CS 319 - Object-Oriented Software Engineering Analysis Report

Infinite Tale

Group 1-H

Erim Erdal
Halil İbrahim Azak
Can Özgürel
Hüseyin Taşkesen

1. Introduction

Infinite Tale is a turn based strategy game played on a hexagonal map. Infinite Tale supports single player game in which a player can play against multiple AI factions. Player and other factions have resources like money and soldiers to either protect their lands or try to conquer other faction's lands. The game is played on a hexagonal map and lands are hexagon tiles on the map. Tiles have different geographical features that affect battles happening in them. To conquer an area, the player must conquer the whole province that has the area. Provinces represent cities in the game and sources are controlled through provinces. Main purpose of the factions is defeating all other factions and conquering all lands.

Infinite Tale is a desktop game that will be developed using Java thus giving it the ability to run on any platform that has Java Virtual Machine. Game's UI will be easy to use and player will use mouse and keyboard.

2. Overview

Infinite tale will be a desktop application and player will play the game using keyboard and mouse. The game consists of several displays including the main menu user will see at the launch of the application, game UI that will show map and informations about the player's session at that moment, a pause menu, a save&load menu and a settings menu.

Main menu shows the user different options like New Game, Load Game and Settings. It is the menu players will see first. While user in this menu, application will not have any game session loaded to decrease required computer resources. Game UI is the display the player will interact most in a session of game. Game UI shows all informations

related to the game session and it is the main display of the game. The map users will see is a hexagonal map and areas are represented as hexagon tiles. Pause menu is the menu users will see if they want to go back to menu in the middle of a game session. Pause menu is very similar to main menu and only differences are the new options including Resume Game, Save Game and Back to Main Menu. The save&load menu is for saving the game session or loading a previously saved session. Finally the settings menu is for adjusting different settings like name of the game session and level of the music.

2.a Managing Game Sessions

Players can start to a new game through main menu at any time of a session. If it is the middle of a game, the active session data will be lost if the player does not save it. When a new game starts, the application creates the map and factions based on data files. These files can be edited outside of the game. When player wants to load a saved session, the application gets the required data from the save file and creates game classes based on the data. Similarly when a player wants to save the session, all required game data will be converted into a data that can be written into a file.

2.b Gameplay

Infinite Tale is a single player game in which the player will play against AI factions. Every faction in game is enemy of each other and the game continues until all but 1 faction is defeated. The player controls one of the factions. The game UI shows the game map from the perspective of the player and informations the player can have at the current situation.

The game is a turn based game. All factions play in turns and at the end of the cycle, resources and map updated according to the things happened at that cycle. Factions use different resources to build armies and fight against each other.

2.c Resources

The game has 2 types of resources. One is the gold and the other is the population. Gold is used to feed armies and recruit new soldiers. The amount of gold factions will get at the end of a cycle of turns is based on the population they have. The amount of gold they will lose is based on the armies they have. The difference of the two is added or reduced from the treasury of the faction. Factions will also spend gold to recruit new soldiers and it is reduced at the time of recruitment.

Other resource of the game is the population. Population is resource of both gold and soldiers. While factions do not control the population directly, their actions affect the number of people they have. For example when new soldiers are recruited, the amount is reduced from the population of the place they are recruited. At the end of a cycle of turns, population is increased based on the previous value. Each different province in the game has different limit of population and the population can never go over the limit.

2.d Armies

Armies consist of different types of soldiers and factions can have multiple armies located on different places on the game area. Armies will have limited amount of tiles they can move per turn. This limit refreshes next turn. A movement into a tile that has enemy armies in it results in a battle.

2.e Battles and Conquest

In Infinite Tale, two armies battle against each other when one of them moves to the tile other one is located. The result of the battle is affected by several things. One is the types and the amount of soldiers armies have. Another is their position in battle as in being the attacker or the defendant. Geographical features of the location also affect the battles and the effect is different for different types of units and attacker and defendant.

For conquest, we considered that it is not logical to be able to conquer a province completely while there are enemy soldiers belonging to that province's tiles. It would be more realistic for the enemy soldiers to not give up on province before all the tiles' soldiers in that province die.

3. Functional requirements

3.a Start Game

Infinite Tale will use data files to create and launch a new game or it will create and launch a game based on the saved data.

3.b Play Game

Infinite Tale will be a turn based strategy game that can be played using a keyboard and a mouse. Players will use their mouse to interact with game map and control their faction through game UI. Other factions in the game will be controlled by the game. There is no time limit in the game. A faction is defeated if it loses all of its lands.

3.c Pause & Resume Game

Players can pause a game session anytime they want. In that case they will see the pause menu where they can resume playing or do other things like saving or loading a game and changing settings.

3.d Change Settings

In the application, there are a couple of things user can adjust according to their preferences. Players will be able to change those settings through settings menu. Settings include the level of the volume and the name of the session.

3.e Saving and Loading a Game Session

Players can save the game session they are playing and load a previously saved session if they want. Game sessions are saved using required game information written to files. Those saved sessions can be transported and used in anywhere by simply having the save files.

3.1. Additional Functional Requirements

Infinite Tale will have new features that will improve UI, game logic and battle system. These new features will be discussed in relation to the subsystems they will be implemented in.

3.1.a UI Changes

UI changes include different types of map view like political view and geographical view. This feature requires the map view component to be able to show map in different styles and to switch between them.

Another new feature is the ability to load map from data files. This feature requires functional changes in several UI components and the facade class of the UI subsystem to be able to create other classes of the game from the data read from files.

3.1.b Map Changes

Conqueror will not start with a single tile as he did before, he will start to the game with several tiles and provinces which will make decision-making of the game harder and more fun for the player.

Neutral lands will be changed to different enemy lands because we thought that it would be a backdoor to gain easy lands and gold, therefore would be abused by many users to make the game easier.

3.1.c Read Lore

A new upgrade to the game is that player will be able to choose to read the story of the Infinite Tale from main menu which will add a new taste to the gaming experience, knowing what went on before the story took place.

3.1.d Battle Mechanics Improvement

Battle mechanics were simply determined by the height and weather of the Tile where clash were happening, we will improve it by making a smarter and more complex battle simulation, adding luck as a new mechanism and giving the defenders a small head start to the battle like in real life.

3.1.e Better Visualization Experience

Our game had a simple UI before and the game screen was not very interactive or smooth-looking. We will change this by improving graphics quality and making continuing pictures appear hence a smoother experience of the game is created.

3.1.f New Musics to Explore

Music options were fairly limited before, actually there were just one. Now we include a variety of beautiful music for your taste.

4. Nonfunctional requirements

4.a Performance

Performance is one of the main concerns that played an important role in planning of Infinite Tale. Since Infinite Tale will be turn based and a 2D game, it will not require high amounts of computer resources thus making it runnable in low-end devices.

