CS-319 TERM PROJECT

Section 2

Group 2A

Monopoly

Design Report

Project Group Members:

1- Can Kırımca

2- Burak Yiğit Uslu

3- Cemre Biltekin

4- Can Kırşallıoba

5- Mustafa Yaşar

6- Emre Açıkgöz

Supervisor: Eray Tüzün

Table Of Contents

1. Introduction

1.1 Purpose Of The System

Monopoly that will be designed is a digital 2-D real-estate board game which aims to make the players have an enjoyable time alone or with friends by creating a competitive and entertaining environment. Different from the original game, this digital version of the game provides customization options for the players to personalize their game to maximize their enjoyment. The player's goal is to be the richest person in the game.

1.2 Design Goals

Following subsections show the design goals that are being considered in the making of the Monopoly game in the light of nonfunctional requirements.

1.2.1 Top Design Goals

Performance Criteria

Response Time

The system is a turn-based game. Consequently, the duration of a single game depends on the duration of the individual turns. Since the player waits for other players' turn when his/her turn passes, it is of importance that the period of turns to be minimized by technical adjustments like minimizing the response time, so that the turn's length only depends on the player itself, but not affected by the delays produced by the system. Therefore, the system is designed to have a maximum response time of less than 2 seconds. This ensures to keep the players' attention in the game, and let them perform quicker operations for a smooth play.

End User Criteria

Usability

Monopoly has over ten detailed rules which makes it hard to play the game intuitively, so the players should have an idea about the concepts of the game and how it is played. Since it is undesirable for the player to check the rules, which is a long body of text, frequently while playing, the user interface is designed to aid the players to minimize their need to refer to the manual. The board layout, texts, cards, icons are chosen to appeal to familiarity (with the original board game and real life events/objects) in such a way that the players will not have difficulty while reading and understanding them. Additionally, there will be a “How to Play” screen that contains the detailed rules of the game which can be accessed any time.

Design Criteria

Robustness

Monopoly is a game that ends after playing for a considerably long time. Some games could last more than one hour, and as a result, a bug from invalid input that crashes the program has the potential to cause lost data which might make the players go for a new game from scratch. Therefore, bugs that can disrupt the game or even crash the program must not occur. It is undesired for the efforts of the players to be wasted. In order to achieve this, the system is designed to restrict the invalid user inputs via user interface (ex. disabling trade button when no items subject to trade are selected), and include more concrete exception handling mechanisms for error-prone scenarios. This allows the survival of the data of the game, and ensures an uninterrupted game experience.

1.2.2 Further Design Goals

Design Criteria

Reliability

The game is designed to behave in an expected logical way (according to the game rules and desired actions) to ensure the continuity of the game. The unintended system behaviors are minimized by making the system 99% reliable since a single unintended behavior might disrupt the flow of the game and the player might not tolerate this behavior and exit the game.

Maintenance Criteria

Extensibility

The game is implemented by following the object oriented software engineering concepts. Since these concepts allow us to design the software in a reasonably systematic way, they also will provide us with the ease to change the system after deployment like adding new functionalities according to user feedback. The system is composed of hierarchical subsystems, therefore, whenever a change is made for a feature, the change will affect only a portion of the system which can be a class addition or a modification of a class.

2. System Architecture

2.1 Subsystem Architecture

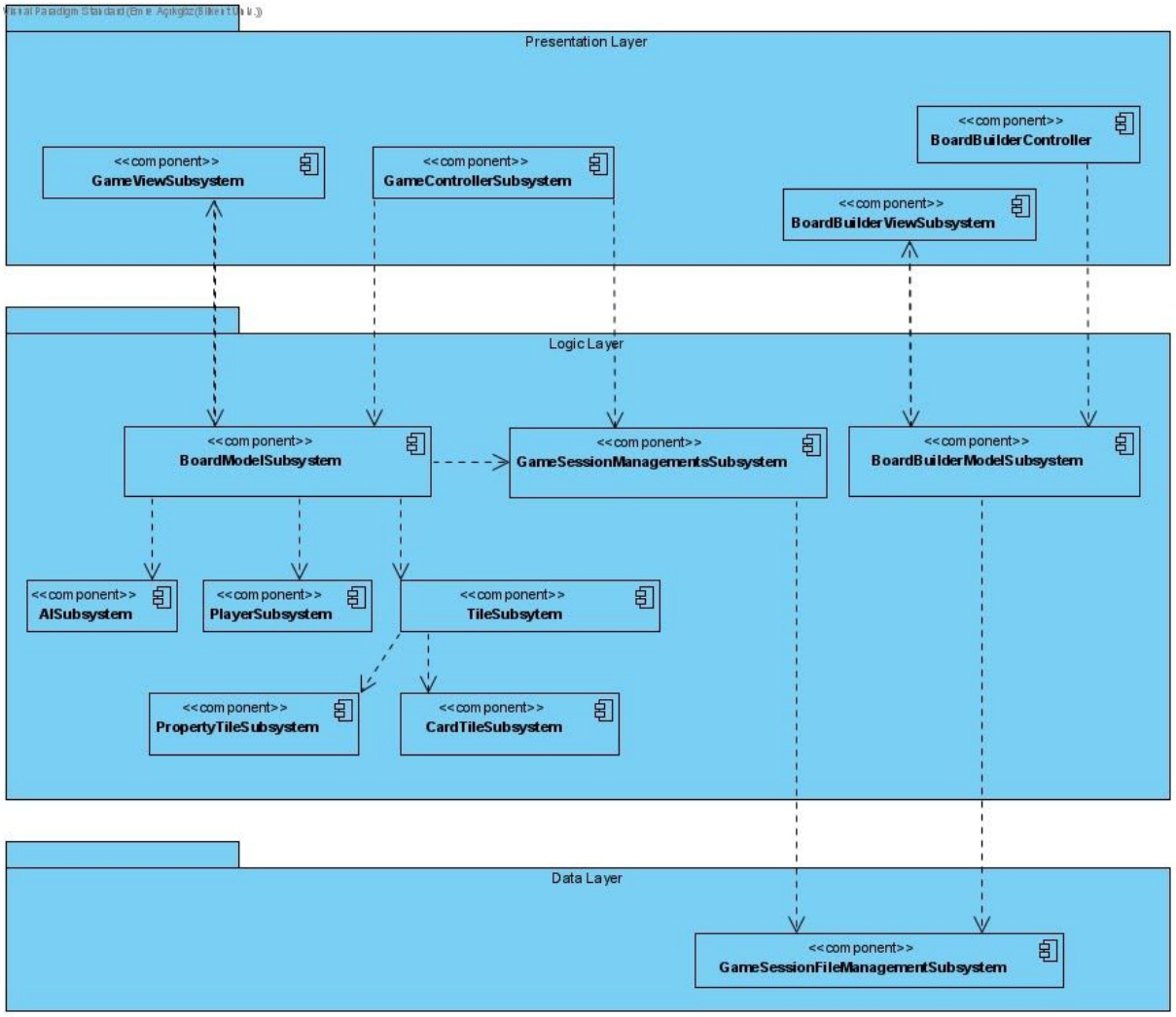

Fig. 1 *Subsystem Decomposition*

In order to decompose our system, we have divided the system into 3 layers (Presentation Layer, Logic Layer, Data Layer) and many subsystems. Our main purpose in decomposition is to minimize the coupling between the subsystems and to maximize the coherence of the components. Decomposition enables us to modify the existent features of the game and add extra features into the game.

For the system architecture, we have decided to combine **MVC (Model View Controller)** system design pattern and **3-tier architecture** as can be seen in Figure 1. This combined design approach is a suitable approach for the system because we have components suitable for their principles as follows:

- GameViewSubsystem, BoardBuilderViewSubsystem are our **Views,** which include the user interface related classes. These include anything the user can see graphically and contains useful information to the end user. They exist in the **Presentation Layer,** which is the front-end layer, in accordance with 3-tier principles.
- GameControllerSubsystem, BoardBuilderController are our **Controllers**, which accepts inputs (ex. the game modes) and updates the View and Model components for the system to reflect the inputs and changes. They exist in the **Presentation Layer,** since these components provide communication to other subsystems/layers.
- The remaining subsystems are our **Models,** since these subsystems include objects (ex. TileSubsystem includes Tile objects in the game) and data/files (ex. GameSessionFileManagementSubsystem includes save files and configuration files). They exist in **Logic Layer** and **Data Layer** because Logic Layer contains functional capabilities and Data Layer provides data access. Since data access is handled by classes in GameSessionFileManagementSubsystem, only it exists in the Data Layer.

