CS 319 - Object-Oriented Software Engineering

Design Report

Ping The Risk

GROUP-3J

Denizhan Kemeröz

Ahmet Ayberk Yılmaz

Süleyman Semih Demir

Mustafa Tuna Acar

Table of Contents

1. Introduction

1.1. Purpose of the system

Ping the Risk is a turn based multiplayer cyberwarfare simulator which will be played on desktop which is expected to be an entertaining game. Aim of the players, which are called hackers, is to attack the other countries to conquer the world.

In order to achieve entertainment, the system should have well-designed, user-friendly interface, no bugs, along with smooth gameplay. One of the noteworthy features of Ping the Risk is that the game is multiplayer. In order to maintain this feature, the connection of the players must be flawless.

1.2. Design goals

1.2.1. High-performance

High-performance means that maximum speed and minimum memory usage. While designing Ping the Risk we aimed to system to be as fast as possible for the best user experience while using the lowest amount of memory.

1.2.2. Reliability

Reliability means that the minimum difference between specified and observed behavior. Ping the Risk designed to be reliable because the opposite means that the game will perform unwanted operations and a frustrating experience for players.

1.2.3. Availability

Availability means that percentage of time that system can be used to accomplish normal tasks. Ping the Risk should be available as much as possible because it is an online multiplayer game so the server needs to be up to play the game.

1.2.4. Extensibility

Extensibility means how easy it is to add functionality or new classes to the system. While designing Ping the Risk we used object-oriented programming so adding new classes without interrupting the system is possible. This is important for adding new features to the system.

1.2.5. Modifiability

Modifiability means how easy it is to change the functionality of the system. We designed Ping the Risk multilayered because in the lifetime of the system certainly there will be some parts that need to be changed. The subsystems have low coupling which means they can be modified without affecting other subsystems.

1.2.6. Adaptability

Adaptability means how easy it is to port the system to different application domains. We designed Ping the Risk such a way that classes and subsystems are reusable so that the parts of the system can also be used for other systems.

1.2.7. Portability

Portability means how easy it is to port the system to different platforms. It is important for us to run our system in different platforms as players may have different platforms. Therefore, we decided to use Java for the development because Java is a platform-independent language.

1.2.8. Usability

Usability means how easy it is for the user to use the system. We designed Ping the Risk such a way that any player can easily play our game. User-interface is very simple and clear. Also, we added the

"How to Play" section that describes the game very detailed. Thus, no users will have hardships using our system.

2. High-level Software Architecture

2.1. Subsystem Decomposition

We decomposed the whole system of the game into three different subsystems namely User Interface Management, Game Management and Database Management. During the decomposing process, we divided the system in such a way that at the end, each subsystem could be given to a group member. This enables us to create the whole project by integrating various applications constructed by different group members with themselves or with different interfaces and data. It means that after all the subsystems are constructed, there will be tens of applications which are meaningless when they are alone. Here, the key point is reusability of these sub-applications. For example, in the game management subsystem, the package Turn Management can be used for other parts of the similar project or even totally different projects.

The first benefit of decomposing the system for us was that we can change the parts in a subsystem without thinking about any possible errors in different subsystems. In other words, by decreasing the number of dependencies between classes or packages, we are now more relaxed while we need to make a change in a subsystem. For example, if we need to add or remove some things into scenes in the user interface management subsystem, we will consider less possible errors affecting other subsystems. In this way, we achieved to decrease the level of coupling.

The first subsystem, called User Interface Management consists of all scenes in the game which lets the players direct the game menus as they want. Most resources are generally image files such as "background.jpeg" or "header.png". By using the Scene Builder tool, these scenes are created. The classes in this subsystem are affected by the manager classes which are in the second subsystem, called Game Management.

In the Game Management subsystem, there are many classes and methods inside them whose job is to do the actual job, controlling everything in the game such as scenes, turns in the game, hacking part of any turn, fetching or uploading any data to the database etc. This subsystem includes a package named “Turn Management” which includes the classes related to one of the main parts of our game, turns. In general, this subsystem gets the related data from the database and embeds it to the related screen, calculates any data and uploads it to the database. Thus, it acts like a bridge between data and user interface.

