Bilkent University
Department of Computer Engineering

CS - 319 Object-Oriented Software Engineering

Term Project - Design Report
Iteration 2

Project Name : Risk
Group No: 1F
Group Members: Rumeysa Özaydın
Merve Kılıçarslan
Ahmet Serdar Gürbüz
Elnur Aliyev
Osman Burak İntişah

Table of Contents

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 with an user friendly interface which enables players to play the game easily and mak experience enjoyable and mind challenging.

The fundamental rules represented in the game will implemented. There will be continents and the continents will have territories. Each player will have territories and a player will battle with another in a territory. The battle specifications will be declared by the player (soldier number vs.) and the outcome of the battle will be concluded by die tosses.

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 Usability

One of the most important non-functional properties of the game will be ease of use of the game. The players will be able to play the game without making any complex operations.In the game, every control has to be easily accessible for a better user experience. In order to achieve this purpose, controls (buttons etc.) will be placed directly on the screen and can be accessed by just one click. All the operations a player has to do in their turn will located on the main game screen. Comprehensible image icons will be placed around the world map, hence moves will be made on this screen.

1.2.2 Performance

The game can be played with players who are connected to each other on the same network. However, to provide a better and smooth user experience, the game state should be updated at most in 45 seconds when a player releases its turn.

1.2.3 Supportability

In order for more player to play this game, it has to be compatible with more computers. To make this happen, the game is implemented in C++ for Windows 64 bit which is commonly used.

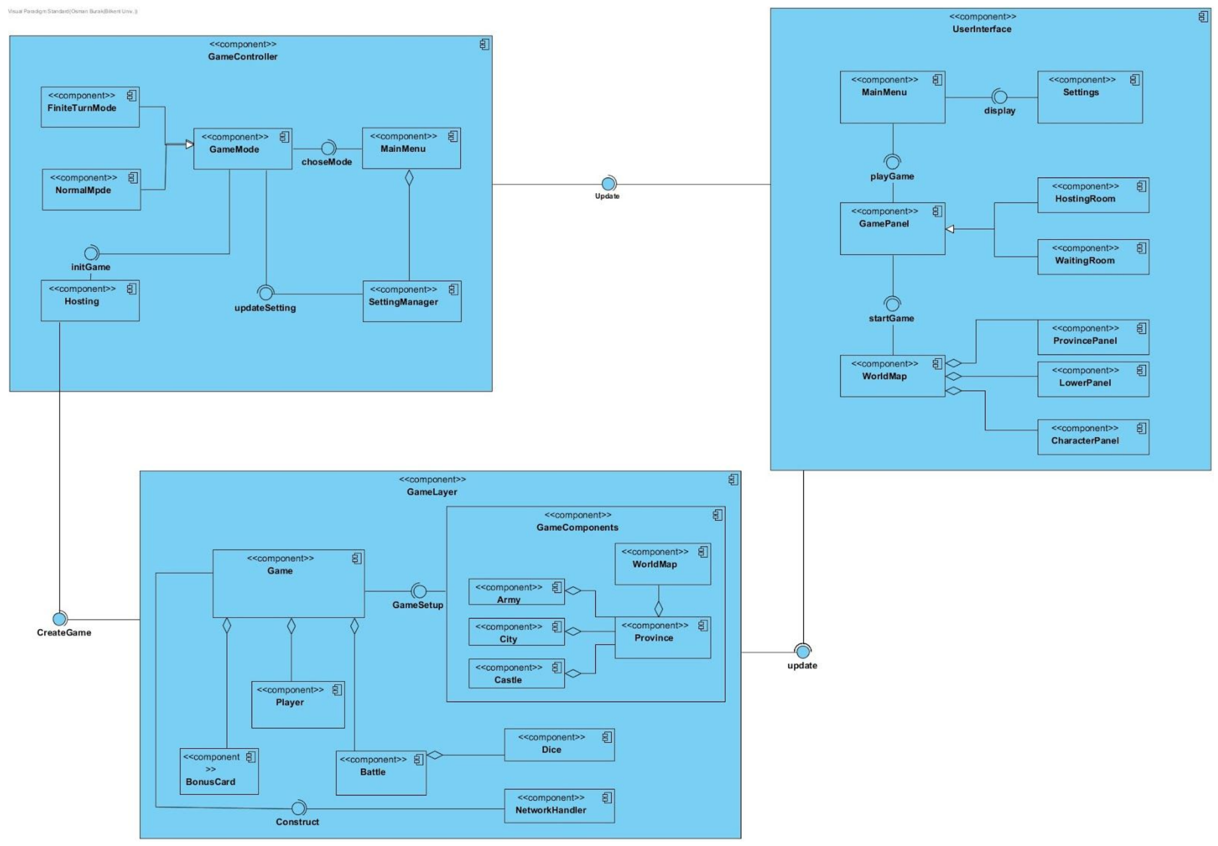
2. High-level software architecture

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. When decomposing our system architecture, the MVC (Model-View-Controller) pattern is used. As developers we have decided to divide the system into 3 manageable pieces deploying MVC pattern properties.