4.b Graphics

Infinite Tale uses JavaFx as the graphics library because JavaFx is one of the libraries that is still improved and expected to be the successor of old graphics libraries. It is being a modern library makes it developed with latest needs in mind and in Infinite Tale, we wanted to use modern ways to present Game UI.

4.c User Friendly UI

Design of UI of Infinite Tale focuses on being user friendly. While this lets users to easily grasp the game and start playing it should be noted that it also limits the functionality of UI. In Infinite Tale, user friendliness preferred because the game is not too complicated and can be played with simple actions that can be done through UI.

4.1. Additional Nonfunctional requirements

4.1.a Improved Reliability

We will take careful steps to improve our code's robustness and it more reliable. It is impossible for a crash to occur during the game even if computer is disconnected from power, the game will simply exit. Network outages are not an issue either.

4.1.b Better Usability

We will make it easier to understand and therefore use for the player to have a better gaming experience. The flow of the game will be smooth-sailing.

4.1.c Increased Supportability

With addition of supportability, we will remove the system requirement to make changes while stopping the flow of the game.

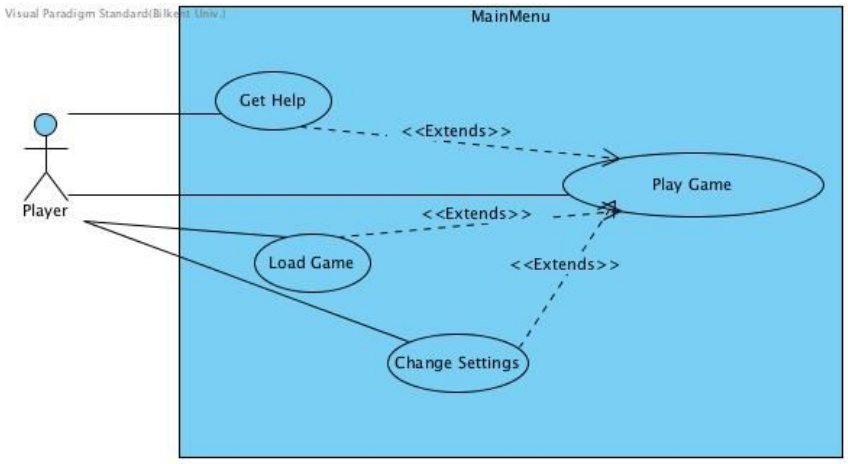
Also Infinite Tale while support different kinds of musics if they are in the right format.

5. System models

5.1. Use case model

5.1.a MainMenu Use Case Model

MainMenu use case of Infinite Tale is explained with detailed below:



5.1.a.1 MainMenu Use Case Definitions

Use Case Name: Get Help

Including Actors: Player

Pre-conditions: system must keep help instructions.

Entry Conditions: Player Clicks on “Help” button via using main menu interface.

Exit Conditions: Player clicks on back icon.

Flow of Events:

1. Player clicks on help button.
2. System displays pre-recorded help instructions.
3. Player clicks on back icon, then help interface closes.

Use Case Name: Play Game

Including Actors: Player

Pre-Conditions: Music volume is set to default unless it configured via using setting option.

Entry condition: Player clicks on “Play Game” button via using main menu interface.

Exit condition:

- Player can select exit via in game menu.
- Player has to conquer all tiles and must beat all boots.
- Losing all owned tiles ends the game and force player to return main menu.

Flow of Events:

1. Game starts by clicking “Play Game” button.
2. System generates the level by placing all actors to their default state.
3. Player moves and attacks its opponents.
4. Player conquers all enemy owned tiles.
5. Game finishes and returns to main menu.

Alternate Flow of Events:

1. Game starts by clicking “Play Game” button.
2. System generates the level by placing all actors to their default state.
3. Player exits game via in-game exit option.

Use Case Name: Load Game

Including Actors: Player

Pre-condition: There must be a Pre-saved game in system, system saves relevant information

Entry condition: Player selects “Load Game” button via using Main Menu interface

Exit condition:

1. Player selects any saved games and system recreates game from where it is left.
2. Player exits load Game interface by using back icon.

Flow of Events:

1. Player selects “Load Game”.
2. System regenerate the game according to saved game’s information.

Alternate Flow of Events:

1. Player selects “Load Game”
2. System cannot find any saved games
3. System returns to main menu.

Use Case Name: Change Settings

Including Actors: Player

Precondition: System is holding music files and set them default.

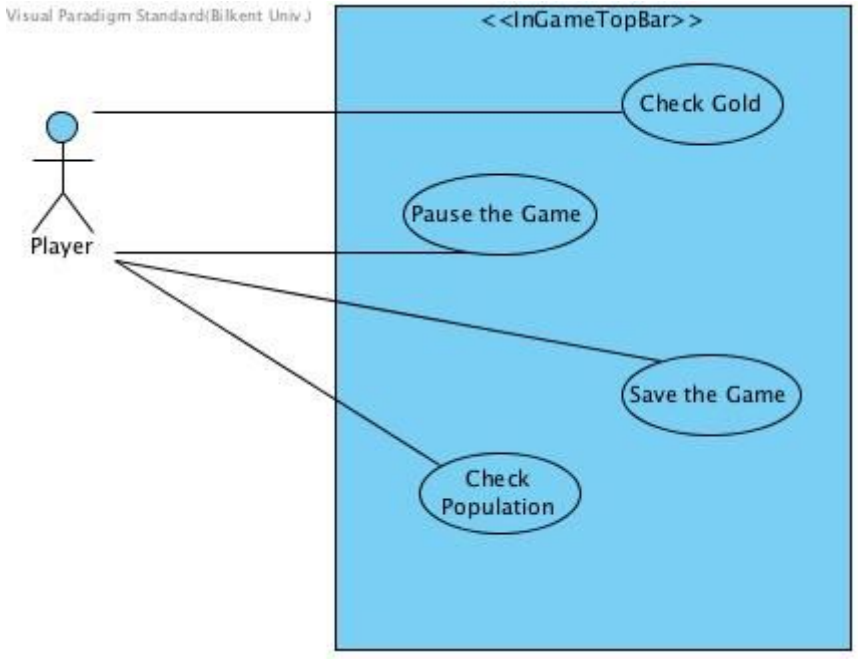
Entry condition: Player clicks on “Settings Button” in main menu OR
Player clicks on “Settings Button” after pausing the game.

Exit condition: Player clicks on back icon and return to main menu.
Player clicks on back icon and return to the game.

