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

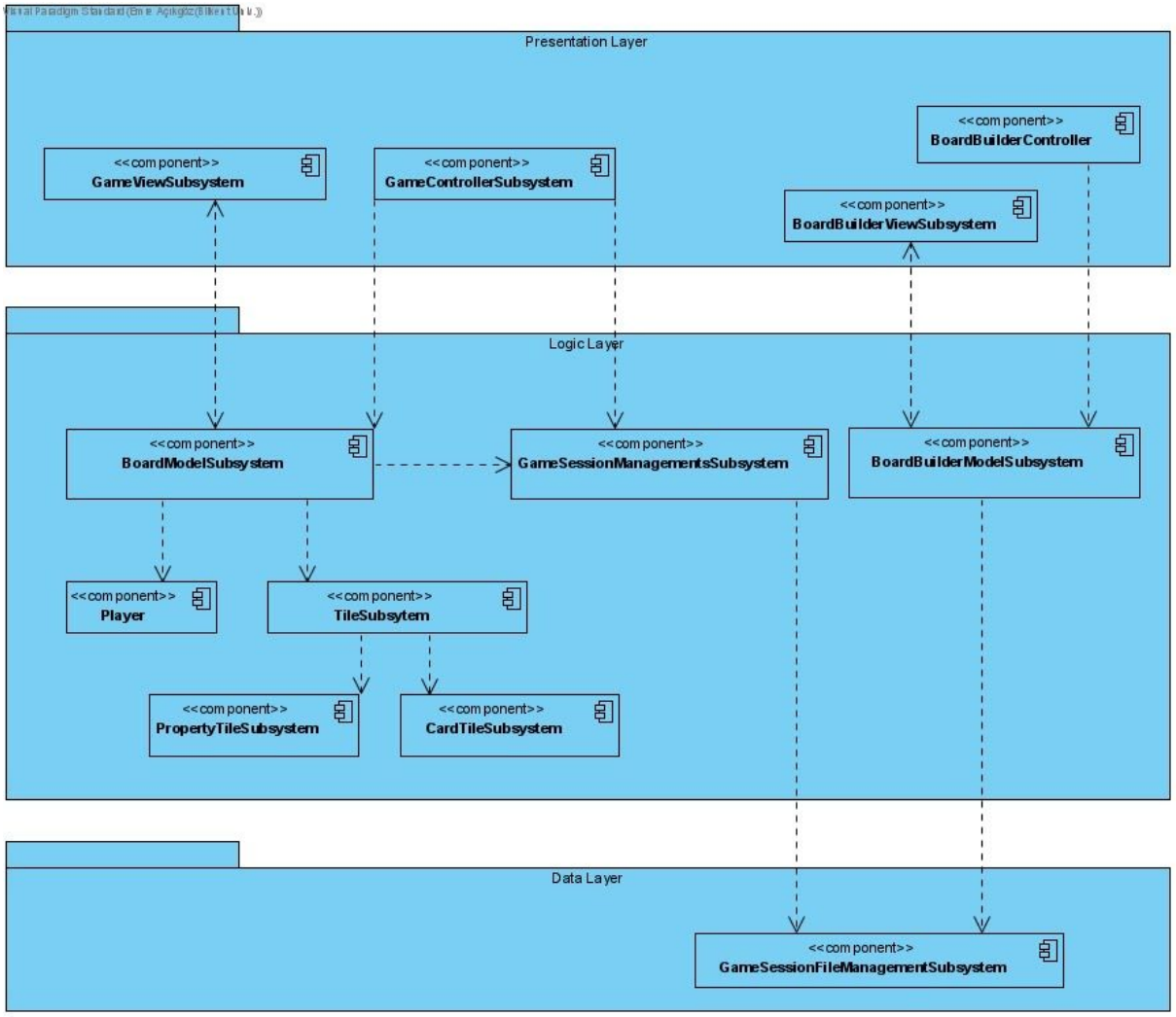

Figure 1. Subsystem Decomposition

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 Model View Controller 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, CyclingIterable, CyclingIterator, are the classes and Iterable is the interface in the Board Model Subsystem. BoardModelSubsystem is responsible for the creation of general logic of the board. Such as creating the board and making the ground for iterating on the

board. In Board class we have used the Iterator Pattern, as it can be seen from the CyclingIterable interface, and CyclingIterator class. In general, in this subsystem, we used Abstract Factory Design Pattern and Builder Design Pattern.