The subsystem decomposition is done, taking design goals into consideration. Moreover, the system is divided in such a way that the dependencies between the subsystems are minimum and the complexity of the system is less. Through the subsystem decomposition, assignment of the components of the system will be easier.

The subsystem decomposition in our system is as follows:



GameController Subsystem:The subsystem is the “controller” component of the MVC system. This component will manage the interaction between the player and the software. User inputs will be taken and the game state will be changed accordingly and the outcome(output) of the user actions will be updated through the servers.

User Interface Subsystem: This subsystem will be responsible for the User Interface and it corresponds to the “view” component of the MVC architectural style. It consists of the panels and the interfaces of the game’s main screen, settings screen and the main menu screen. This subsystem also contains buttons, listeners, frames of the game.

GameLayer Subsystem: This subsystem maintains the domain knowledge and it corresponds to the “model” component of the MVC style. The subsystem is the repository that keeps track of the all the different components in the game like the world map, provinces, players, battle, dice etc.

The subsystem decomposition is done so that the dependencies between the subsystems (coupling) are minimized. However, the dependencies(cohesion) within a subsystem and its components are maximized. This decomposition affects the implementation of the game positively since division of layer can be done more efficiently.

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 and a mouse 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). There will be draggle objects for soldier replacement and building castles etc. In battles, province selection and soldier selection will be done by clicking into the map. For the storage of the game, we will use text files.

The system will not work without network connection since single player is not available.

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. The game state will not be saved as a result a data management system is not required.

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.

Login system will not be used in the game hence no security measures will be regarded for the database or login system. The user will be able to change the settings and the modes of the game addition game’s functionalities.

The updated version of the map and the player attributes will be updated in other screens of the players after one of the player releases his/her turn. The updates will be handled by the SFML library components.

There will be no levels or achievements however with respect to bonus cards some of the functionalities of the users can be detained for a time period.

2.5 Control Flow

The game will be played by 4-6 players and the single player mode for the game is not available. Player will choose different options from the menus and will make decisions in the game. The game will be created according to the specifications in the menu and game attributes will be changed during the game according to the decisions made by players.

2.6 Boundary conditions

2.6.1.Application Setup

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

The system will not require the installation of new softwares.

In order to play the game multiplayer, every pc must be connect to same wifi or with connection can be supported with LAN. For the players to play the game, all players must be connected. After the host created game, other players will join the game and host will select the colors for each player.

2.6.2.Terminating the Application

In order to be terminated, the exit button in the game window should be clicked. However, termination can be done with operating system commands, too. Since game state is not saved, no measures are taken even if the game is closed in a critical situation. However, even if only one player exits the game, the game will be finished for all players.

2.6.3. Input/Output Exceptions

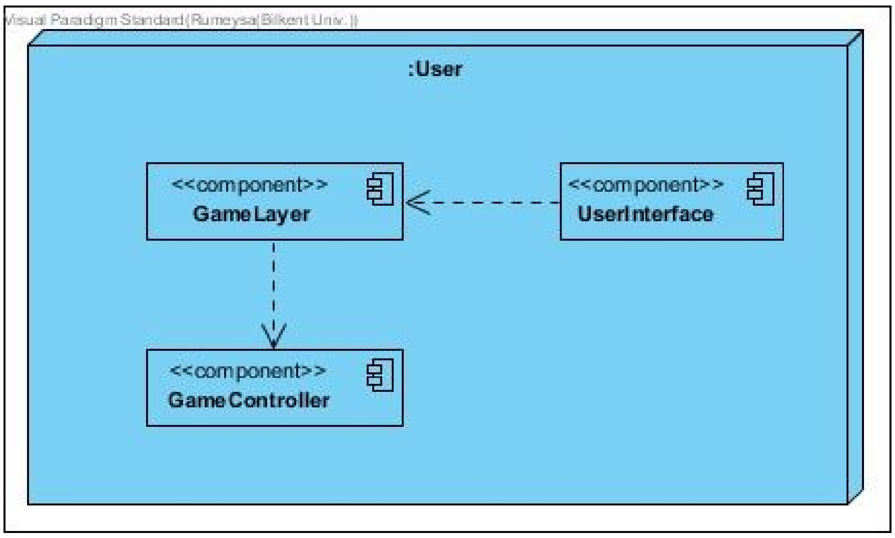
There might be some file reading problems in the game due to an exception or an invalid challenge. The game may start without sound and images if such errors occurs. In addition, if the user enters a too long or invalid string an error messages will be displayed.

