Calls the addCard() method from Player class with the cardID as the input.

public void drawChanceCard(): Takes the last element of the chanceCards ArrayList and removes it. Calls the addCard() method from Player class with the cardID as the input.

Figure 23: GoToJail Class

Methods:

public void jailPlayer(Player p): This method takes Player object as an input and uses setJail() method of the Player class with `true` as the parameter.

Figure 24: Go Class

Methods:

public void jailPlayer(Player p): This method takes Player object as an input and uses setBalance() method of the Player class with the predetermined salary amount.

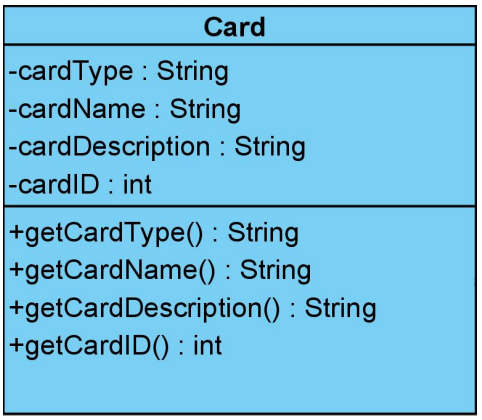

Figure 25: Card Class

Attributes:

private String cardType: Stores the card type as a String. It can be “TitleDeedCard”, “ChanceCard” and “CommunityChestCard”

private String cardName: Stores the title of the card as a String.

private String cardDescription: Stores the description of the card as a String.

private int cardID: Stores the unique card ID as an integer.

Methods:

public String getCardType(): Returns the cardType as a String.

public String getCardName(): Returns the cardName as a String.

public String getCardDescription(): Returns the cardDescription as a String.

public int getCardID(): Returns the cardID as an integer.

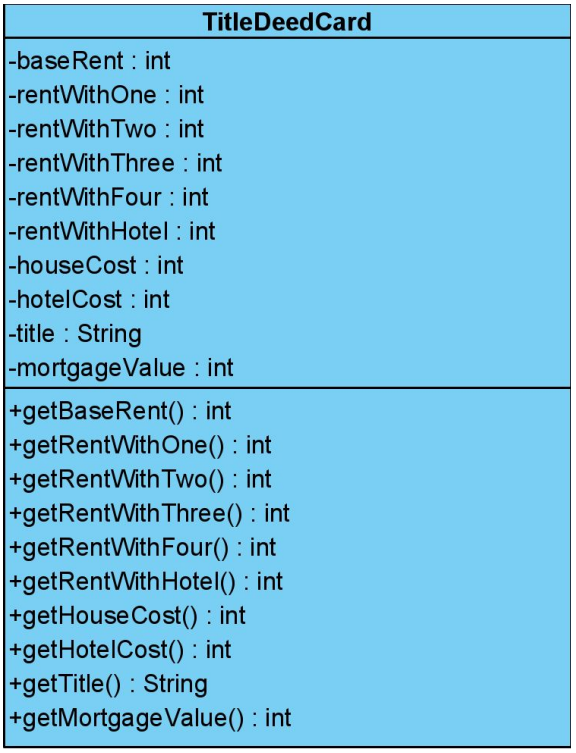

Figure 26: TitleDeedCard Class

Attributes:

private int baseRent: Base rent of a property without any buildings on it.

private int rentWithOne, rentWithTwo, rentWithThree, rentWithFour, rentWithHotel: Rent of a property with 1-4 houses or a hotel.

private int houseCost: Cost of buying a house for a property.

private int hotelCost: Cost of buying a hotel for a property.

private String title: Title of a title deed card as a String.

private int mortgageValue: The amount of money the bank pays for the mortgage of the property.

Methods:

public int getBaseRent(): Returns baseRent attribute.

public int getRentWithOne(), getRentWithTwo(), getRentWithThree(), getRentWithFour(), getRentWithHotel(): Returns one of the associated attributes rentWithOne, rentWithTwo, rentWithThree, rentWithFour, rentWithHotel.

public int getHouseCost(): Returns houseCost attribute.

public int getHotelCost(): Returns hotelCost attribute.

public String getTitle(): Returns title attribute.

public int getMortgageValue(): Returns mortgageValue attribute.