GameSessionManagementSubsystem: This subsystem includes the GameSessionManager. This subsystem controls the game, and it is more of an entity than a 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. In this subsystem we used Abstract Factory Design Pattern and the Builder Design Pattern.

TileSubsystem: Tile (Abstract), TeleportTile, GoTile, GoToJailTile, JailTile, FreeParking Tile, IncomeTaxTile, TeleportTileActionStrategy, GoTileActionStrategy, GoToJailTileActionStrategy, JailTileActionStrategy, FreeParkingTileActionStrategy and IncomeTaxTileActionStrategy are the classes in TileSubsystem. Observable and Representable are the interfaces in 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.

PropertyTileSubsystem: PropertyTile, PropertyTileActionStrategy 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.

PlayerSubsystem: PlayerSubsystem includes the interface Player, abstract classes AbstractPlayer and AbstractPlayerFactory, and the classes HumanPlayer, AIPlayer, PlayerBuilder, PlayerFactory, PlayerToken as well as the enumeration AICharacteristic, which are central classes in logic layer. This subsystem is mainly responsible for the representation and management of human and AI players playing the game and their tokens on the board. All AI operations regarding decision making are also a part of this subsystem.

There are several design patterns implemented in this subsystem, namely the Builder Design Pattern and the Abstract Factory Design Pattern. PlayerBuilder class is designed according to the Builder Design Pattern, as it builds an array of Players, and AbstractPlayerFactory implements the Abstract Factory Design Pattern because it creates a single Player object.

Players in the Monopoly game are initialized by the combination of these 2 patterns; PlayerBuilder class makes use of AbstractPlayerFactory to create Players.

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 XMLHandler are the classes belonging to GameSessionManagementSubystem. 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 xml 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 does not require any additional specific hardware. The keyboard is only used to enter the offered amount of money during a trade process. Every other operation can be done by using the mouse.

The game will be implemented by using Java programming language, with JDK 8. For the graphical user interface, we are going to use JavaFX. 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 xml files to be accessed while creating or loading a game. 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 in a periodic fashion.