The Data Management subsystem simply consists of the classes which are related both to remote and local databases. In this project we will use mySQL database in an existing domain. There are many tables and the classes in this subsystem helps the system to understand the types and values of the data in the database.

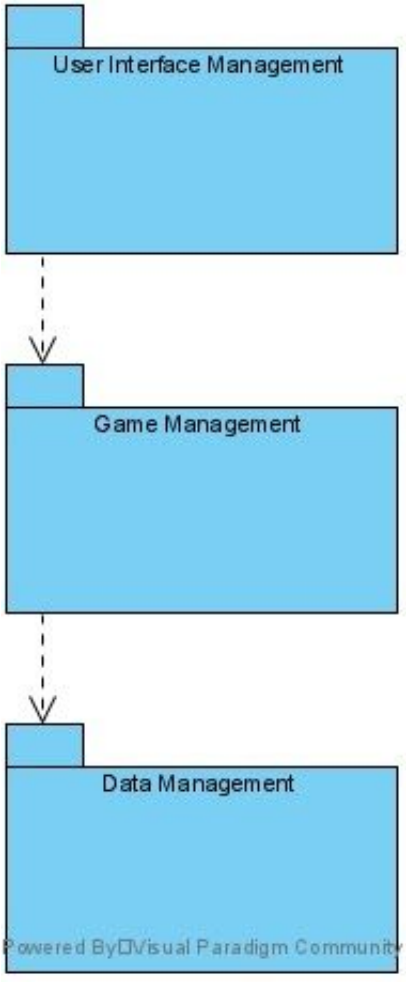

Figure 1: Basic version of subsystem decomposition

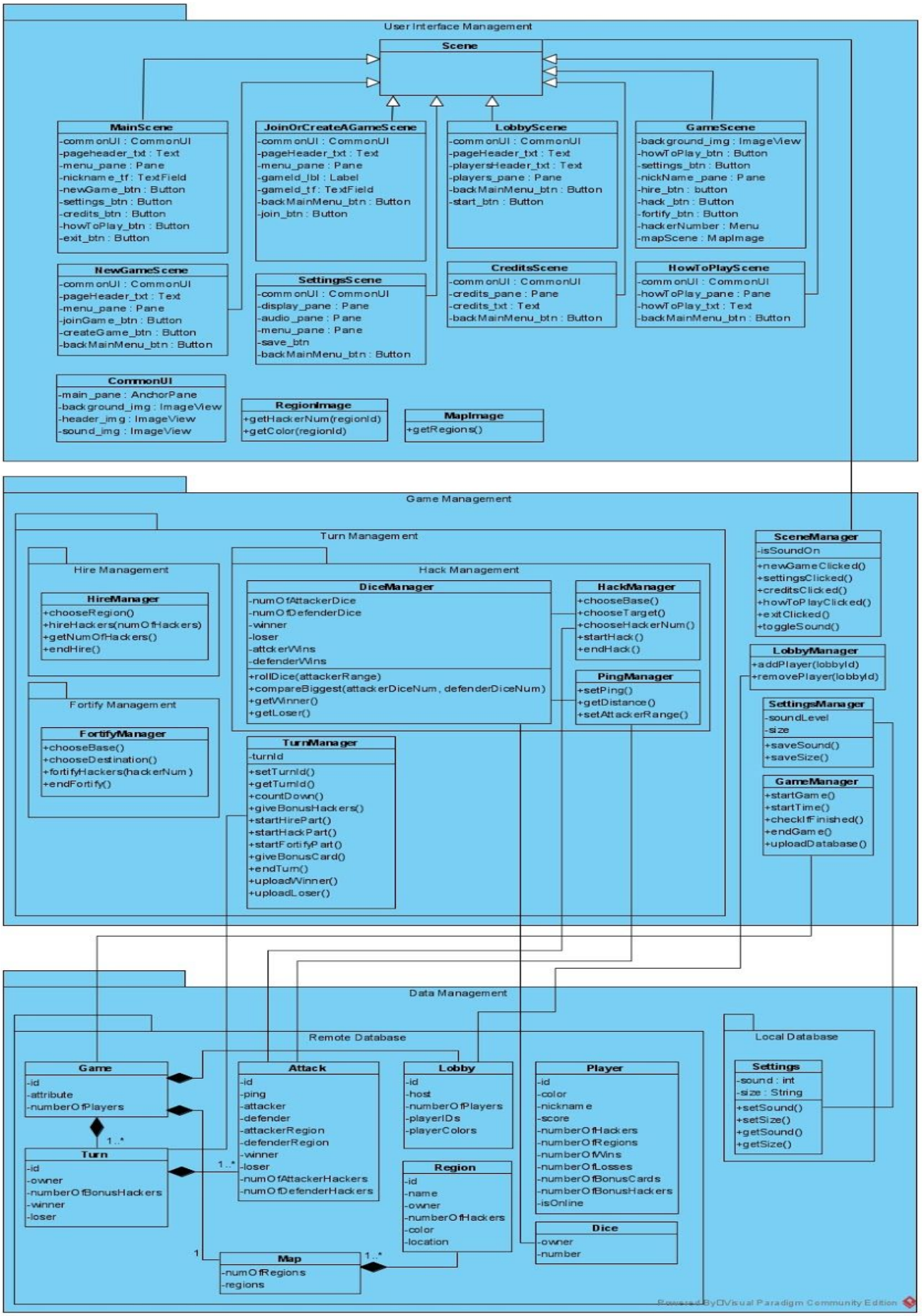

Figure 2: The detailed version of subsystem decomposition

2.2. Hardware/Software Mapping

Ping the Risk game will be implemented in the Java programming language, by using JAVAFX libraries. The game requires the Java Runtime Environment to be used by users who have JDK 8 or newer versions that have JAVAFX libraries.

As a hardware requirement, the user should have a mouse to press the buttons to open the settings, credits, help, and play the game.

These requirements are minimal to play a game so that it can be more accessible and playable for more users

For storage of the game (scores, names, gameplay, settings) our database will be used. It means that people should have internet connection as a one more requirement.

2.3. Persistent Data Management

Since the players will be playing online and the data needs to update for each player simultaneously, the gameplay data will be kept in an online server database. The gameplay data will change according to the events that occur at the end of each turn. After each event, the updated version of data will be returned into the database via MySQL code that contains the current condition of game instances, objects and the map which then will be processed for each of the players in the lobby. The user preferences like sound and display options will be kept in the user hard disk since it does not require to be kept in a database.

2.4. Access Control and Security

