CS 319 - Object-Oriented Software Engineering

Design Report

Risk Game

Group 1-D

Yigit Kutay GULBEN

Nurlan FARZALIYEV

Esad Burak ALTINYAZAR

Burak YENI

Burak MUTLU

Anar HUSEYNOV

Supervisor: Eray TUZUN

Contents

1. Introduction

1.1 Purpose of the System:

The purpose of implementing digital version of Risk box game can be thought as playing such a fun game on a digital platform so that playing the game would be much quicker and additional features may be added into game. The reason for being much quicker experience is that you do not need to setup physical conditions to play game with friends. By implementing additional features, we aim to offer a better quality and unique experience for users.

1.2 Design Goals

1.2.1 Performance Criteria:

***Efficiency*:** While implementing program especially game-like programs which highly demands on the process power, the efficiency of memory and time complexity of algorithms should be taken into account. For this respect, we regard the time complexity of algorithms used to simulate the game mechanics since they may have high effect on the game performance. Also, memory related issues may affect the execution of game functionalities in long term. For the sake of having good performance, such factors should be always in first priority among the implementation of game.

***Game Performance*:** In essence of a game, the users are the main focus of games, they are the target community of implemented program. Since the reflected performance to the users affects the experience they get, game performance is an important consideration. That’s why we should focus to optimize the viewed performance by users for offering much flowing game experience.

1.2.2 Dependability Criteria:

***Reliability*:** The last thing a user want to experience is the crash of a game in the middle of the game. In the matter of whole offered options in the game, there should not be any bug in its complete boundaries, and, the game should fully perform its options. The structure of our program and game mechanisms will be designed to not allow any undesired situations. For other respects, in game mechanics, we should not leave any open-end relation such that a smart user may realize it and try to abuse over such open-end relation. That would be a very annoying experience for the players, especially in multiplayer game mode.

Robustness**:** Our game will have a solid stand in the view of users through its additional features and well thoughts characteristics. For instance, the mentioned well thoughts characteristics of the game may be explained as its fascinating visual effects, its addictive game experience, offering a competitive multiplayer or strategically deep-minded computer opponents.

Playability**:** Our game will be understandable for the players as their first look into game; we will offer supportive hints for most essential rules and mechanisms of the game and also a tutorial game mode for beginners to get adapted with the game. As an additional feature, we may implement a specific feature like in-game dynamic difficulty, which refers to changing difficulty into much easier or much harder ones dependently on the player score gap with others.

1.2.3 Maintenance Criteria:

Usability**:** The "Risk" game is a board game originally. What is problematic about the board games is those kinds of games require some other players to be present at a time in order to play the game, and managing the game itself can also take some time. However, the usability of the version that we will be implementing will be very high since it will take just a couple of seconds to open the game and start playing. Moreover, there is no need for other players to be present, there will be "Single player" option as well.

Extensibility**:** It should be always exciting to update the game with additional features after implementing of its primal version. That's why we will always be ready for further improvements and modifications and we will realize such game implementation that it would be available to integrate with implementations of upcoming features. Since we are using Java to implement the game, our design is going to be an object-oriented software design. This design ensures that by future it will be easy, will not be problematic, to make changes or add new features in case of any problems or enhancements. For example, we are planning to have one default difficulty level before sum up the whole implementation. Afterwards, we will add new difficulty levels as well, and it will not take too much time.

Adaptability**:** The main reason we choose to implement our game in Java is it provides platform independency. The compiled Java code can be run in any platform that runs JVM.

2. Software Architecture

2.1 Subsystem decomposition

While implementing a game, the most basic way to make complex intertwined things much easier to handle is dividing the functionalities into subcomponents. For such purpose, we choose to implement MVC (Model-View-Controller) programming approach. By using MVC, we will divide the complex game functionalities into such three segments that all three executes its own responsible stuff and communicates with each other properly to give a program running without any sort of error.

Model: Model describes the structure of data and business logic. Objects of model retrieve and maintain the state of the model in a program. Applies the game mechanics into their current state and provides the required data to the other components, which are View and Controller.

View: View is also known as user interface. In its most basic sense, it describes the visual reflections of the corresponding model objects. The interface not only displays the model to the user and but also makes the data modifiable since it is related with user inputs. It will transmit the inputs not directly into model components but the controller. That's a better way of implementing things.