2.6.4. Critical Error

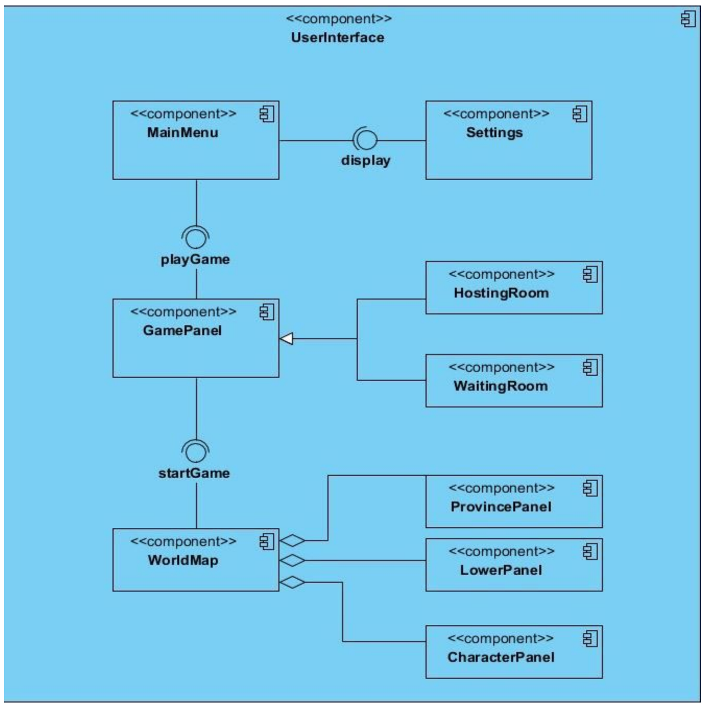
If the game collapses in any case, all the data will be lost during the game.

3. Subsystem services

The services and their relations can be summarized as the following deployment diagram. MVC model can be clearly seen from this figure.



3.1 User Interface



Subsystem is used for displaying menu interfaces to the user. This subsystem will be linked with the GameLayer in order to present an interactive experience for the user. The subsystem is responsible for the visual parts of the game like menus and main game screen.

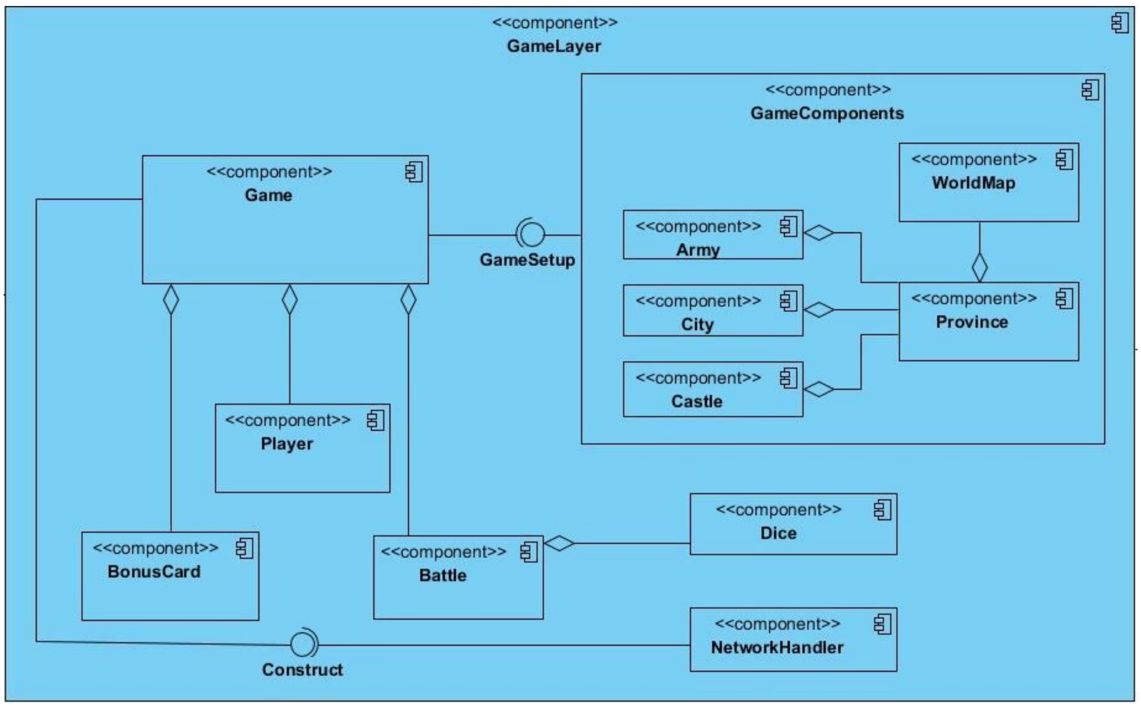
We have decided to use SFML libraries to implement user interface.Buttons and functional components for the menu additional screens will be implemented using SFML library basic functions.

