Cs319 Term Project
Design Report

Section - 03

Group - 3E

Coronapoly

Group Members

1. Osman Buğra Aydın - 21704100
2. Ertuğrul Aktaş - 21802801
3. Muhammed Doğancan Yılmazoğlu - 21801804
4. Oğuzhan Angın - 21501910
5. Mehmet Alperen Yalçın - 21502273

Table Of Contents

1. Introduction

1.1 Purpose of the system

Coronapoly is a two dimensional board game that is established upon the ideas of the game Monopoly. Player starts with some amount of currency and rolls dice to move around the board of the game. Players can accomplish actions like buying or upgrading properties after rolling the dice. Player needs to complete the tasks like paying rent or draw a community card to fulfill the corresponding requirements of the landed tile after the dice roll. Players continue moving around the board and accomplishing the tasks of buying, selling or renting until the player is bankrupt. Goal of the player is to form a monopoly over the other players while trying not to become bankrupt. Players also need to deal with the Covid-19 situation while trying to achieve the goal of becoming the monopoly of the game. If a player fails the task of not becoming bankrupt, then that player is eliminated from the game. Last standing player is the winner of the game.

1.2 Criteria

Functional

Coronapoly games is designed to be played other players or bots. In both cases player needs to interact with other players or with bots. Also other players and bots need interact with each other as well. We can also count players interactions with the user interface and game. When all of that is considered, there are a lot of interactions needs to handled and processed. For this reason, functionality plays an important role in Coronapoly game. Inputs and outputs of the methods in the classes are designed and checked according to this need.

Easy To Play

Above in the tradeoffs part it is mentioned that in this project functionality is more important than usability or understandability due to the

high amount of interactions in its nature. Yet this does not mean that these properties are not important because Coronapoly is a game and it needs to be easily playable for the player. Simple and plain design of the user interface and menus aims to increase usability of the game to enhance the gameplay experience of the player. The How To Play menu aims to increase understandability of the game by introducing players to the mechanics and concepts of the game before playing. When both plain UI and How To Play menu combined aims to create a user friendly experience for the player while understanding and playing the Coronapoly game.

Fast

Games are the piece of software that often require high performance to function properly. For this reason Coronapoly game is designed to be fast and smooth in terms of performance to achieve high frame per seconds and fluent gameplay.

Portable

Coronapoly game is designed to be playable on different machines with different systems to reach a wide variety of players. For this reason Coronapoly is implemented using Java language. Java Virtual Machine (JVM) enables us to run this game on different platforms and systems. This situation increases the portability of the Coronapoly game.

Modifiable

Coronapoly game is implemented as a Model View Controller (MVC) design structure. MVC design structure can be modified easily since it separates each model, view and controller. This helps us to modify one of them while not worrying about others. Corresponding changes can be reflected to corresponding parts of the MVC design structure.

Extendable

To keep the game always fresh and interesting, Coronapoly game needs to be extendable. It should be easy to implement new features into the game according to user feedback or development decisions. Game design of the Coronapoly game gives us an opportunity to add, remove or modify the aspects and features of the game. These changes may include adding new cards to

community chests, changing values of the Corona settings, editing the contents of the loot box feature and many other changes.

2. High Level Software Architecture

2.1 Subsystem Decomposition

The System Architecture section depicts the decomposition of the monopoly game into smaller parts. The decomposition procedure aims to divide the system into subparts that would create efficiency when writing the initial code and doing maintenance in the later stages. For this project, Model-View-Controller (MVC) Architectural Style will be used for the following reasons:

1) The project itself is a game that requires presentation of internally processed data that MVC supports by its view system for the visualizing, model system for data processing and decision making and controller system to let the user do changes in the game.

2) The interactions in the system are not linear since the view model can also communicate with the data layer.

Here is a visual representation of the subsystems:

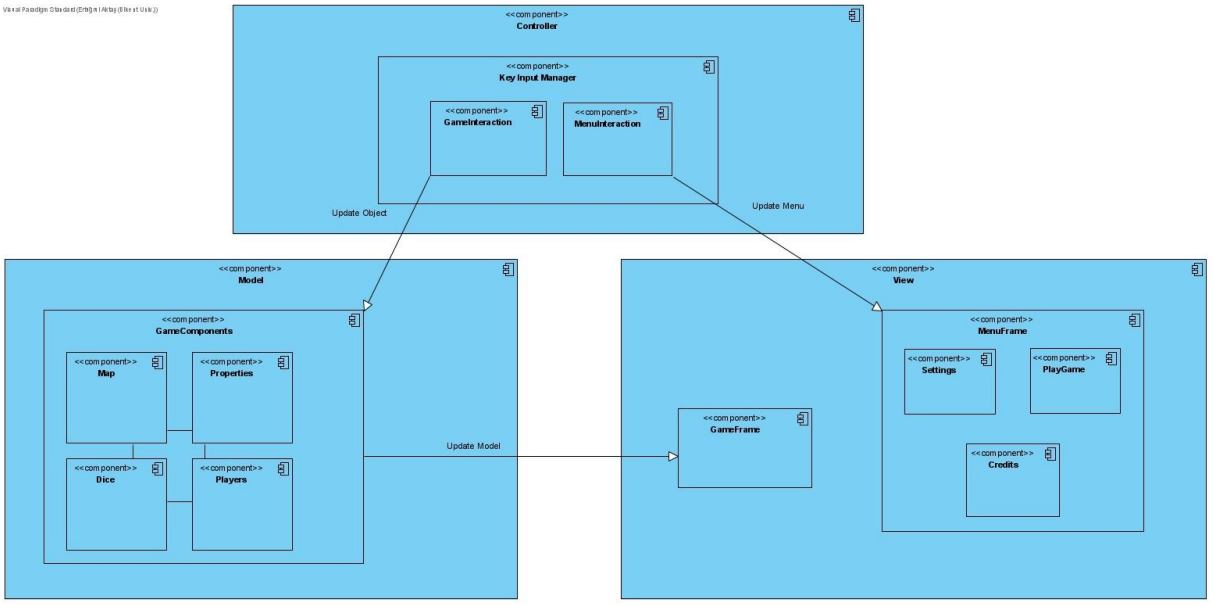

Figure 2.1.1: Subsystem Decomposition

Figure 2.1.1 shows the interactions between subsystems. The controller subsystem interacts with the user and decides on the action to do depending on the input of the user. The input could either be a game input where the user performs an action in the game or a menu interaction where the user could start or pause an ongoing game, enter settings or the credits screen. Game interaction subsystem which is a part of the controller, allows the system to update the game objects that could require internal processing. After the game entities are updated, their view is also updated so that game frame and the actual game is synchronized. The key idea here is that the view seen by the user could be both altered by the controller and the model system. This decomposition of the system fits with the MVC model that will provide a convenient implementation for the developers.

2.2 Hardware/Software Mapping

Coronapoly game depends on getting actions of the user from the user interface. Since the player movement just consist of rolling a dice and automatically moving on the board, keyboard will be rarely used for just entering numbers to user interface and other simple actions. Yet it is still a hardware requirement to play Coronapoly. This game heavily depend on mouse interactions, inputs and outputs to control the user interface. For this reason, a mouse is another hardware requirement for Coronapoly game. Since Coronapoly is not an online game, active internet connection is not required. Coronapoly is singleplayer game and will only run and played on a single computer. Everything will be processed locally, so there is no need to use a database. Since stored data is in text file format and not large in size, any computer with modern hardware and software should be able to run this game due to its low system requirements.

Coronapoly games are being developed by using the Java language. For this reason Java Runtime Environment (JRE) is needed as a software requirement to be able to run the game. Users need to install JRE to be able to run the jar executable of the game.

2.3 Persistent Data Management

Coronapoly will not be an online game and for this reason data of the player will be recorded and processed locally. Status of the game and data of the players will be stored in a save file which will be stored in the hard disk of the user. Exact location will be designed as users directory of the machine independent from the operating system of the machine. For windows machines exact location will be decided as c:\users\username while for linux machines /home/username will be decided. Save file type will be a text file. This save file will include data like current money of each player, players turn, players properties and their current positions on the map. Its format will be similar to JSON but it will be custom and have its own differences as it is implemented. Settings data will be stored inside a different save file to be easily modified before, during or after execution. Sound files used in game will be stored as .waw format while image files used in game will be stored as .gif format.

2.4 Access Control and Security

Security is not a big concern for Coronapoly game because it is not an online game and everything is executed locally on a computer. For this reason in this game there will not be a need to use an internet connection to a server or database. Installing game on a computer is enough to play the game. Since this is not an online game but a local game, there is no attempt to create a security system to prevent cheats, leaks or attacks. Not being an online makes it hard for an attacker to attack or gain access to the machine or the game. If user wants to cheat in a game, that will not create any issues because all of data belongs to players computer and stored locally. No one will be affected from this except the player. That’s the user's choice about how they want to play the game. However, there might be some cheatings in a multiplayer game whenever a human player leaves the computer to go to the toilet or to grab a cup of coffee. While that player is away from the computer, other human players can open the trade screen and transfer all wealthy to themselves. To avoid that every user will have a simple password to confirm the trade.

2.5 Boundary Conditions

Coronapoly does not to be installed to be playable but instead it will have a .jar executable. Game will be executed by opening the jar file of the game. Jar file preferred over .exe executable to make the game portable for machines with different operating systems. Coronapoly game can be downloaded from the internet or copied from a different computer to be played on the intended computer.

After the Coronapoly game is executed, the initialization stage will begin. Game will check for if there are any previous save files on the computer. If there are no save files are detected in the save folder, then a new save file will be created in the save folder. If there are save files in the save folder, data stored in the save files will be loaded and initialized. Settings will be loaded from the save files as well during the initialization. Sound files and images will be loaded into the program during initialization.

The Coronapoly game can be terminated using the exit buttons found on the pause and main menus of the game. Close button on the window of the game can be also used to exit the game. If the user wants to exit the game, then the data of the player and status of the game will be recorded to the save file. Java garbage collector will deal with releasing the memory and deleting objects of the game.

If the game collapses during the execution, progress of the player may not be saved. This can also happen if the process is terminated using the close button of the window or by another program. Also crash of the program can corrupt the data stored in the save file. Backups of the save files and auto saves during execution may prevent these issues from happening. This feature will create a copy of the save file to the same directory during the execution and will delete that copy after the controlled termination after we implement it.

3 Low-level design

3.1 Object Design Trade-offs

Speed against Size

For our game we value speed more than size in the memory. Constant freezes or drops may annoy the player more than the larger size. Speed is important to convey a more fluent and less problematic gameplay experience to players and size can be traded for this cause.