The main reason for usage of MVC (especially in the presentation layer) is to avoid having to change other components in order to change a single component,

and the main reason for usage of 3-tier with it is to separate business logic from the data. Therefore, the whole of the system is organized.

2.1.1 Presentation Layer

GameViewSubsystem: This subsystem creates the user interface for the gameplay. The gameplay user interface consists of the pause menu, trade menu, auction menu, the game screen itself. **Observer pattern** will be used as a part of the **MVC Design Pattern.**

GameControllerSubsystem: Game Controller Subsystem will handle the communications between the logic classes of the game and the graphical user interface. Again **Observer Design Pattern** along with **MVC Design Pattern** will be used.

BoardBuilderViewSubsystem: This subsystem creates the interface for building a custom board. The user interacts with this subsystem by customizing the properties and changing the Go Tile income. **Observer Pattern** will be used with **MVC Design Pattern.**

BoardBuilderController: This subsystem will handle the communications between the logic classes of the map builder subsystem and the map builder graphical user interface. **Observer Pattern** will be used with **MVC Design Pattern.**

2.1.2 Logic Layer

BoardModelSubsystem: Board, BoardFactory, AuctionModel, TradeModel, TurnManager are the classes in the BoardModelSubsystem. BoardModelSubsystem is responsible for the creation of general logic of the board like creating the board and making the ground for iterating on the

board. In Board class we have used the **Iterator Pattern** to easily iterate over the board tiles**.** In general, in this subsystem, we used **Abstract Factory Design Pattern** and **Builder Design Pattern.** Since a board is created from a configuration file and a board is a complicated structure, there is a need for a builder design pattern.

GameSessionManagementSubsystem: This subsystem includes the GameSessionManager, GameSession, GameSessionBuilder which contain and manage information about a single game session. In this subsystem, we have used **Builder Design Pattern**, so that the client code is not affected by the complexity of the GameSession class.

BoardBuilderModelSubsystem: BoardBuilder is the class of the BoardBuilderModelSubsystem. BoardBuilderModelSubsystem handles the logical operations on the board such as changing the names of the tiles, the value of the rent, and the property value.

TileSubsystem: Tile (Abstract), TeleportTile, GoTile, GoToJailTile, JailTile, FreeParking Tile, IncomeTaxTile, TeleportTileActionStrategy, GoTileActionStrategy, GoToJailTileActionStrategy, JailTileActionStrategy, FreeParkingTileActionStrategy, IncomeTaxTileActionStrategy, TileFactory, AbstractTileFactory, AbstractGameActionFactory, GameActionFactory, ActionStrategy, GameAction are classes of the TileSubsystem. This subsystem is responsible for the operations that will be performed when the player lands on a particular tile (except Card and Property tiles). These

operations include transporting to another tile, going to jail, paying tax etc. In this subsystem, we used **Command Design Pattern, Strategy Design Pattern, Abstract Factory Design Pattern** and **Builder Design Pattern.** Command pattern is used because we wanted to reduce the coupling between the logic layer and the presentation layer as much as possible. Strategy pattern is used because there are different tiles that have different responsibilities in different times. We solve this problem with a strategy pattern to assign the functionality of a tile during runtime. We have a very complex subsystem due to these two patterns. Therefore, we utilize builder and abstract factory patterns.

PropertyTileSubsystem: PropertyTile, PropertyTileActionStrategy, PropertyActionStrategy, ColorGroup, Color, and TitleDeedCard are the classes in PropertyTileSubsystem. Observable and Representable are the interfaces in PropertyTileSubsystem. This subsystem is responsible for the actions taken for the general property operations such as, buying or selling that property. These actions will be taken from the PropertyTileAction Strategy. Different operations that can be done with the property tiles are represented as the different strategies. In this subsystem, we used **Command Design Pattern, Strategy Design Pattern, Abstract Factory Design Pattern** and **Builder Design Pattern.** The reasons we use them are the same with that of the TileSubsystem.

PlayerSubsystem: PlayerSubsystem includes the interface Player, abstract classes AbstractPlayer and AbstractPlayerFactory, and the classes

HumanPlayer, PlayerFactory, PlayerToken, Dice which are central classes in the logic layer. This subsystem is mainly responsible for the representation and management of players playing the game and their tokens on the board.

We use the **Abstract Factory Design Pattern** for the mentioned purposes. AbstractPlayerFactory implements the Abstract Factory Design Pattern because it creates a single Player object. Players in the Monopoly game are initialized with this pattern; the system makes use of AbstractPlayerFactory to create Players.

AISubsystem: All AI operations regarding decision making are a part of this subsystem. AIPlayer, AIStrategy, StingyAIStrategy, AdventurousAIStrategy, BalancedAIStrategy, GameStatics classes. In this subsystem, we have used **Strategy Design Pattern** to implement AI characteristics since each AI characteristic enforces a different behaviour on the AI player, and this design pattern enables us to extend and introduce new AI characteristics in future versions of the game by simply adding a new strategy class.

CardTileSubsystem: CardTile, CardTileActionStrategy are the classes in CardTileSubsystem. Observable and Representable are the interfaces in CardTileSubsystem. This subsystem is responsible for the actions when a player lands on a Chance or Community Chest card tile. These actions include drawing a card and performing the consequence of this card such as paying a fee or going to jail. Each of these actions are represented as different strategies. In this subsystem, we used **Command Design Pattern,**

Strategy Design Pattern, Abstract Factory Design Pattern and **Builder Design Pattern.**

2.1.3 Data Layer

GameSessionFileManagementSubsystem: SerializationHandler, BoardConfiguration, FileManager and ConfigHandler are the classes belonging to GameSessionFileManagementSubystem. This subsystem is responsible for saving and loading the games and saving and loading the user created board configurations when initiated, which includes interacting with text and JSON files on local storage. It is the part of the system that interacts with the storage “hardware node” according to **3-Tier Architecture.**

2.2. Hardware/Software Mapping

In order to play the game of Monopoly, the user needs a computer, a mouse, and a keyboard. The game will be implemented by using Java programming language, the minimum Java Version to run the game is 8. The game will run on Java Runtime Environment. So, the user must install Java to his or her computer in order to run the game.

2.3 Persistent Data Management

For the initial version of the project, we are not going to use any sort of database. In order to store game data, we are going to use a file system. We thought that this approach would be more suitable because there will be only one writer. When the user saves a game, the save data is going to be stored

in text files inside game folders for further access. The configuration of the default and custom game boards are going to be stored in binary and JSON files to be accessed while creating or loading a game. The classes that will be serialized are GameSession, Board, Tile, GameAction, TitleDeedCard, Card, CardDeck, PlayerToken, GameStatistics, AIStrategy, TurnManager, BoardConfiguration, Player and Dice. Moreover, we are going to edit the configuration files according to the users' preferred settings such as sound and music level.

2.4 Access Control and Security

The game does not require any sort of installation other than simply unzipping the game folder and opening the .jar file. We are not going to implement any safety measures, because we are not taking any sensitive user data. There will not be any account-based system. Any user who starts the game can access all features and saved game data (that is previously saved). The game will not need access to internet connection.

2.5 Boundary Conditions

For the sake of portability, the game will be able for download in .jar format. The users can carry the game in a USB flash drive so that they can play the game on different computers.

If there is any sort of error whilst playing the game and the game shuts down. Due to this error, the data will be most likely saved due to our auto-save feature which is done automatically at the end of a turn.

The game will exit when the user clicks on the “Exit Game” button on the main menu. If they wish to exit mid-game, they can do so by pausing the game by clicking on the “Menu” button, our recommendation is to save the game before quitting. After saving the game, the user can click on “Return to Main Menu” and exit the game.

3. Low-Level Design

3.1 Object Design Trade-Offs

Space vs Speed

In Monopoly, there is no excessive amount of data to be stored such as a large number of arrays or matrices. Therefore, memory can be spared to speed up the system. For instance, the same variable will be stored in different memory locations so that the search for this variable is going to take less time, so, the most of the performance of the CPU will be focusing on the game logic.

