Bilkent University

Department of Computer Engineering

CS - 319 Object-Oriented Software Engineering

Term Project – Design Report

Iteration 1

Project Name: Risk

Group No: 1F

Group Members: Rumeysa Özaydın
Merve Kılıçarslan
Ahmet Serdar Gürbüz
Elnur Aliyev
Osman Burak İntişah

1) Introduction

1.1 Purpose of the system

Risk is a desktop game which aim to entertain players while improving their strategy skills. Desktop version will be implemented in order to provide with a user friendly interface which enables players to play the game easy, enjoyable and mind challenging. Our Risk game differs from the original board game with some extra features and rules. There will be bonus card implemented which are single use cars that helps the player sabotage other players. These cards will be drawn if you successfully conquered a territory in a particular turn. Bonus cards will be allowed to be used at the end of every turn hence it will be affecting the next turn’s rules. Moreover, a new object will be added to the game called “castle” and “coins”. Coins will be given to players in the beginning of the game and the players will continue winning coins with respect to size of their territories. Coins earned can be increased or decreased in very turn. Coins will be used to buy “castle” or soldier in the main game frame. “Castle” will be an object that a player can place in their territories. A territory can have more than one castles and castle will protect the territory by allowing the attacker to use only 1-2 soldier in battle mode. Since Risk is a strategy and diplomacy based game, implementation will support the multiplayer mode only. This implementation will allow people to have interactions and conversations when attacking and taking provinces. Risk is a logical game hence players wants to think a lot but since the game is not in real life circumstances, measures should be taken to prevent long waiting times. In order to decrease waiting time of other player, a limited time slot will be given to players in which they will have to act or their turn will passed.

1.2 Design goals

1.2.1 Ease of Use

Risk is a strategy game so the use have to spend time to understand the rules before beginning. Once the user understood the rules, they can easily adapt the interface of the game. The main purpose is to conquer all provinces in the map.

1.2.2 Reliability

We don’t store user’s data in anywhere since the game does not require to do so. The players can play the game from the same network or from the same computer. Therefore, there will not be any security problems.

1.2.3 Efficiency

The goal is to make the game as efficient as possible. To address this goal, we organized classes in a minimal way and we did not store any unnecessary information. Yet, since the game can be played from different computers which requires updating the game from all the computers after each turn, there might be some performance issues.

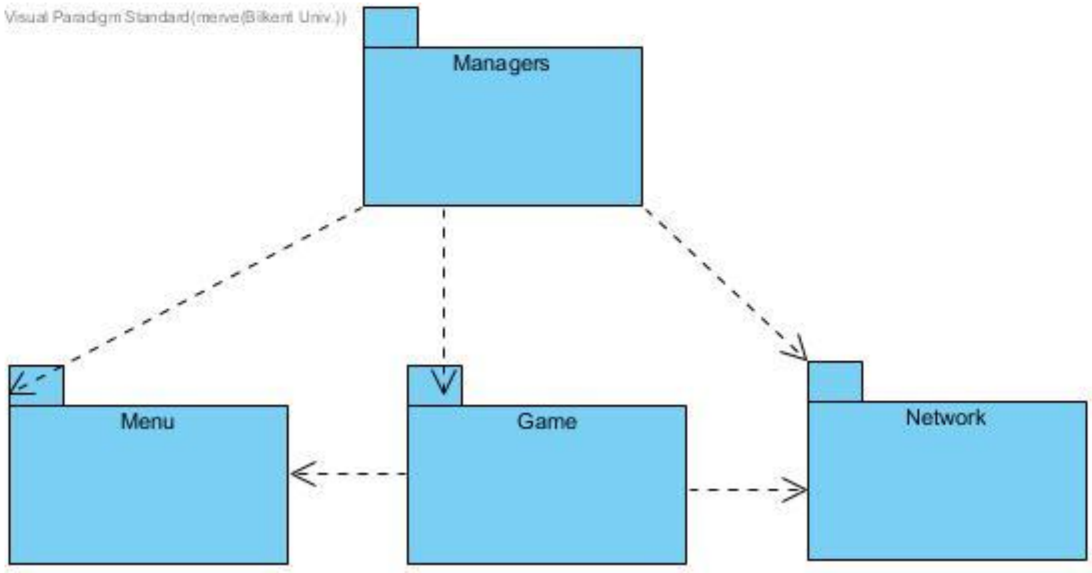
1.2.4 Extendibility

The game will be implemented with an extendible and well organized design so that new features and new modes can be easily added afterwards. In addition, it can be turned into online multiplayer game.