Ping the Risk game will use an internet connection, but lobby access is required to access the game with other players, so malicious actions can be prevented ingame. Because of the online usage and the way the data is collected, there might be some security problems. The score will be collected throughout the gameplay, and will be deleted after. That is why the score and the game result cannot be changed after the gameplay is ended.

2.5. Boundary Conditions

Ping The Risk will be launched from a jar file to avoid spending any time at installation and it will increase the portability of the game.

The game can be terminated by pressing the quit button form the esc or main menu. It can also be closed by alt + F4 keyboard shortcut.

If there is any system failure or hardware problem due to corrupt or missing files, the player will not be able to continue because the game does not offer a reconnect to the game function and the lobby that the player dropped from will continue playing until the game is finished.

3. Low-Level Design

3.1. Object Design Trade-offs

3.1.1. Development Time vs. Functionality

While we are designing Ping the Risk we found lots of ideas to add such as single player game mode with AI, multiplayer game mode on the same computer, multiplayer online game mode or different maps. However, we got limited time to develop Ping the Risk so we needed to decide what features we will keep and what features we will let go. At the end we decided to go with multiplayer online game mode with one map. This caused less functionality.

3.1.2. Availability vs. User Friendliness

We designed Ping the Risk as an online game instead of offline. By doing that we gave up on availability because in online games to play the game players will need to have an internet connection and our server to be up and running but in offline they need one computer to run the game and that's it. However, we gave users the chance to play where they want and regardless of the distance between them.

3.1.3. Portability vs High-performance

We decided to implement Ping the Risk in Java language because Java is a platform-independent language. This makes our system portable. However, we could have used for example C++ to maximize the performance but portability is more important for us.