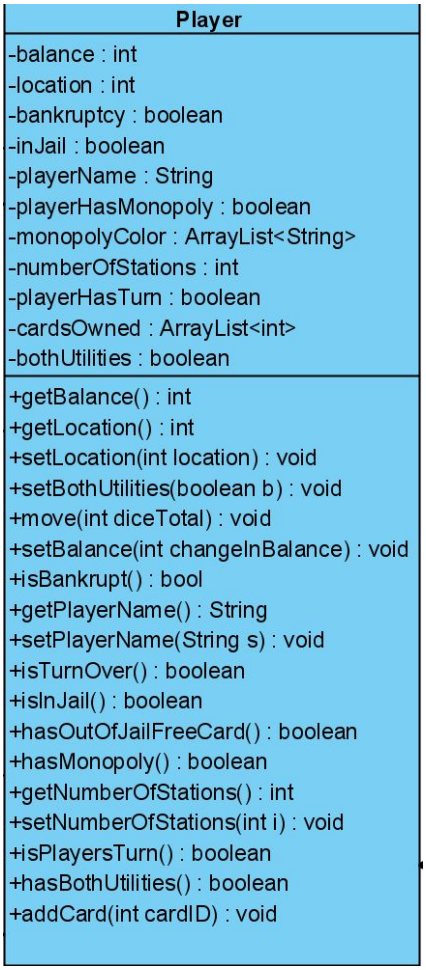

Figure 27: Player Class

Attributes:

private int balance: Current money balance of the player.

private int location: Current location of a player as integers from 1 to 40.

private boolean bankruptcy: Current status of Player's bankruptcy as a boolean variable.

private boolean inJail: Current status of Player's jail situation as a boolean variable.

private String playerName: Name of the player stored as a String.

private boolean playerHasMonopoly: Current status of Player's monopoly situation as a boolean variable.

private ArrayList<String> monopolyColor: An array list of Strings holding the colors of monopolies the Player has.

private int numberOfStations: Number of stations player owns. Used for calculating the rent of stations.

private boolean playerHasTurn: Current status of Player's turn stored as a boolean variable.

private ArrayList<int> cardsOwned: ID's of the cards Player owns. Can be TitleDeedCard, ChanceCard, CommunityChestCard.

private boolean bothUtilities: Shows if player owns both utilities on the board or not. Used for calculating the rent of a utility.

private boolean outOfJailCard: Shows if player has outOfJailCard or not.

Methods:

public int getBalance(): Returns balance attribute.

public int getLocation(): Returns location attribute.

public void setLocation(int location): Sets the location attribute.

public void setBothUtilities(boolean b): Sets the bothUtilities attribute to boolean b.

public void move(int diceTotal): Adds diceTotal to the location attribute. Then sets the location attribute to the remainder of its division by 40 (Since there are 40 total tiles).

public void setBalance(int changeInBalance): Adds changeInBalance to balance. If result is smaller than 0, sets bankruptcy attribute to `true`.

public boolean isBankrupt(): Returns bankruptcy attribute.

public String getPlayerName(): Returns playerName attribute.

public void setPlayerName(String s): Sets the playerName attribute to string s. Used at the beginning of the game to set the player's name.

public boolean isInJail(): Returns inJail attribute.

public boolean hasOutOfJailFreeCard(): Returns outOfJailCard attribute.

public boolean hasMonopoly(): Returns playerHasMonopoly attribute.

public int getNumberOfStations(): Returns numberOfStations attribute.

public void setNumberOfStations(int i): Adds int i to the numberOfStations attribute. int i can be +1 or -1.

public boolean isPlayersTurn(): Returns playerHasTurn attribute.

public boolean hasBothUtilities(): Returns bothUtilities attribute.

public void addCard(int cardID): Adds the int cardID to cardsOwned ArrayList.