Functionality vs Usability

In the game, there are many activities the player can initiate due to a large set of functionalities. The system is designed to include user interface (UI) elements (different pop up windows, different buttons) to allow the player to execute these activities in his/her turn without having to drown in an abundance of functionalities on one screen. Thus, some functionalities are hidden under a single UI element to favor user friendliness and ease.

Readability vs Robustness

In order to make the game robust, i.e, be able not to crash or exhibit undesired behavior in case of an invalid user input, the implementation needs more exception handling mechanisms which means more code. Since there is more code in the implementation, the readability of the implementation decreases.

3.2 Final Object Design

Fig. 2 *Final object design*

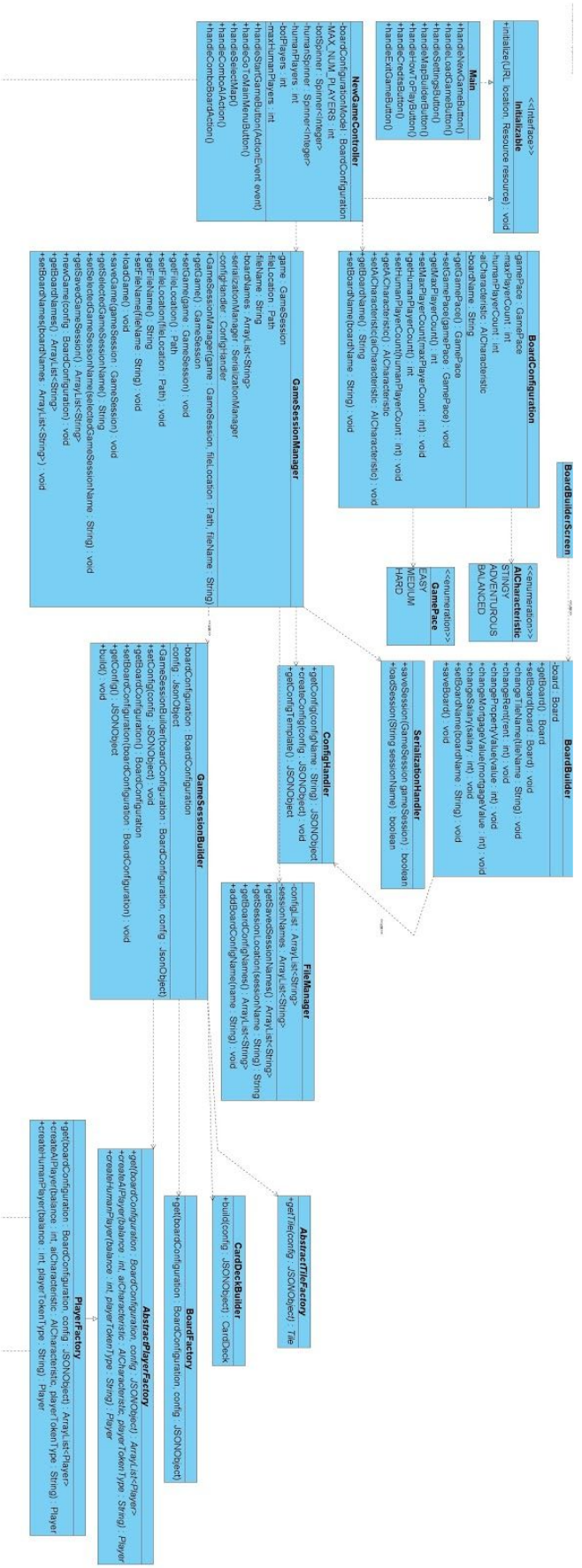

Fig. 3 *Abstraction of final object design*

Fig. 4 *Abstraction of final object design*

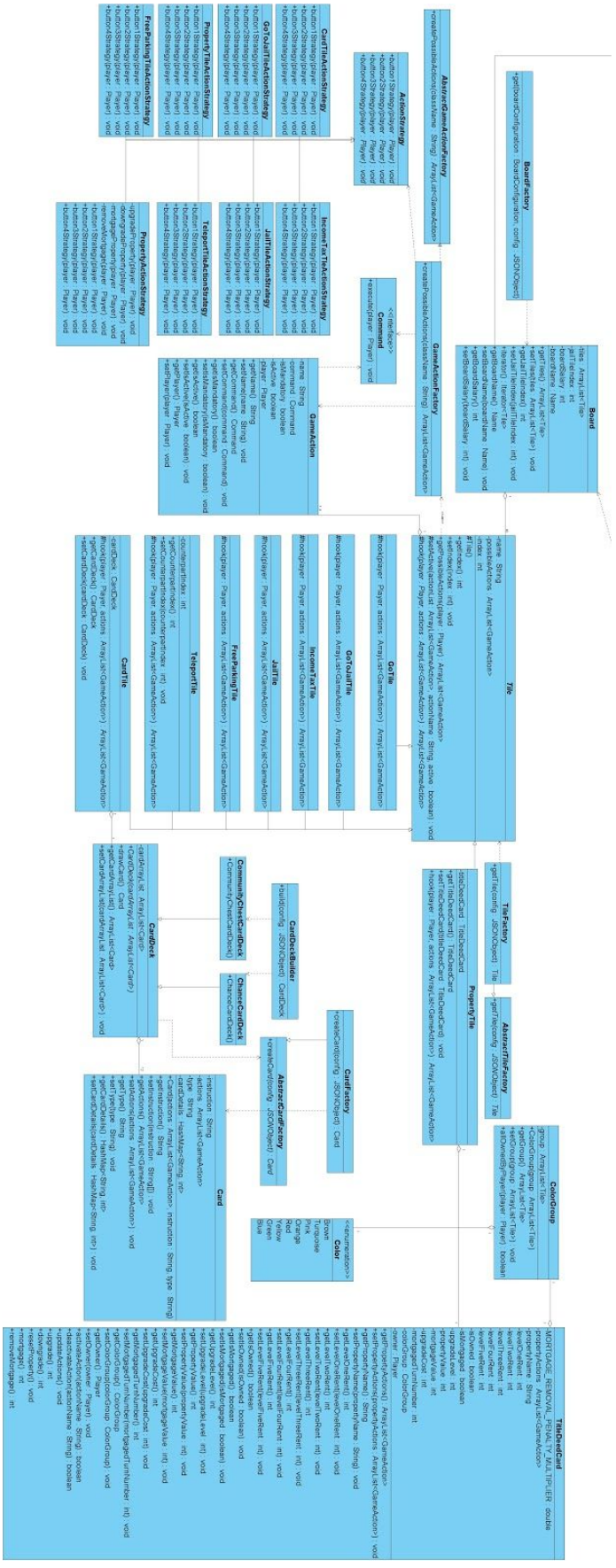

Fig. 5 *Abstraction of final object design*

3.3 Packages

3.3.1 src.controller Package

This package includes the controller classes of the MVC pattern.

3.3.2 src.data Package

This package has classes related to data and file management in the system.

3.3.3 src.model Package

Model has no classes for it but it creates a namespace.

3.3.4 src.view Package

The view package includes user interface related classes.

3.3.5 src.model.board Package

This package includes the creation and management of board objects and operations.

3.3.6 src.model.player Package

This package includes creation of human and AI player objects, and token of players and manages them. It contains information about AI characteristics, players' states, and the dice result.

3.3.7 src.model.session Package

This package includes classes related to management of a game session and a turn of the game.

3.3.8 src.model.tiles Package

This package includes different tile types classes like GoTile, JailTile etc. and manages tile objects.

3.3.9 src.model.tiles.actionStrategy Package

This package includes the game functionality classes implemented with the strategy pattern and the pattern.

3.3.10 src.model.tiles.card Package

This package includes the creation of cards (Community Chest and Chance) and card deck objects and manages them.

3.3.11 src.model.tiles.property Package

This package includes the objects related to property in the game like color groups and title deed cards, and contains information about them and manages them.

3.4 Class Interfaces

3.4.1 GameSessionManager Class

Explanation: This class contains the game session information and initiates saving and loading functionalities.

Constructor:

GameSessionManager(GameSession game, Path fileLocation, String fileName): Constructor creates a GameSessionManager object and initializes it with given parameters.