2) High-level software architecture

2.1 Subsystem Decomposition

We have decided to decompose our system in order to see dependencies between classes in the same subsystem. Hence, our system is easy to understand and easy to implement in coding environment. For example, game will start with the menu and screens for the different options for the game will be controlled here. Game package will consists of players, their instances and game logic etc. since they have dependency to each other and partitioning them from other packages results in flexibility and easy distribution of roles in the group members. Network part of the system will be implemented using packages and separation from other parts ensures to identify errors and package related issues easier.



2.1.1 Managers

This is the part that for managing game state and resources like a game engine. Managers controls the file management, data operations, screen transitions and data to handle the network relations.

2.1.2 Menu

This subsystem is used for displaying menu interfaces to the user. It is also responsible for processing image-based inputs (with Listeners) and directing the user to selected screens. However, the Menu do not include any real logic about the game. Therefore, all the game components are in Game subsystem.

2.1.3 Game

It is the core system for the game. This subsystem is used for enabling the user playing the game on the screen. It will keep all the data for the current state of the game.

2.1.4 Network

This subsystem enables users to play from different computers. It controls the changes for each state after the turn has finished by the user. And updates the current situation for all the players in the game.

2.2 Hardware/software mapping

Game of risk will be implemented in C++ programming language which makes our implementation easier and faster. For screen displays, sound outputs and mouse inputs we will use SFML libraries. Our graphics and sound effects will be mapped to SFML’s hardware interface. Moreover, “Risk” will require a keyboard as a hardware requirement. It will be used for I/O tool of the game. Users will control menu selections, attack movements, soldier movements and activities with coin (buying a castle). For the storage of the game, we will use text files.

2.3 Persistent data management

“Risk” does not require any complex data storage system or database. The instances of the game will be stored in the hard drive of the user. Some of the files will be arranged in a way that cannot be modified during the game, but, some will be arranged to be instantiated during the game.

2.4 Access control and security

As mentioned in previous chapters (2.3) “Risk” does not require any database. Yet, in order to make the game available to be played multiplayer from different computers, we will use sockets which enables players to join the game from the same network but it won’t give any chance to do any malicious actions or data leaks.

2.5 Boundary conditions

2.5.1 Application Setup
“Risk” will have an executable .exe extension. We will provide corresponding libraries for the user in installer so that they can run our game in a Windows Machine.

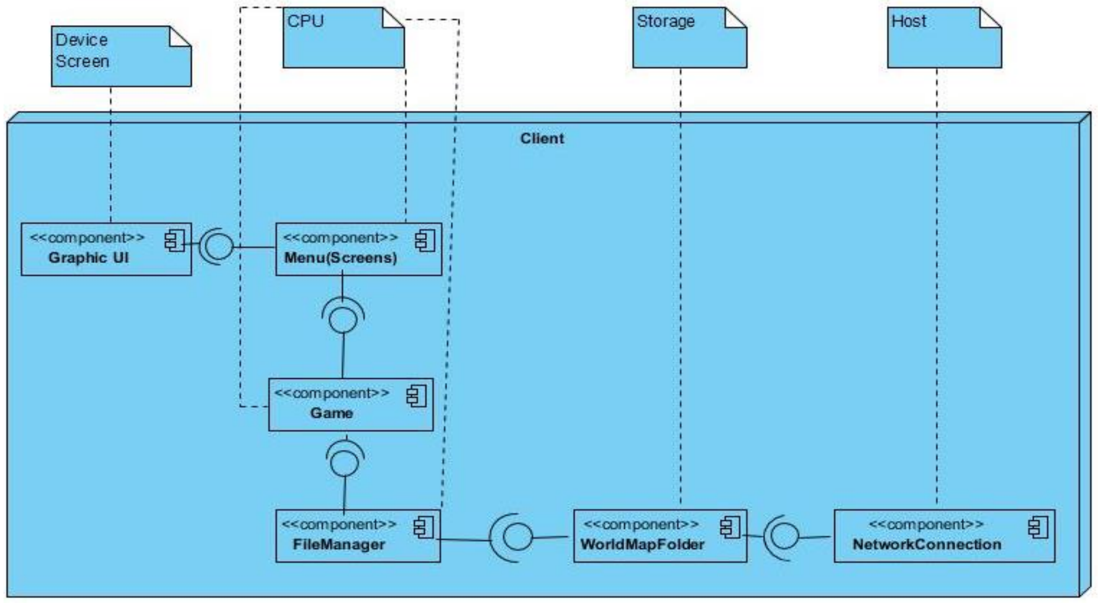
2.5.2 Terminating the Application
In order to be terminated, the exit button in the game window should be clicked.