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

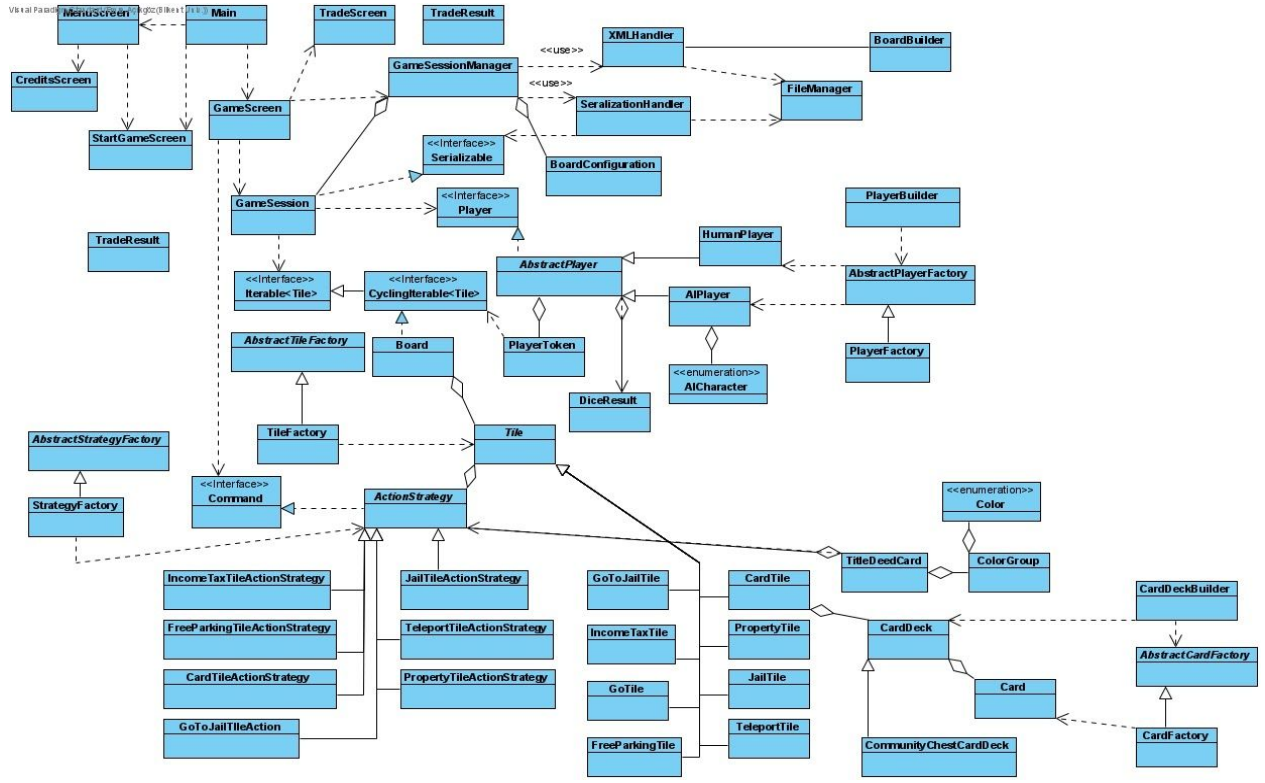

Figure 2. Abstraction of final object design

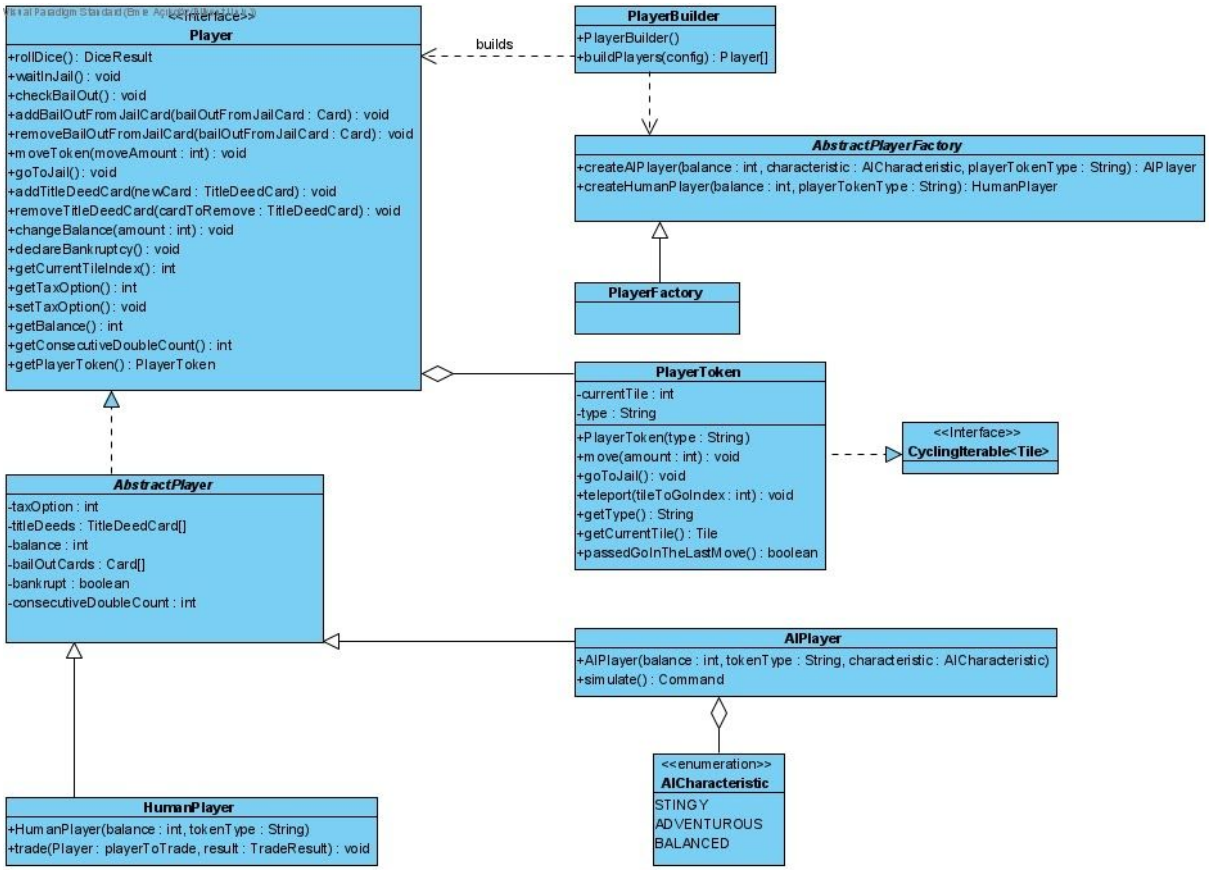

Figure 3. Classes related to Player.

