Object Oriented Software Engineering Project

CS319 Project : Infinite Tale

Design Report

Erim Erdal

Halil İbrahim Azak

Can Özgürel

1 - Introduction

1.1 - Purpose of the System

Our purpose in creating Infinite Tale is offering the player a quality experience of strategy gaming. By creating a simple user interface and making the rules very similar to those were in real life of middle ages, we give the user an easy grasp of basic concepts so that they can start playing almost immediately. As the gamer keeps playing, they will discover the depths of lying strategies that one can take which is what makes Infinite Tale special.

1.2 - Design Goals

1.2.a - Performance Criteria

Efficiency: We think efficiency is extremely crucial when creating an interesting game. That's the reason we made our game as efficient as possible in a Java system because before even starting detailed design of the game, we thought about the tradeoffs we should make and efficiency was our number one priority. In several places where we could make our system more memory efficient instead we sacrificed our memory and reduces our number of iterations to maximize performance. We decreased the weight on our game manager as much as possible to squeeze out every possible bit of performance there.

Response Time: Since our game is simple enough, it does not include any database connection or web services which makes it response time instant.

Memory: Very little space in hard drive and almost no space in RAM is enough for running our game.

1.2.b - Dependability Criteria

Reliability: Our game will be very consistent in its boundary conditions, it will be almost impossible to make our system crash since we will handle almost every exception possible by debugging and testing our game in a lot of different use cases. Boundary conditions will be properly defined and obeyed with discipline in order to not give the user any chance to crash the game.

Robustness: As our game being reliable, this also makes it robust. No kind of input that user can enter will be able to crash our system. Having a robust system is much easier in simple applications.

Fault Tolerance: Our game tolerates erroneous inputs and keeps working correctly under them.

1.2.c - Maintenance Criteria

Extensibility: Our object oriented software design approach was to divide each and every step that we could divide reasonably. This design approach helped us to build a system similar to a castle of lego bricks. With the detailed documentation given and very small coupling of the system, one could easily take one part off or add another one without having too much difficulties, changing and modifying nearly every single attribute of our game; musics, background pictures, starting gold and troops, almost anything.

Usability: Our game is extremely straightforward to open and start playing. There are some learning curves that the player might have to pass but this does not mean usability of our system is low, this is mainly caused of the numerous ways of strategies that can be pulled off in the game and one has to think and has to weigh their risks sometimes before making their moves in order to be successful in the game. We tried to make this as easy as possible by providing in-game help to user and also a very detailed documentation explaining everything.

Adaptability: The reason we chose Java except than it was preferred by instructors was that it has the ByteCode intermediate step which is used to make the application cross-platform enabled. This makes our program able to run in all JRE installed platforms, in the cost of sacrificing some important amount of efficiency with not creating our game in C/C++.

1.2.d - Tradeoffs

Efficiency - Reusability: Creating and stabilizing efficiency in this game was a far more important topic than reusability because reusability was never a concern for us when creating this game. We do not consider implementing the classes we created in here to different softwares, our classes are specifically designed for efficiency. Trying to make it reusable would make it lose this attribute.

Functionality - Usability: In competitive software products in market we can see that sometimes they tend to give up usability in order to create a much more functional and detailed software. It is understandable since the users of this product will be professionals and will have been taken the necessary education for using these kind of software. But in our case this tradeoff was nothing to even consider for. Our aim is not creating a fully functional swiss army knife. We just simply created a high usability game because our aim to make people chill and play some strategy games.

Memory - Performance: Among the listed this was by far the hardest tradeoff to consider but from the start we knew that performance would be more important than using memory recklessly. In huge games with high resolution graphics sometimes engineers must consider the fact that memory requirements of the game could be huge, but then again this was not similar to our case. Even if use memory recklessly we would not go any far because we use simple 2d graphics. On the other hand performance was a core key in our game to make it enjoyable that’s the main reason we chose performance over memory. For an example stored things in separate arrays that did not need to be actually stored in separate arrays in order to fasten up our processes.

2 - Software Architecture

2.1 - Subsystem Decomposition

In this section, structure will be explained as smaller parts and how those parts work together. The purpose of this is to see how each part works both independently and in connection with other parts. To understand the MVC structure behind the game and what are the tasks of each part, parts should be examined and explained individually.

The game is structured using Model-View-Controller (MVC) structure. MVC structure lets developing each part separately and helps controlling information and limiting access to it. In the game information about the status of the game is hold and processed in Model classes. These classes include game element classes like Tile and Province classes. Each class holds an information about the game and modifying of this information is made by calling the owner object’s methods. This way, access to information and processing of it is controlled by its owner which creates a more readable and better structured code which is crucial when multiple person developing in parallel.

View classes of the game are the UI classes like Map and InformationHeader class. View classes controls the presentation of UI and the information they will get and use is determined from the beginning. This let's developing and testing UI classes independently and changes in other classes does not require revise of view classes. Likewise a change or a runtime problem in view classes does not affect game classes so in a situation of error game does not lose information about the status of the game.