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 GameLayer.

User Interface will be updated through the signals from the GameLayer.

Since other subsystems in the game does not directly depend on this subsystem, but interactions with the GameLayer is essential for the functionality. However, due to decoupling, game can be changed with changing this subsystem which makes the game easy to manipulate.

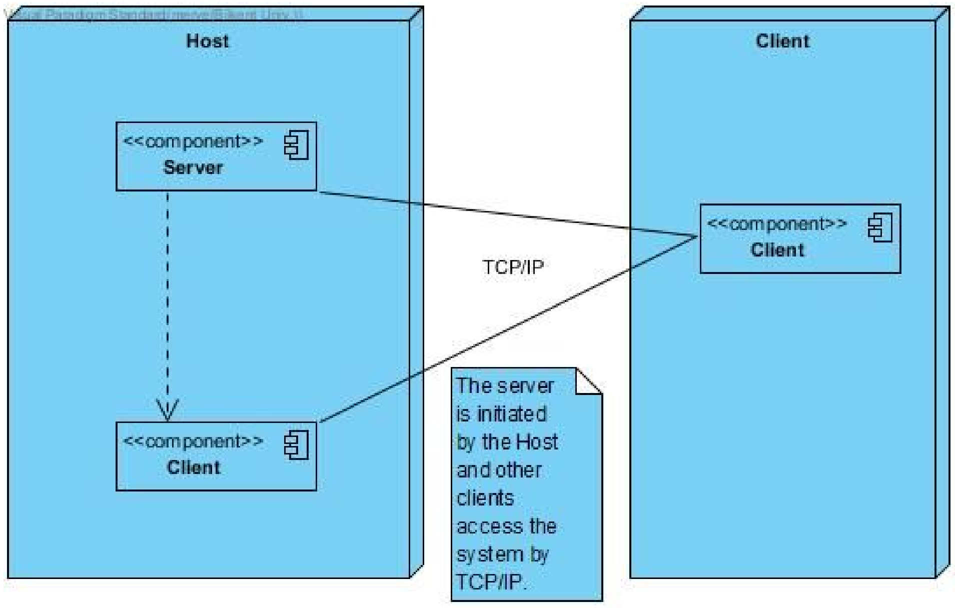
3.2 GameLayer



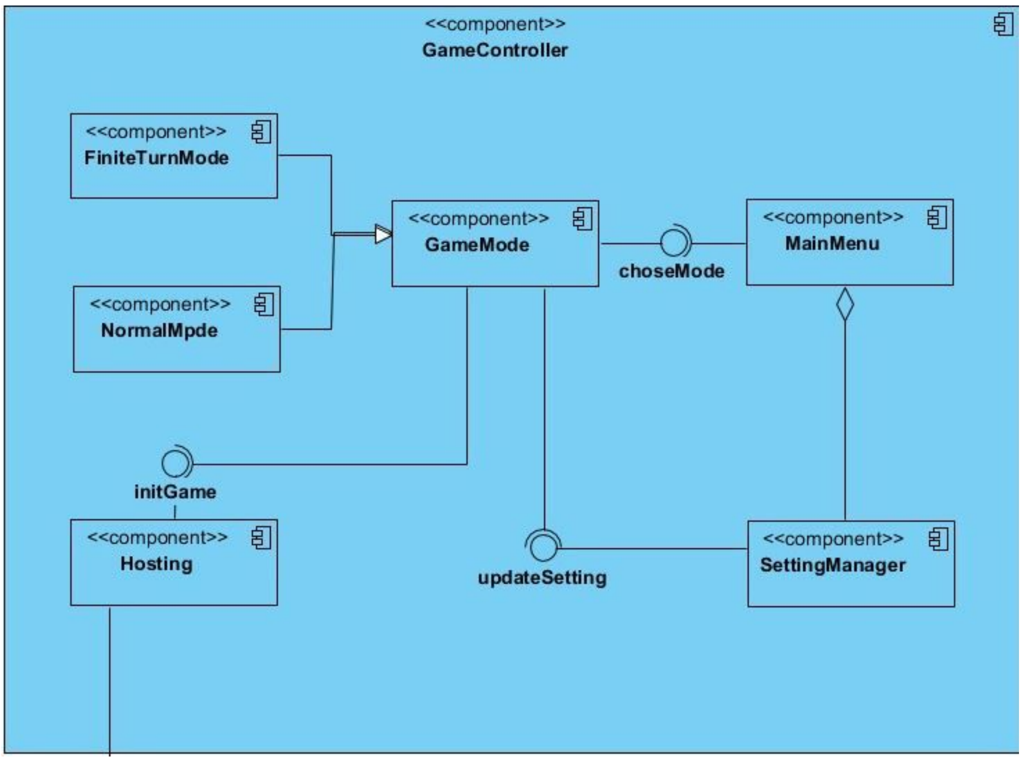
This subsystem includes components for game entities which are basically game objects. GameLayer has a main component called GameComponents which represent different types of game objects/entities. The WorldMap is composed of provinces and provinces are composed of armies, cities and castles. All the game entities can be updated by a signal came from the GameController subsystem. The components of the GameLayer can update its view by sending the proper signals with proper methods which make changes in the UserInterface subsystem. Also, the Game is consist of Player, BonusCard, Battle and NetworkHandler.

- **Player:** Represents the player and is consist of id, gold, color, nickname and bonus cards.

- **Battle:** Represents the battle in a game. It is consist of dice(which decides the winning side), armyDefender(the number of soldier that the defender has) and armyAttacker(the number of soldier that the attacker attacks).
- **NetworkHandler:** 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. The figure below shows the host-client interaction.


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.

3.3 GameController



This subsystem includes controls components. Game will start with the menu and screens for the different options for the game (like finite turn mode, normal mode or the options on the settings screen), and the game will be controlled in this subsytem. The controller components get data inputs from the components of the user interface subsystem and in terms of the actions has been chosen the related task evaluated for the manipulation of the game model. SettingManager controls the features if the game and has ability to change them such as sound level, music play etc. As it seen from the diagram host is initiating the game and it has the ability to create the game. GameController communicates with GameLayer subsystem. And the host is requesting the create the game option from this subsystem.

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. Only a finite turn mode will be implemented to finish the game in a fixed number of player turns. 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 Design Decision and Patterns

4.2.1 MVC Design Pattern

Our main subsystem decomposition has been done according to the MVC design pattern. This design pattern is suitable for our game structure since we can divide our software with respect to model, view and controller. Controller part is addressed as GameController and interactions are handled in this subsystem. Model is addressed as GameLayer and domain knowledge on entities is in this division. Lastly, UserInterface is the view and the interface components displayed to the users is here.