Controller: Controller manages the user request and executes required operations over model components. Generally, user firstly interacts with the View and after that needed URL request will be generated by this component, that is, generated request will be fulfilled by a controller.

2.2 Persistent Data Management

There will be background photos in the game, music will be one of the features. The files of such features may be stored in local storage. The all required game data needed to be saved like game instances, objects and all of these kinds of data will be saved in an ordinary file. Since the persistent data is not too much, we have decided to save our game as a database stored in local memory. The saved stuff will be encrypted and user will not have access to the saved files.

2.3 Access Control and Security

In order to make the data secure we decided to encrypt the saved files and those files will not be user accessible. The encryption always makes things safer even if it is not needed. The network handles the requests and corresponding operations of such requests, which resulted in that the user not able to manipulate the game data directly. For such manner, the user is able to only communicate with networks that is supposed to execute the plausible requests of users, that is, it is a middle layer restricts the user access and provides better security.

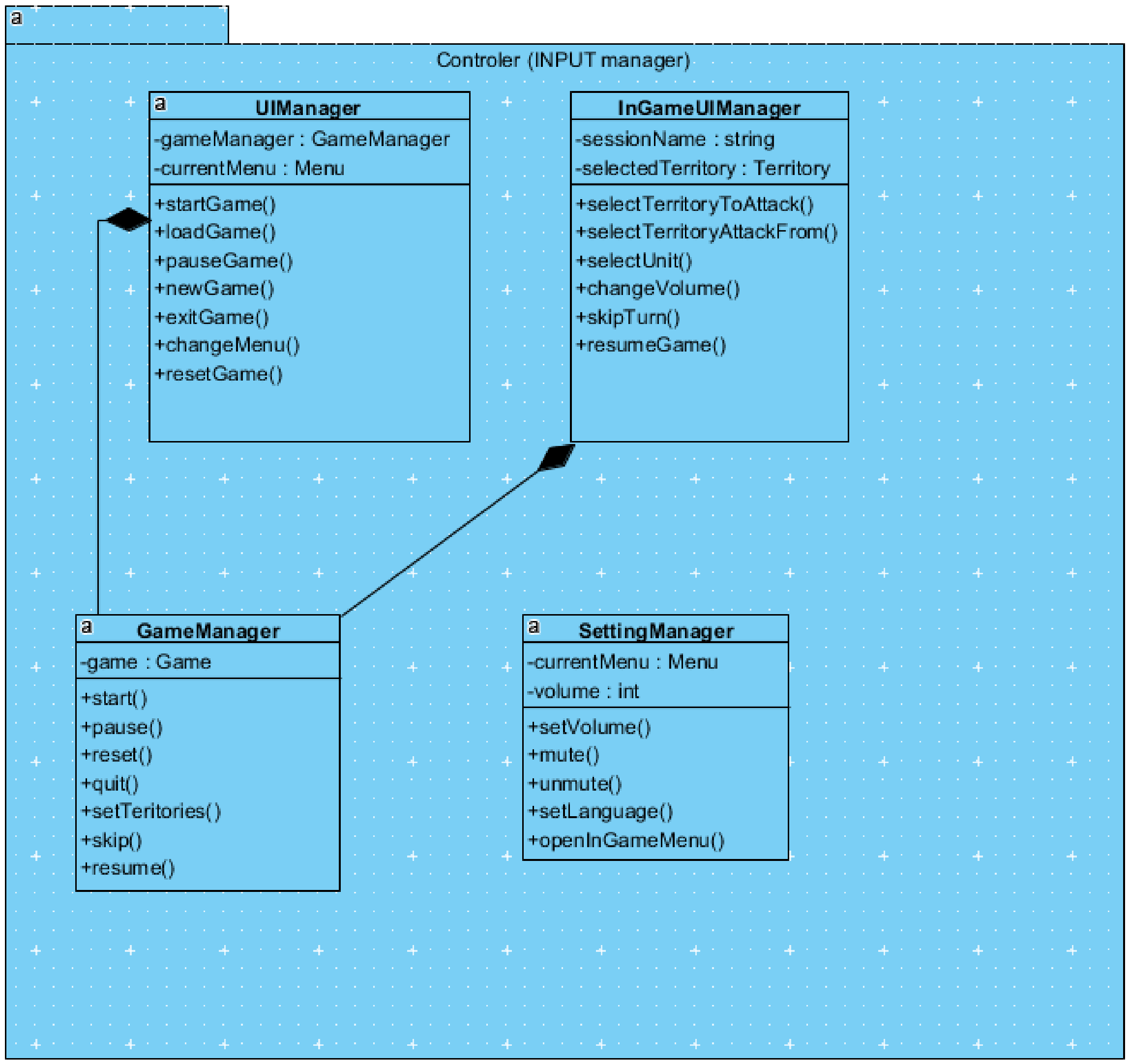
2.4 Boundary Conditions

It does not matter in which perspective you look, the games either offer to the players some freedom and determines some restrictions over their options. We need to set such boundaries on the options game offered in order to balance game experience and not to lead any boundary abusing. If any kind of error occurs during the game the program will exit. Player cannot run the game while it is already on run, so first application will exit automatically if the player reruns the game.

3. Subsystem Services

3.1 Input Management Subsystem

UI Manager, InGameUIManager and GameManager are the base classes for all the considered input, they are responsible for all the territory selections, attacking, adding unit and so on. They take the input from the player through the mouse and buttons, and passes that information to the GameManager and SettingManager.


Controler (INPUT manager)
UIManager
-gameManager : GameManager
