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.

1.2.9. Design Patterns

We applied strategy design pattern while implementing the point strategy for the cards. This strategy pattern made it possible to add different point gain behaviours in a single method which is add points. This makes it possible to add or modify the algorithms to the existing code without making code smell.

The second pattern that we added is the state pattern which is applicable to our TurnPart class because this class runs three sequential states which are Hire, Hack and Fortify and it adds simplicity for these parts. Therefore, state design pattern is the right fit.

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.

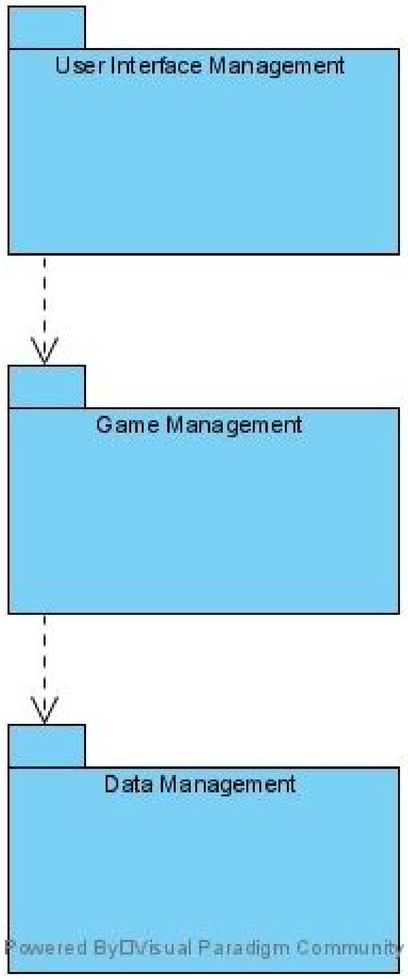

User Interface Management
Game Management
Data Management
Powered By Visual Paradigm Community