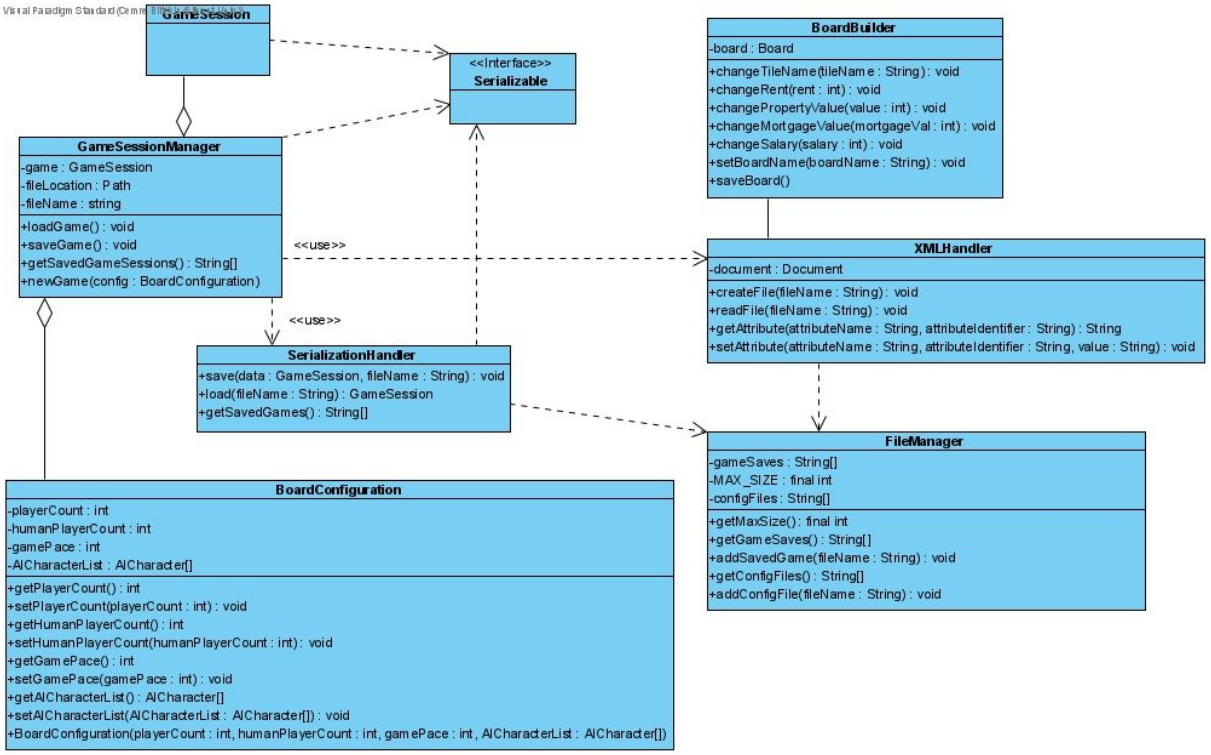

Figure 4. Classes related to game configurations & file handling.