-currentMenu : Menu
+startGame()
+loadGame()
+pauseGame()
+newGame()
+exitGame()
+changeMenu()
+resetGame()
InGameUIManager
-sessionName : string
-selectedTerritory : Territory
+selectTerritoryToAttack()
+selectTerritoryAttackFrom()
+selectUnit()
+changeVolume()
+skipTurn()
+resumeGame()
GameManager
-game : Game
+start()
+pause()
+reset()
+quit()
+setTeritories()
+skip()
+resume()
SettingManager
-currentMenu : Menu
-volume : int
+setVolume()
+mute()
+unmute()
+setLanguage()
+openInGameMenu()

UIManager:

public startGame() - this method is for starting to the chosen game

public loadGame() - this method is for choosing the last saved game for starting

public pauseGame() - this method is for pausing the game

public newGame() - this method is for choosing the “new game” to start the game

public exitGame() - this is for exiting from the game. After user exits from the game, it will be saved automatically. User will be able to start the game again by loadGame() method

public changeMenu() - this method is for opening or exiting from the different components of the menu

public resetGame() - this is method is for starting the game again from the beginning

InGameUIManager:

public selectTerritoryToAttack() - this method is for selecting the target territory that will be attacked

public selectTerritoryAttackFrom() - this method is for selecting the attacking territory

public selectUnit() - this method is for selecting the units that will be added to specified territory

public changeVolume() - this method is for adjusting the volume of the game

public skipTurn() - this method is for skipping the turn to next player

Game Manager:

public start() - this method will be called by startGame() method to start the game

public pause() - this method will be called by pauseGame() method to pause the game

public reset() - this is method will be called by resetGame() method for reseting the game

public quit() - this method will be called by exitGame() method to exit from the game

public setTerritories() - this method assigns 2 territories to combat

public skip() - when this method is called while in a turn it skips the turn to other player that is waiting

public resume() - If the game is paused this method resumes the game

SettingManager:

public setVolume() - While the setting menu is on this by using this medhod volume is adjusted