Figure 1: Basic version of subsystem decomposition

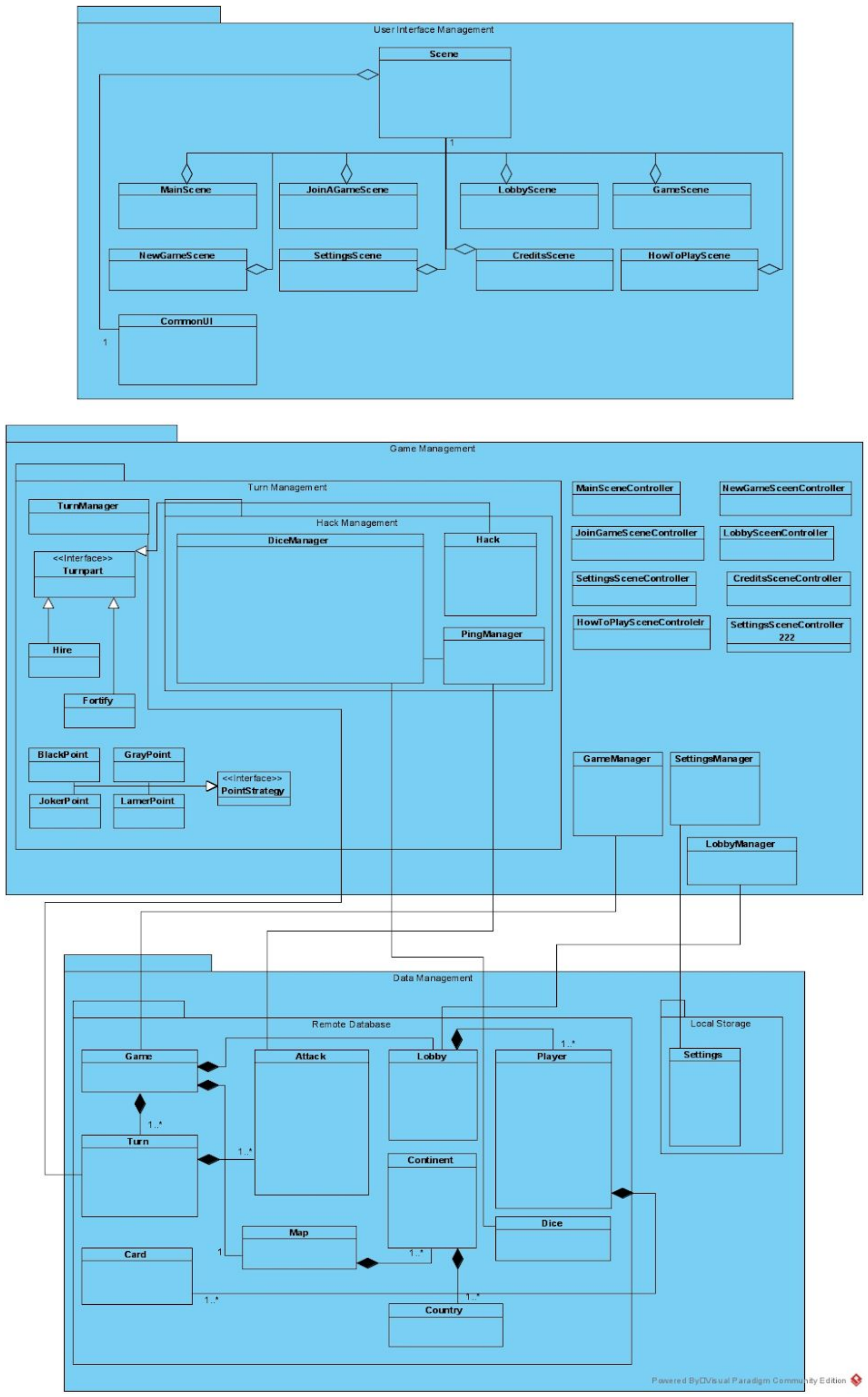
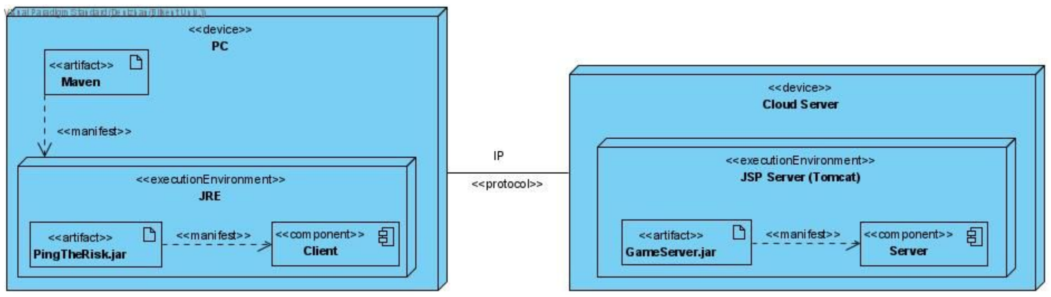


Figure 2: The detailed version of subsystem decomposition

2.2. Hardware/Software Mapping



The game requires the Java Runtime Environment to be used by users who have JDK 8 or newer versions.

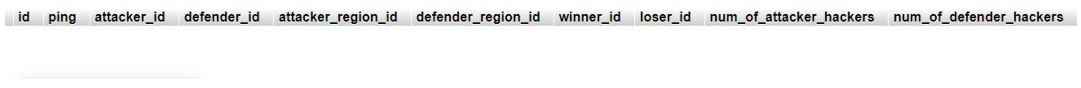
These requirements are minimal to play a game so that it can be more accessible and playable for more users

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

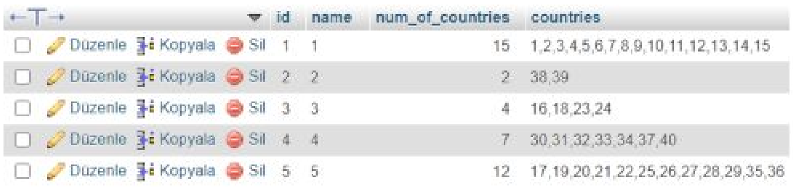
Since the players will be playing online and the data needs to update for each player simultaneously, the gameplay data will be kept in an online server database. The gameplay data will change according to the events that occur at the end of each turn.We are going to use AWS (Amazon Web Services) and MySQL for the server implementation of our game. After each event, the updated version of data will be returned into the database via MySQL 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.