3.2. Final Object Design

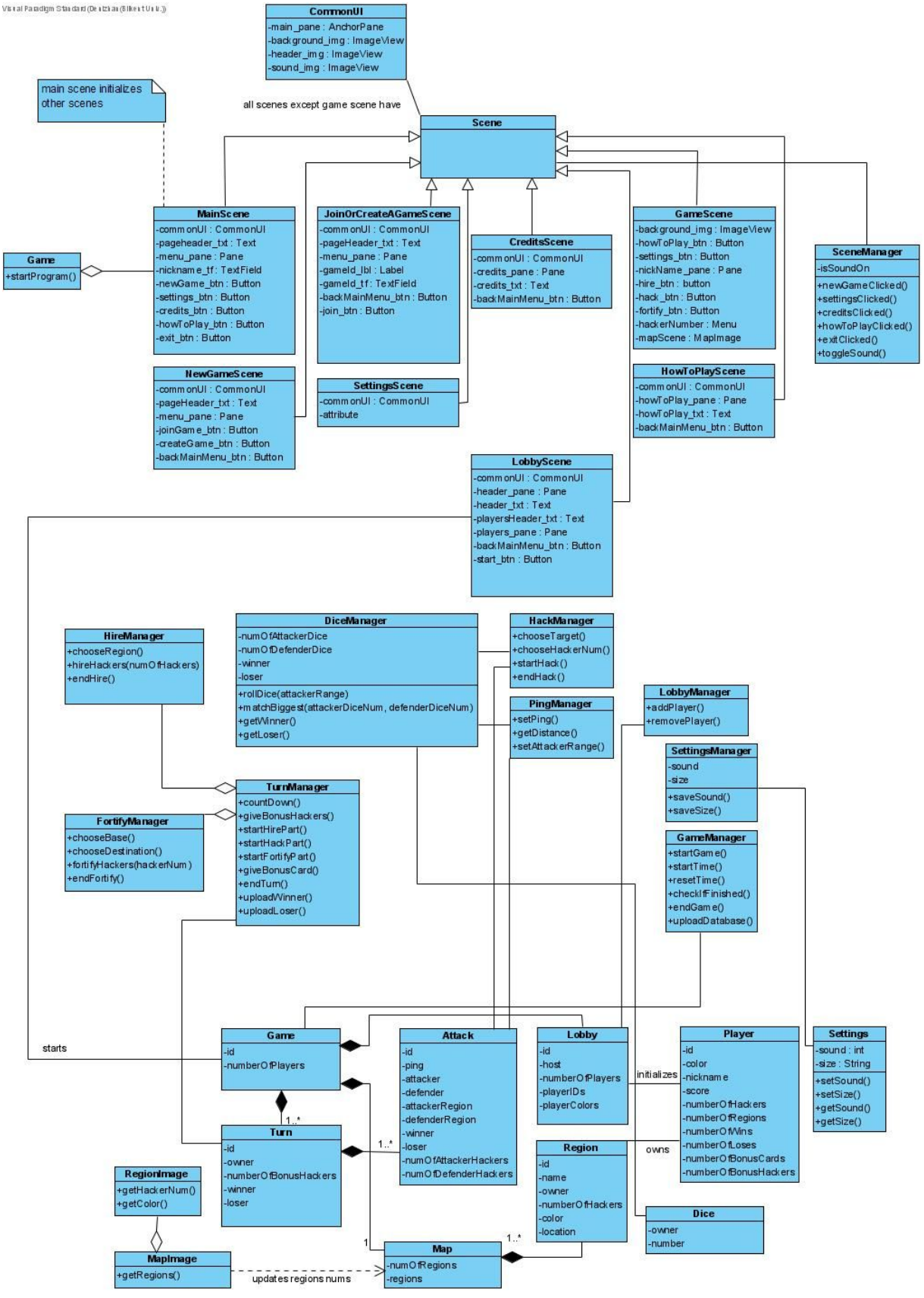

Figure 3: Final Object Design

3.3. Packages

Ping the Risk will use JavaFX library of Java. So, any UI class that is created for the game will be packaged from JavaFX.

3.3.1. Packages Used From JavaFX

javafx.fxml Package: Package that implements object hierarchy

javafx.scene Package: Package that implements Scene class.

javafx.event Package: Package that implements Event class.

java.util.ResourceBundle: Package that contains locale specific items.

javafx.application: Package that enables the usage of Scene and Event class.