Attributes:

private GameSession game: This attribute contains the current game session information such as board information, players' information etc.

private Path fileLocation: This attribute contains the information of which the game session is going to be saved into.

private String fileName: This attribute contains the file name information of save files.

private ArrayList<String> boardNames: This attribute contains the name of the boards that are created.

private ConfigHandler configHandler: This attribute is responsible for reading the config file in the JSON format and then initializing the board's properties according to the config file.

private SerializationManager serializationManager: This attribute is an instance of SerializationManager which will be handling serialization operation for saving and loading games.

Methods:

public void loadGame(): This method loads the game session information from a file specified by fileName attribute in the path provided by fileLocation attribute. It calls SerializationHandler's load function to perform reading from the file (deserialization).

public void saveGame(): This method saves the game session information to a file specified by fileName attribute in the path provided by fileLocation attribute. It calls SerializationHandler's save function to perform writing on the file (serialization).

public String[] getSavedGameSessions(): This method returns the file names of the saved files as a String array. This method calls SerializationHandler's getSavedGames function to access those files.

public void newGame(BoardConfiguration config): This method creates a new save game with initial board configuration information upon starting a new game.

public GameSession getGame(): This method returns the GameSession attribute which holds the current game information.

public GameSession setGame(GameSession game): This method sets the current game with the game parameter.

public Path getFileLocation(): This method returns the file location as Path of the current game session.

public void setFileLocation(Path newPath): This method sets the file location of the current game session.

public void getFileName(): This method returns the name of the file of the current game session.

public void setFileName(String name): This method sets the name of the file of the current game session.

public ArrayList<String> getBoardNames(): This method returns boardNames attribute which contains every board name created.

public void setBoardNames(ArrayList<String> names): This method sets every board name created according to the parameter given.

3.4.2 Board Class

Explanation:

This class is a representation of the board of the Monopoly game.

Attributes:

ArrayList<Tile> tiles: This list holds the tiles that are present on the board.

int jailTileIndex: This attribute holds the index number of the jailTile. Changes according to config file.

int boardSalary: This attribute is the salary a receives when he/she passes the GoTile. Changes according to config file.

String boardName: Name of the initialized board. Changes according to config file.

Methods:

public int getBoardSalary(): Getter method for the board salary.

public void setBoardSalary(int boardSalary): Setter method for the board salary.

public String getBoardName(): Getter method for the board name.

public void setBoardName(String boardName): Setter method for the board name.

public ArrayList<Tile> getTiles(): This method returns the tile list.

public void setTiles(ArrayList<Tile> tiles): This method sets the tile list it takes from the arguments to the attribute of the class.

public int getJailTileIndex(): This method returns the jail tile index.

public void setJailTileIndex(int jailTileIndex): This method sets the jail tile index.

3.4.3 Player Interface

Explanation: This interface outlines the methods extending Player subsystem classes must implement. It is the collection of operations a player must be able to do in a Monopoly game.

Methods:

public void waitInJail(): This method allows the player to wait for 3 turns maximum when in jail.

public void checkBailOut(): This method checks if the player satisfies the conditions for bailing out of jail, if it does then the player bails out of jail.

public void addBailOutFromJailCard(Card bailOutFromJailCard): This method adds a bail out of jail card to the player, it takes a bailOutFromJailCard as argument.

public void removeBailOutFromJailCard(Card bailOutFromJailCard): This method removes the bailOutFromJailCard from the player's inventory.

public void moveToken(int moveAmount): This method moves the token on the board for the passed arguments amount.

public void goToJail(): This method makes the player go to jail.

public void addTitleDeedCard(TitleDeedCard newCard): This method adds a title deed card to the players inventory, it takes said title deed card as an argument to the method.

public void removeTitleDeedCard(TitleDeedCard cardToRemove): This method removes the title deed card form the players inventory, it takes said title deed card as an argument to the method.

public void changeBalance(int amount): This method changes the players balance for the specified amount in the argument of the method (For instance, -20).

public void declareBankruptcy(): This method makes the player declare bankruptcy.

public int getTaxOption(): This method returns the tax option the player chooses.

public void setTaxOption(): This method sets the tax option for the player.

public int getBalance(): This method returns the balance of the player.

public void setBalance(int balance): Sets the player's balance to the parameter balance.

public int getConsecutiveDoubleCount(): This method returns the current consecutive double count of the player.

public void setConsecutiveDoubleCount(int consecutiveDoubleCount): Sets the player's consecutiveDoubleCount to the parameter consecutiveDoubleCount.

public void startAuction(ArrayList<TitleDeedCard> titleDeeds): This method starts the auction with the properties it is passed from the arguments.

public Tile getCurrentTile(): This method returns the tile that the player is currently on.

public TitleDeedCard getSelectedTitleDeedCard(): This method returns the selected title deed card.

public PlayerToken getPlayerToken(): This method returns the player's token.

public void setPlayerToken(PlayerToken playerToken): This method sets the player token to a specific unique token.

public void setCurrentlyDrawnCard(Card card): This method sets the drawn card.

public void getCurrentlyDrawnCard(Card card): This method returns the drawn card.

public void setSelectedTitleDeedCard(TitleDeedCard titleDeedCard): This method sets the selected title deed card.

public int getGetOutOfJailChoice(): This method returns the option of bailing out of jail chosen by the player.

public void setGetOutOfJailChoice(int getOutOfJailChoice): This method sets the bailing out of jail choice, given the choice as an integer.

public int getTotalWorth(): This method calculates and returns the total worth of the player, i.e., addition of the money they have and the value of the properties they have.

public void setTotalWorth(int totalWorth): This method sets the total worth of the player.

public int getLiquidTotalWorth(): This method calculates and returns the liquid total worth of the player. Liquid total worth is the money that the player can have when they sell every property and building of themselves plus the money they have.

public void setLiquidTotalWorth(int liquidTotalWorth): This method sets the liquid total worth of the player.

public AuctionModel getAuctionModel(): Gets the AuctionModel attribute of implementing classes.

public void setAuctionModel(AuctionModel auctionModel): Sets the AuctionModel attribute of implementing classes to auctionModel parameter.

public TradeModel getTradeModel(): Gets the TradeModel attribute of implementing classes.

public void setTradeModel(TradeModel tradeModel): Sets the TradeModel attribute of implementing classes to tradeModel parameter.

public void startTrade(Player otherPlayer): This method starts a trade between the player object and the otherPlayer.

public boolean isBankrupt(): Getter method for bankrupt.

public void playTurn(): By calling other methods, rollDice, moveToken, does the actions a player should do in a turn. Calls other methods necessary for the updating the state of the player after playing the turn (For example, if a double dice is rolled, it

updates the consecutiveDoubleCount attribute of implementing player classes. If 3 consecutiveDoubleCount reaches 3, the player is sent to jail).

Dice rollDice(): Rolls the dice, and returns the result as a Dice object.

3.4.4 AbstractPlayer Abstract Class

Explanation: This abstract class implements the Player interface, and serves as a base for the extending HumanPlayer and AIPlayer classes. Outlines the attributes a player in a Monopoly game must have.

Attributes:

private TaxOption taxOption: This attribute represents the user selected option for the method of paying tax in the game.

private ArrayList<TitleDeedCard> titleDeedCards: This attribute is an arraylist of the title deed cards held by the player, representing the properties owned by the player.

private int balance: This attribute represents the balance of the player.

private ArrayList<Card> bailOutOfJailCards: This attribute represents the arraylist of bail out of jail cards the player holds.

private boolean bankrupt: This attribute represents the bankruptcy status of the player.

private int consecutiveDoubleCount: This attribute represents the count of double dice the player has thrown in one turn.

private PlayerToken playerToken: This attribute represents the players token.

private int numberOfTurnsSpentInJail: This attribute is the number of turns the player has spent in jail.

private boolean isInJail: This attribute is true if the player is in jail, false otherwise.

private int totalWorth: This method calculates and returns the total worth of the player, i.e., addition of the money they have and the value of the properties they have.

private int liquidTotalWorth: This attribute is the liquid total worth of the player. Liquid total worth is the money that the player can immediately obtain when they mortgage every property and sell all buildings (downgrade properties) they own plus the money they have.

private int getOutOfJailChoice: This attribute represents the bail out choice chosen by the player that is in jail.

private boolean canBailOut: This attribute is true if the player can successfully bail out of jail, false otherwise.

private AuctionModel auctionModel: To activate the auction, the player holds a reference to the auction model.

private TradeModel tradeModel: To activate the trade, the player holds a reference to the trade model.

private TitleDeedCard selectedTitleDeed: This attribute contains the information about the title deed card that player has selected.

private Card currentlyDrawnCard: This attribute is the card that the player has drawn from either of the card decks.