2.3.1. Database Tables:



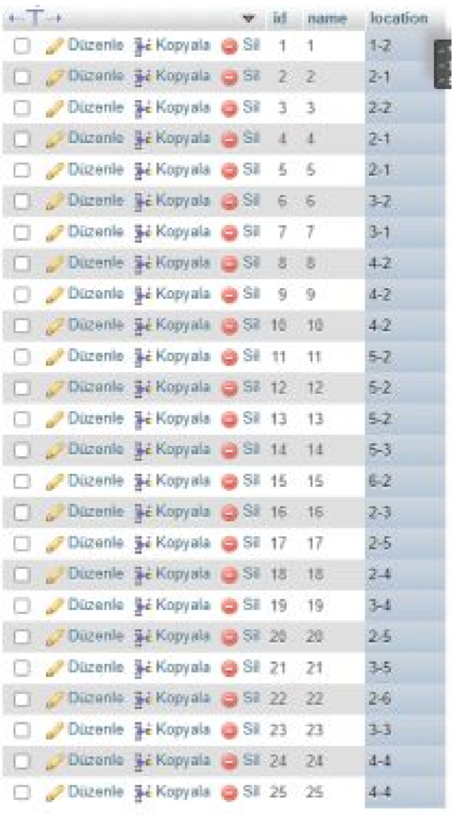
id	ping	attacker_id	defender_id	attacker_region_id	defender_region_id	winner_id	loser_id	num_of_attacker_hackers	num_of_defender_hackers

Figure 1: “attack” table in database



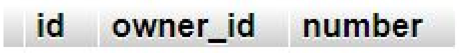
				id	name	num_of_countries	countries
☐	Düzenle	Kopyala	Sil	1	1	15	1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
☐	Düzenle	Kopyala	Sil	2	2	2	38,39
☐	Düzenle	Kopyala	Sil	3	3	4	16,18,23,24
☐	Düzenle	Kopyala	Sil	4	4	7	30,31,32,33,34,37,40
☐	Düzenle	Kopyala	Sil	5	5	12	17,19,20,21,22,25,26,27,28,29,35,36

Figure 2: “continent” table in database



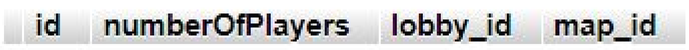
				id	name	location
☐	Düzenle	Kopyala	Sil	1	1	1-2
☐	Düzenle	Kopyala	Sil	2	2	2-1
☐	Düzenle	Kopyala	Sil	3	3	2-2
☐	Düzenle	Kopyala	Sil	4	4	2-1
☐	Düzenle	Kopyala	Sil	5	5	2-1
☐	Düzenle	Kopyala	Sil	6	6	3-2
☐	Düzenle	Kopyala	Sil	7	7	3-1
☐	Düzenle	Kopyala	Sil	8	8	4-2
☐	Düzenle	Kopyala	Sil	9	9	4-2
☐	Düzenle	Kopyala	Sil	10	10	4-2
☐	Düzenle	Kopyala	Sil	11	11	5-2
☐	Düzenle	Kopyala	Sil	12	12	5-2
☐	Düzenle	Kopyala	Sil	13	13	5-2
☐	Düzenle	Kopyala	Sil	14	14	5-3
☐	Düzenle	Kopyala	Sil	15	15	6-2
☐	Düzenle	Kopyala	Sil	16	16	2-3
☐	Düzenle	Kopyala	Sil	17	17	2-5
☐	Düzenle	Kopyala	Sil	18	18	2-4
☐	Düzenle	Kopyala	Sil	19	19	3-4
☐	Düzenle	Kopyala	Sil	20	20	2-5
☐	Düzenle	Kopyala	Sil	21	21	3-5
☐	Düzenle	Kopyala	Sil	22	22	2-6
☐	Düzenle	Kopyala	Sil	23	23	3-3
☐	Düzenle	Kopyala	Sil	24	24	4-4
☐	Düzenle	Kopyala	Sil	25	25	4-4