3.3.2. Scene Package

3.3.2.1. User Interface Management Subsystem

Scene package contains User Interface Management Subsystem which will be used to navigate through the game screens scene by scene. Every scene will represent a different screen for the game to display. This package will contain “CommonUI”, “MainScene”, “JoinOrCreateAGameScene”, “LobbyScene”, “GameScene”, “NewGameScene”, “SettingsScene”, “CreditsScene” and the “HowToPlayScene”

classes which will also use methods and classes from the JavaFX library.

3.3.3. Gameplay Package

3.3.3.1. Game Management Subsystem

The Gameplay Package will contain the Game Management Subsystem which will allow the user to interact with the game UI while playing the game.This package contains the Turn Management Subsystem which contains its own classes and manages actions and outcomes of the user's decisions at the end of each turn, "SceneManager" class, "PingManager" class, "SettingsManager" class and "LobbyManager" class. Game Management Subsystem depends on User Interface Management Subsystem since the "SceneManager" class that exists in this subsystem also uses the Scene package. "PingManager", "SettingsManager" and "LobbyManager" classes are used for the movement of the data from the Gameplay Package and the Data Package.

3.3.3.1.1. Turn Management Subsystem

Turn Management subsystem exists inside the gameplay system which manages what the user does and the outcome of users decisions. This subsystem contains "DiceManager", "HireManager",

“FortifyManager”, “TurnManager”, “HackManager” and “PingManager” classes which are used to manage the actions and outcomes of the user’s decisions.

3.3.4. Data Package

3.3.4.1. Data Management Subsystem

Data Package will contain a Data Management Subsystem which will deal with the movement of the data that is going to change after each interaction by the user. This subsystem contains the Remote Database Subsystem and Local Database Subsystem. Remote Database Subsystem deals with the movement of the data between user and the server via MySQL. Local Database Subsystem is used for the users preferences throughout the game that are not needed to be sent into the server because it does not concern other players in the lobby.The other throughout the game. This subsystem depends on the Game Management Subsystem because “Settings”, “Dice”, “Lobby”, “Attack”, “Game” and “Turn” classes of this package are dependent on the according manager classes in the Management Package.

3.3.4.1.1. Remote Database Subsystem

Remote Database Subsystem will be used to implement data transactions between the game and

server via MySQL. This subsystem includes “Dice”, “Lobby”, “Attack”, “Game”, “Turn”, “Map”, “Player” and “Region” classes as components. Since “Dice”, “Lobby”, “Attack”, “Game” and “Turn” classes are connected to the Game Management Package, this system depends on the Game Management Package as well.

3.3.4.1.2. Local Database Subsystem

Local Database Subsystem will be used to store users preferences such as music and display which does not need to be on the server because it will not affect other players' gameplay. This subsystem includes the “Settings” class. Since “Settings” class is also connected to the Game Management Package, this system depends on the Game Management Package as well.

3.4. Class Interfaces

For any developer to implement all the functionalities in a more efficient way, we explained all attributes and operations in all classes.

3.4.1. CommonUI

This class consists of JavaFX components which are used in more than one scene classes. It helps us with avoiding writing the same codes for more than one classes.

main_pane: This attribute is the main AnchorPane component of JavaFX. All nodes will be located in this pane.

background_img: This attribute is an ImageView object and shows a .jpeg file.

header_img: This attribute is the ImageView object consisting of the header of the game.

sound_img: This attribute is there for visualizing whether the sound is on or off. The image resource changes accordingly.

3.4.2. MainScene

commonUI: This attribute is an object of CommonUI class.

pageHeader_txt: Text object to represent the name of the menu, which is “Main Menu” in this case.

menu_pane: Pane object consisting of a text field and 5 buttons by which the user can direct the game menus.

nickname_tf: TextField object for the user to enter a nickname before creating or joining a game.

newGame_btn: Button object which puts JoinOrCreateAGameScene to the stage.

settings_btn: Button object which puts SettingsScene to the stage.

credits_btn: Button object which puts CreditsScene to the stage.

howToPlay_btn: Button object which puts HowToPlayScene to the stage.

exit_btn: Button object which exits the system and closes the stage.