public mute() - This method mutes the game sounds

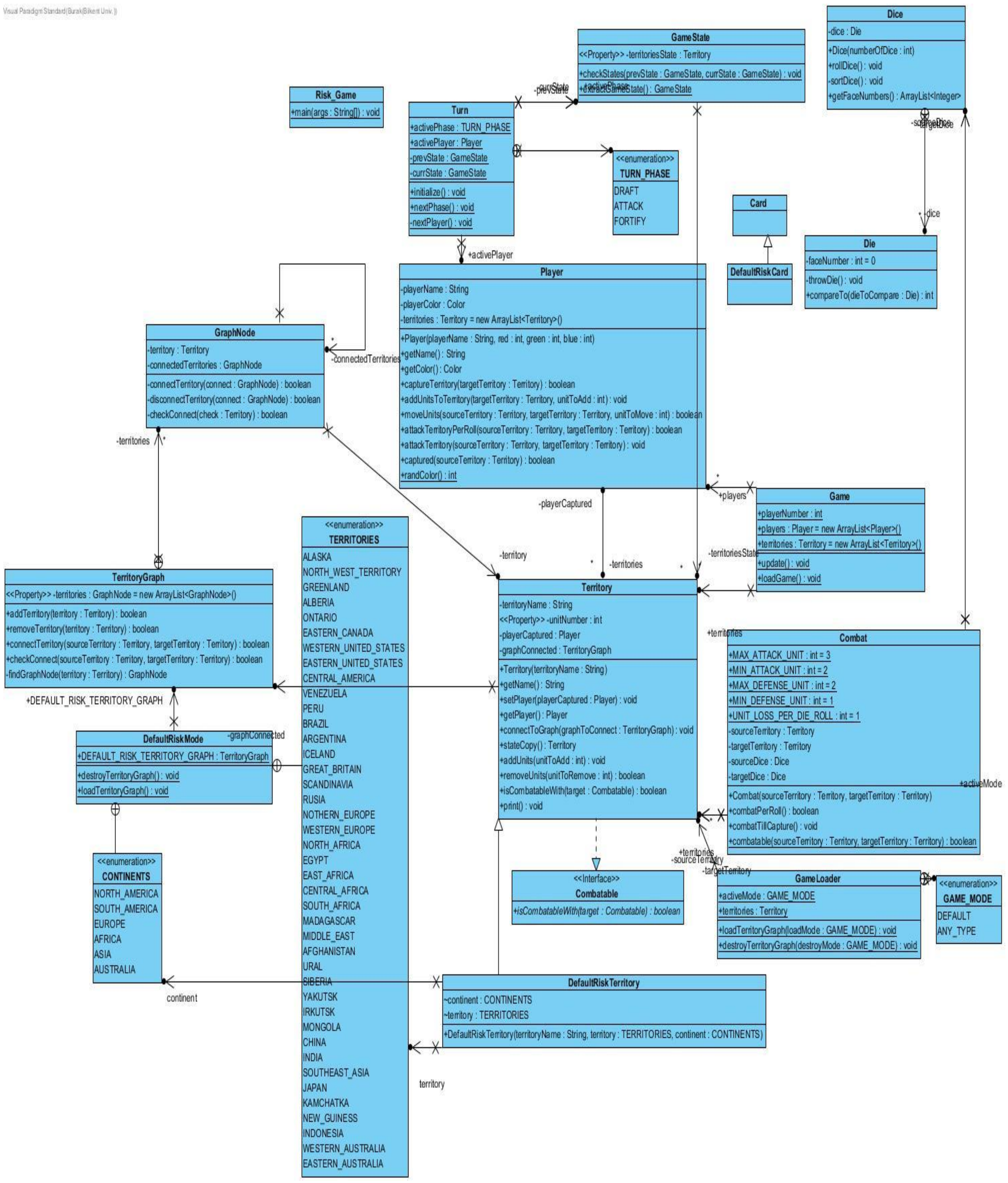
public unmute() - This method unmutes the game sounds

public setLanguage() - By using this method player can change the game's and menu's language

public openInGameMenu() - While the game is running by using this methed player can open the in game menu and adjust volume, language and so on.

3.2 Model Subsystem

Model classes can be listed as Game, Turn, GameState which are top-level running game classes. Basically, they will determine which map is used, in which player turn is and current phase of the game. Much lower classes determines the cards needed to be given players after a turn, the moves a player is able to do and how combat mechanics works.