Flow of events:

1. Player clicks on “Setting” button.
2. Player changes music setting to off.
3. Exits by clicking back icon.

5.2.a InGameTopBar Use Case Model



5.2.a.1 InGameTopBar Use Case Definitions

Use Case Name: Check Gold

Including Actors: Player, System

Precondition: System contains gold variables of each player.

Entry condition: Player clicks on “Gold” button via using InGameTopBar interface.

Exit condition: Player stops pressing on “Gold” Button.

Flow of Events:

1. Player clicks and holds on “Gold” button.
2. Player releases the button.
3. Pop-up gold interface closes.

Use Case Name: Check Population

Including Actors: Player, System

Pre-conditions: System contains population variables of each player.

Entry condition: Player clicks on “Population” button via using InGameTopBar interface.

Exit condition: Player stops pressing on “Population” Button

Flow of Events:

1. Player clicks and holds on “Population” button.
2. Selects check current population.
3. Player releases the button.
4. Pop-up gold interface closes.

Alternate of Events:

1. Player clicks and holds on “Population” button.
2. Selects check soldier population.
3. Player releases the button.
4. Pop-up gold interface closes.

Use Case Name: Pause the Game

Including actors: Player

Pre-condition: Game has to be running.

Entry condition: Player clicks on “Pause” Button via using InGameTopBar interface.

Exit condition: Player clicks on “Pause” Button via using InGameTopBar interface.

Flow of events:

1. Player pushes “Pause” button.
2. Game stops receiving input until it gets unpaused.

Use Case Name: Save the Game

Including Actors: Player

Precondition: System has to have space to save game variables, game must be running.

Entry condition: Player clicks on “Save” Button via using InGameTopBar interface.

Exit condition: After save game returns to normal state.

Flow of events:

1. Player clicks on “Save” button.
2. System saves game variables.
3. Game returns to normal state.

5.2. Dynamic models

In this section, the player’s interaction with Infinite Tale will be explained using sequence diagrams. Player can interact with game in different ways and game classes interact with each other. Sequence diagrams below will show and let us explain some of the common actions a player can take and how classes will interact with each other in those cases.

5.2.a Starting a New Game

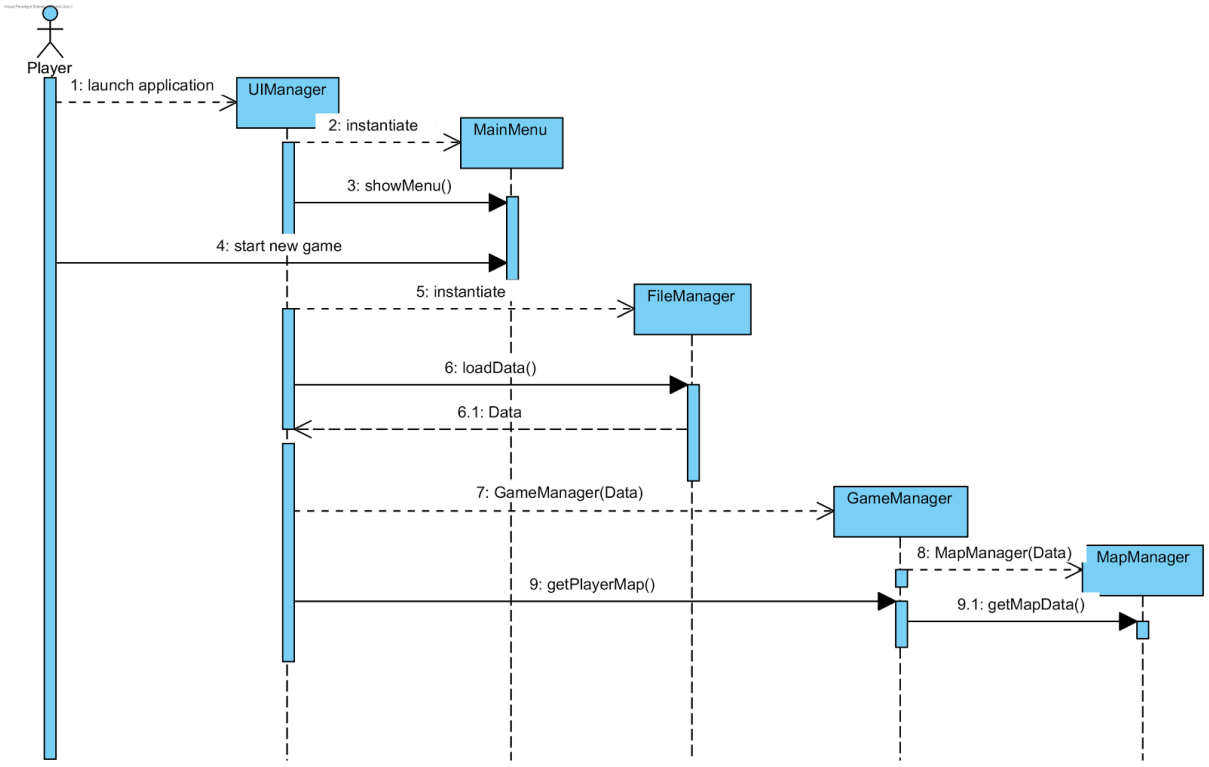


Figure 5.2.a.1 - New Game Sequence Diagram

When the player launches the application, main method of the UIManager class will run and initialize the application. At first it doesn't initialize game related classes to reduce use of computer resources. Player is presented with the main menu which includes different options like New Game and Load Game. In this scenario starting of a new game will be explained.

As it can be seen in the diagram, when player starts a new game UIManager class first creates FileManager class to load data files. Data files will have the information about map and other components of the game. After getting the required data, UIManager creates GameManager class using this data and GameManager class creates MapManager class using same data. These classes hold and control state information about the state of the game so the data acquired from files is needed to create them. When these classes are successfully created, UIManager request the map the player will see to show the game screen.