Figure 3: “country” table in database



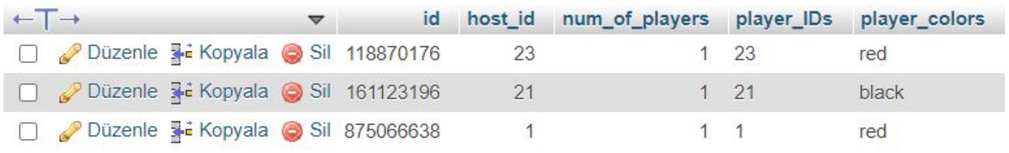
id	owner_id	number

Figure 4: “dice” table in database



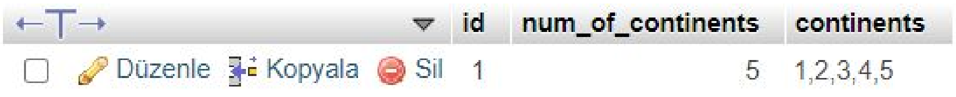
id	numberOfPlayers	lobby_id	map_id

Figure 5: “game” table in database



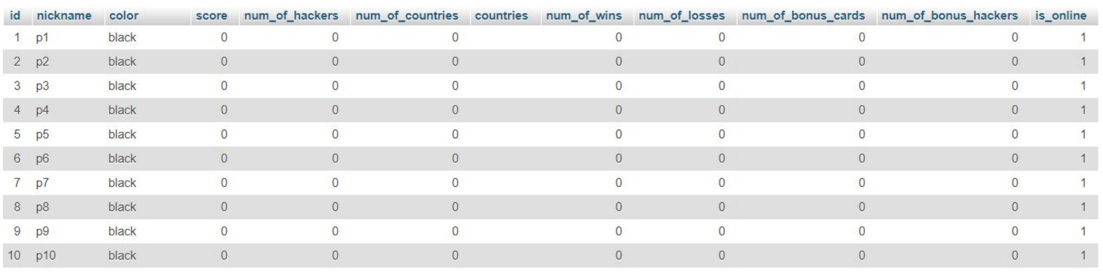
←T→				id	host_id	num_of_players	player_IDs	player_colors
☐	Düzenle	Kopyala	Sil	118870176	23	1	23	red
☐	Düzenle	Kopyala	Sil	161123196	21	1	21	black
☐	Düzenle	Kopyala	Sil	875066638	1	1	1	red

Figure 6: “lobby” table in database



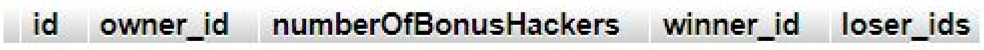
←T→				id	num_of_continents	continents
☐	Düzenle	Kopyala	Sil	1	5	1,2,3,4,5

Figure 7: “map” table in database



id	nickname	color	score	num_of_hackers	num_of_countries	countries	num_of_wins	num_of_losses	num_of_bonus_cards	num_of_bonus_hackers	is_online
1	p1	black	0	0	0		0	0	0	0	1
2	p2	black	0	0	0		0	0	0	0	1
3	p3	black	0	0	0		0	0	0	0	1
4	p4	black	0	0	0		0	0	0	0	1
5	p5	black	0	0	0		0	0	0	0	1
6	p6	black	0	0	0		0	0	0	0	1
7	p7	black	0	0	0		0	0	0	0	1
8	p8	black	0	0	0		0	0	0	0	1
9	p9	black	0	0	0		0	0	0	0	1
10	p10	black	0	0	0		0	0	0	0	1

Figure 8: “player” table in database



id	owner_id	numberOfBonusHackers	winner_id	loser_ids

Figure 9: “turn” table in 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 such as changing the game data. However, 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