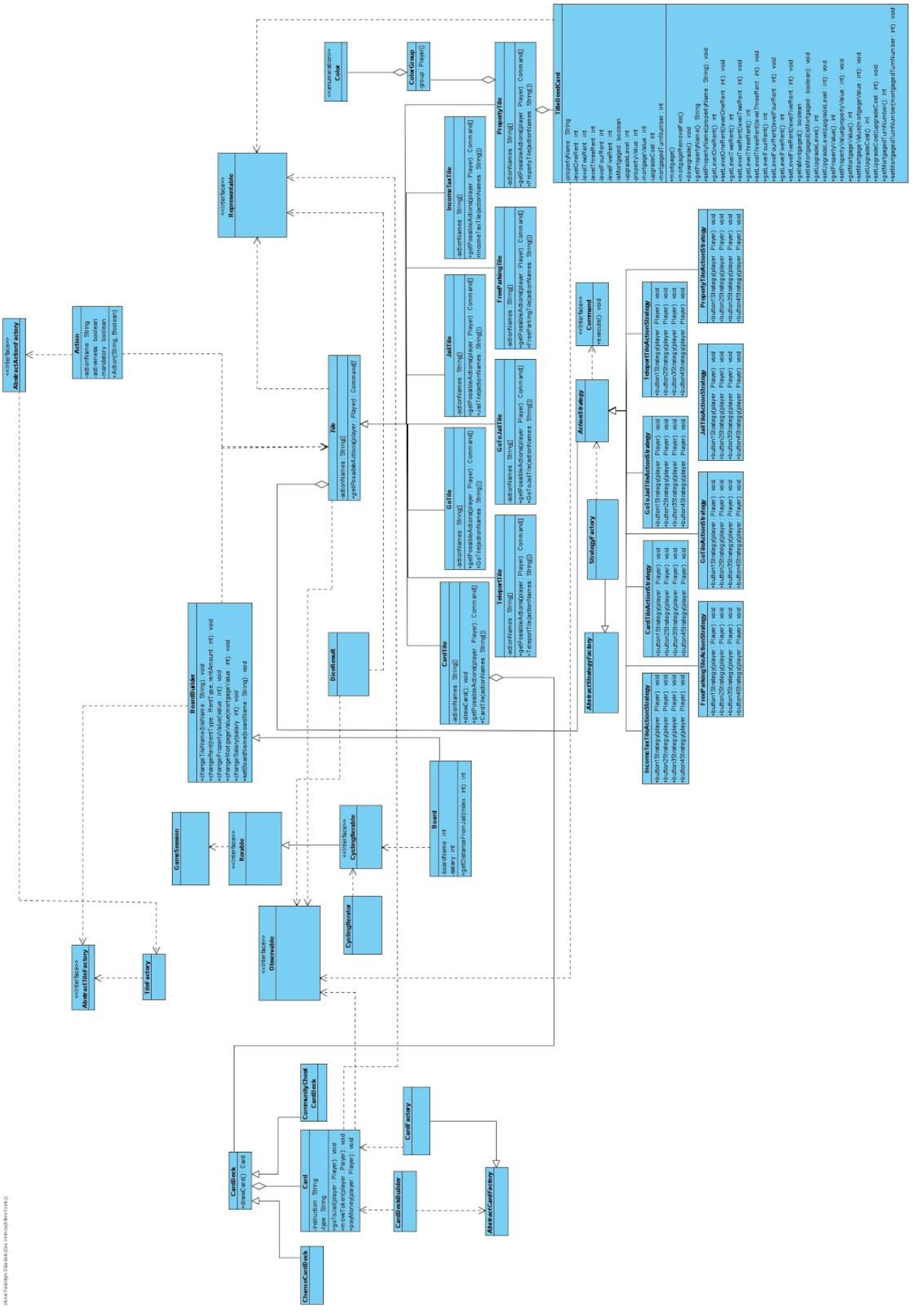

Figure 5. Classes related to board & tiles.

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.

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.

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

3.4.2 Board Class

Explanation:

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

Attributes:

private String boardName: This attribute holds the name of the board.

private int salary: This attribute holds the salary.

Methods:

public int getDistanceFromJail(int index): This method returns the distance from the jail, it takes the index of the current player token.

3.4.3 Iterable Interface

Explanation: This interface makes the iterators move.

3.4.4 CyclingIterable Interface

Explanation: This interface makes the iterator move through the tiles endlessly, in a circular fashion.

3.4.5 CyclingIterator Class

Explanation: Iterates through objects (in this case tiles) of classes implementing CyclingIterable Interface.

3.4.6 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 DiceResult rollDice(): This method rolls the dice and returns the result as DiceResult object.

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

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

public void moveToken(moveAmount: int): 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(newCard: TitleDeedCard): 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(cardToRemove: TitleDeedCard): 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(amount: int): 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 getCurrentTileIndex(): This method returns the index of the tile that the player is currently on.

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

public PlayerToken getTokenType(): This method returns the token the player has.

3.4.8 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 int taxOption: This attribute represents the user selected option for the method of paying tax throughout the game.

private TitleDeedCard[] titleDeeds: This attribute is an array of the title deed cards.

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

private Card[] bailOutCards: This attribute represents the array of bail out from 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 go.

private PlayerToken playerToken: This attribute represents the players token.