5.2.b Moving Units on Map to Battle

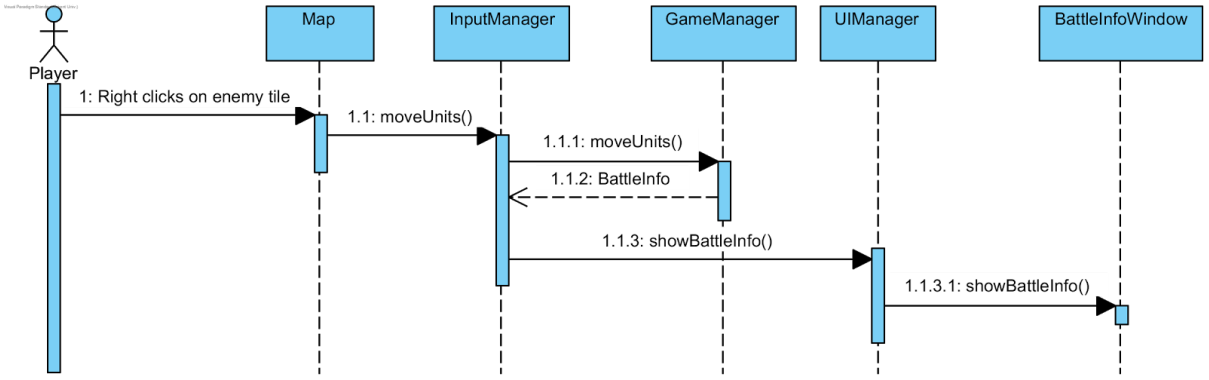


Figure 5.2.b.1 - Move Units Sequence Diagram

When an owned tile selected on the map if it has owned units that can still move, the player can right click a neighbouring tile to move units. In the diagram above, this command and interactions between classes after clicking are shown. When the Map class detects this command, it uses InputManager class to send Move Units command to the GameManager class. GameManager calculates the result of the action with the help of MapManager class (this interaction is not added in the diagram to keep diagram