Functionality against Usability

This game might be simple from the surface but as it goes deeper it gets more complex as interactions between players with players and players with properties increases. For this reason functionality plays a more important role than usability. If the game is not functional as pretended then problems may occur during the interactions which are the important aspect of the game since Coronapoly is a strategy game.

Security against Usability

To play this game, the player does not need to provide personal or critical data. Also Coronapoly is not an online game so they do not need to create or connect an account to the game. This game is meant to be played locally. For this reason security comes after the usability. Usability is more important because fun of the game comes from easy control and interactions. Simple UI is created to increase usability. Also How To Play screens increases usability to even further.

Understandability against Functionality

In Coronapoly, both understandability and functionality play important roles and are required for the final product. Yet functionality comes before the understandability because understandability is about ease of understanding the function of something. Functionality is about if that is functioning as intended or not. This game involves a lot of interactions between players, properties and the game itself. Understanding something can be redundant if that thing is not

functioning properly. This is important since the most part of the game is about interactions between the players and other units of the game.

3.2 Final Class Design

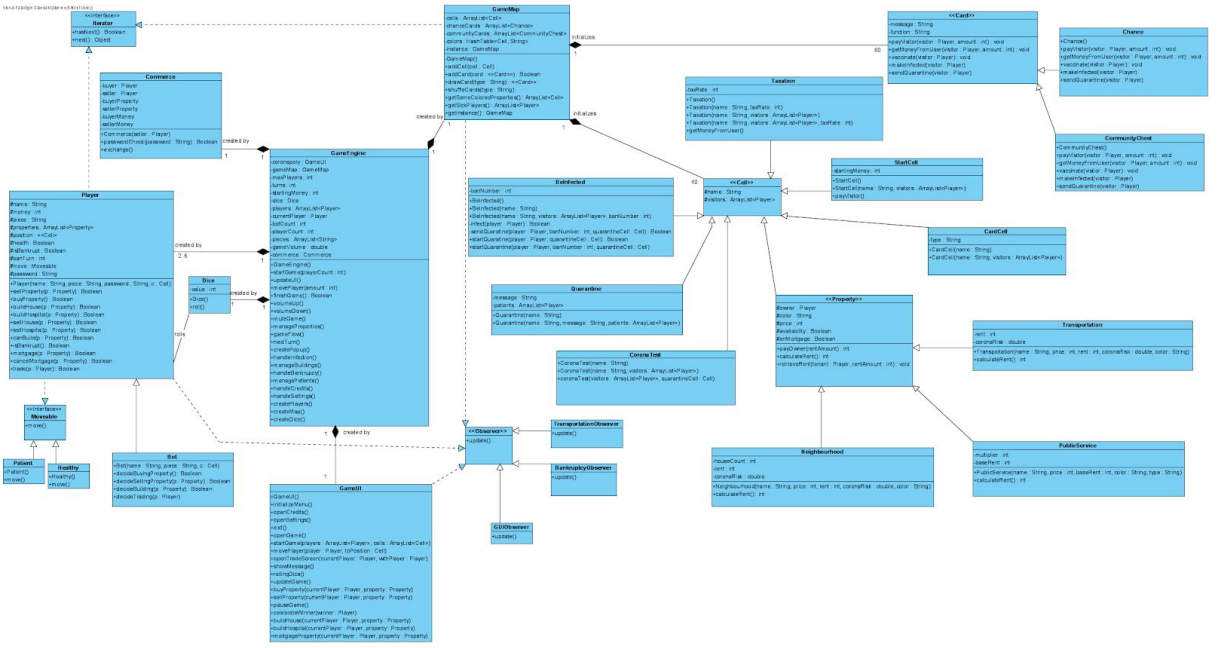

Figure 3.2.1: Final Object Model

3.3 Design Patterns

Façade

The façade pattern is an efficient pattern to lower the coupling between classes or subsystems as well as it has a disadvantage. The GameEngine class functions as a façade pattern. To exemplify, Player class cannot make any relations with GameMap. However, GameEngine can provide relations between Player and GameMap within itself. Also, GameUI cannot directly retrieve the classes which are GameMap, Player, etc. It can only reach them by GameEngine class. The disadvantage is not being able to communicate directly between classes. The GameEngine will always interfere the interactions between subsystems, cause more time and memory.

Observer

The observer pattern is used for being able to react directly and efficiently to the changes that occur on the system. Beside, more classes implement s observer class can be added to the system dynamically and without changing any code on the system. In our system, there are three observer classes that have been used for different and specific purposes. TransportationObserver class will observe the transportation cells. If there is a player who comes to a transportation cell and doesn't buy it, there will be an auction to sell that transportation cell to the other players. BankruptcyObserver class will observe if there is a new player who just became bankrupt. Then, it will create a popup to say goodbye to that user. GUIObersever will observe the changes that must be updated in the user interface.

Singleton

The singleton pattern is used to create a safe developing interface and ensure that there will be one instance of the particular class. We used singleton pattern for the GameMap class because there will be one GameMap class and it will be handled or processed in the GameEngine class. By this pattern, we have ensured that GameMap cannot be used by other classes and created a safe developing area.

Iterator

The iterator pattern is a highly useful design pattern that will be used in our system. It is used for iterating a class instance directly. It has been included in our system because we need an efficient and readable implementation in our code for the classes of GameMap and Player.