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

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

3.4.10 HumanPlayer Class

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

Constructors:

public HumanPlayer(balance : int, tokenType : String): Constructs the HumanPlayer object with the balance and tokenType.

Methods:

public void trade(playerToTrade: Player, result: TradeResult): This method sets the balance and the property list of the trade parties according to the trade result that comes from the trade's GUI screen.

3.4.11 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(balance : int, tokenType : String, characteristic : AICharacteristic): 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.

3.4.12 PlayerBuilder Class

Explanation: Responsible for building an array of Players at the beginning of the game. Uses AbstractPlayerFactory class for this functionality.

Constructors:

public PlayerBuilder(): Default constructor.

Methods:

public Player[] buildPlayers(config): Builds an array of the players to initialize the game. The attributes of individual players depends on the config parameter.

3.4.13 PlayerFactory Class

Explanation: Extends the AbstractPlayerFactory abstract class.

Constructors:

public PlayerFactory(): Default constructor.

3.4.14 PlayerToken Class

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

Attributes:

private int currentTile: Holds the index of the tile PlayerToken is on.

private String type: Stores the type of the token,

Constructors:

PlayerToken(type : String): Constructs the player token with its type.

Methods:

public void move(amount: int): 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(currentTileIndex: int): 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 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.

3.4.15 AICharacteristics 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.

3.4.16 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:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.17 TeleportTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public TeleportTile(actionNames: String[]): This constructor constructs a TeleportTile object with the specified action names.

Methods:

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

3.4.18 GoTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public GoTile(actionNames: String[]): This constructor constructs a GoTile object with the specified action names.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.19 GoToJailTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public GoToJailTile(actionNames: String[]): This constructor constructs a GoToJailTile object with the specified action names.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.20 JailTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public JailTile(actionNames: String[]): This constructor constructs a JailTile object with the specified action names.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.21 FreeParkingTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public FreeParkingTile(actionNames: String[]): This constructor constructs a FreeParkingTile object with the specified action names.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.22 IncomeTaxTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

public IncomeTaxTile(actionNames: String[]): This constructor constructs an IncomeTaxTile object with the specified action names.

Methods:

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.23 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): This method teleports the player to another one of its teleport tile kind.

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.24 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>

3.4.25 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>

3.4.26 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 bailout from jail card.

3.4.27 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>

3.4.28 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 for the defined amount of money.

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

public void button3Strategy(Player player): <Empty>

public void button4Strategy(Player player): <Empty>

3.4.29 Observable Interface

Explanation: This interface represents an observable object.

3.4.30 Representable Interface

Explanation: This interface points to the graphics in the XML files.

3.4.31 PropertyTile Class

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructors:

PropertyTile(actionNames: String[]): This constructor constructs a PropertyTile object with the given arguments.

Methods:

Command[] getPossibleActions(Player player): This method returns all possible actions associated with this type of tile.

3.4.32 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 sell the property.

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

public void button4Strategy(Player player): <Empty>

3.4.33 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:

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

int levelOneRent: This attribute holds the property's rent.

int levelTwoRent: This attribute holds level 2 property's rent.

int levelThreeRent: This attribute holds level 3 property's rent.

int levelFourRent: This attribute holds level 4 property's rent.

int levelFiveRent: This attribute holds level 5 property's rent.

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

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

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

Methods:

String getPropertyName(): This method returns the property's name.

void setPropertyName(String propertyName): This method sets the property's name.

int getLevelOneRent(): This method returns the level one rent.

void setLevelOneRent(int levelOneRent): This method sets the level one rent.

int getLevelTwoRent(): This method gets the level two rent.

void setLevelTwoRent(int levelTwoRent): This method sets the level two rent.

int getLevelThreeRent(): This method gets the level three rent.

void setLevelThreeRent(int levelThreeRent): This method sets the level three rent.

int getLevelFourRent(): This method gets the level four rent.

void setLevelFourRent(int levelFourRent): This method sets the level four rent.

int getLevelFiveRent(): This method gets level five rent.

void setLevelFiveRent(int levelFiveRent): This method sets the level five rent.

boolean getIsMortgaged(): This method returns the mortgage status of the property.

void setIsMortgaged(boolean isMortgaged): This method sets the mortgage status of the property.

int getUpgradeLevel(): This method returns the upgrade level of the property.

void setUpgradeLevel(int upgradeLevel): This method sets the upgrade level of the property with its argument.

int getPropertyValue(): This method returns the value of the property.

void setPropertyValue(int mortgageValue): This method sets the property's value.

int getMortgageValue(): This method returns the mortgage value of the property.

void setMortgageValue(int mortgageValue): This method sets the mortgage value of the property.

int getUpgradeCost(): This method returns the upgrade cost of the property.

void setUpgradeCost(int upgradeCost): This method sets the upgrade cost of the property.

int getMortgagedTurnNumber(): This method returns the mortgaged turn number of the property.

void setMortgagedTurnNumber(int mortgagedTurnNumber): This method sets the mortgaged turn number of the property.