simple) and creates a BattleInfo object that has the information about result. A move unit command can result in a simple movement or in a battle. In this scenario, it is taken as the enemy tile has enemy units which will turn the movement into a battle. When InputManager class gets the result and it is a battle, it sends this information to UIManager. Finally UIManager calls the method of the BattleInfoWindow class to show this information.

5.2.c Adjusting Music Level Through Settings Menu

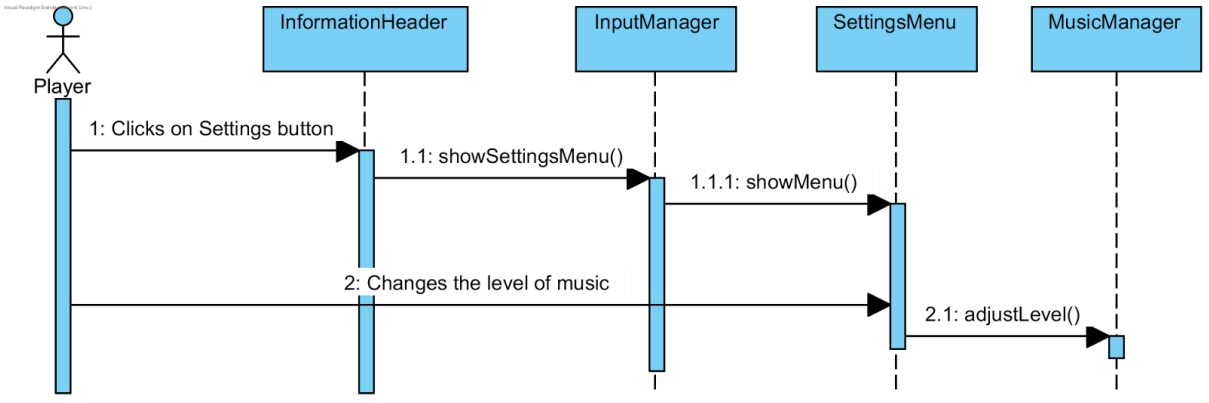
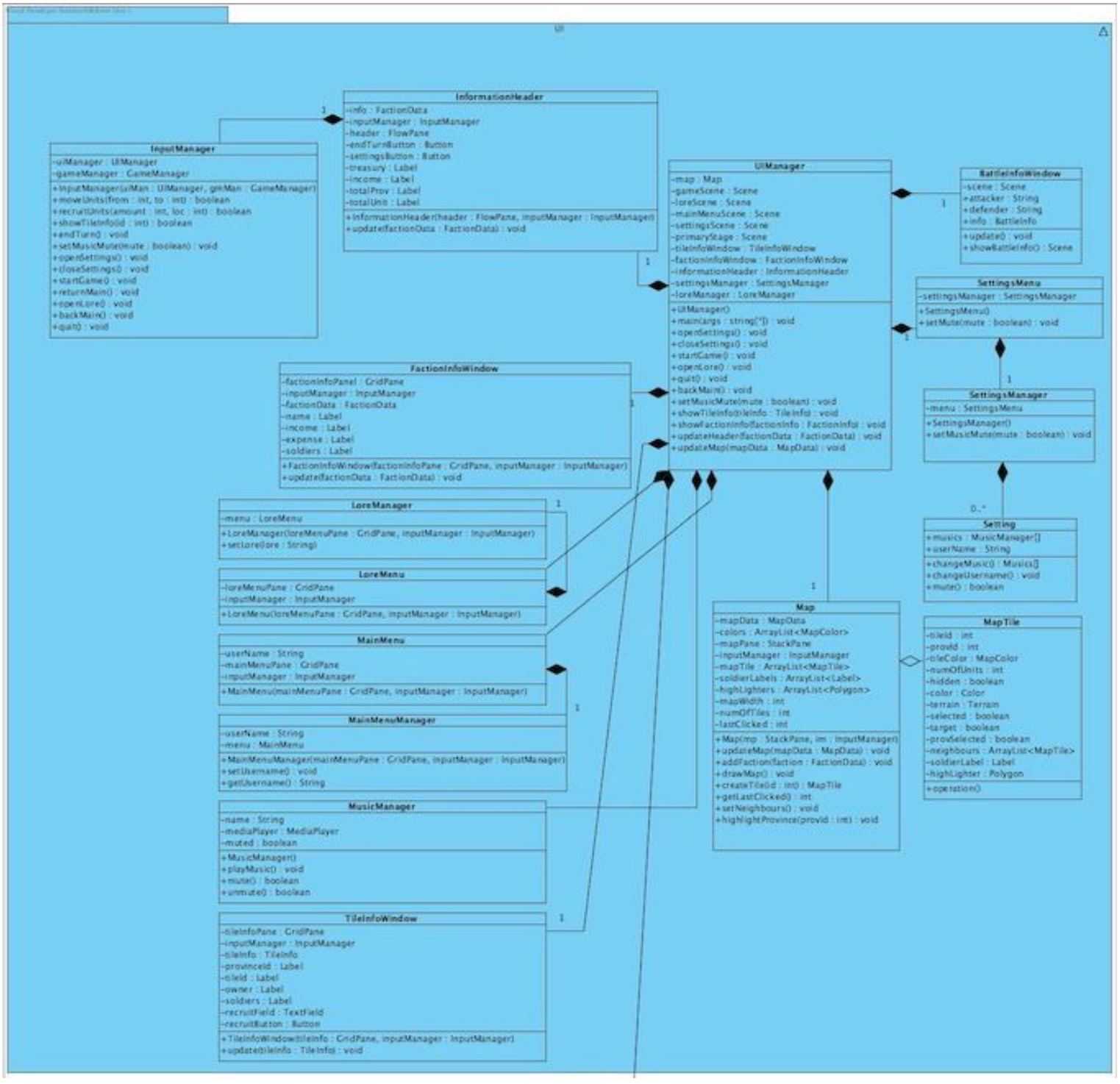
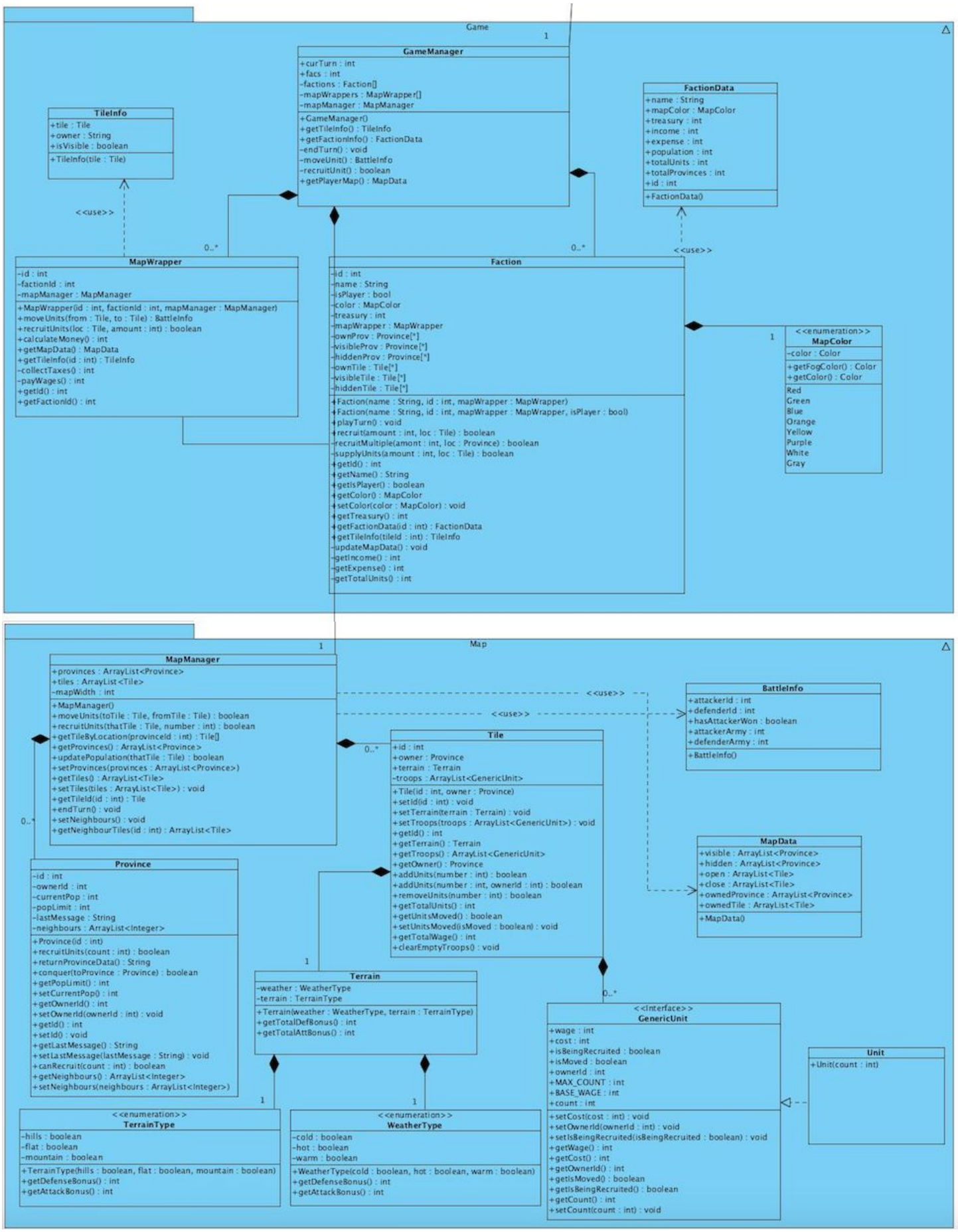