Strategy

The strategy pattern provides a flexible and efficient implementation of a program. It is included in the Player class because there are two ways of moving in the map for a player. The first way is to not move because a player can be in the quarantine. The other way is to move according to the dice that the player rolled. While moving the player, we will just use the move method and it will act according to the class that it implemented Moveable interface. Also, there will be a huge flexibility since the type of moving is decided before and implemented in different class. The possible disadvantage is having more class in our system.

3.4 Packages

There are some packages that facilitate the system to be implemented.

java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)[1].

javafx.scene

Provides the core set of base classes for the JavaFX Scene Graph API[2].

javafx.scene.input

Provides the set of classes for mouse and keyboard input event handling[3].

javafx.scene.image

Provides the set of classes for loading and displaying images[4].

javafx.animation

Provides the set of classes for ease of use transition based animations[5].

javafx.event

Provides basic framework for FX events, their delivery and handling[6].

javafx.stage

The JavaFX Stage class is the top level JavaFX container[7].

java.io.File

An abstract representation of file and directory pathnames[8].

3.5 Class Interfaces

Cell Abstract Class

This class represents the main structure of cells which are neighbours, public services, card, corona testing slot, etc. The abstraction of this class is crucial to avoid code smell. Abstract Cell class contains the essential and common properties and methods. All of the cells in the game inherit this class and use its properties and methods.

Properties:

protected String name: This represents the name of the cell. The name will be shown on the game interface. It is protected to be shared with the classes that inherit Cell class. This property has getter and setter methods.

protected List<Player> visitors: It is the players who are on the cell. It is an array list because there might be more than one players who stand on the same

cell. It is protected to be shared with the classes that inherit Cell class.This property has getter and setter methods.

Methods:

public void addVisitor(Player p): This class adds a player as a visitor in a cell's visitor list when a player comes to a cell.

BeInfected Class

This class represents the infection cell on the game. In the infection cell, there is no chance that players who stand on the cell couldn't get the coronavirus. Visitors are directly affected by the disease and sent to the quarantine cell to wait for some turns. This process is handled by this class.

Properties:

private int banNumber: It is the number of turns that are forbidden to move for players who stand on the BeInfected cell and will be sent to the Quarantine cell. It is not shared with any class. This property has getter and setter methods.

Constructors:

public BeInfected(String name, List visitors): This is the constructor which doesn't change the default value of ban turn but takes new values to the name and visitors properties.

public BeInfected(String name, List visitors, int banNumber): This is a detailed constructor that enables programmers to create an instance of this class with specified values of the name, visitors and ban turn properties.

Methods:

private Boolean infect(Player player): This method takes the player who stands on the BeInfected cell as parameter. Then, It changes the health condition of the player. This means that people have the virus now. This is a private function to be used in the startQuarantine() method.

private Boolean sendQuarantine(Player player, int banNumber, Cell quarantineCell): This method takes the banTurn, quarantine cell and player who stand on the BeInfected cell as parameters. Ban turn of the player is decided and the cell that the player will start its quarantine is given in the parameters. This is a private function to be used in the startQuarantine() method.

public Boolean startQuarantine(Player player, int banNumber, Cell quarantineCell): This is the main function that private functions will be used for infecting the player and sending it to the quarantine cell. Also, banTurn can be adjusted within the parameters.

public Boolean startQuarantine(Player player, Cell quarantineCell): This is the same function as above. The only difference is that this method uses the class' own banTurn property.

Quarantine Class

The instances of this class keep the patients who are diagnosed with coronavirus. There is a certain amount of time that patients will be kept in the cell. Visitors will just pass this cell on the contrary to the patients.

Properties:

private String message: This is the message that will be sent to the coronavirus patients who are obligated to stay here for a specified amount of ban turns in this cell by the visitors. This property has getter and setter methods.

private List patients: The patients are players that are infected with the virus and have to stay in this cell for a specified amount of ban turns. This property has getter and setter methods.

Constructors:

public Quarantine(): This is the default constructor that has default message and initialization of patients.

public Quarantine(String message, List patients): This is the constructor that can be given specific values to the properties of this class.

CoronaTest Class

This class' aim is to find every coronavirus carrier and put them into a quarantine cell. Therefore, it tests for every visitor and executes its aim according to the results of the tests.

Constructors:

public CoronaTest(String name, List visitors): This is the constructor that can be given specific values to the properties of this class.

Methods:

public Boolean coronaTest(List visitors, Cell quarantineCell): This method iterates every player who is on this cell. Then, it makes a corona test for every visitor and results are announced right away. Players with positive test results are sent to the quarantine cell which is specified in parameters. Also, ban turn of the players is specified in the method. Players with negative test results can continue their game.

Taxation Class

This class creates a cell that enforces every visitor of this kind of cell to pay money proportional to their capital. Tax rate will determine the amount of money that will withdraw from the player.

Properties:

private double taxRate: This is a percentage which will be multiplied by the money that visitors have. This property has getter and setter methods.

Constructors:

public Taxation(int taxRate): This is the constructor that has default name and initialization of visitors and can be given specific value to the taxRate property of this class.

public Taxation(String name, List visitors): This is the constructor that can be given specific values to the name and visitors.

public Taxation(String name, List visitors, int taxRate): This is the constructor that can be given specific values to the taxRate, name and visitors.