Risk_Game
+main(args : String[]) : void
Turn
+activePhase : TURN_PHASE
+activePlayer : Player
-prevState : GameState
-currState : GameState
+initialize() : void
+nextPhase() : void
-nextPlayer() : void
GameState
<<Property>> -territoriesState : Territory
+checkStates(prevState : GameState, currState : GameState) : void
+extractGameState() : GameState
-currState
-prevState
+activePhase
Dice
-dice : Die
+Dice(numberOfDice : int)
+rollDice() : void
-sortDice() : void
+getFaceNumbers() : ArrayList<Integer>
-sourceDice
-targetDice
*
-dice
<<enumeration>>
TURN_PHASE
DRAFT
ATTACK
FORTIFY
Card
DefaultRiskCard
Die
-faceNumber : int = 0
-throwDie() : void
+compareTo(dieToCompare : Die) : int
+activePlayer
Player
-playerName : String
-playerColor : Color
-territories : Territory = new ArrayList<Territory>()
+Player(playerName : String, red : int, green : int, blue : int)
+getName() : String
+getColor() : Color
+captureTerritory(targetTerritory : Territory) : boolean
+addUnitsToTerritory(targetTerritory : Territory, unitToAdd : int) : void
+moveUnits(sourceTerritory : Territory, targetTerritory : Territory, unitToMove : int) : boolean
+attackTerritoryPerRoll(sourceTerritory : Territory, targetTerritory : Territory) : boolean
+attackTerritory(sourceTerritory : Territory, targetTerritory : Territory) : void
+captured(sourceTerritory : Territory) : boolean
+randColor() : int
GraphNode
-territory : Territory
-connectedTerritories : GraphNode
-connectTerritory(connect : GraphNode) : boolean
-disconnectTerritory(connect : GraphNode) : boolean
-checkConnect(check : Territory) : boolean
*
-connectedTerritories
-territories
*
-playerCaptured
*
+players
Game
+playerNumber : int
+players : Player = new ArrayList<Player>()
+territories : Territory = new ArrayList<Territory>()
+update() : void
+loadGame() : void
-territory
*
-territories
*
-territoriesState
<<enumeration>>
TERRITORIES
ALASKA
NORTH_WEST_TERRITORY
GREENLAND
ALBERIA
ONTARIO
EASTERN_CANADA
WESTERN_UNITED_STATES
EASTERN_UNITED_STATES
CENTRAL_AMERICA
VENEZUELA
PERU
BRAZIL
ARGENTINA
ICELAND
GREAT_BRITAIN
SCANDINAVIA
RUSIA
NOTHERN_EUROPE
WESTERN_EUROPE
NORTH_AFRICA
EGYPT
EAST_AFRICA
CENTRAL_AFRICA
SOUTH_AFRICA
MADAGASCAR
MIDDLE_EAST
AFGHANISTAN
URAL
SIBERIA
YAKUTSK
IRKUTSK
MONGOLA
CHINA
INDIA
SOUTHEAST_ASIA
JAPAN
KAMCHATKA
NEW_GUINESS
INDONESIA
WESTERN_AUSTRALIA
EASTERN_AUSTRALIA
TerritoryGraph
<<Property>> -territories : GraphNode = new ArrayList<GraphNode>()
+addTerritory(territory : Territory) : boolean
+removeTerritory(territory : Territory) : boolean
+connectTerritory(sourceTerritory : Territory, targetTerritory : Territory) : boolean
+checkConnect(sourceTerritory : Territory, targetTerritory : Territory) : boolean
-findGraphNode(territory : Territory) : GraphNode
+DEFAULT_RISK_TERRITORY_GRAPH
-graphConnected
DefaultRiskMode
+DEFAULT_RISK_TERRITORY_GRAPH : TerritoryGraph
+destroyTerritoryGraph() : void
+loadTerritoryGraph() : void
<<enumeration>>
CONTINENTS
NORTH_AMERICA
SOUTH_AMERICA
EUROPE
AFRICA
ASIA
AUSTRALIA
continent
Territory
-territoryName : String
<<Property>> -unitNumber : int
-playerCaptured : Player
-graphConnected : TerritoryGraph
+Territory(territoryName : String)
+getName() : String
+setPlayer(playerCaptured : Player) : void
+getPlayer() : Player
+connectToGraph(graphToConnect : TerritoryGraph) : void
+stateCopy() : Territory
+addUnits(unitToAdd : int) : void
+removeUnits(unitToRemove : int) : boolean
+isCombatableWith(target : Combatable) : boolean
+print() : void
+territories
Combat
+MAX_ATTACK_UNIT : int = 3
+MIN_ATTACK_UNIT : int = 2
+MAX_DEFENSE_UNIT : int = 2
+MIN_DEFENSE_UNIT : int = 1
+UNIT_LOSS_PER_DIE_ROLL : int = 1
-sourceTerritory : Territory
-targetTerritory : Territory
-sourceDice : Dice
-targetDice : Dice
+Combat(sourceTerritory : Territory, targetTerritory : Territory)
+combatPerRoll() : boolean
+combatTillCapture() : void
+combatable(sourceTerritory : Territory, targetTerritory : Territory) : boolean
+activeMode
+territories
-sourceTerritory
-targetTerritory
<<Interface>>
Combatable
+isCombatableWith(target : Combatable) : boolean
GameLoader
+activeMode : GAME_MODE
+territories : Territory
+loadTerritoryGraph(loadMode : GAME_MODE) : void
+destroyTerritoryGraph(destroyMode : GAME_MODE) : void
<<enumeration>>
GAME_MODE
DEFAULT
ANY_TYPE
DefaultRiskTerritory
~continent : CONTINENTS
~territory : TERRITORIES
+DefaultRiskTerritory(territoryName : String, territory : TERRITORIES, continent : CONTINENTS)
territory