Figure 5.2.c.1 - Settings Menu Sequence Diagram

In Infinite Tale, player can adjust different settings through the Settings Menu. In this scenario, how the player will be able to adjust the level of music is shown. First, player needs to click the Settings button. Then, InformationHeader class informs InputManager class about the action. InputManager class calls related method of SettingsMenu class to show the Settings Menu. Player can adjust the level of music in this menu. When the player changes the level, SettingsMenu class informs MusicManager about the new level.

5.3. Object and class model




Game
GameManager
+curTurn : int
+facs : int
-factions : Faction[]
-mapWrappers : MapWrapper[]
-mapManager : MapManager
+GameManager()
+getTileInfo() : TileInfo
+getFactionInfo() : FactionData
-endTurn() : void
-moveUnit() : BattleInfo
-recruitUnit() : boolean
+getPlayerMap() : MapData
TileInfo
+tile : Tile
+owner : String
+isVisible : boolean
+TileInfo(tile : Tile)
FactionData
+name : String
+mapColor : MapColor
+treasury : int
+income : int
+expense : int
+population : int
+totalUnits : int
+totalProvinces : int
+id : int
+FactionData()
<<use>>
MapWrapper
-id : int
-factionId : int
-mapManager : MapManager
+MapWrapper(id : int, factionId : int, mapManager : MapManager)
+moveUnits(from : Tile, to : Tile) : BattleInfo
+recruitUnits(loc : Tile, amount : int) : boolean
+calculateMoney() : int
+getMapData() : MapData
+getTileInfo(id : int) : TileInfo
-collectTaxes() : int
-payWages() : int
+getId() : int
+getFactionId() : int
Faction
-id : int
-name : String
-isPlayer : bool
-color : MapColor
-treasury : int
-mapWrapper : MapWrapper
-ownProv : Province[*]
-visibleProv : Province[*]
-hiddenProv : Province[*]
-ownTile : Tile[*]
-visibleTile : Tile[*]
-hiddenTile : Tile[*]
+Faction(name : String, id : int, mapWrapper : MapWrapper)
+Faction(name : String, id : int, mapWrapper : MapWrapper, isPlayer : bool)
+playTurn() : void
+recruit(amount : int, loc : Tile) : boolean
-recruitMultiple(amont : int, loc : Province) : boolean
-supplyUnits(amount : int, loc : Tile) : boolean
+getId() : int
+getName() : String
+getIsPlayer() : boolean
+getColor() : MapColor
+setColor(color : MapColor) : void
+getTreasury() : int
+getFactionData(id : int) : FactionData
+getTileInfo(tileId : int) : TileInfo
-updateMapData() : void
-getIncome() : int
-getExpense() : int
-getTotalUnits() : int
<<enumeration>>
MapColor
-color : Color
+getFogColor() : Color
+getColor() : Color
Red
Green
Blue
Orange
Yellow
Purple
White
Gray
Map
MapManager
+provinces : ArrayList<Province>
+tiles : ArrayList<Tile>
-mapWidth : int
+MapManager()
+moveUnits(toTile : Tile, fromTile : Tile) : boolean
+recruitUnits(thatTile : Tile, number : int) : boolean
+getTileByLocation(provinceId : int) : Tile[]
+getProvinces() : ArrayList<Province>
+updatePopulation(thatTile : Tile) : boolean
+setProvinces(provinces : ArrayList<Province>)
+getTiles() : ArrayList<Tile>
+setTiles(tiles : ArrayList<Tile>) : void
+getTileId(id : int) : Tile
+endTurn() : void
+setNeighbours() : void
+getNeighbourTiles(id : int) : ArrayList<Tile>
BattleInfo
+attackerId : int
+defenderId : int
+hasAttackerWon : boolean
+attackerArmy : int
+defenderArmy : int
+BattleInfo()
Tile
+id : int
+owner : Province
+terrain : Terrain
-troops : ArrayList<GenericUnit>
+Tile(id : int, owner : Province)
+setId(id : int) : void
+setTerrain(terrain : Terrain) : void
+setTroops(troops : ArrayList<GenericUnit>) : void
+getId() : int
+getTerrain() : Terrain
+getTroops() : ArrayList<GenericUnit>
+getOwner() : Province
+addUnits(number : int) : boolean
+addUnits(number : int, ownerId : int) : boolean
+removeUnits(number : int) : boolean
+getTotalUnits() : int
+getUnitsMoved() : boolean
+setUnitsMoved(isMoved : boolean) : void
+getTotalWage() : int
+clearEmptyTroops() : void
MapData
+visible : ArrayList<Province>
+hidden : ArrayList<Province>
+open : ArrayList<Tile>
+close : ArrayList<Tile>
+ownedProvince : ArrayList<Province>
+ownedTile : ArrayList<Tile>
+MapData()
Province
-id : int
-ownerId : int
-currentPop : int
-popLimit : int
-lastMessage : String
-neighbours : ArrayList<Integer>
+Province(id : int)
+recruitUnits(count : int) : boolean
+returnProvinceData() : String
+conquer(toProvince : Province) : boolean
+getPopLimit() : int
+setCurrentPop() : int
+getOwnerId() : int
+setOwnerId(ownerId : int) : void
+getId() : int
+setId() : void
+getLastMessage() : String
+setLastMessage(lastMessage : String) : void
+canRecruit(count : int) : boolean
+getNeighbours() : ArrayList<Integer>
+setNeighbours(neighbours : ArrayList<Integer>)
Terrain
-weather : WeatherType
-terrain : TerrainType
+Terrain(weather : WeatherType, terrain : TerrainType)
+getTotalDefBonus() : int
+getTotalAttBonus() : int
<<interface>>
GenericUnit
+wage : int
+cost : int
+isBeingRecruited : boolean
+isMoved : boolean
+ownerId : int
+MAX_COUNT : int
+BASE_WAGE : int
+count : int
+setCost(cost : int) : void
+setOwnerId(ownerId : int) : void
+setIsBeingRecruited(isBeingRecruited : boolean) : void
+getWage() : int
+getCost() : int
+getOwnerId() : int
+getIsMoved() : boolean
+getIsBeingRecruited() : boolean
+getCount() : int
+setCount(count : int) : void
Unit
+Unit(count : int)
<<enumeration>>
TerrainType
-hills : boolean
-flat : boolean
-mountain : boolean
+TerrainType(hills : boolean, flat : boolean, mountain : boolean)
+getDefenseBonus() : int
+getAttackBonus() : int
<<enumeration>>
WeatherType
-cold : boolean
-hot : boolean
-warm : boolean
+WeatherType(cold : boolean, hot : boolean, warm : boolean)
+getDefenseBonus() : int
+getAttackBonus() : int

A Small Explanation For Classes in Our Class Diagram

WeatherType: An enumeration class for determining the weather.

TerrainType: Determines the terrain type.

GenericUnit: A class for determining troops.

Unit: An inherited class of GenericUnit. They represent different unit types.

MapData: Hold the data of the map to be able to draw it later depending on players.

BattleInfo: Holds the battle information. Does some calculations to determine the outcome of the battle.

Tile: Represents the hexagonal tiles in game.

Terrain: Represents the terrain of the tiles. This will determine attack and defence bonuses.

Province: Represents provinces in the game which consists of 4 tiles. A province will not be invaded until all tiles in that province are not conquered.

MapManager: The main class that takes care of other classes and handles their inputs and outputs.

GameManager: The main class of Game that takes care of other classes in game, handles inputs and redirects outputs.

TileInfo: Class that stores information about tiles.

FactionData: Class that stores information about factions.

MapWrapper: MapWrapper class regulates information, calculates wages and gold of whole map, gives a general look.

Faction: Handles look of map, checks data about visible and unvisible data, handles fog of war etc.

MapColor: Enumeration class that holds colors of the map.
InformationHeader: This class is responsible for showing player informations about the state of his faction like wealth.

LoreManager: This class controls the lore of the game and has related methods.

LoreMenu: This class controls the Lore Menu and used to show the lore of the game.

MainMenu: This class controls the Main Menu and shows the main menu, basically displaying the screen.

MainMenuManager: This class controls the Main Menu.

MusicManager: This class holds tracks and stores the necessary musics in game that will be played in the course of game.

TileInfoWindow: This class is used to show information about the selected tile on map.

Map: Map class is the class that controls game map.