2.5.3 Input / Output Exceptions
There might be some file reading problems in the game due to an exception or an invalid challenge. The game may start without sound and images if such errors occurs. In addition, if the user enters a too long or invalid string an error messages will be displayed.

2.5.4 Critical Error
If the game collapses in any case, all the data will be lost.

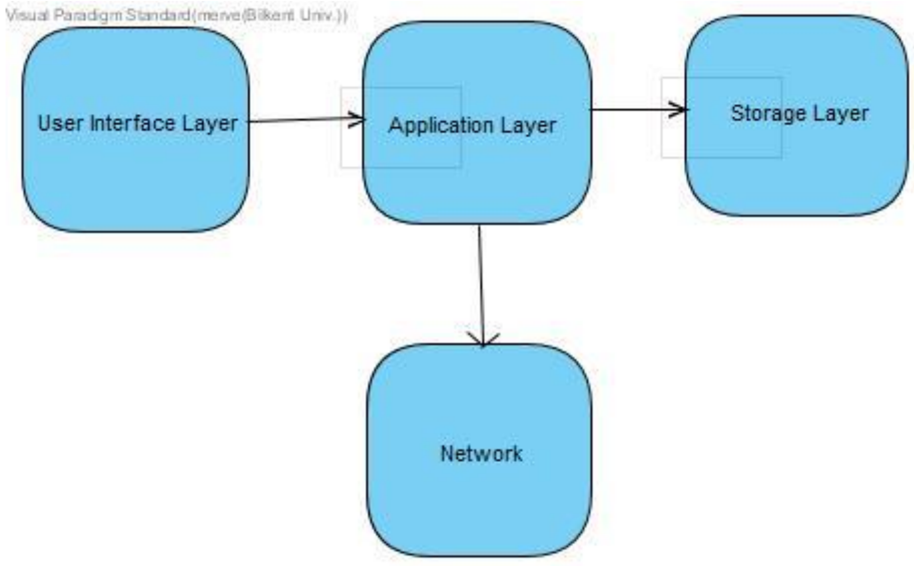
3) Subsystem services

The services and their relations can be summarized as the following deployment diagram.



Device Screen
CPU
Storage
Host
Client
<<component>> Graphic UI
<<component>> Menu(Screens)
<<component>> Game
<<component>> FileManager
<<component>> WorldMapFolder
<<component>> NetworkConnection

These services belongs to different layers however they depend on each other. Layer flow is shown in the following figure.



3.1 User Interface Layer

We have decided to use SFML libraries to implement user interface. Our GUI structure will be a single component that generates interface. The maps and other instances will be initialized and displayed on the screen. Our layer will also have soldiers, bonus cards and dice as a more complex part of the UI. Buttons and functional components for the menu additional screens will be implemented using SFML library basic functions.
Mouse Listener will be embedded into the UI layer to get the mouse events in the game. Mouse Listener accuracy especially important for the game since mouse parser’s coordinates will be the main indicator for the select provinces. Mouse Listener coordinates will be directed to the Application Layer. Soldier counts and soldier replacements representations will be done on the UI by Application Layer.