Territory:

public getName() - this method returns the name of the territory

public setPlayer() - this method sets the player to a specific territory

public getPlayer() - this method returns the player who owns the specific territory

public addUnits() - this method is used to add units to a specific territory

public removeUnits() - this method removes units from a specific territory

public isCombatableWith() - this method finds if the selected territory can be combatted

TerritoryGraph:

public addTerritory() - this method adds a territory into specied territory graph

public connectTerritories() - this method connects two specified territories with each other, which makes them boundary related territories, that is, the interaction over such territories is possible afterwards

public checkConnected() - this method returns whether two territories connected in terms of boundary relation to use such data futher operations in the game

Player:

public getName() - this method returns the name of the player

public getColor() - this method return the color that is assigned to the player

public captureTerritory() - this method adds a specific territory to a player and shows it on map

public addUnitsToTerritory() - this method adds units to the specific territory of the player

public moveUnits() - this method moves units from one territory of the player to another

public attackTerritoryPerRoll() - this method used during combat to attack to the specified territory over a player but only one dice roll.

public attackTerritory() - this method is used when It is possible player attacks to the other territories

public captured() - this method detects if player captured the attacked territory or not

Combat:

public combatPerRoll() - this methods operates a combat but only one dice roll scenario and manipulates the territorial properties regarding with the result of combat

Public combatTillCapture() - this methods calls combatPerRoll over and over again until the combat resulted with a capture or defeat

public combatable() - this methods checks whether a combat can be executable over two specified territories by checking players, connections, the unit number

Game:

public update() - this method executes game mechanics calling related methods of mentioned model classes

public loadGame() - this method loads the data to instantiate required game objects

GameState:

public static checkStates() - this method compares 2 states perevious and currState, according to the result gives cards to the player

public static extractGameState() - this methods returns a game state by checking the current state of game objects, which stores required data to compare the state of game between turns to distribute cards among players

Dice:

public rollDice() - this method generates a random variable

public getFaceNumbers() - this method returns the values of the dices

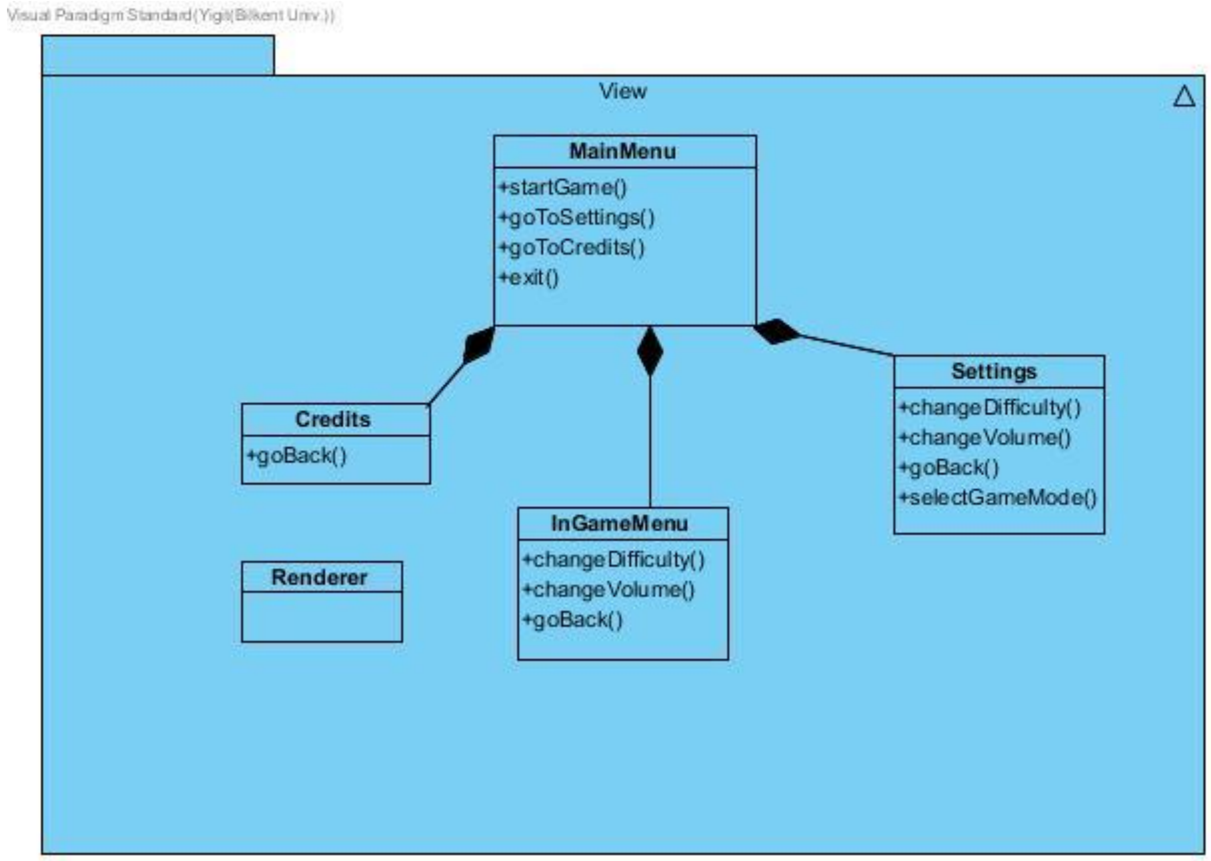
GameLoader:

public loadGameTerritoryGraph() - this method load the territory graph dependent upon the selected game mode, in which territory graph determines the relations over territories

Public destroyTerritoryGraph() - this method deletes the data of game objects from memory, which releases the memory

3.3 View Subsystem

This View Classes monitors the game to the users with respect to Input Managers and the model classes. The Game will react to the users input which are taken by the UIManagers and by using this input games continues by changing the model's properties. And the Viewclasses monitors the model classes and reflect the game to the screen.



MainMenu:

startGame: This method will be called by startGame() method to start the game

goToSettings: This method is calling the settings object and calling the proper user interface

goToCredits: This method opens a window that explains the game also gives information about the designers

exit: Terminates the program

InGameMenu:

changeDifficulty: Setting the difficulty level of the game

changeVolume: As the Input Controller calls this method, this method sets the volume as muted unmuted or as a given number

goBack: Returns to the game

Settings:

changeDifficulty: Setting the difficulty level of the game

changeVolume: As the Input Controller calls this method, this method sets the volume as muted unmuted or as a given number

goBack: Returns to the main menu

selectGameMode: Changes the game mode as a single player or multiplayer

Credits:

goBack: Returns to the main menu

4. Low Level Design

4.1 Design Pattern

Certain design pattern will be used in our project to make our system more understandable. In this section there will be examples given and explained why we used them.

4.1.1 Facade Design Pattern

Facade Design Pattern is known as a design pattern which helps designers to hide complexity of the systems and provides a huge but a single interface that users can interact with. It increases the simplicity of the program by making that class to handle all kinds of inputs and outputs in that particular subsystem.

4.2 Object design trade-offs

Memory - Performance: As it is an issue in all types of programs, we have faced the same question that is related to memory and time complexity. Therefore, we can state the fact that Memory - Performance problem was the hardest trade-off that cannot be ignored. On the other hand, it was clear that the performance was more important than memory in our trade. Mostly in games which require to run highly complex graphical engines also bring huge data with them, and that increases the memory size in the program. However, in our game since we use only 2D graphics, it is highly unlikely that we would choose memory over performance. Therefore, the aim in that manner was increasing our performance in order to keep the game simple and fun for the user aspect.

5. Glossary

MVC:	Model-View-Controller architecture

6. References

https://www.tutorialsteacher.com/mvc/mvc-architecture

https://medium.com/code-smells/model-view-controller-architecture-78a855b8ab56