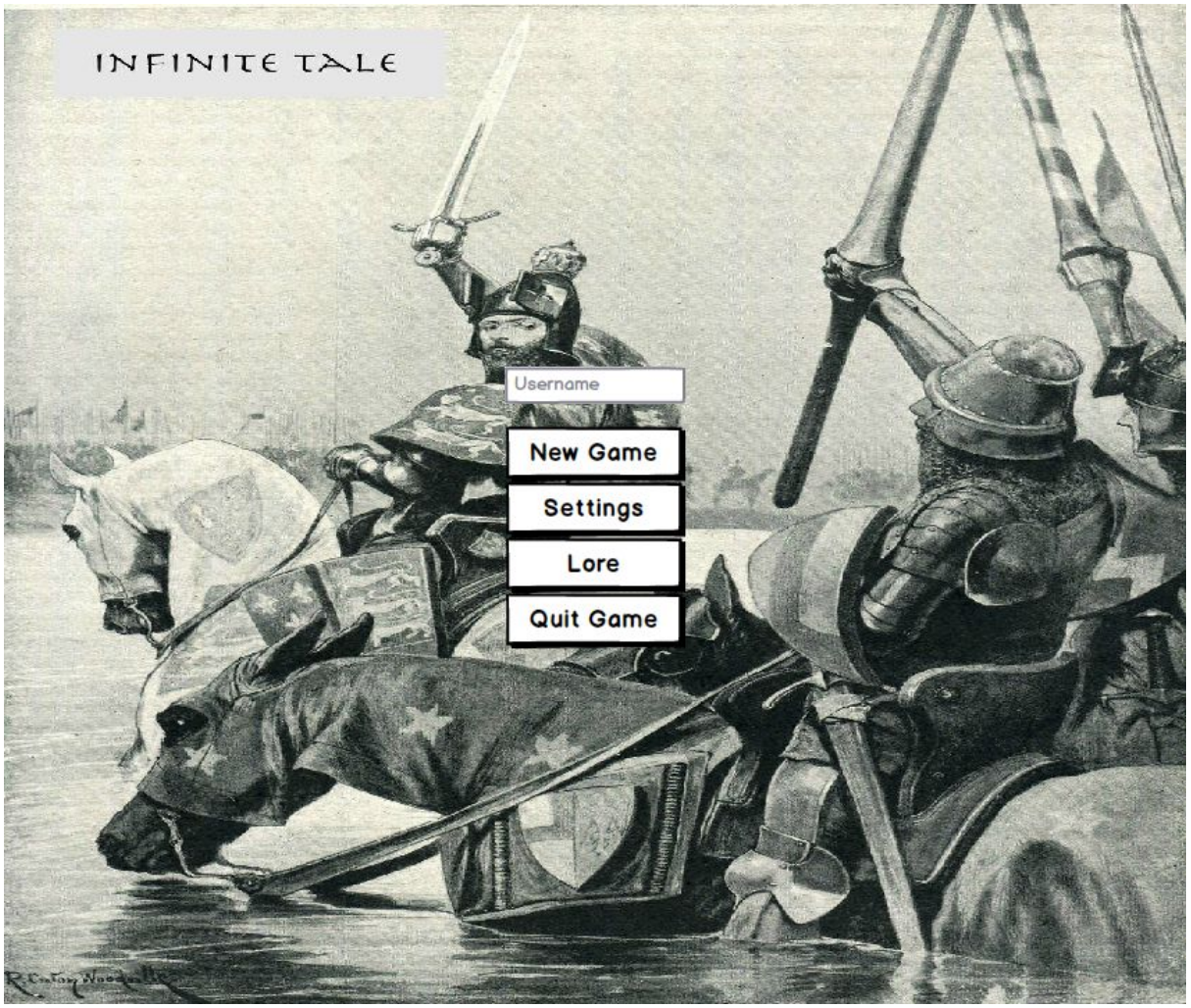
MapTile: MapTile represents a single tile on the map. It is used to store data related to visuals of the tile. Extends JavaFx's Polygon class.

SettingMenu: This class is used to show the settings menu in the game, where you can pause and mute also go back to main menu, etc.

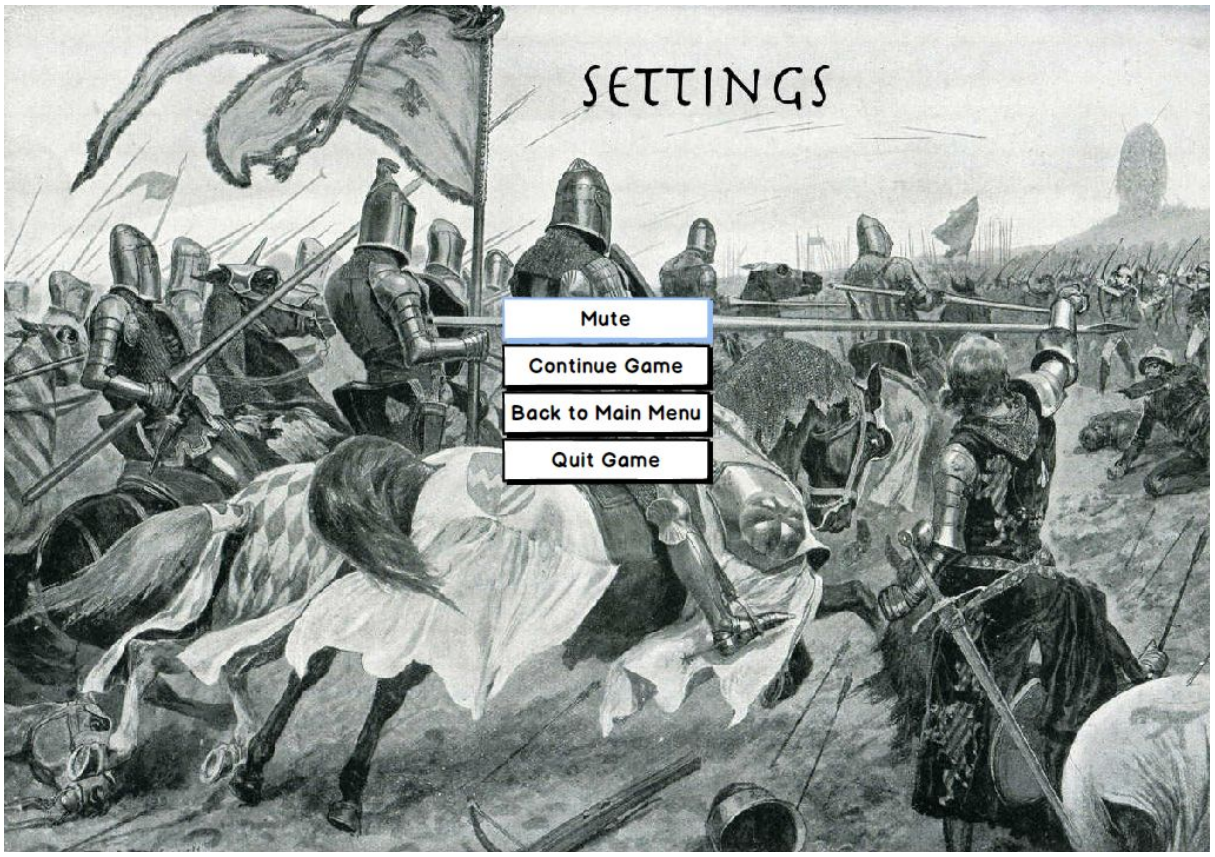
SettingManager: This class manages the view of menu of the settings.

BattleInfoWindow: Battle Information Window is controlled and showed through this class.