3.2 Application Layer

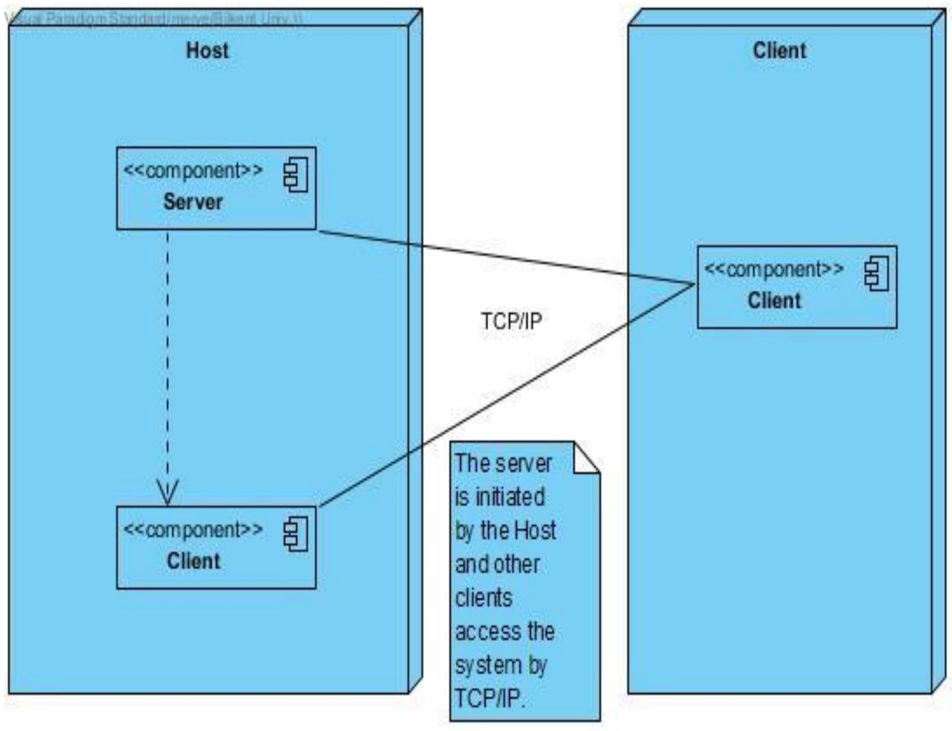
Getting the accurate province border in the World Map will be challenging for the implementation. In order to overcome this challenge we have decided to get the location of the provinces by using colors. Game Maps will consist of two layer. In the first layer different colors will be assigned to different provinces however players will not be able to see the details due to second layer. In the second layer of the map, actual World Map with provinces and continents will be represented. Provinces clicked on and the coordinates will be taken according to the first layer of the map.
Every play will have different attributes for each turn interchangeably and the updates for some attributes will also be updated in the other player’s screens. We have decided to put these attributes into the Application Layer in order to prevent continuous duplication. Updates will be accessed via network from the World Map and the constantly changing data will be accessed by players.

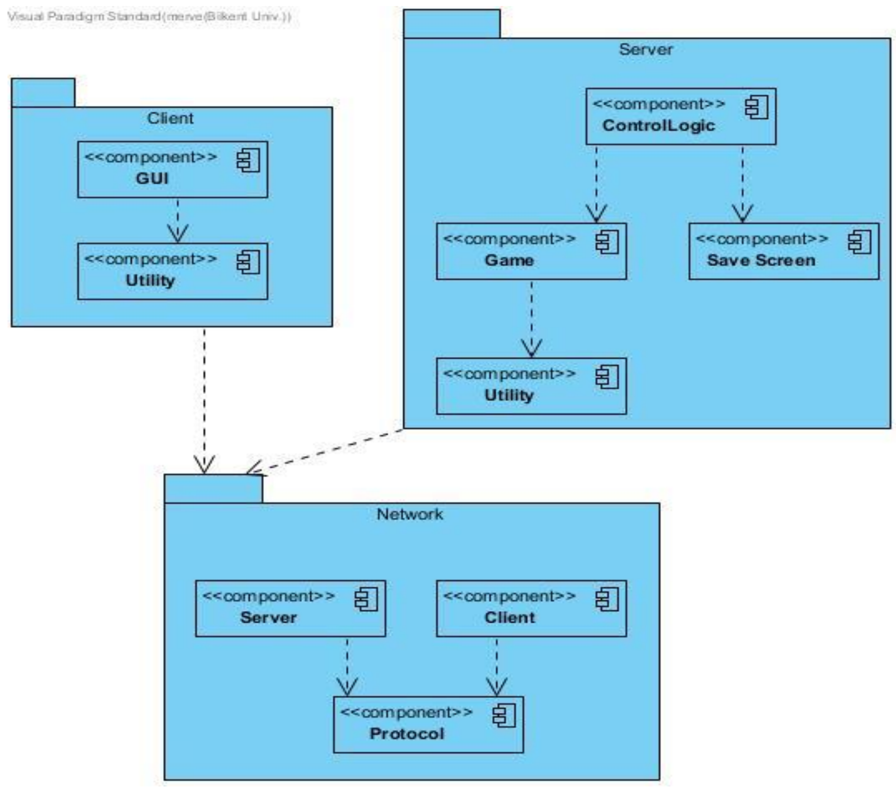
3.3 Storage Layer

Our game will require access to storage devices in order to get Credits and World Map, soldier, castle and dice pictures. These will be interact by Application and UI layer. However, we will not require storage for storing the game state since game will not be saved.

3.4 Network

Our game will use sockets in order to provide an multiplayer game for the players who are in the same network. The sockets will act like a channel and provide the current state of the game for each player after a turn. The information of the players will be accessible only for that player.


Host
<<component>>
Server
<<component>>
Client
TCP/IP
Client
<<component>>
Client
The server is initiated by the Host and other clients access the system by TCP/IP.