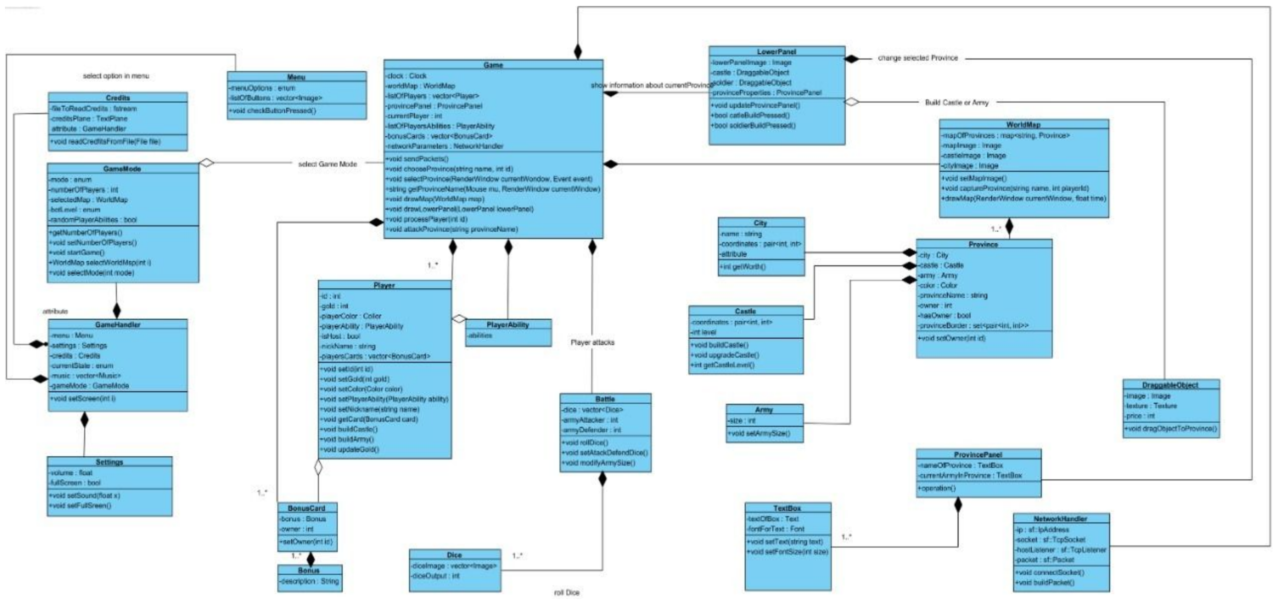
4.2.2 Singleton Design Pattern

In our game, in order to make certain moves, game logic algorithms and the main attributes of the the game must be accessed. These algorithms are in the subsystem and in the WorldMap class. To avoid confusion in the system, WorldMap class will have only one instance at a time. As a result, same reference will be done to the same object or manager throughout the game. A single instance of a WorldMap will be present.

4.2.3 Factory Design Pattern

In the Risk game, many of the entities are fixed in terms of numbers. A single World map, number of provinces and cities, the number of dice are fixed. Hence, as a design decision, we have decided to use the factory design pattern for some of the entities in our game. Classes, City, Province, Dice, BonusCards and WorldMap are entities that are limited and known beforehand. Consequently, it is practical to predefine and control these entities from a factory classes.

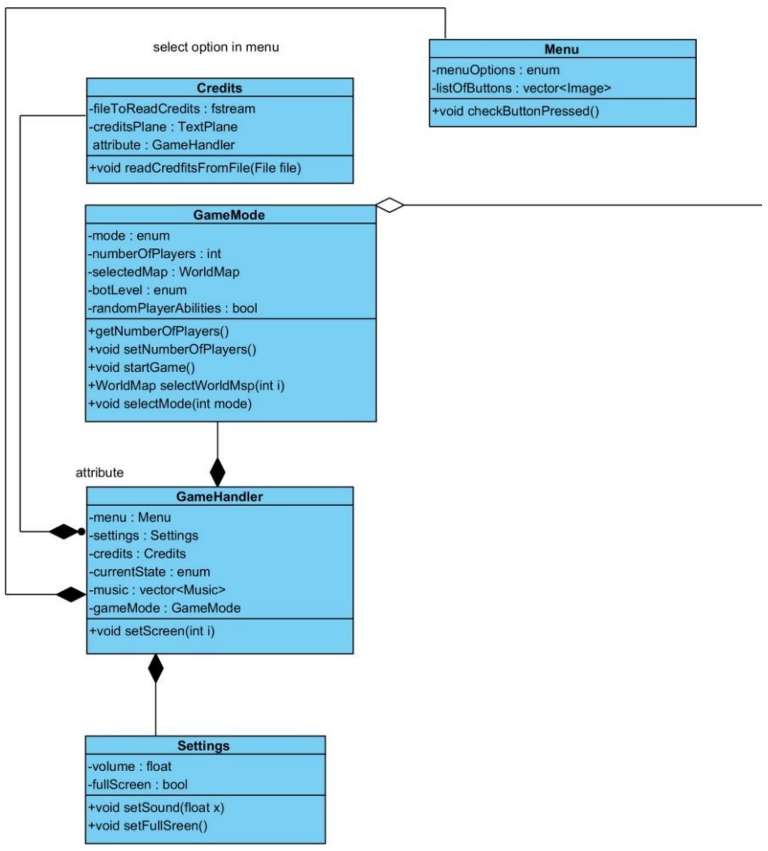
4.3 Final object design



In this section we are going to discuss about our classes and their design. We have divided our classes in order to maintain an 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.