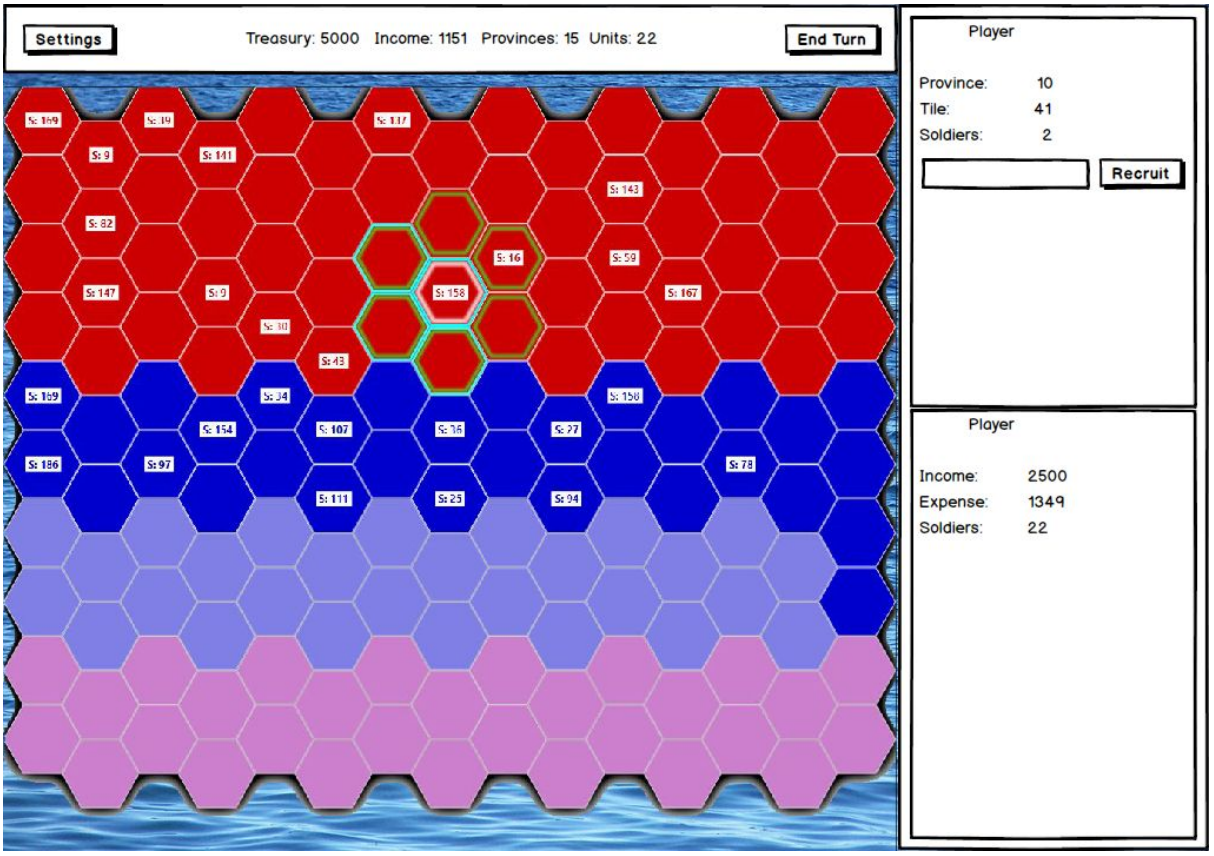
5.4. User interface - navigational paths and screen mock-ups



This picture above is the Main Menu screen. From here you can choose to start a new game, go to settings, display the lore or quit game.



This picture above is inside the settings menu. Here you can mute the music playing in the background, choose to continue game where you left it, go back to main menu to start a new game or quit game altogether.



This picture above is inside the game menu. From here you can play the game by using functionalities our game offers. You can also see information from the panes on right and on top.

6. Improvement Summary

In the analysis of the Infinite Tale, we focused on improving the structure of the game and adding some crucial features like different types of units.

6.1 Structure Changes

While reconsidering the class structure of the game we realized that some classes grow a lot because of some not related functionalities. An example of this is the UIManager class which was a facade class for UI related classes but was also handling all inputs. In the new structure, input related methods and properties will be separated in a new class called

InputManager. This way the control of input handling will be better and more functionality will be provided by the reuse of already written codes.

Another structural change we decided to make is using Singleton pattern for manager and facade classes. In the previous structure while we didn't had any, it was still possible to have multiple objects of some classes like Map class. We realized that this may have cause bugs and decided to use Singleton pattern to limit number of instance of some classes to one.

6.2 New Features

While Infinite Tale had basic functionality to be a turn based strategy game, there were some important features that would make it even more enjoyable. An example of this is different types of units with different properties.

7 - Conclusion

In this analysis report, we specified our design and implementation methods to create a turn based strategy game, named "Infinite Tale". Our analysis report analyzes our game in two main parts. First of the main part is requirement specification and second one is system model. The focus of the analysis process is the decisions that we make as a team to create a well-designed game.

Requirements specification is the part where we determine functional and nonfunctional units of the game. In functional units, we represent the parts of the game that must work correctly in order to have a playable game. Start game is the core of this project and it is analyzed deeply in use case and sequence diagrams. Also other options must be available for an operating game. Nonfunctional requirements are mostly about the implementation process of our project such as performance, graphics and extendibility.

Our report includes the basic gameplay elements, rules and features of the game. As second, system architecture has been specified with help of use case models, dynamic model and objects and class model diagrams.

Models used are given below in order:

1 - Use Case Model

2 - Dynamic Models

3 - Class and Object Model

4 - User Interface Mockups

After considering which cases we should be using in which conditions, we have found out the most important cases that are a must to include in this report. Use case reports have two use cases, first of them is display of main menu. Main menu is the heart of our game since the most important option, start game lays here. Also one can change settings or display help from this screen. Second one is ingame top bar where player can see all his resources and construct a strategy from it.

In dynamic model, we included important sequence diagrams. One of them is starting the game. Actually this sequence diagram is complicated because starting the game nearly needs all components of the game in use, such as creating a small number of troops, creating enemies and drawing the visualization of the game, also dealing resources. Dynamic Models help us to create the heart and core of the game. For this reason we took dynamic models most seriously, in case we might face some problems in implementation it should not be caused by poor modelling.

As the last crucial part we have mock-ups. Mock-ups are indeed crucial because the visualization of the game is one of the most important parts. A game with very bad graphics might be frustrating to play with and when dealing with designing objects one needs to find creative images of these objects to represent within game.

To conclude, we think this analysis report is very detailed in terms of explaining many different cases in the game. Also this report helped us to create a solid underground for the start of our project.