4) Low-level design

4.1 Object Design Trade-Offs

4.1.1 Functionality vs Usability

In our Risk game, a control system in the main game screen is used to control the game. Game map and the options for the player's turn such as attack, build castle, release turn etc. are present on the side of the map in the main screen. All inputs for the game are taken from mouse in order to increase usability. Since the Risk Game is complicated and has many functionalities, increasing the usability with only one input device is preferred. However, we could implement a keyboard input to indicate the number of soldiers which will be used for an attack.

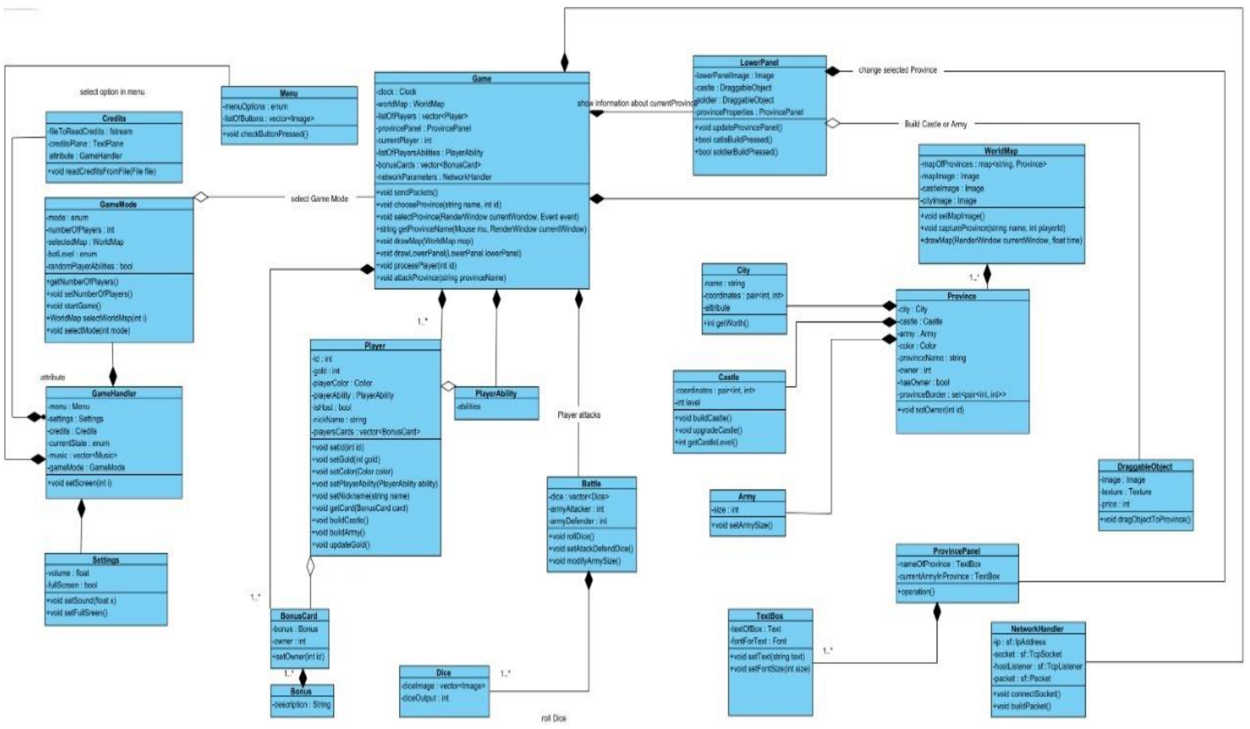
4.1.2 Efficiency vs Portability

Our game will be implemented in C++. We have chosen C++ since the game have many interactive layers and will have a large map with zooming properties hence we have chosen efficiency over portability. We concluded that if the game has fast response time, users will be more comfortable to play such a complex game. Moreover, game can only be played with multiple participants. To achieve network connections between the players sockets, packets, web requests and file transfers are implemented using SFML library.

4.1.3 Variety vs Rapid Development

The Risk game has many functionalities due to complex rules. There are different restrictions with respect to different moves. These rules will be implemented to game however different modes of the game will not implemented due to time limitations. In the physical Risk game there are different functionalities of the objects for different game modes however our Risk implementation supports only one game mode. It can be said that our game mode is a mixture of 3 game modes in real game and some new innovations.