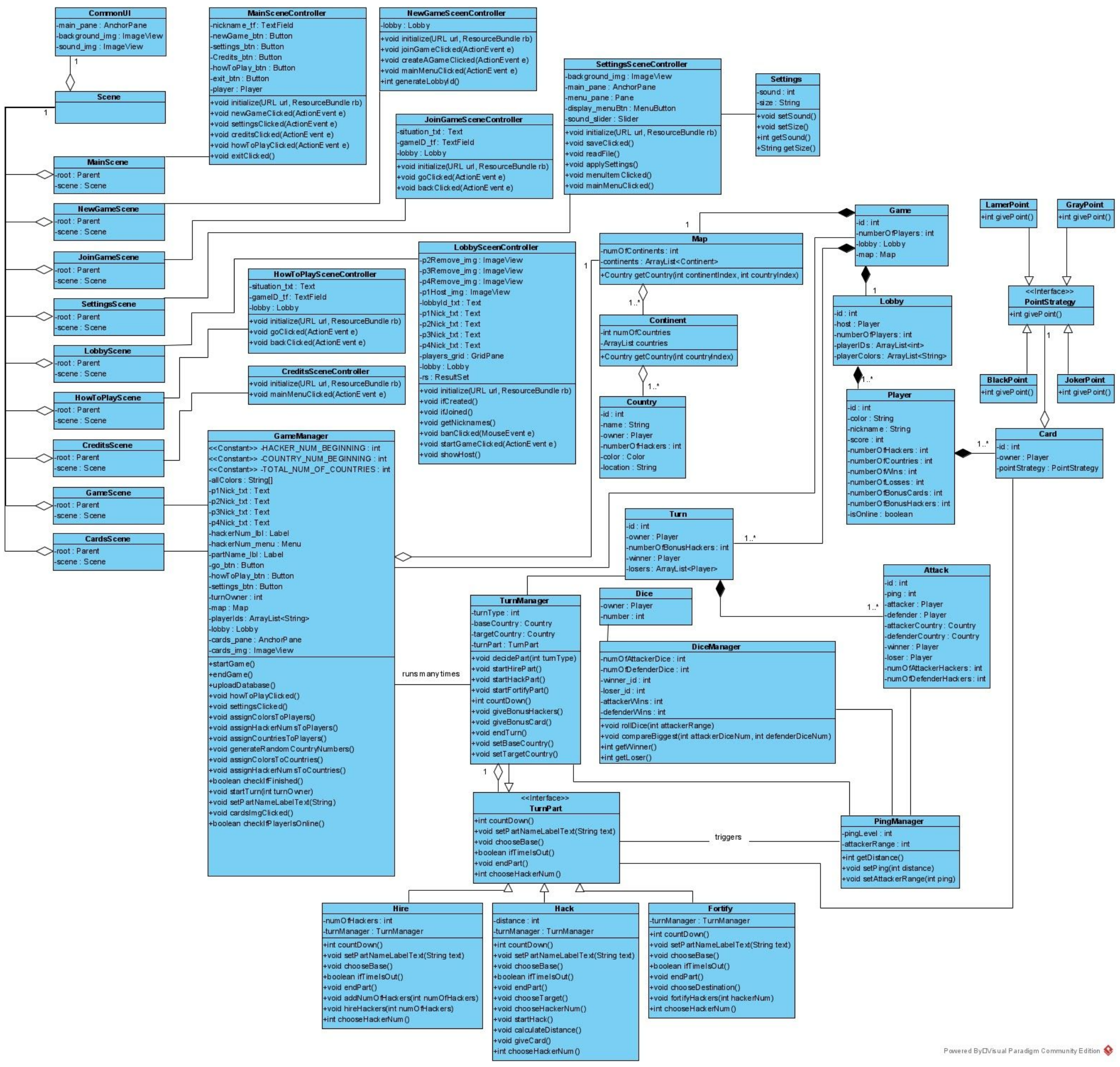


Figure 3: Final Object Design

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.

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.

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.

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.

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.

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.

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

3.4.2. MainSceneController

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.

player: Player object of player class.

void initialize(URL url, ResourceBundle rb): Initializing the url with bundle.

void newGameClicked(ActionEvent e): Action event for new game clicked.

void settingsClicked(ActionEvent e) : Action event for settings button.

void creditsClicked(ActionEvent e) : Action event for credits button.

void howToPlayClicked(ActionEvent e) : Action event for howToPlay button.

void exitClicked(): Action event for exit button.