3.4.3. JoinOrCreateAGameScene

commonUI: This attribute is an object of CommonUI class.

pageHeader_txt: Text object to represent the name of the menu, which is “Join A Game” or “Create A Game” in these cases.

menu_pane: Pane object consisting of a label, a text field and 2 buttons by which the user can direct the game menus.

gameId_lbl: Label object which represents the aim of the text field “gameId_tf”.

gameId_tf: TextField object that gets game ID input from the player.

backMainMenu_btn: Button object to go back to the main menu.

join_btn: Button object which puts LobbyScene to the stage.

3.4.4. LobbyScene

commonUI: This attribute is an object of CommonUI class.

pageHeader_txt: Text object to represent the name of the menu, which is “Lobby” in this case.

playersHeader_txt: Text object to represent the name of the pane that shows players, which is “Players”.

backMainMenu_btn: Button object which puts MainScene to the stage.

start_btn: Button object to start the game. Visible only for the host of the lobby.

3.4.5. GameScene

background_img: Because this scene has a different view than the ones explained above, it does not use a CommonUI object. Instead, this attribute is used.

howToPlay_btn: Button object which puts HowToPlayScene to the stage.

settings_btn: Button object which puts SettingsScene to the stage.

nickname_pane: Pane object to show the nicknames of all players.

hire_btn: Button object to hire hackers. Visible for all players.

hack_btn: Button object to start hacking a country. Visible for all players.

fortify_btn: Button object to fortify hackers from one region to another. Visible for all players.

hackerNumber: Menu object to let the players choose the number of hackers in each stage (hire, hack, fortify).

mapScene: MapImage object to collect and represent all regions in a map.

3.4.6. NewGameScene

commonUI: This attribute is an object of CommonUI class.

pageHeader_txt: Text object to represent the name of the menu, which is “New Game” in this case.

menu_pane: Pane object consisting of 3 buttons by which the user can direct the game menus.

joinGame_btn: Button object which puts JoinOrCreateAGameScene to the stage..

createAGame_btn: Button object which puts JoinOrCreateAGameScene to the stage..

backMainMenu_btn: Button object which puts the MainScene to the stage.

3.4.7. SettingsScene

commonUI: This attribute is an object of CommonUI class.

display_pane: Pane object consisting of a label and a menu object for display settings.

audio_pane: Pane object consisting of a label and a Slider object for audio settings.

menu_pane: Pane object consisting of “display_pane”, “audio_pane” and 2 buttons.

save_btn: Button object to upload settings to local database.

backMainMenu_btn: Button object which puts the MainScene to the stage.

3.4.8. CreditsScene

commonUI: This attribute is an object of CommonUI class.

credits_pane: Pane object consisting of a text object, “credits_txt”.

credits_txt: Text object to represent credits.

backMainMenu_btn: Button object which puts the MainScene to the stage.

3.4.9. HowToPlayScene

commonUI: This attribute is an object of CommonUI class.

howToPlay_pane: Pane object consisting of a text object, “howToPlay_txt”.

howToPlay_txt: Text object to represent the information about gameplay.

backMainMenu_btn: Button object which puts the MainScene to the stage.

3.4.10. RegionImage

int getHackerNum(int regionId): Gets the number of hackers in the region specified.

Color getColor(int regionId): Gets the color of region specified in parameter.

3.4.11. MapImage

ImageView getRegions(): Returns a map image uploaded to the database by developers.

3.4.12. SceneManager

private boolean isSoundOn: Represents the situation whether sound is on or off.

void newGameClicked(): Puts the NewGameScene to the stage.

void settingsClicked(): Puts the SettingsScene to the stage.

void creditsClicked(): Puts the CreditsScene to the stage.

void howToPlayClicked(): Puts the HowToPlayScene to the stage.

void exitClicked(): Closes the system.

void toggleSound(): Switches sound situation and changes the sound image accordingly.

3.4.13. LobbyManager

void addPlayer(int lobbyId): When a player joins the lobby, it runs and adds this player to the “playerIDs” list in “Lobby” class in the database. It also uploads the colors of the player to the “playerColors” list in “Lobby” class in the database.

void removePlayer(int lobbyId): When a player leaves the lobby, it runs and removes this player from the “playerIDs” list in “Lobby” class in the database. It also removes the color of the player from the “playerColors” list in “Lobby” class in the database.