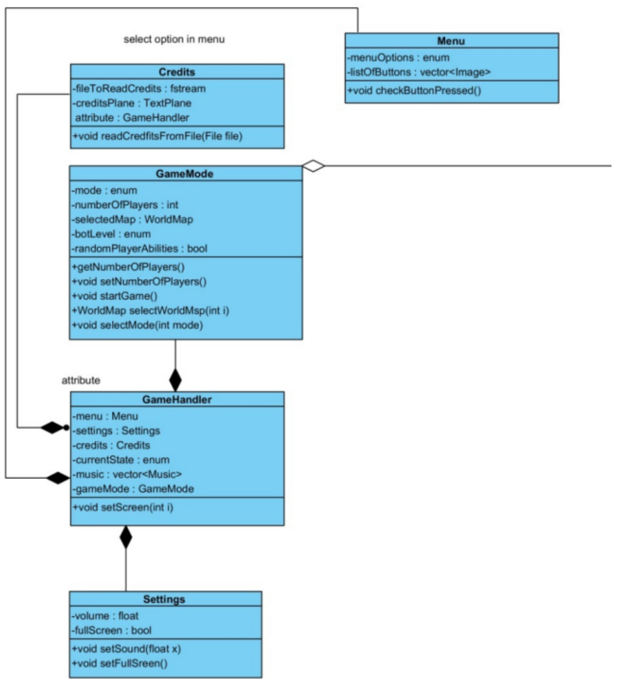
4.2 Final object design



```
Menu
-menuOptions : enum
-listOfButtons : vector<Image>
+void checkButtonPressed()

Credits
-fileToReadCredits : fstream
-creditsPlane : TextPlane
attribute : GameHandler
+void readCreditsFromFile(File file)

GameMode
-mode : enum
-numberOfPlayers : int
-selectedMap : WorldMap
-botLevel : enum
-randomPlayerAbilities : bool
+getNumberOfPlayers()
+void setNumberOfPlayers()
+void startGame()
+WorldMap selectWorldMap(int i)
+void selectMode(int mode)

GameHandler
-menu : Menu
-settings : Settings
-credits : Credits
-currentState : enum
-music : vector<Music>
-gameMode : GameMode
+void setScreen(int i)

Settings
-volume : float
-fullScreen : bool
+void setSound(float x)
+void setFullSreen()

Game
-clock : Clock
-worldMap : WorldMap
-listOfPlayers : vector<Player>
-provincePanel : ProvincePanel
-currentPlayer : int
-listOfPlayersAbilities : PlayerAbility
-bonusCards : vector<BonusCard>
-networkParameters : NetworkHandler
+void sendPackets()
+void chooseProvince(string name, int id)
+void selectProvince(RenderWindow currentWondow, Event event)
+string getProvinceName(Mouse mu, RenderWindow currentWindow)
+void drawMap(WorldMap map)
+void drawLowerPanel(LowerPanel lowerPanel)
+void processPlayer(int id)
+void attackProvince(string provinceName)

Player
-id : int
-gold : int
-playerColor : Color
-playerAbility : PlayerAbility
-isHost : bool
-nickName : string
-playersCards : vector<BonusCard>
+void setId(int id)
+void setGold(int gold)
+void setColor(Color color)
+void setPlayerAbility(PlayerAbility ability)
+void setNickname(string name)
+void getCard(BonusCard card)
+void buildCastle()
+void buildArmy()
+void updateGold()

PlayerAbility
-abilities

BonusCard
-bonus : Bonus
-owner : int
+setOwner(int id)

Bonus
-description : String

Dice
-diceImage : vector<Image>
-diceOutput : int

Battle
-dice : vector<Dice>
-armyAttacker : int
-armyDefender : int
+void rollDice()
+void setAtackDefendDice()
+void modifyArmySize()

LowerPanel
-lowerPanelImage : Image
-castle : DraggableObject
-soldier : DraggableObject
-provinceProperties : ProvincePanel
+void updateProvincePanel()
+bool catleBuildPressed()
+bool soldierBuildPressed()

WorldMap
-mapOfProvinces : map<string, Province>
-mapImage : Image
-castleImage : Image
-cityImage : Image
+void setMapImage()
+void captureProvince(string name, int playerId)
+drawMap(RenderWindow currentWindow, float time)

City
-name : string
-coordinates : pair<int, int>
-attribute
+int getWorth()

Castle
-coordinates : pair<int, int>
-int level
+void buildCastle()
+void upgradeCastle()
+int getCastleLevel()

Army
-size : int
+void setArmySize()

Province
-city : City
-castle : Castle
-army : Army
-color : Color
-provinceName : string
-owner : int
-hasOwner : bool
-provinceBorder : set<pair<int, int>>
+void setOwner(int id)

DraggableObject
-image : Image
-texture : Texture
-price : int
+void dragObjectToProvince()

ProvincePanel
-nameOfProvince : TextBox
-currentArmyInProvince : TextBox
+operation()

TextBox
-textOfBox : Text
-fontForText : Font
+void setText(string text)
+void setFontSize(int size)

NetworkHandler
-ip : sf::IpAddress
-socket : sf::TcpSocket
-hostListener : sf::TcpListener
-packet : sf::Packet
+void connectSocket()
+void buildPacket()
```

In this section we are going to discuss about our classes and their design. We have divided our classes in order to maintain a understandable diagram. One of our parts is for menu classes and the other one is for the game play. The following subsections will demonstrate the further details of the classes.