3.4.3. JoinGameSceneController

situation_txt: Text that explains the situation.

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

lobby: a Lobby object created for the player to access the lobby.

void initialize(URL url, ResourceBundle rb): Initializing the url with bundle.

void goClicked(ActionEvent e): Action event for go button.

void backClicked(ActionEvent e): Action event for back button.

3.4.4. NewGameSceenController

lobby: a Lobby object created for the player to access the lobby.

void initialize(URL url, ResourceBundle rb): Initializing the url with bundle.

void joinGameClicked(ActionEvent e): Action event for join game button.

void createAGameClicked(ActionEvent e): Action event for create a game button.

void mainMenuClicked(ActionEvent e): Action event for main menu button.

int generateLobby(): method which creates a lobby in the database and returns game id to be used for joining.

3.4.5. Scene

Scene is a child of CommonUI class.

3.4.6. GameScene

GameScene is a child of Scene class.

3.4.7. GameManager

HACKER_NUM_BEGINNING: Constant integer that keeps the maximum number of hacker numbers for the beginning of the game.

COUNTRY_NUM_BEGINNING: Constant integer that keeps the maximum number of countries to be assigned for the beginning of the game.

TOTAL_NUM_OF_COUNTRIES: Constant integer that keeps the total number of countries in the whole map of the game.

allColors[ ]: String array that keeps all defined colors.

p1Nick_txt: Text object that keeps the nickname of the player 1.

p2Nick_txt: Text object that keeps the nickname of the player 2.

p3Nick_txt: Text object that keeps the nickname of the player 3.

p4Nick_txt: Text object that keeps the nickname of the player 4.

hackerNum_lbl: Label object that keeps “hacker” string constantly during the game.

hackerNum_menu: Menu object that allows players to choose a hacker number for each turn.

partName_lbl: Label object that keeps the name of the part (hire, hack, fortify).

go_btn: Button object that allows players to go to the next part.

howToPlay_btn: Button object to see the how to play menu during the game.

settings_btn: Button object to change the settings during the game.

turnOwner: An integer which is used to decide which part will be run.

map: Map object which consists of many continents. It’s view changes many times during the game.

playerIds: An arraylist of String objects which keeps the id numbers of all players in the game.

lobby: Lobby object that is used for getting various information about the lobby.

cards_pane: AnchorPane object which consists of an image called “cards_img”.

cards_img: ImageView object that keeps the image of cards icon and that is clickable to open the cards menu.

void startGame(): Method that starts the game by getting the result of checkIfFinished() function. If the result is true, then it does not start. If the result is false, this means there is no problem to start the game and it runs.

void endGame(): Method that ends the game by getting the result of checkIfFinished() function. If the result is true, then it runs and ends the game. If the result is false, this means there is no need to end the game and it does not run. If it runs, it calls uploadDatabase() function to upload the information about the whole game to the database.

void howToPlayClicked(): When the howToPlay_btn is clicked, this method runs and opens the “how to play menu”.

void settingsClicked(): When the settings_btn is clicked, this method runs

void assignColorsToPlayers(): Assign each player a different colour to display it on the map.

void AssignHackerNumToPlayers: At the beginning of the game randomly assign 30 hackers to each player.

void assignCountriesToPlayer: At the beginning of the game randomly assign 40/4 = 10 countries to each player.

void generateRandomCountryNumbers(): Randomly generate each country a number.

void assignColoursToCountries(): Assign each country a colour to show which player owns them.

void assignHackerNumsToCountries(): Randomly assign 30 hackers to each 10 countries for each player.

boolean checkIfFinished: Checks if only one player remains in the game and turns true if so.

void startTurn(int turnOwner): Starts the turn and gives the turn to the integer turnOwner.

void setPartNameLabelText(String): Sets the part name label string.

void cardsImgClicked(): Method which shows the card image that is clicked.

boolean checkIfPlayerIsOnline(): Method that checks if the target player is online and turns true if so.