3.4.14. SettingsManager

private int soundLevel: Keeps the sound level.

private String size: Keeps the size option.

void saveSound(): Uploads the sound level to “Settings” class in the local database.

void saveSize(): Uploads the size option to “Settings” class in the local database.

3.4.15. GameManager

void startGame(): Runs the game and turns.

void startTime(): Starts and counts the time of the game.

boolean checkIfFinished(): There are conditions indicating whether the game is finished or not. Returns true if the game is finished, false if not.

void endGame(): If checkIfFinished() function returns true, it ends the game.

void uploadDatabase(): In each step, it runs the other database related functions in many different classes.

3.4.16. TurnManager

private int turnId: Keeps the id of the turn for all functions in this class.

void setTurnId(): Setter for the attribute “turnId”.

int getTurnId(): Getter for the attribute “turnId”.

void countDown(): Counts down the given time for each player and if a player is late to complete the step, then it finishes the part and passes the next part.

void giveBonusHackers(): Runs in the “hire” part and gets the data called “numberOfBonusHackers” from the class “Player” in the database and adds this number to 5 hackers.

void startHirePart(): Starts the hire part for the player specified as the owner of the turn in “Turn” class in the database.

void startHackpart(): Starts the hack part for the player specified as the owner of the turn in “Turn” class in the database.

void startFortifyPart(): Starts the fortify part for the player specified as the owner of the turn in “Turn” class in the database.

void giveBonusCard(): Runs after the hack part and gets the data called “numberOfBonusCards” from the class “Player” in the database and adds one to this number.

Player uploadWinner(): Uploads the winner of the hack part to Turn class (winner) and Player class (increments the data numberOfWins) in the database.

Player uploadLoser(): Uploads the loser of the hack part to Turn class (loser) and Player class (increments the data numberOfLoses) in the database.

void endTurn(): Finishes the turn and passes another player.

3.4.17. HireManager

void chooseRegion(): When the player clicks on one of his/her regions during the hire part, this function runs and chooses the region for hireHackers() function.

void hireHackers(int numOfHackers): Gets the number of hackers to be hired from the owner of the current turn and when the player chooses the number of hackers and clicks “Hire” button, it increments the number of hackers in the destination region.

int getNumOfHackers(): Returns the number of hackers in the region returned by chooseRegion() function.

void endHire(): Finishes the hire part and invokes the starter function of the hack part.

3.4.18. HackManager

void chooseBase(): Chooses the base for hack attack when the player clicks one of his/her regions in the hack part.

void chooseTarget(): Chooses the target for hack attack when the player clicks one of the enemy regions in the hack part.

void chooseHackerNum(): When the player chooses the number of hackers to attend the attack from the choose menu, this function runs.

void startHack(): Starts the attack. Also, invokes some other functions in different classes such as Ping Manager and Dice Manager.

void endHack(): Finishes the hack part and passes the fortify part.

3.4.19. DiceManager

private int numOfAttackerDice: Keeps the attacker's dice number.

private int numOfDefenderDice: Keeps the defender's dice number.

private Player winner: Keeps the winner of the comparison of biggest dice.

private Player loser: Keeps the loser of the comparison of biggest dice.

private int attackerWins: Keeps the number of wins of the attacker during the dice comparison.

private int defenderWins: Keeps the number of wins of the defender during the dice comparison.

int[] rollDice(int attackerRange): Returns the numbers on each dice rolled. Parameter "attackerRange" keeps the biggest value for the dice and is declared in Ping Manager.

void compareBiggest(int attackerDiceNum, int defenderDiceNum): Finds the biggest dice of each side and compares them. Increments either attackerWins or defenderWins attribute.

Player getWinner(): Compares attackerWins and defenderWins attributes and if attackerWins is greater then returns attacker as winner, if defenderWins is greater then returns defender as winner.

Player getLoser(): Returns the opponent player returned by getWinner() function.