4.2.1 Menu Classes



4.2.1.1 Menu

Menu class instantiates **menuOptions** and direct user to the selected option by keeping the options in a vector called **listOfButtons**.

4.2.1.2 GameHandler

According to the user input, it handles the menu options. It instantiates menu, settings, credits, currentState music and gameMode.

4.2.1.3 GameMode

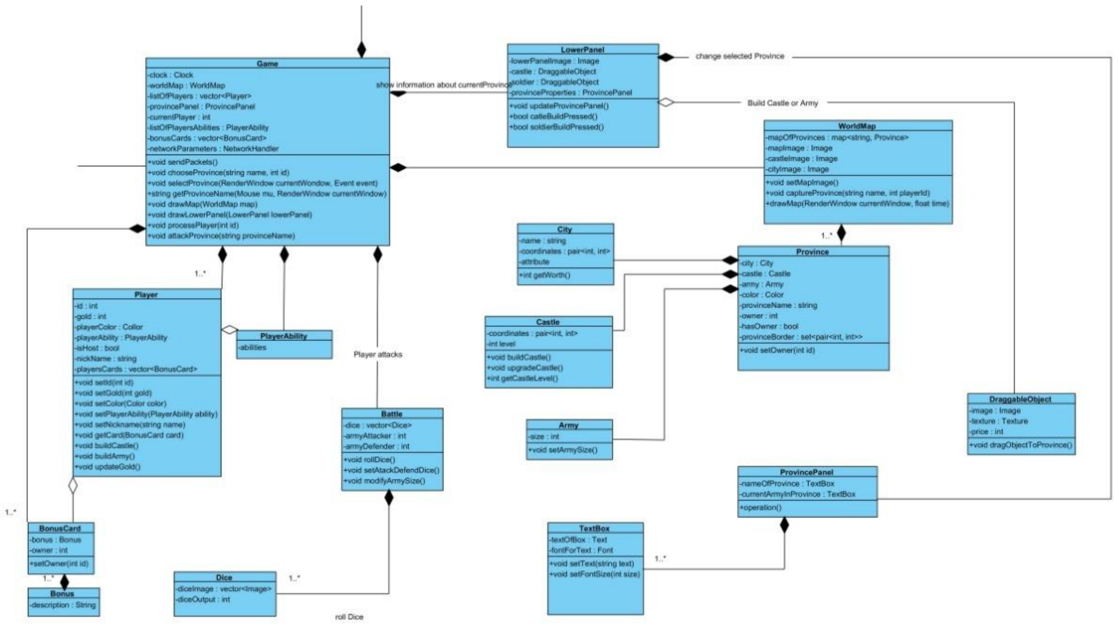
According to the user input for creating a game or joining a game, it will ask for game settings. In other words, it instantiates **mode, numberOfPlayers, selectedMap, randomPlayerAbilities**. When the host of game press start button the game starts by calling the **Game** class.

4.2.1.4 Settings
It enables users to sets the screen resolutions and volume level.

4.2.1.5 Credits
The user will access this screen from the home screen using “Credits” button. Developers of the game and their contact information are listed in addition to the GitHub link of the game.