Methods:

public void waitInJail(): This method is called for a player that is in jail, who chooses to wait for one more turn.

public boolean checkBailOut(): This method returns true if the player has opportunity to bail out instead of waiting in his/her turn; false, otherwise.

public void goToJail(): This method instructs the player to go to jail (tile) and changes the player's status to in jail.

public void changeBalance(int amount): This method changes the balance of the player with calculation when the player loses or earns money by an amount.

public void declareBankruptcy(): This method is used to declare bankruptcy for the player, should the player go bankrupt or choose to retire from the game.

public int getTaxOption(): This method returns the tax option selected by the player, should the player land on income tax tile.

public void setTaxOption(TaxOption taxOption): This method sets the tax option that is chosen by the player.

public int getBalance(): This method returns the current balance of the player.

public void setBalance(int balance): This method sets the balance of the player to the given amount.

public int getConsecutiveDoubleCount(): This method returns the consecutive number of rolling doubles in one turn.

public void setConsecutiveDoubleCount(int consecutiveDoubleCount): This method sets the consecutive number of rolling a double in one turn.

public int getTotalWorth(): This method returns the total worth of the player.

public void setTotalWorth(int totalWorth): This method sets the total worth of the player.

public int getLiquidTotalWorth(): This method returns the liquid total worth of the player.

public void setLiquidTotalWorth(int liquidTotalWorth): This method sets the liquid total worth.

public void startAuction(ArrayList<TitleDeedCard> tittleDeeds): This method initiates an auction with the titleDeedCards it is given. This action can be initiated by a player when he or she is buying a property.

public AuctionModel getAuctionModel(): This method returns an Auction model.

public void setAuctionModel(AuctionModel auctionModel): This method sets the auction model it is passed through the argument list.

public TradeModel getTradeModel(): This method returns the trade model.

public void setTradeModel(TradeModel tradeModel): This method sets the trade model it is passed through the argument list.

public void startTrade(Player otherPlayer): This method starts a trade between the player and the other player.

public void addBailOutFromJailCard(Card card):

public void removeBailOutFromJailCard(Card card):

public void addTitleDeedCard(TitleDeedCard card): This method adds the title deed card to the title deed card list of the player.

public void removeTitleDeedCard(TitleDeedCard card): This method removes the title deed card given as parameter from the title deed card list of the player.

public TitleDeedCard getSelectedTitleDeed(): This method returns the title deed card that the player selected.

public void setSelectedTitleDeed(TitleDeedCard selectedTitleDeed):

public Card getCurrentlyDrawnCard(): This method returns the Card that the player drew currently.

public void setCurrentlyDrawnCard(Card card): This method sets the currently drawn card that the user pulled.

public boolean isBankrupt(): This method returns if the player is bankrupt or not.

3.4.5 AbstractPlayerFactory Abstract Class

Explanation: Abstract class for the PlayerFactory class. Contains the methods for creating/initializing a single AIPlayer or a single HumanPlayer.

Methods:

public abstract ArrayList<Player> get(BoardConfiguration boardConfiguration, ConfigAdapter configPlaceHolder): This method returns the list of the players that are playing the game.

public abstract Player createAIPlayer(int balance, AICharacteristic characteristic, String playerTokenType): Creates an AIPlayer, initializing the respective attributes of the AIPlayer with its parameters.

public abstract Player createHumanPlayer(int balance, String playerTokenType): Creates an HumanPlayer, initializing the respective attributes of the AIPlayer with its parameters.

3.4.6 HumanPlayer Class

Explanation: This class implements the Player interface and represents the human players of the Monopoly game.

Constructors:

public HumanPlayer(): Constructs the HumanPlayer object.

Methods:

public Dice rollDice(): This method allows the player to roll the dice.

3.4.7 AIPlayer Class

Explanation: This class implements the Player interface and represents the players powered with artificial intelligence, namely bots.

Attributes:

private AICharacteristic characteristic: This attribute holds the characteristic of the AI player.

Constructors:

public AIPlayer(int balance, String tokenType, AICharacteristic characteristic): Constructs the AIPlayer with a balance, tokenType and a characteristic.

Methods:

public Command simulate(): This method calculates a score of the probable actions the AI can take in its turn by going through conditional statements. It returns a Command object that holds the action that the AIPlayer is going to take in that round.

3.4.8 AIStrategy Class

Explanation: This class is the super class of the AI strategy classes.

Attributes:

private GameStatistics gameStatistics: This attribute holds the game statistics for the AI to take action on.

Methods:

public abstract playTurn(Player currentPlayer): This is an abstract method that enables the players to play their turns.

public Board getBoardStatistics(): Getter methods for boardStatistics attribute.

public void setBoardStatistics(Board boardStatistics): Setter method for the boardStatistics attribute.

public GameStatistics getGameStatistics(): Getter method for the game statistics attribute.

public void setGameStatistics(GameStatistics gameStatistics): Setter method for the game statistics attribute.

3.4.9 BalancedAIStrategy Class

Explanation: This class sets the behaviour of the balanced AI.

Methods:

public void playTurn(Player currentPlayer): Plays the turn for AIPlayer according to its characteristic (Balanced).

3.4.10 StingyAIStrategy Class

Explanation: This class sets the behaviour of the stingy AI.

Methods:

public void playTurn(Player currentPlayer): Plays the turn for AIPlayer according to its characteristic (Stingy).

3.4.11 AdventurousAIStrategy Class

Explanation: This class sets the behaviour of the adventurous AI.

Methods:

public void playTurn(Player currentPlayer): Plays the turn for AIPlayer according to its characteristic (Adventurous).

3.4.12 GameStatistics Class

Explanation: This class holds the information necessary for AI players to evaluate the board and base their play on.

Attributes:

private Board board: A reference to the game board.

private ArrayList<Player> players: Holds the players in the game.

Methods:

public int calculateAverageRent(): This method calculates the average rent of the properties.

public int getMaximumRent(): Getter method for the maximumRent attribute.

public int getMinimumRent(): Getter method for the minimumRent attribute.

public int calculateNumberOfPropertiesWithoutOwner(): This method calculates the number properties without an owner.

public int calculateNetWorthOfPlayers(): This method calculates the total net worth of the players.

3.4.13 Dice Class

Explanation: This class represents the dice of the game.

Attributes:

private int dice1: This attribute holds the value of the first dice.

private int dice2: This attribute holds the value of the second dice.

Methods:

public void rollDice(): This method allows the user to roll the dice.

public int getDiceResultSum(): Getter method for the sum of the dice results.

public int getDice1(): Getter method for the dice1 attribute.

public int getDice2(): Getter method for the dice2 attribute.

3.4.14 BailOutChoice Enumeration

Explanation: This enumeration has four types: bail out card, money, double dice, wait. These choices are offered to the player when the player is in jail.

3.4.15 TaxOption Enumeration

Explanation: This enumeration has two types which are the tax with ratio and the fixed tax amount. These enumeration types are offered as a choice to the player.

3.4.16 PlayerFactory Class

Explanation: Extends the AbstractPlayerFactory abstract class.

Constructors:

public PlayerFactory(): Default constructor.

3.4.17 AuctionModel Class

Explanation: Contains all of the necessary information and attributes for an auction. Is responsible for handling an auction between 2 players.

Attributes:

private boolean active: Attribute denoting whether the auction is happening right now.

private Player highestBiddingPlayer: Player that has bid the highest amount to the property in auction.

private int remainingTime: The time left on the clock for auction. Various methods of this class modify this attribute.

private int highestBid: The highest money bid on the auction at the particular moment in the game.

Constructors:

public AuctionModel(): The default constructor for an AuctionModel.