4.4 Class Interfaces

4.4.1 Menu Classes



4.4.1 Menu

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

important methods:

- **void playMenu(RenderWindow &currentWindow, Event &event, int & currentState)** : this method is displaying main menu of the game.

4.4.2 GameHandler

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

important methods:

- **void selectState(int state) :** this method selects state of the menu pages.
- **void startMusic():** starts playing music.
- **void setMusic(Music &music):** selects music to play.

4.4.3 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.

important methods:

- **void setMode(int mode):** sets Mode of game. It is called when player selects mode of game.
- **void setNumberOfPlayers(int num):** sets number of players.
- **void setMap(String map):** sets map of game from the “map” folder of the game directory.
- **void setBotLevel(int i):** sets level of bot
- **void setRandomaAbilities(bool b):** sets true/false option for random abilities of players.

4.4.4 Settings

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

important methods:

- **void setVolume (float volume):** sets volume
- **void setScreenSize(int x,int y):** sets size of screen
- **void setFullScreen(bool t):** enables/disables fullScreen mode
- **void saveSettings():** saves and applies new settings.

4.4.5 Credits

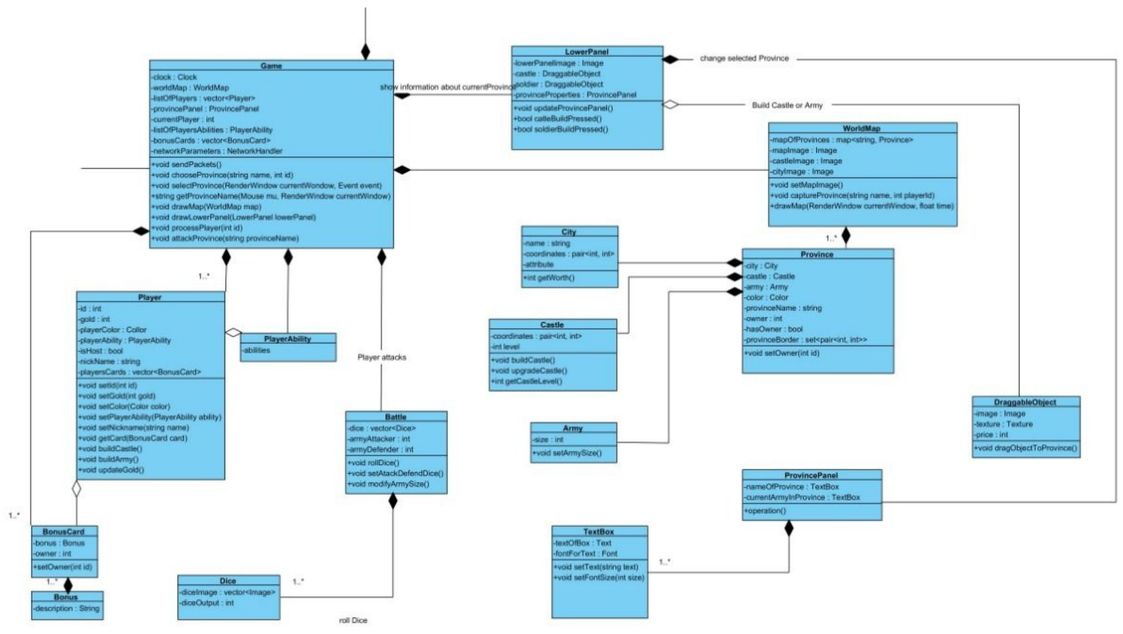
The user will access this screen from the home screen using “Credits” button.

Developers of the game and their contact information are listed in addition to the GitHub link of the game.

important methods:

- **string getText(fstream file):** reads game credits from a file.
- **void showCredits():** displays credits of Game.

4.5 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()
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()
ProvincePanel
-nameOfProvince : TextBox
-currentArmyInProvince : TextBox
+operation()
1..*
BonusCard
-bonus : Bonus
-owner : int
+setOwner(int id)
1..*
Bonus
-description : String
Dice
-diceImage : vector<Image>
-diceOutput : int
1..*
roll Dice
TextBox
-textOfBox : Text
-fontForText : Font
+void setText(string text)
+void setFontSize(int size)
1..*

4.5.1 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.

important methods:

- **void playGame():** continues/starts game
- **void createListOfPlayers():** creates listOfPlayers and takes their names.
- **void startClock():** initilizes clocks.
- **void updateProvincePanel():** updates Panel of detailed information of province.
- **void moveToNextPlayer():** changes turn to the next player.
- **void submitMove():** submitsMove
- **string getProvinceName(Mouse mu):** get name of province that player clicked
- **void chooseProvince(Mouse mu):** select province that player clicked.
- **void processKeyboard(Keyboard keyboard):** processes every key press on keyboard and moves, zooms map.
- void **sendPacketsToHOST():** sends builded packets with information to server.