Methods:

public void getMoneyFromUser(): This method takes the players that stand on the cell and iterates them to take tax which is taxRate percentage of their whole capital.

StartCell Class

This class is basically the initial cell where all players begin the game. Also, it provides players a steady income as it gives a certain amount of money in every time players complete a tour around the game board.

Properties:

private final int STARTINGMONEY: This is the money that players will earn whenever they reach or pass the starting cell.

Constructors:

public StartCell(): This is the default constructor that has default name and initialization of visitors.

public StartCell(String name, List visitors): This is the constructor that can be given specific values to the properties of this class.

Methods:

public void payVisitors(): This method takes the players that stand on or pass the cell and iterates them to give starting money. Money will be added to their current capital.

CardCell Class

This class is inherited from the cell class. Represents the map cells for Community Chest and Chance cards. It is just a basic representation containing type of the card that will be drawn from the deck.

Properties:

private String type: Denotes the type of the CardCell and the card that will be drawn from the deck. Can be either “Chance” or “CommunityChest”.

Constructors:

CardCell(String name): Initializes the cell with only name. Type can be set later.

CardCell(String name, String type): This is the constructor that can be given specific values to the properties of this class.

Card Abstract Class

This is the abstract parent class of the CommunityChest and Chance classes which will be both derived from. Instances of this class will be contained as CommunityChest or Chance type in the GameMap class. It is the chance center of the game. Card class can initialize lots of different objects that can create significantly different results to the players.

Properties:

private String message: This is the message that will be displayed whenever a player stands on the card. This property has getter and setter methods.

private String cardFunction: This is the variable that decides for the function about what method to execute. This property has getter and setter methods.

Constructors:

public Card(String name): This is the constructor which forces programmers to give a name. The reason is that there are just two different names that this class can have. Visitors property is also initialized.

public Card(String name, List visitors): This constructor takes the name and visitors as parameters.

public Card(String name, List visitors, String message, String cardFunction): This constructor gives specified values to the each property of the class.

Methods:

public void executeCardFunction(String cardfunction): This function takes the cardfunction string to decide which method to use inside of it. Every decision of action that a Card class can apply on a player is decided in this function.

private void payVisitor(Player visitor, int amount): This method takes player and amount as parameters. The value of the amount will be added to the visitor's whole capital.

private void getMoneyFromUser(Player visitor, int amount): This function will take money from the player given in parameters. The amount of money that will be retrieved is the amount in the parameter.

private void getMoneyFromUser(Player visitor, double rate): This function will take money from the player given in parameters. The amount of money that will be retrieved is calculated by the multiplication of the player's whole capital and rate given in the parameter.

private void vaccinate(Player visitor): This method makes the player immune to the coronavirus for a few turns.

CommunityChest Class

This class represents the community chest type of cards. It extends Card abstract class. Most of the methods will be the same with chance type of cards.

Constructors:

public CommunityChest(String message): Initializes the card with an only message to be shown to the player. Function can be set later.

public CommunityChest(String message, String function): This constructor gives specified values to the each property of the class.

Chance Class

This class represents the chance type of cards. It extends Card abstract class. Most of the methods will be the same with community chest type of cards.

Constructors:

public Chance(String message): Initializes the card with an only message to be shown to the player. Function can be set later.

public Chance(String message, String function): This constructor gives specified values to the each property of the class.

Property Abstract Class

This is the abstraction class to write a clean code and construct a successful inheritance. This class represents the majority of the cells which have rent income.

Properties:

protected Player owner: This property represents the player that owns this cell. This property has getter and setter methods.

protected String color: This property indicates the color that the cell will be painted. This property has getter and setter methods.

protected int price: This is the amount of money needed to buy this place. This property has getter and setter methods.

protected Boolean availability: This is the indicator that shows whether this cell is ready to be bought or not. This property has getter and setter methods.

protected Boolean onMortgage: This property shows the condition of the place. This property has getter and setter methods.

Methods:

public void payOwner(int rentAmount): This method is triggered whenever a visitor stands on this place. It gives the rent amount to the owner of this property.

public void retrieveRent(Player tenant, int rentAmount): This method withdraws money from the visitor or tenant of this class.

public abstract int calculateRent(): This is an abstract method to be implemented by the classes that inherit this class.

PublicService Class

This class stands for the public services that players will pay money or operate the place. Chance factor is important in this cell since the amount of rent that the owner of this place will get depends on the dices that the visitor rolls.

Properties:

private int multiplier: This property is equal to the summation of dices that the player has rolled before it comes to this cell. This property has getter and setter methods.

private int baseRent: This is the base value that will be multiplied by multiplier. This property has getter and setter methods.

Constructors:

public PublicService(int price, int baseRent, String color): This is the constructor that determines the price, base rent and color of the cell. Also, it declares inherited properties such as owner, availability, onMortgage and name.

Methods:

public int calculateRent(): This function calculates the rent by multiplying the amount of base rent with multiplier. Also, rent is doubled if both public services are owned by the same player.

Transportation Class

This class is a type of cell that stands for the transportation options. Every transportation option has a price to be bought and increases its rent if a player owns more than one transportation cell.

Properties:

private int rent: It is the amount of money that visitor will pay to the owner. This property has getter and setter methods.

private double coronaRisk: This is the risk of being infected by coronavirus in the cell. This property has getter and setter methods.