Methods:

public void startAuction(ArrayList<TitleDeedCard> titleDeedCards): Starts the auction for the TitleDeedCard ArrayList. Sets active to true.

public void bid (Player player): Increases the highest bid changes the highestBiddingPlayer to the parameter player.

public void endAuction(): Ends the current auction by setting active to false.

public int getHighestBid(): Getter method for the highestBid.

public void setHighestBid(int highestBid): Setter method for the highestBid.

public int getHighestBiddingPlayer(): Getter method for the highestBiddingPlayer.

public void setHighestBiddingPlayer(int highestBid): Setter method for the highestBiddingPlayer.

public boolean getActive(): Getter method for the active.

public void setActive(): Setter method for the active.

3.4.18 TradeModel Class

Explanation:

Attributes:

private Player player1: This attribute holds the first performer of the trade.

private Player player2: This attribute holds the second performer of the trade.

private ArrayList<TitleDeedCard> titleDeedCardsPlayer1: This attribute holds the title deed cards offered by player1.

private ArrayList<TitleDeedCard> titleDeedCardsPlayer2: This attribute holds the title deed cards offered by player2.

private int moneyPlayer1: This attribute holds the amount of money offered by player1.

private int moneyPlayer1: This attribute holds the amount of money offered by player2.

Methods:

public void startTrade(Player player1, Player player2): This method allows the players to initiate a trade.

public void completeTrade(): This method concludes the trade.

public void cancelTrade(): This method cancels the ongoing trade.

public Player getPlayer1(): Getter method for the player1 attribute.

public void setPlayer1(Player player1): Setter method for the player1 attribute.

public Player getPlayer2(): Getter method for the player1 attribute.

public void setPlayer2(Player player2): Setter method for the player2 attribute.

public ArrayList<TitleDeedCard> getTitleDeedCardsPlayer1(): Getter method for the titleDeedCardsPlayer2 attribute.

public void setTitleDeedCardsPlayer1(ArrayList<TitleDeedCard> titleDeedCardsPlayer1): Setter method for the titleDeedCardsPlayer1 attribute.

public ArrayList<TitleDeedCard> getTitleDeedCardsPlayer2(): Getter method for the titleDeedCardsPlayer2 attribute.

public void setTitleDeedCardsPlayer2(ArrayList<TitleDeedCard> titleDeedCardsPlayer2): Setter method for the titleDeedCardsPlayer2 attribute.

public int getMoneyPlayer1(): Getter method for the moneyPlayer1 attribute.

public void setMoneyPlayer1(int moneyPlayer1): Setter method for the moneyPlayer1 attribute.

public int getMoneyPlayer2(): Getter method for the moneyPlayer2 attribute.

public void setMoneyPlayer2(int moneyPlayer2): Setter method for the moneyPlayer2 attribute.

3.4.19 Command Interface

Explanation: The interface that creates a base for the extending GameAction class.

Methods:

public void execute(Player player): Executes the specified action for the player passed as a parameter

3.4.20 GameActionFactory Class

Explanation: It uses the factory design pattern to create GameAction objects.

Methods:

public ArrayList<GameAction> createPossibleActions(String classname): This method creates possible actions.

3.4.21 AbstractGameActionFactory Abstract Class

Explanation: Abstract class that serves as a base for GameActionFactory class.

Methods:

public ArrayList<GameAction> createPossibleActions(String classname): This method creates possible actions.

3.4.22 GameAction Class

Explanation: This class represents the actions that are taken for certain ingame operations.

Attributes:

private String name: This attribute holds the name of the action.

private Command command: This attribute holds the command object.

private boolean isMandatory: This attribute holds the boolean value of the mandatory situation of the action.

private boolean isActive: This attribute holds the boolean value of the activeness of the action.

private Player player: This attribute holds the player object.

Methods:

public String getName(): This method returns the name attribute.

public void setName(String name): This method sets the name attribute.

public Command getCommand(): This method returns the command object.

public void setCommand(Command command): This method sets the command object.

public boolean getIsMandatory(): This method returns the isMandatory attribute.

public void setIsMandatory(boolean isMandatory): This method sets the isMandatory attribute.

public boolean getIsActive(): This method returns the isActive attribute.

public void setIsActive(boolean isActive): This method sets the isActive attribute.

public Player getPlayer(): This method returns the player attribute.

public void setPlayer(Player player): This method sets the player attribute.

3.4.23 PlayerToken Class

Explanation: This class represents each player's respective unique token on the board.

Attributes:

private Tile currentTile: This attribute holds the tile object the PlayerToken is on.

private String type: This attribute stores the type of the token with its type name.

private int currentTileIndex: This attribute stores the current tile index the PlayerToken is on.

private Board board: This attribute stores the board object of the game.

private int lastDiceRoll: This attribute has information about the last dice roll value.

Constructors:

PlayerToken(String type, Board board): It constructs the player token with its type and with board information.

PlayerToken(): It is the default constructor.

Methods:

public void move(int amount): This method moves the token for the specified amount in the argument list.

public void goToJail(): This method moves the token to the jail.

public void teleport(int teleportTileIndex): This method teleports the token to the other teleport tile of the same kind, with the given index in the argument list.

public String getType(): This method returns the type of the token.

public String setType(): This method sets the type of the token.

public Tile getCurrentTile(): Returns the current tile token is on.

public boolean passedGoInTheLastMove(): Returns true if the Player has passed from the Go Tile with its last move. Returns false otherwise.

3.4.24 AICharacteristic Enumeration

Explanation: This enumeration influences AI's decision making. This enumeration is used in AIPlayer's simulate method to determine the Command to be returned. The enumeration types are Stingy, Adventurous, Balanced.

3.4.25 Tile Abstract Class

Explanation:

This class is the abstract class that creates the prototype of getPossibleActions method. Its dependent classes implement this method in their own ways.

Attributes:

private String name: This attribute holds the name of the tile.

private ArrayList<GameAction> possibleActions: This attribute holds all possible actions associated with landing on the tile.

private int index: This attribute indicates the location of the tile on the board.

Constructor: Default constructor.

Methods:

public int getIndex(): Getter method for the index attribute.

public void setIndex(int index): Setter method for the index attribute.

public ArrayList<GameActions> getPossibleActions(Player player): This method returns all possible actions associated with this type of tile.

public void setActive(ArrayList<GameActions> actionList, String actionName, boolean active): This method modifies the activeness of an action associated with the tile.

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionalities of different types of tiles.

3.4.26 TeleportTile Class

Explanation:

This class represents the Teleport Tile on the board.

Attributes:

private int counterpartIndex: This attribute holds the counterpart of one of the teleport tiles.

Constructors: Default constructor.

Methods:

public int getCounterpartIndex(): This method returns the teleport tile's counterpart's index.

public void setCounterpartIndex(int counterpartIndex): This method sets the teleport tile's counterpart's index.

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.27 GoTile Class

Explanation:

This class represents the Go Tile on the board.

Constructors: Default constructor.

Methods:

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.28 GoToJailTile Class

Explanation:

This class represents the Go To Jail Tile on the board.

Constructors: Default constructor.

Methods:

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.29 JailTile Class

Explanation:

This class represents the Jail Tile on the board.

Constructors: Default constructor.

Methods:

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.30 FreeParkingTile Class

Explanation:

This class represents the Free Parking Tile on the board.

Constructors: Default constructor.

Methods:

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.31 IncomeTaxTile Class

Explanation:

This class represents income tax tiles on the board.

Constructor: Default constructor.

Methods:

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction> actions): This method implements the special functionality of this type of tile.

3.4.32 TeleportTileActionStrategy Class

Explanation:

This class holds the possible actions that can be performed on the teleport tiles.

Methods:

public void button1Strategy(Player player): This method teleports the player to another one of its teleport tile kind.

public void button2Strategy(Player player): <Empty>

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.33 GoTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the go tiles.

Methods:

public void button1Strategy(Player player): This method makes it so that the player gets a predefined amount of money when passing the go tile.

public void button2Strategy(Player player): <Empty>

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.34 GoToJailTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the go to jail tile.

Methods:

public void button1Strategy(Player player): This method makes it so that the player goes to the jail tile when he lands on the go to jail tile.

public void button2Strategy(Player player): <Empty>

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.35 JailTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Jail tiles.

Methods:

public void button1Strategy(Player player): This method allows the player to wait for three turns to bail out of the jail.

public void button2Strategy(Player player): This method allows the player to pay the fee to bail out of jail.

public void button3Strategy(Player player): This method allows the player to bail out from the jail if he throws a double dice in three tries.

public void button4Strategy(Player player): This method allows the player to bail out from jail if he has and uses a bail out from jail card.