4.5.2 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.

important methods:

- **void addProvince(int a, string name, string cityName, int cityCoordX, int cityCoordY, int castleCoordX, int castleCoordY, int armyCoordX, int armyCoordY):** adds a province to the map. Information is stored in file in "map" directory of program
- **void captureProvince(string name, int playerNumber):** changes owner of province of map.
- **string findProvince(int a):** find province according to its ID.
- **void drawAllProvinces(RenderWindow &currentWindow, float time):** Draws all buildings and armies of all Provinces of Map.

4.5.3 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 a object from the City class.

castle: Which is a object from the Castle class.

army: Which is a object from the Army class

important methods:

- **void setOwner(int player):** sets the owner of province
- **void buildCastle():** building castle in province
- **void destroyCastle():** destroying castle in province
- **void drawProvince():** drawing all elements of province.

4.5.4 Army

It instantiates the size of the army.

important methods:

- **void setSize(int x):** sets size of army
- **void getSize():** gets size of army

4.5.5 Castle

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

important methods:

- **void upgradeCastle():** upgrades level of castle

4.5.6 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.

important methods:

- **void draw(RenderWindow & currentWindow):** draws lower panel with information about province.

4.5.7 ProvincePanel

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

important methods:

- **void getProvinceName():** gets name of currentProvince
- **void getProvinceArmySize():** gets size of army of a selected province

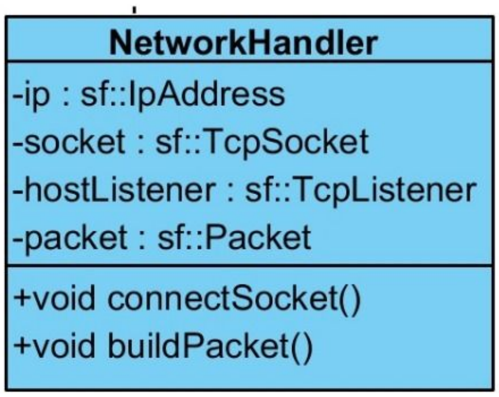
4.5.8 DraggableObject

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

important methods:

- **void activateMovable():** sets inMove to true when player drags it.
- **void disableMovable():** sets inMove to false when player releases object.

4.6 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 Addresses of the computers.

important methods:

- **void connectSocket():** connects socket to a given IP
- **void buildPacket():** upload information to packets.2

4.7 Packages

In the system implementation, we have two types of packages. One of them is Internal Subsystem Packages which consists of the classes that we have implemented. The other one is External Packages which contains SFML Library.

4.7.1. 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.

4.7.1.1 GameController

GameController package includes the classes that communicate with both UserInterface and GameLayer packages' classes. This package has the game logic according to inputs from UserInterface package. It manages game objects instances in the GameLayer.

4.7.1.2 GameLayer

GameLayer package includes the classes that are real game objects such as province, soldier, tower, city etc.

4.7.1.3 UserInterface

UserInterface package includes the classes that have listeners waiting for the input from the user. These package also includes the view components classes which is the layer between the user and the system.

4.7.2. 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.

A sound buffer will be used to loadFromStream to provide music for the player and enhance the game experience.

- sfml-window.lib will be used to create window for the game.

RenderWindow and its functions will be used to create User Interface. Mainly draw(), clear() functions of the library are essential to provide a window.

- sfml-graphics.lib will be used to implement User Interface.

This package is essential to implement Listeners etc. in order to provide an interactive User Interface.

- sfml-network.lib will be used to implement to communicating with sockets.

With Socket and TCPSocket packages the network will be established between users.

5. Improvement Summary

In our second iteration, we have rethought our subsystem decomposition and proposed a new decomposition. We have made some changes on our classes and attributes. Further clarifications are made and some diagrams are added to the design phase.

1. Design goals have been revised. Association between the nonfunctional requirements have been made.
2. Subsystem decomposition has been revised and a new subsystem decomposition has been proposed.
3. Explanation about the new subsystem decomposition has been made and further clarifications have been made.
4. We have created a new deployment graph for our subsystem decomposition.
5. We have corrected the deployment diagram for client/server communication.
6. We have decided to proceed with the MVC architectural design for the implementation and division of jobs in our team.