Constructors:

public Transportation(String name, int price, int rent, double coronaRisk, String color): This is the constructor that determines the name, price, rent, color and risk of being infected with coronavirus of the cell. Also, it declares inherited properties such as availability, owner, onMortgage and name.

Methods:

public int calculateRent(): This method calculates the rent according to the type of the transportation and the number that the owner of this place has transportation cells.

Neighbourhood Class

This class represents a neighbourhood from Ankara. The players can buy this property to have a steady income based on visitors that come on the cell. Besides, owners can develop the neighbourhood which causes rent to rise.

Properties:

private int houseCount: This is the number of houses on the property. This number increases the rent. This property has getter and setter methods.

private int rent: This property is the amount of money that a visitor will pay if it comes to it. This property has getter and setter methods.

private double coronaRisk: This is the risk of being infected by coronavirus in the cell. This property has getter and setter methods.

Constructors:

public Neighbourhood(String name, int price, int rent, double coronaRisk, String color): This is the constructor that determines the name, price, rent, color and risk of being infected with coronavirus of the cell. Also, it declares inherited properties such as availability, owner, onMortgage and name.

Methods:

public int calculateRent( ): This function calculates the rent according to the houses that stand on the cell. There is a base rent if there is not a house.

Player Class

Player class is responsible for the players which have the biggest role in a Monopoly game. One of the players will be the monopoly with using his/her move. Therefore, there are a lot of attributes and methods in this class.

Properties:

private String name: Before the beginning of the game all players must enter a name and this attribute represents names of the players. This property has getter and setter methods.

private int money: All players have money and this money is not being stable. This attribute represents money that the users have. This property has getter and setter methods.

private String piece: There are pieces to represent users on the game board. All players must choose a piece before starting. This property has getter and setter methods.

private List properties: This is an array list to put properties for each user.

private Cell position: There are cells on the game board and this attribute represents the cell that user on it. This property has getter and setter methods.

private Boolean health: This represents the health situation of the users. Users can be covid-19 positive or negative. This property has getter and setter methods.

private Boolean isBankrupt: This represents the bankruptcy status of the user. If the value is true, this player loses the game. This property has getter and setter methods.

private int banTurn: If a user in quarantina he/she is banned for three turns. This represents when the player can play again. This property has getter and setter methods.

private Moveable move: If a user in quarantina he/she cannot move, otherwise movement is allowed.

private String password: If a user in a trade situation, he/she must enter the password correctly. This property has a getter method.

Constructor:

Player(String name, String piece, String password, Cell c): The constructor takes a name, a piece, a password and a cell as a parameter and creates a user with this name, piece, password and a cell which is the start cell. Other attributes also set in the constructor like users' money should be equal to the starting money.

Methods:

public Boolean sellProperty(Property p): Players can sell their property when they run out of the money with this method.

public Boolean buyProperty(): Players can buy a property when they are on the properties' cell if they have enough money to buy it.

public Boolean buildHouse(Property p): This method builds a house on a property if canBuild() method returns true.

public Boolean buildHospital(Property p): This method builds a hospital on a property if canBuild() method returns true.

public Boolean canBuild(Property p): This method gets the availability to build a house or hospital on a property.

public Boolean sellHouse(Property p): This method sells a house on a property when they are running out of the money with this method.

public Boolean sellHospital(Property p): This method sells a hospital on a property when they are running out of the money with this method.

public int rollDice(): This method roll dice on players turn and return an integer.

public Boolean mortgage(Property p): Players can mortgage properties with this method.

public Boolean cancelMortgage(Property p): Players can cancel the properties' mortgage with this method.

public Boolean trade(Player p): Players can trade their properties with other players using this method.

Bot Class

Bot class is responsible for non-human players. When there are not enough people to play, users can play Coronapoly with bots that have decision mechanics.

Constructor:

Bot(): Constructor of the bot class takes no argument. Their names and pieces are taken randomly and set other attributes as Player class' constructor.

Methods:

public Boolean decideBuyingProperty(): This method makes a decision to buy a property that bot is on and if returns true bot will buy the property using buyProperty method.

public Boolean decideSellingProperty(Property p): This method makes a decision to sell a property and if returns true bot will sell the property using sellProperty method.

public Boolean decideBuilding(Property p): This method makes a decision to build a building and if returns true bot will build a house or hospital using buildHouse or buildHospital method.

public Boolean decideTrading(Player p): This method makes a decision to trade with other players and if returns true bot will make trade with using trade method.

Commerce Class

This class carries out operations with given arguments triggered by the game engine class. It manages the commerce operations such as trading

properties between players and auctioning certain properties. An instance will be created when one of the said operations needs to be executed.

Properties:

private Player buyer: The player who sends the trade offer to another player.

private Player seller: The player who receives the trade offer and answers to the offer.

private Property buyerProperty: Offered property against the requested property of the seller.

private Property sellerProperty: Requested property against the offered property of the buyer.

private int buyerMoney: Offered money in order to compensate value difference between properties against the requested property of the seller.

private int sellerMoney: Requested money in order to compensate value difference between properties against the offered property of the buyer.

Constructors:

public Commerce(Player buyer): Instantiates the commerce object with the current player of the turn who is the only one capable of sending trade offers.

Methods:

public boolean passwordCheck(): This method checks the trade offered players password for finalization of a trade. Prevents illegal trades of players.

public boolean exchange(): This method finalizes the trade and exchanges properties and money between given players.