4.2.1.6 HowToPlay
The user will be informed by a tutorial which will be either in video format or gifs.

4.2.2 Game


Game
-clock : Clock
-worldMap : WorldMap
-listOfPlayers : vector<Player>
-provincePanel : ProvincePanel
-currentPlayer : int
-listOfPlayersAbilities : PlayerAbility
-bonusCards : vector<BonusCard>
-networkParameters : NetworkHandler
+void sendPackets()
+void chooseProvince(string name, int id)
+void selectProvince(RenderWindow currentWondow, Event event)
+string getProvinceName(Mouse mu, RenderWindow currentWindow)
+void drawMap(WorldMap map)
+void drawLowerPanel(LowerPanel lowerPanel)
+void processPlayer(int id)
+void attackProvince(string provinceName)
show information about currentProvince
LowerPanel
-lowerPanelImage : Image
-castle : DraggableObject
-soldier : DraggableObject
-provinceProperties : ProvincePanel
+void updateProvincePanel()
+bool catleBuildPressed()
+bool soldierBuildPressed()
change selected Province
Build Castle or Army
WorldMap
-mapOfProvinces : map<string, Province>
-mapImage : Image
-castleImage : Image
-cityImage : Image
+void setMapImage()
+void captureProvince(string name, int playerId)
+drawMap(RenderWindow currentWindow, float time)
1..*
City
-name : string
-coordinates : pair<int, int>
-attribute
+int getWorth()
Province
-city : City
-castle : Castle
-army : Army
-color : Color
-provinceName : string
-owner : int
-hasOwner : bool
-provinceBorder : set<pair<int, int>>
+void setOwner(int id)
Castle
-coordinates : pair<int, int>
-int level
+void buildCastle()
+void upgradeCastle()
+int getCastleLevel()
Army
-size : int
+void setArmySize()
DraggableObject
-image : Image
-texture : Texture
-price : int
+void dragObjectToProvince()
ProvincePanel
-nameOfProvince : TextBox
-currentArmyInProvince : TextBox
+operation()
TextBox
-textOfBox : Text
-fontForText : Font
+void setText(string text)
+void setFontSize(int size)
1..*
1..*
Player
-id : int
-gold : int
-playerColor : Collor
-playerAbility : PlayerAbility
-isHost : bool
-nickName : string
-playersCards : vector<BonusCard>
+void setId(int id)
+void setGold(int gold)
+void setColor(Color color)
+void setPlayerAbility(PlayerAbility ability)
+void setNickname(string name)
+void getCard(BonusCard card)
+void buildCastle()
+void buildArmy()
+void updateGold()
PlayerAbility
-abilities
Player attacks
Battle
-dice : vector<Dice>
-armyAttacker : int
-armyDefender : int
+void rollDice()
+void setAtackDefendDice()
+void modifyArmySize()
1..*
BonusCard
-bonus : Bonus
-owner : int
+setOwner(int id)
1..*
Bonus
-description : String
Dice
-diceImage : vector<Image>
-diceOutput : int
1..*
roll Dice

4.2.2.1 Game
The game class includes all the necessary information such as **clock**, **worldMap**, **listOfPlayers**, **provincePanel**, **currentPlayer**,

listOfPlayerAbilities, **bonusCards** and **networkParameters**. Also it keeps the changes during the game.

4.2.2.2 WorldMap

It includes castleImage, cityImage and mapImage to show the user. Also, it keeps a map of strings and Provinces to understand which province has been selected from the user.

4.2.2.3 Province

It has attributes which are:
provinceBorder: It sets the border of each province.
hasOwner: It returns if the province is taken or not.
owner: It returns the number of the player who has the province.
provinceName: It keeps the unique name of the province.
color: It keeps the color of the province. Which enables to understand which continent the province is in.
city: Which is an object from the City class.
castle: Which is an object from the Castle class.
army: Which is an object from the Army class

4.2.2.4 Army

It instantiates the size of the army.

4.2.2.5 Castle

It instaties the coordinates where the castle is built and the level of the castle.

4.2.2.6 LowerPanel

Object of this class will enable user to add castle and soldier to the selected province. It will include castle and soldier images which are going to be draggable. Also it will show the selected provinceProperties.

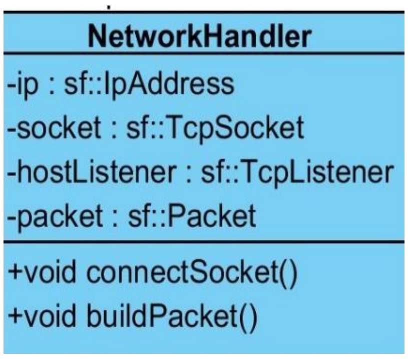
4.2.2.7 ProvincePanel

It will have properties which are nameOftheProvince and currentArmyInProvince. And it will display this properties.

4.2.2.8 DraggableObject

It will display the image, texture and price of the draggable object on the lower panel.

4.2.3 NetworkHandler



Object of this class will be capable of communicating in the server side of the game. It will send sockets from one computer to the other after clicking **end turn** button. It will keep IP Adress' of the computers.

4.3 Packages

4.3.1 Internal Subsystem Packages

We split our classes into subsystem they belong. Every subsystem has its package. This organization will make the implementation process more efficient. The following packages are set in our game: **managers, game, menu and network.**

4.3.2 External Packages

Our external packages are from SFML library. Some of the packages that we will use:

- sfml-audio.lib will be used to integrate audio in the game.
- sfml-window.lib will be used to create window for the game.
- sfml-graphics.lib will be used to implement User Interface.