3.4.20. PingManager

void setPing(): Assign a ping value for the attacker. This value is assigned according to the return of getDistance() function. It is used in setAttackerRange() function.

int getDistance(): Return the distance between base and target regions in a hack attack.

void setAttackerRange(): Calculates a range value, which is from 1 to 6, for rollDice() function in DiceManager class. As the distance returned by getDistance() function gets longer, this function returns a smaller value to decrease the possibility of the attacker to win.

3.4.21. FortifyManager

void chooseBase(): Chooses the base region as the player clicks the first region.

void chooseDestination(): Chooses the destination region as the player clicks the second region.

void fortifyHackers(hackerNum): Fortifies the hackers from base region to destination region. Also, uploads the numberOfHacker for both regions in the database.

endFortify(): Finishes the fortify part and calls endTurn() function.

3.4.22. Game

private int id: Keeps the unique id value which is there for each game.

private Player winner: Keeps the winner player.

private int numberOfPlayers: Keeps the player number in the game.

3.4.23. Turn

private int id: Keeps the unique id value which is there for each turn.

private Player owner: Player object which keeps the owner of the turn.

private int numberOfBonusHackers: Gets the numberOfBonusHackers value from Player class by “owner” object.

private Player winner: Keeps the winner player of the turn.

private Player loser: Keeps the winner player of the turn.

3.4.24. Attack

private int id: Keeps the id number of the lobby which is unique for each lobby.

private int ping: Keeps the ping level which is calculated in PingManager Class.

private Player attacker: Keeps the attacker player of the turn.

private Player defender: Keeps the defender player of the turn.

private Region attackerRegion: Keeps the base region which belongs to the attacker player.

private Region defenderRegion: Keeps the target region which belongs to the defender player.

private Player winner: Keeps the winner player of the attack.

private Player loser: Keeps the loser player of the attack.

private int numOfAttackerHackers: Keeps the number of hackers included in the attack by the attacker player. It is maximum 3.

private int numOfDefenderHackers: Keeps the number of hackers included in the attack by the defender player. It is maximum 2.

3.4.25. Map

private int numOfRegions: Keeps the total number of regions included in the map of the game.

private Region[] regions: Array to keep the regions as objects.

3.4.26. Lobby

private int id: Keeps the id number of the lobby which is unique for each lobby.

private Player host: Keeps the player who created the lobby

private int numberOfPlayers: Keeps the number of players who joined the lobby.

private int[] playerIDs: Keeps the id numbers of all players in the lobby.

private Color[] playerColors: Keeps the color values of all players in the lobby.

3.4.27. Region

private int id: Keeps the id number of the region which is unique for each region.

private String name: Keeps the name of the region.

private Player owner: Keeps the current owner of the region. During the game it may change many times.

private int numberOfHackers: Keeps the number of hackers that the owner player hired in the region.

private Color color: Keeps the color of the region which is actually the color of the player.

private int location: Keeps the location value of the region. Location value is defined with a number which increments from left to right and top to bottom throughout the map, respectively.

3.4.28. Player

private int id: Keeps the id number of the player which is unique for each player.

private Color color: Keeps the color of the player.

private String nickname: Keeps the nickname entered by the player.

private int score: Keeps the score of the player.

private int numberOfHackers: Keeps the total number of hackers the player has.

private int numberOfRegions: Keeps the total number of regions the player has.

private int numberOfWins: Keeps the total number of wins the player has.

private int numberOfLosses: Keeps the total number of losses the player has.

private int numberOfBonusCards: Keeps the number of bonus cards given by the game to the player.

private int numberOfBonusHackers: Keeps the number of bonus hackers which is the correspondence to number of bonus cards the player has.

private boolean isOnline: True if the player is online with the nickname, false if not.

3.4.29. Dice

private int id: Keeps the id number of the dice which is unique for each dice.

private Player owner: Keeps the owner player of the dice. With this information, the system finds the interlocutor player of the dice rolled.

private int number: Keeps the value on the dice.

3.4.30. Settings

This class is kept in the local database which is actually a txt file. The sound level and the size option is kept as a string in this txt

file. Thus, the personal settings will belong to only one player, not all players.

private int sound: Keeps the volume level.

private String size: Keeps a string value such as “1920x1080”. The system reads this string and defines the size of the main stage object (JavaFX) of the game.

4. Glossary and References