GameMap Class

This class represents the game board of Coronapoly that has cells of the quarantine, test centers etc. This is the center of the game that players move on to play Coronapoly. Firstly, all players shown on the start cell and a player who has the greatest score from dice will move first.

Attributes:

private List cells: This represents the cells of the game map and they are initialized when the start game button is pressed. Cells can be neighbourhood, public services etc. Each cell has a name and players can move on it as a visitor.

private List chanceCards: There are two types of cards in the game and they are shown on the game board. This attribute represents the chance cards. When a player comes to a card cell, he/she will take a card and must do the instruction on the card. This attribute has a getter method.

private List communityCards: This attribute represents the community chest cards. This attribute has a getter method.

private Hashtable<Cell, String> colors: Neighbourhoods have a color and players can build houses or hospitals when they have all properties with a single color. This table represents all colors. This attribute has a getter method.

private GameMap instance: This attribute contains the only instance of the game map object and helps us to create the singleton design pattern. This attribute has a getter method.

Methods:

public List getSameColoredProperties(): This method returns the properties with the same color as an arraylist. Players can build houses or hospitals when they have all properties with a single color.

public List getSickPlayers(): This method returns all the sick players as an arraylist. Every player can see which player is sick.

public Boolean addCell(Cell cell): This method adds cells on the game board before the game starts.

public Boolean addCards(Card card): This method adds cards on the game board before the game starts.

public Card drawCard(String type): When a player comes on a card cell, he/she must draw a card and do the instruction. This method returns a card for this player.

public void shuffleCards(String type): This method basically shuffle the cards on the game board when the game starts.

GameUI Class:

This class will be responsible for initializing and adjusting the graphics of the game elements on the screen. JavaFX objects and functions will be utilized in this class. Will be driven mostly by the directives of the game engine class.

Constructors:

public GameUI(): A basic constructor for the class taking no arguments. Will be creating a game scene with the predefined values.

Methods:

public void initializeMenu(): Initializes menu elements and places them accordingly.

public void openCredits(): Opens the credits screen where people worked on the development of the game is listed.

public void openSettings(): Opens the settings menu where the user can change the game settings such as volume.

public void exit(): Runs on clicking exit button. Terminates the game screen.

public void openGame(): Opens a menu where the user can adjust player count, bot count and starting money.

public void startGame(List players): Runs on clicking start game button after adjusting game variables mentioned in openGame() method. Initializes and opens the game scene where the game board and elements will be present.

public void movePlayer(Player player, Cell toPosition): Moves the given player to a position on the game board. Mostly will run after rolling dice but sometimes it will be used for moving the player to the quarantine.

public void openTradeScreen(Player currentPlayer, Player withPlayer): Opens a trade pop up for the current player with the player he chooses to trade.

public void showMessage(): Will be used for showing the message of the drawn card

public void rollDice(): Triggers the dice rolling animation.

public void updateGame(): Updates the state of the game board between turns.

public void buyProperty(Player currentPlayer, Property property): Adds the bought property to the inventory of the current player.

public void sellProperty(Player currentPlayer, Property property): Removes the sold property from the current player's inventory

public void pauseGame(): Basically pauses the game and shows a resume or exit dialog box

public void celebrateWinner(Player winner): Triggers the celebration animation for a given player

public void buildHouse(Player currentPlayer, Property property): Puts a house on the property after the current player builds a house on that property.

public void buildHospital(Player currentPlayer, Property property Puts a hospital on the property after current player builds a hospital on that property.

public void mortgageProperty(Player currentPlayer, Property property): Puts given property of the current player in the mortgage state.

GameEngine Class:

This class is one of the most important classes in the project. It basically represents the brain of the Coronapoly. This class will be responsible for controlling every action taken by the both user and the game. Every single operation will be happening inside the engine and with the use of GameUI class they will be presented to users.

Attributes:

private final int MAX_PLAYERS: This attribute represents the maximum number of players to play Coronapoly. This attribute has a getter method.

private final int STARTING_MONEY: Amount of the money that will be given to players at the start of the game. This attribute has a getter method.

private int playerCount: Number of the players that will be playing the game in the current session. This attribute has a getter and a setter method.

private int botCount: Number of the computer controlled player that will be added for the current session. This attribute has a getter and a setter method.

private int turns: Number of the turns taken from the start of the game session. This attribute has a getter method.

private float gameVolume: Current game volume. Will be at 50% initially. This attribute has a getter and a setter method.

private GameUI gameUI: Reference to the gameui component. This attribute has a getter method.

private GameMap gameMap: Reference to the gamemap component. This attribute has a getter method.

private List players: Reference to all players. This attribute has a getter method.

private Player currentPlayer: This variable holds the current player of the turn. This attribute has a getter method.

private Dice dice: Reference to the dice object. This attribute has a getter method.

Constructors:

public GameEngine(): Default constructor for the game engine taking no parameters. Most of the class variables will be adjusted on the start of the game.