3.4.36 FreeParkingTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Free Parking Tiles.

Methods:

public void button1Strategy(Player player): <Empty>

public void button2Strategy(Player player): <Empty>

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.37 IncomeTaxTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Income Tax Tile.

Methods:

public void button1Strategy(Player player): This method allows the player to pay the tax in money format, by selecting 0 in the setTaxOption() method.

public void button2Strategy(Player player): This method allows the player to pay the tax in his total money percentage-wise, by selecting 1 in the setTaxOption() method.

public void button3Strategy(Player player): This method allows the player to pay the tax by calling the payTax() method.

public void button4Strategy(Player player): <Empty>

3.4.38 PropertyTile Class

Explanation: This class holds its constructor, and the method to contact with its strategy class.

Attributes:

private TitleDeedCard titleDeedCard: This attribute holds the title deed card associated with the property tile.

Constructor: Default constructor.

Methods:

public TitleDeedCard getTitleDeedCard(): Getter method for the titleDeedCard attribute.

public void setTitleDeedCard(TitleDeedCard titleDeedCard): Setter method for the titleDeedCard attribute.

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction> actions): This method implements the special functionality of this type of tile.

3.4.39 PropertyTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Property Tiles.

Methods:

public void button1Strategy(Player player): This method allows the user to buy the property.

public void button2Strategy(Player player): This method allows the user to pay the rent, this operation also has the possibility to trigger a trade operation.

public void button3Strategy(Player player): This method allows the user to start the auction for a property.

public void button4Strategy(Player player): <Empty>

3.4.40 PropertyActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Property Tiles.

Methods:

private void upgradeProperty(Player player): This is a helper method that is used in the button1Strategy method.

private void downgradeProperty(Player player): This is a helper method that is used in the button2Strategy method.

private void mortgageProperty(Player player): This is a helper method that is used in the button3Strategy method.

private void removeMortgage(Player player): This is a helper method that is used in the button4Strategy method.

public void button1Strategy(Player player): This method allows the user to upgrade the property.

public void button2Strategy(Player player): This method allows the user to downgrade the property.

public void button3Strategy(Player player): This method allows the user to mortgage the property.

public void button4Strategy(Player player): This method allows the user to remove mortgage from a property.

3.4.41 TitleDeedCard Class

Explanations: This class holds all the attributes and methods of the title deed cards. The operations of properties (Pay rent, upgrade, mortgage etc.) are performed according to the attributes in this class.

Attributes:

private String propertyName: This attribute determines the name of the property.

private int levelOneRent: This attribute holds the property's rent when the property is not upgraded.

private int levelTwoRent: This attribute holds the property's rent when the property is upgraded to level 2.

private int levelThreeRent: This attribute holds the property's rent when the property is upgraded to level 3.

private int levelFourRent: This attribute holds the property's rent when the property is upgraded to level 4.

private int levelFiveRent: This attribute holds the property's rent when the property is upgraded to level 5.

private boolean isMortgaged: This attribute holds the property's mortgage status.

private int upgradeLevel: This attribute indicates the upgrade level of property. When the property is upgraded, the value of this variable is incremented.

private int upgradeCost: This attribute holds the cost of upgrading the property.

private int propertyValue: This attribute holds the total value of the property.

private int mortgageValue: This attribute holds the amount of money that will be obtained by mortgaging the property.

private ColorGroup colorGroup: This attribute holds the color group (red, blue etc.) of the property.

private int mortgagedTurnNumber: This attribute holds the number of turns the property is mortgaged.

private ArrayList<GameAction> propertyActions: This attribute holds all possible actions associated with the property.

private Player owner: This attribute holds the Player that owns the property.

private final double MORTGAGE_REMOVAL_PENALTY_MULTIPLIER: This constant value will be used to calculate the mortgage removal fee.

Methods:

public String getPropertyName(): Getter method for the propertyName attribute.

public void setPropertyName(String propertyName): Setter method for the propertyName attribute.

public int getLevelOneRent(): Getter method for the levelOneRent attribute.

public void setLevelOneRent(int levelOneRent): Setter method for the levelOneRent attribute.

public int getLevelTwoRent(): Getter method for the levelTwoRent attribute.

public void setLevelTwoRent(int levelTwoRent): Setter method for the levelTwoRent attribute.

public int getLevelThreeRent(): Getter method for the levelThreeRent attribute.

public void setLevelThreeRent(int levelThreeRent): Setter method for the levelThreeRent attribute.

public int getLevelFourRent(): Getter method for the levelFourRent attribute.

public void setLevelFourRent(int levelFourRent): Setter method for the levelFourRent attribute.

public int getLevelFiveRent(): Getter method for the levelFiveRent attribute.

public void setLevelFiveRent(int levelFiveRent): Setter method for the levelFiveRent attribute.

public boolean isMortgaged(): Getter method for the isMortgaged attribute.

public void setIsMortgaged(boolean isMortgaged): Setter method for the isMortgaged attribute.

public boolean isOwned(): Getter method for the isOwned property.

public void setIsOwned(boolean isOwned): Setter method for the isOwned attribute.

public int getUpgradeLevel(): Getter method for the upgradeLevel attribute.

public void setUpgradeLevel(int upgradeLevel): Setter method for the upgradeLevel attribute.

public int getPropertyValue(): Getter method for the propertyValue attribute.

public void setPropertyValue(int propertyValue): Setter method for the propertyValue attribute.

public int getMortgageValue(): Getter method for the mortgageValue attribute.

public void setMortgageValue(int mortgageValue): Setter method for the mortgageValue attribute.

public int getUpgradeCost(): Getter method for the upgradeCost attribute.

public void setUpgradeCost(int upgradeCost): Setter method for the upgradeCost attribute.

public int getMortgagedTurnNumber(): Getter method for the mortgagedTurnNumber attribute.

public void setMortgagedTurnNumber(int mortgagedTurnNumber): Setter method for the mortgagedTurnNumber attribute.

public ColorGroup getColorGroup(): Getter method for the colorGroup attribute.

public void setColorGroup(ColorGroup colorGroup): Setter method for the colorGroup attribute.

public ArrayList<GameActions> getPropertyActions(): Getter method for the propertyActions attribute.

public void setPropertyActions(ArrayList<GameAction> propertyActions): Setter method for the propertyActions attribute.

public Player getOwner(): Getter method for the owner attribute.

public void setOwner(Player player): Setter method for the owner attribute.

public boolean activateAction(String actionName): A helper method that searches the action with the given name on propertyActions ArrayList, and activates that action.

public boolean deactivateAction(String actionName): A helper method that searches the action with the given name on propertyActions ArrayList, and deactivates that action.

public void updateActions(): After each change in the state of property (upgradeLevel, isOwned etc.) this method is called. It activates and deactivates the actions based on the current state of the property.

public int upgradeProperty(): This method increments the property's upgrade level and returns the amount to be taken from the player's balance.

public int downgradeProperty(): This method decrements the property's upgrade level and returns the amount to be added to the player's balance.

public void resetProperty(): This method restores the property back to its initial state.

public int mortgage(): This method changes the property state to "mortgaged" and returns the amount to be added to the player's balance

public int removeMortgage(): This method changes the property state back to "not mortgaged" and returns the amount to be taken from the player's balance.

3.4.42 ColorGroup Class

Explanation:

This class represents the color groups that specific properties belong to.

Attributes:

private ArrayList<Tile> group: This attribute holds the tiles that belong to this color group.

private Color color: This attribute holds the color of this color group.

Constructor:

public ColorGroup(ArrayList<Tile> group): Constructs a ColorGroup object with the specified group of tiles.

Methods:

public ArrayList<Tile> getGroup(): Getter method for the group attribute.

public void setGroup(ArrayList<Tile> group): Setter method for the group attribute.

public Color getColor(): Getter method for the color attribute.

public void setColor(Color color): Setter method for the color attribute.

public boolean allOwnedByPlayer(Player player): A helper method that checks whether all properties in the color group are owned by the specified player. This method is called when the player attempts to upgrade a property.

3.4.43 CardTile Class

Explanation:

This class holds the drawCard(), and getPossibleActions() methods.