3.4.8. NewGameScene

NewGameScene is a child of Scene class.

3.4.9. SettingsScene

SettingsScene is a child of Scene class.

3.4.10. SettingsSceneController

background_img: used to provide a background image.

main_pane: is an AnchorPane object.

menu_pane: is a Pane object.

display_menuBtn: button object for displaying menu.

sound_slider: Slider for changing the sound.

void initialize(URL url, ResourceBundle rb): Initializing the url with bundle.

void saveClicked(): Saves the setting preferences.

void readFile(): Reads and transcribes the file.

void applySettings(): Applies the preferred settings.

void menuItemClicked(): Applies the menu item clicked.

void mainMenuClicked(): Returns to the main menu.

3.4.11. Settings

sound: an integer used to keep track of sound value.

size: a String used to keep track of display size value.

void setSound(): Sets the sound to the adjusted value.

void setSize(): Sets the display to the adjusted value.

int getSound(): Returns the sound value in int.

String getSize(): Returns the size value in String.

3.4.12. CreditsScene

CreditsScene is a child of Scene class.

3.4.13. CreditsSceneController

void initialize(URL url, ResourceBundle rb): Overridden method from the Initializable class. Runs when the fxml root page runs.

void MainMenuClicked(ActionEvent e): When the main menu button is clicked in how to play a scene, this method runs and lets the user turn back to the main menu.

3.4.13. HowToPlayScene

HowToPlayScene is a child of Scene class.

3.4.14. HowToPlaySceneController

void initialize(URL url, ResourceBundle rb): Overridden method from the Initializable class. Runs when the fxml root page runs.

void MainMenuClicked(ActionEvent e): When the main menu button is clicked in how to play a scene, this method runs and lets the user turn back to the main menu.

3.4.15. LobbyScene

LobbyScene is a child of Scene class.

3.4.16. LobbySceneController

p2Remove_img: ImageView object keeps the remove icon for player2.

p3Remove_img: ImageView object keeps the remove icon for player3.

p4Remove_img: ImageView object keeps the remove icon for player4.

p1Host_img: ImageView object keeps the host icon for player 1.

lobbyId_txt: Text object that shows the lobby Id.

p1Nick_txt: Text object that keeps the nickname of player 1.

p2Nick_txt: Text object that keeps the nickname of player 2.

p3Nick_txt: Text object that keeps the nickname of player 3.

p4Nick_txt: Text object that keeps the nickname of player 4.

players_grid: GridPane object that keeps all objects above in an order.

lobby: Lobby object which will be used for getting information about the lobby.

rs: ResultSet object to keep the result of the database query. It is declared globally because it is used in many functions in this class.

void initialize(URL url, ResourceBundle rb): Overridden method from the Initializable class. Runs when the fxml root page runs.

void ifCreated(): If the player came to this menu by creating a game, this method runs.

void ifJoined(): If the player came to this menu by joining an existing game, this method runs.

void getNicknames(): Fetches the nicknames of the lobby players from the database and puts them to the appropriate text objects.

void banClicked(MouseEvent e): If one of the “remove” images is clicked by the host of the lobby, this function runs and bans the selected player.

void startGameClicked(ActionEvent e): If the host clicks the start game button, this method runs and starts the GameManager.

void showHost(): Gets the host player from the lobby object and puts host icon to the p1Host_img object.

3.4.17. CardsScene

CardsScene is a child of Scene class.

3.4.18. JoinGameScene

JoinGameScene is a child of Scene class.

3.4.19. MainScene

MainScene is a child of Scene class.

3.4.20. TurnManager

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

Country baseCountry: Keeps the country object that is selected first.

Country targetCountry: Keeps the country object that is selected second.

TurnPart turnPart: Keeps the turnPart object for the turns.

void decidePart(int turnType): Decides the turn part according to the turnType value.

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.

int 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 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.

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

void setBaseCountry(): Sets the baseCountry object to the selected country.

void setTargetCountry(): Sets the targetCountry object to the selected country.