Methods:

public void startGame(int playerCount): Intializes a game session with the given parameter.

public void updateUI(): Triggers game ui to update between each turn.

public void movePlayer(Cell toCell): Moves current player to the given cell. Also triggers the corresponding operations in the GameUI component.

public void finishGame(): Ends the game if every player is bankrupt except one. Also triggers the corresponding operations in the GameUI component.

public void volumeUp(): Increases the game volume by a unit.

public void volumeDown(): Decreases the game volume by a unit.

public void muteGame(): Makes the game volume zero.

public void manageProperties(): Manages the operations on the properties of the current player. Also triggers the corresponding operations in the GameUI component.

public void gameFlow(): Updates the game state between turns. Also triggers the corresponding operations in the GameUI component.

public void nextTurn(): Prepare game for the next turn and get to the next turn. Also triggers the corresponding operations in the GameUI component.

public void createPopup(): Creates a pop up to get user interaction. Also triggers the corresponding operations in the GameUI component.

public void handleInfection(): Checks the infection risk of the cell and adjusts infection status of the current player. Also triggers the corresponding operations in the GameUI component.

public void manageBuildings(): Manages the operations on the buildings of the current player. Also triggers the corresponding operations in the GameUI component.

public void handleBankruptcy(): Checks the bankruptcy status of each player and adjusts their bankruptcy status accordingly. Also triggers the corresponding operations in the GameUI component.

public void managePatients(){} Manages the players in quarantine. Sets them free if they took enough turns in the quarantine. Also triggers the corresponding operations in the GameUI component.

public void handleCredits(): After getting a user input triggers GameUI to open the credits scene.

public void handleSettings(): After getting a user input triggers GameUI to open the game setting menu.

public void createPlayers(): Initialized players.

public void createMap(): Initializes the game map for the current game session.

public void createDice(): Initializes the dice object for the current game session.

Dice Class:

This class represents a basic dice for the coronapoly game. It will be functioning exactly as it is in the table-top version of the monopoly game.

Attributes:

private int value: This value represents the value we get after rolling the dice.

Constructors:

public Dice(): Basic constructor for the dice class with no parameters.

Methods:

public int roll(): Rolls the dice for the current user and returns its value.

Moveable Interface

This interface represents the movement capability of a player in terms of players' health.

Methods:

public void move(): Move the player or not with looking at her/his health situation.

Patient Class

This class represents the movement of a player that is patient. It implements the Moveable interface.

Methods:

public void move(): This method does not allow the player to move.

Healthy Class

This class represents the movement of a player that is healthy. It implements the Moveable interface.

Methods:

public void move(): This method allows the player to move.

Iterator Interface

This interface represents navigating and editing sequences. GameMap class and Player class implement this interface.

Methods:

public Boolean hasNext(): This method checks whether the sequence has an element at next index or not and returns a boolean.

public Object next(): This method returns the next element in a sequence.

Observer Abstract Class

This class helps us in creating observer design pattern. Some of the elements in the game such as gui, bankruptcy status, auctioning of the

transportation properties must be updated when a certain actions occur. So this structure helps them to listen the changes that are required to trigger operations of the said objects.

Methods:

public void update(): Updates the current object according to the triggering that is received from the observed objects.

GUIObserver Class

This class helps us to listen to the game manager and update the user interface of the game every turn.

Methods:

public void update(): Overrides the observer class' method.

BankruptcyObserver Class

This class helps us to listen to the players' bankruptcy situation to check whether the game is over or not every turn.

Methods:

public void update(): Overrides the observer class' method.

TransportationObserver Class

This class helps us to listen to the players' offer in auction when a player comes to a transportation cell but he/she does not buy it.

Methods:

public void update(): Overrides the observer class' method.

4. Improvement Summary

Some improvements are done for the second iteration of this report according to given feedback. For the criteria and trade-off parts, no changes were made because there were no obvious issues or feedback given for those parts of the report. In the hardware software mapping part, It was forgotten to mention that the game is going to be played on a single machine as a singleplayer game. This is now mentioned in the report according to feedback. Also in this part development and running issues were confused and JAVAFX and JDK were mentioned. These statements about JAVAFX and JDK are now removed and only JRE is mentioned as a software requirement for this game. In the persistent data management part some details about the save file was missing. Planned exact location of the save file for both windows and linux operating systems are now being mentioned in the report. Also the contents and structure of the save file is detailed according to feedback.

Format-wise improvements were made concerning font sizes, figure captions, title placement and other small improvements to provide an easy to follow, better reading experience in the report.

Design patterns section is added in the table of contents and there are five design patterns choosed to implement the system. They are all described in a detailed way and by giving concrete examples in class diagram.

In the Access Control and Security section, a new function to secure the game experience has been updated. Trading action is secured by the

new class Commerce. This was a need when a human player is away from the keyboard in the multiplayer game.

In the class interface section, there are some additions to the classes. First of all, there are new classes to implement design patterns such as Observer, Moveable, etc. Secondly, there are some new classes to increase the functionality of the system such as Commerce and Card. For example, Commerce class will handle all the commercial acts between players.

5. Glossary & references

[1]https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

[2]https://docs.oracle.com/javase/8/javafx/api/javafx/scene/package-summary.html

[3]https://docs.oracle.com/javase/8/javafx/api/javafx/scene/input/package-summary.html

[4]https://docs.oracle.com/javase/8/javafx/api/javafx/scene/image/package-summary.html

[5]https://docs.oracle.com/javase/8/javafx/api/javafx/animation/package-summary.html

[6]https://docs.oracle.com/javase/8/javafx/api/javafx/event/package-summary.html

[7] https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html

[8] https://docs.oracle.com/javase/7/docs/api/java/io/File.html