3.4.34 CardTile Class

Explanation:

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

Attributes:

String[] actionNames: This array holds the names of all possible actions associated with this tile.

Constructor:

public CardTile(actionNames: String[]): This constructor constructs a CardTile object with the given arguments.

Methods:

public void drawCard(): This method draws a card from the card deck and gives that card to the player.

public Command[] getPossibleActions(player: Player): This method returns all possible actions associated with this type of tile.

3.4.35 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>

3.4.36 BoardBuilder Class

Explanation:

This class sets the features of the board and contains them.

Attributes:

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

Methods:

public void changeTileName(tileName: String): 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(rent: int): 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(value: int): 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(mortgageValue: int): 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(salary: int): 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 board that is created and calls createFile function of XMLHandler.

3.4.37 SerializationHandler Class

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

Methods:

public void save(data: GameSession, fileName: String): This method gets the information about the game session from data parameter and the fileName in order to save the game and it serializases the game data by using serialization methods available in Java. It calls FileManager's addSavedGame function to add this save file to storage of save files.

public GameSession load(fileName: String): This method searches the folder in which the games are stored to find the file whose name is given in the parameter and returns a GameSession object by using deserialization methods available in Java.

public String[] getSavedGames(): This method gets the names of every saved game in the folder in which the games are stored.

3.4.38 XMLHandler Class

Explanation: This class creates and reads XML configuration files.

Attributes:

private Document document: This attribute is a document for a configuration file to be parsed to.

Methods:

public void createFile(fileName: String): This method creates an XML file for configuration with the given fileName that is provided by GameSessionManager. It calls FileManager's addConfigFile function to add this configuration file to file storage.

public void readFile(fileName: String): This method reads the configuration XML file.

public String getAttribute(attributeName: String, attributeIdentifier: String): This method gets the attribute value from the XML given the attribute name and the identifier.

public void setAttribute(attributeName: String, attributeIdentifier: String, value: String): This method sets the attribute value for the XML given the attribute name, the identifier, and the value.

3.4.39 FileManager Class

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

Attributes:

private String[] gameSaves: This array contains the name of every game that has been saved.

private final int MAX_SIZE: This attribute contains the maximum number of games and boards that can be saved.

private String[] configFiles: This attribute contains the name of every configuration XML file.

Methods:

public int getMaxSize(): This method returns the MAX_SIZE.

public String[] getGameSaves(): This method returns the list of names of save files that contain game session information.

public void addSavedGame(fileName: String): This method adds a new save file to save file storage (array) by its fileName.

public String[] getConfigFiles(): This method returns the name of every configuration file as an array.

public void addConfigFile(fileName: String): This method adds a new configuration file to configuration file storage (array) by its fileName.

3.4.40 BoardConfiguration Class

Explanation: This class holds information about new game criteria.

Attributes:

private int playerCount: This attribute contains the information about the number of players that is playing the game.

private int humanPlayerCount: This attribute contains the information about the number of human players which means playerCount - number of AI players.

private int gamePace: This attribute contains the information about the pace of the game.

private AICharacter[] AICharacterList: This array contains the AICharacters that are playing the game.

Constructor:

public BoardConfiguration(playerCount:int, humanPlayerCount: int, gamePace: int, AICharacterList: AICharacter[]): It constructs a BoardConfiguration object with the values given in the parameter.

Methods:

public int getPlayerCount(): It returns the number of players playing the game.

public void setPlayerCount(int: playerCount): This method sets the playerCount.

public int getHumanPlayerCount(): It returns the number of human player count.

public void setHumanPlayerCount(): This method sets the human player count.

public int getGamePace(): It returns the game pace.

public AICharacter[] getAICharacterList(): It returns the AICharacter list.

public void setAICharacterList(): It sets the AICharacter list.