3.4.21. TurnPart

int 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.

setPartNameLabelText(String text): Sets the part name label with the given text.

void chooseBase(): Chooses the base for turn parts when the player clicks one of his/her regions.

boolean ifTimeIsOut(): Checks if the time is out or not.

void endPart(): Ends the current part and moves the next one.

int chooseHackerNum(): Chooses the hacker number for every part of the turn.

3.4.22. Hire

int numOfHackers: Keeps the number of hackers selected

TurnManager turnManager: Keeps the turnManager object.

void addNumOfHackers(int numOfHackers): Adds the bonus hackers from the cards to numOfHackers.

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.

3.4.23. Hack

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

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

void giveCard(): If a player hacked at least one region from the hack part then gains a random card.

3.4.24. Fortify

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.

3.4.25. 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.

3.4.26. DiceManager

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

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

private int winner_id: Keeps the winner id of the comparison of biggest dice.

private int loser_id: Keeps the loser id 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.

void rollDice(int attackerRange): Rolls the dice for attacker and defender. 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.

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

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

3.4.27. PingManager

int pingLevel: Keeps the ping level that setted at setPing method.

int attackerRange: Keeps the attacker range that setted at setAttackerRange method.

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.

3.4.28. Game

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

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

private Lobby lobby: Keeps the lobby of the game.

private Map map: Keeps the map of the game.

3.4.29. 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 attackerCountry: Keeps the base country which belongs to the attacker player.

private Region defenderCountry: Keeps the target country 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.

3.4.30. Map

private int numOfContinents: Keeps the total number of continents included in the map of the game.

private ArrayList <Continent> continents: ArrayList to keep the continents as objects.

Country getCountry(int continentIndex, int countryIndex): Gets the desired country.

3.4.31. 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 ArrayList<int> playerIDs: Keeps the id numbers of all players in the lobby.

private ArrayList<Color> playerColors: Keeps the color values of all players in the lobby.

3.4.32. Country

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

private String name: Keeps the name of the country.

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

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

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

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

3.4.33. 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 numberOfCountries: 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.

3.4.34. Continent

private int numOfCountries: Keeps the total number of countries included in the map of the game.

private ArrayList <Countries> countries: ArrayList to keep the countries as objects.

Country getCountry(int continentIndex): Returns the desired country.

3.4.35. Turn

private int id: Keeps the id number of the turn which is unique.

private Player owner: Keeps who has the turn. During the game it changes many times.

private int numberOfBonusHackers: Keeps the number of bonus hackers that the owner player gets.

private Player winner: Keeps the winner player.

private ArrayList <Player> losers: Keeps the player that lost the game.

3.4.36. Card

private int id: Keeps the id number of the card which is unique.

private Player owner: Keeps who has the card.

private PointStrategy pointStrategy : Keeps the point strategy which is different for all types of cards.

3.4.37. PointStrategy

int givePoint(): Returns the point of the card.

3.4.38. BlackPoint

int givePoint(): Returns the point of the Black Card.

3.4.39. GrayPoint

int givePoint(): Returns the point of the Gray Card.

3.4.40. JokerPoint

int givePoint(): Returns the point of the Joker Card.

3.4.41. LamerPoint

int givePoint(): Returns the point of the Lamer Card.

4. Improvement Summary

Scenes are implemented using CommonUI class. Each scene has controller classes that include necessary methods to assign action events to the buttons and they include properties for the scenes. Since the scenes are made with JavaFX, they are also implemented with fxml files in the program.

The different types of card classes are added. To decrease the codes that are used to identify different card objects, point strategy is used. The continent and country is added to differentiate those, rather than just stating as region. The loser count can be more than one, so the loser variable became an ArrayList in Turn class.

HireManager, HackManager, and FortifyManager classes were deleted and instead added TurnPart, Hire, Hack, and Fortify classes which use State Design Pattern for more simplicity.

GameManager class is the class that starts the game play and sets everything for the players. There are 4 max players and each player is randomly assigned colors, 30 hackers and 10 provinces in this class. GameManager checks whether there is only one player remaining and ends the game if so.

5. Glossary and References