Attributes:

private CardDeck cardDeck: This attribute holds the deck that the player will draw cards from when they land on the card tile.

Constructor: Default constructor

Methods:

public CardDeck getCardDeck(): Getter method for the cardDeck attribute.

public void setCardDeck(CardDeck cardDeck): Setter method for the cardDeck attribute.

protected ArrayList<GameAction> hook(Player player, ArrayList<GameAction>): This method implements the special functionality of this type of tile.

3.4.44 CardTileActionStrategy Class

Explanation: This class holds the possible actions that can be performed on the Card Tiles.

Methods:

public void button1Strategy(Player player): This method allows the player to draw and apply what the chance card commands.

public void button2Strategy(Player player): This method allows the player to draw and apply what the community chest card commands..

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.45 Card Class

Explanation:

This class represents the cards that can be drawn by the player during the game.

Attributes:

private String instruction: This attribute holds the instruction which is written on the card.

private ArrayList<GameAction> actions: This attribute holds the actions associated with the specific card.

private String type: This attribute holds the type of the card.

private HashMap(String, int) cardDetails: This attribute holds the card's action with an integer value specifying the amount (Such as, proceed 3 tiles).

Constructor:

public Card(ArrayList<GameAction> actions, String instruction, String type): Constructs a Card object with the given attributes.

Methods:

public String getInstruction(): Getter method for the instruction attribute.

public void setInstruction(String instruction): Setter method for the instruction attribute.

public ArrayList<GameAction> getActions(): Getter method for the actions attribute.

public void setActions(ArrayList<GameAction> actions): Setter method for the actions attribute.

public String getType(): Getter method for the type attribute.

public void setType(String type): Setter method for the type attribute.

3.4.46 AbstractCardFactory Abstract Class

Explanation: This class is responsible for creating cards. This class is currently the superclass of the CardFactory class only. In future, other kinds of card factory classes can be implemented as subclasses of this abstract class.

Methods:

public Card createCard(JSONObject config): Creates a card based on the specifications in the configuration object.

3.4.47 CardFactory Class

Explanation: This class is responsible for creating standard types of chance and community chest cards in the game.

Methods:

public Card createCard(JSONObject config): Creates a card based on the specifications in the configuration object.

3.4.48 AbstractTileFactory Abstract Class

Explanation: This class is responsible for creating the tiles on the game board. This class is currently the superclass of the TileFactory class only. In future, other kinds of tile factory classes can be implemented as subclasses of this abstract class.

Methods:

public Tile getTile(JSONObject config): Creates a tile based on the specifications in the configuration object.

3.4.49 TileFactory Class

Explanation: This class is responsible for creating standard types of tiles on the game board.

Methods:

public Tile getTile(JSONObject config): Creates a tile based on the specifications in the configuration object.

3.4.50 CardDeckBuilder Class

Explanation: This class is responsible for building the card deck builder.

Methods:

public CardDeck build(JSONObject config): This method builds the Card Deck with the argument it gets, which is the configuration object.

3.4.51 CardDeck Abstract Class

Explanation: This class represents a card deck that brings all Chance or Community Chest cards together.

Attributes:

private ArrayList<Card> cardArrayList: An attribute that holds all Card objects in the deck.

Constructor:

public CardDeck(ArrayList<Card> cardArrayList): Constructs a Chance or Community Chest card deck with specified cards.

3.4.52 ChanceCardDeck Class

Explanation: This class is a subclass of CardDeck class. It represents a deck of Chance Cards. Although it consists of Card objects like CommunityChestCardDeck class, the distribution of the card types are different.

3.4.53 CommunityChestCardDeck Class

Explanation: This class is a subclass of CardDeck class. It represents a deck of Community Chest Cards. Although it consists of Card objects like ChanceCardDeck class, the distribution of the card types are different.

3.4.54 BoardBuilder Class

Explanation:

Class that is mainly responsible for getting the user-inputted data required for saving and initializing a board and writes them into a configuration file. It gets the data from the UI screens in the presentation layer.

Attributes:

private Board board: This attribute is an instance of the Board that is being created by the player.

Methods:

public Board getBoard(): This method returns the board object.

public void setBoard(Board board): This method sets the board with the board that is passed to that functions arguments.

public void changeTileName(String tileName): This method gets the name that is written by the player while building a board from the UI screen and sets the name of the tile accordingly.

public void changeRent(int rent): This method gets the integer value that is written by the player while building a board from the UI screen and sets the rent of the tile accordingly.

public void changePropertyValue(int value): This method gets the integer value that is written by the player while building a board from the UI screen and changes the value of the property accordingly.

public void changeMortgageValue(int mortgageValue): This method gets the integer value that is written by the player while building a board from the UI screen and changes the mortgage value of the property accordingly.

public void changeSalary(int salary): This method gets the integer value from the UI screen and changes the salary which is given to every player after they pass the go tile.

public void setBoardName(String boardName): This method gets the string value from the UI screen and changes the name of the board.

public void saveBoard(): This method saves the state of the board

3.4.55 SerializationHandler Class

Explanation: This class performs serialization and deserialization methods for saving and loading game.

Methods:

public boolean saveSession(GameSession gameSession): This method saves the current game session by taking the GameSession object to its argument, and returns a boolean value.

public boolean loadSession(String sessionName): This method loads the session by taking the session name to its arguments, and returns a boolean value.

3.4.56 FileManager Class

Explanation: This class manages the files, acts like a file storage.

Attributes:

private ArrayList<String> configList: This attribute holds the list of all the configurations in a String list.

private ArrayList<String> sessionNames: This attribute holds the session names in a String list.

Methods:

public ArrayList<String> getSavedSessionNames(): This method returns the list of the saved session names.

public String getSessionLocation(String sessionName): This method returns the location of the session location.

public ArrayList<String>getBoardConfigNames(): This method returns the board configuration names in a String list format.

public void addBoardConfigName(String name): This method adds a configuration name.

3.4.57 BoardConfiguration Class

Explanation: This class holds the information about the properties of the board to be initialized. It holds properties including the number of players, number of human and AI players, pace of the game, characters of the AI players.

Attributes:

private GamePace gamePace: This attribute holds the game pace object.

private int maxPlayerCount: This attribute holds the maximum number of players.

private int humanPlayerCount: This attribute holds the number of human players.

private AICharacteristic aiCharacteristic: This attribute holds the ai's characteristics.

private String boardName: This attribute holds the board's name.

Methods:

public GamePace getGamePace(): This method returns the game's pace.

public void setGamePace(GamePace gamePace): This method sets the game's pace.

public int getMaxPlayerCount(): This method returns the maximum number of players.

public void setMaxPlayerCount(int maxPlayerCount): This method sets the maximum number of players.

public int getHumanPlayerCount(): This method returns the number of human players.

public void setHumanPlayerCount(int humanPlayerCount): This method sets the number of human players' count.

public AICharacteristic getAICharacteristic(): This method returns the characteristic of the AI.

public void setAICharacteristic(AICharacteristic aiCharacteristic): This method sets the AI's characteristic.

public String getBoardName(): This method returns the board's name.

public void setBoardName(String boardName): This method sets the board's name.

3.4.58 ConfigHandler Class

Explanation: This class is responsible for interacting with the file system. It creates and reads config files.

Methods:

public JSONObject getConfig(String configName): Reads the config with the given name from the file system.

public void createConfig(JSONObject config): Creates a config file with the name specified in the config file.

public JSONObject getConfigTemplate(): This method returns a template config to change its properties in the board builder.

4. Improvement Summary

In summary, we have revised the methods and attributes of classes in detail and investigated missing classes in the first iteration and added them. We have also got rid of redundant methods. We have changed the way the system stores a single game session in a file; previously, we read and write XML files for configurations, but now the system creates and reads JSON Object files to carry out saving and loading of a game session. We have resolved ambiguities in the report to be concise like how we specified how often the game is automatically saved (that is, at each turn's end). Our AI players (bots) design is decided and its related classes are finalized; the system calculates game statistics for AI players to choose a command to execute in one turn with conditional statements. We have filled the specifications (mostly the handlers) of controller classes of views (screens) in the object model diagram. We have revised the method explanations to clearly display what each method and attribute represent. We have corrected our object model diagram according to these changes and added the multiplicities on it.