Finally controller classes establishes connection between view and model classes and controls the game. Controller classes include UIManager, GameManager and MapManager classes. Since the information view classes need and the way model classes will provide it are determined from the beginning, controller classes can be developed and tested independently like other parts of the game. Controller classes also have parent-child connection between each other. As can be seen from **Figure - 1**, UIManager is at the view classes end of the structure. In **Figure - 2** it can be seen that MapManager is at the model classes end of the structure. Lastly in **Figure - 3**, GameManager class and its components which are located in the middle of the structure can be seen.

When determining the structure of the game, we tried to design a structure that would let every member of the team to develop and test his/her part independently so that every member can work without waiting others to do something for them. Also system would run as expected even if all of the codes is not completed therefore letting testing of the general structure any time. We decided to use MVC pattern to accomplish this.

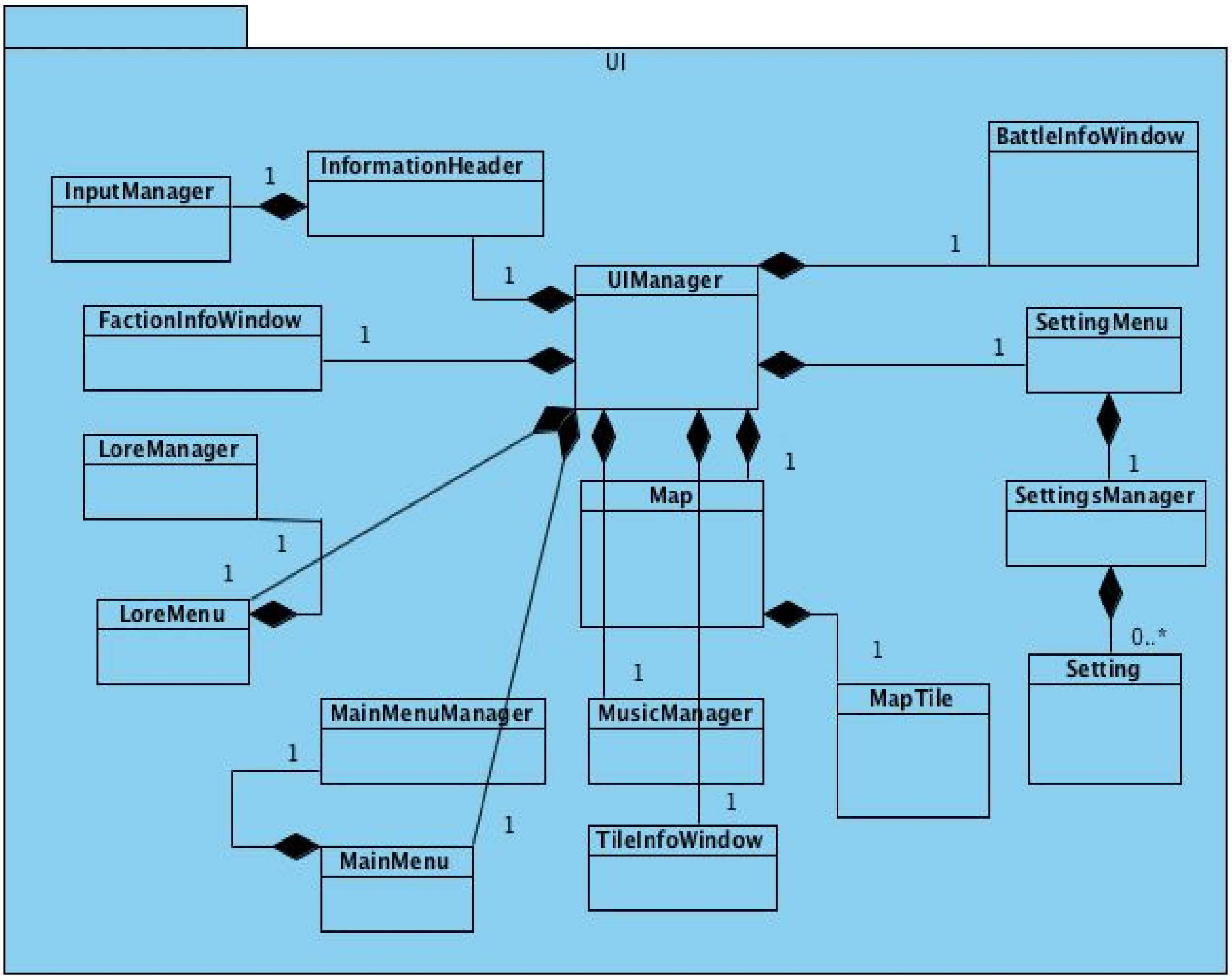


Figure - 1 (View Classes and UIManager)

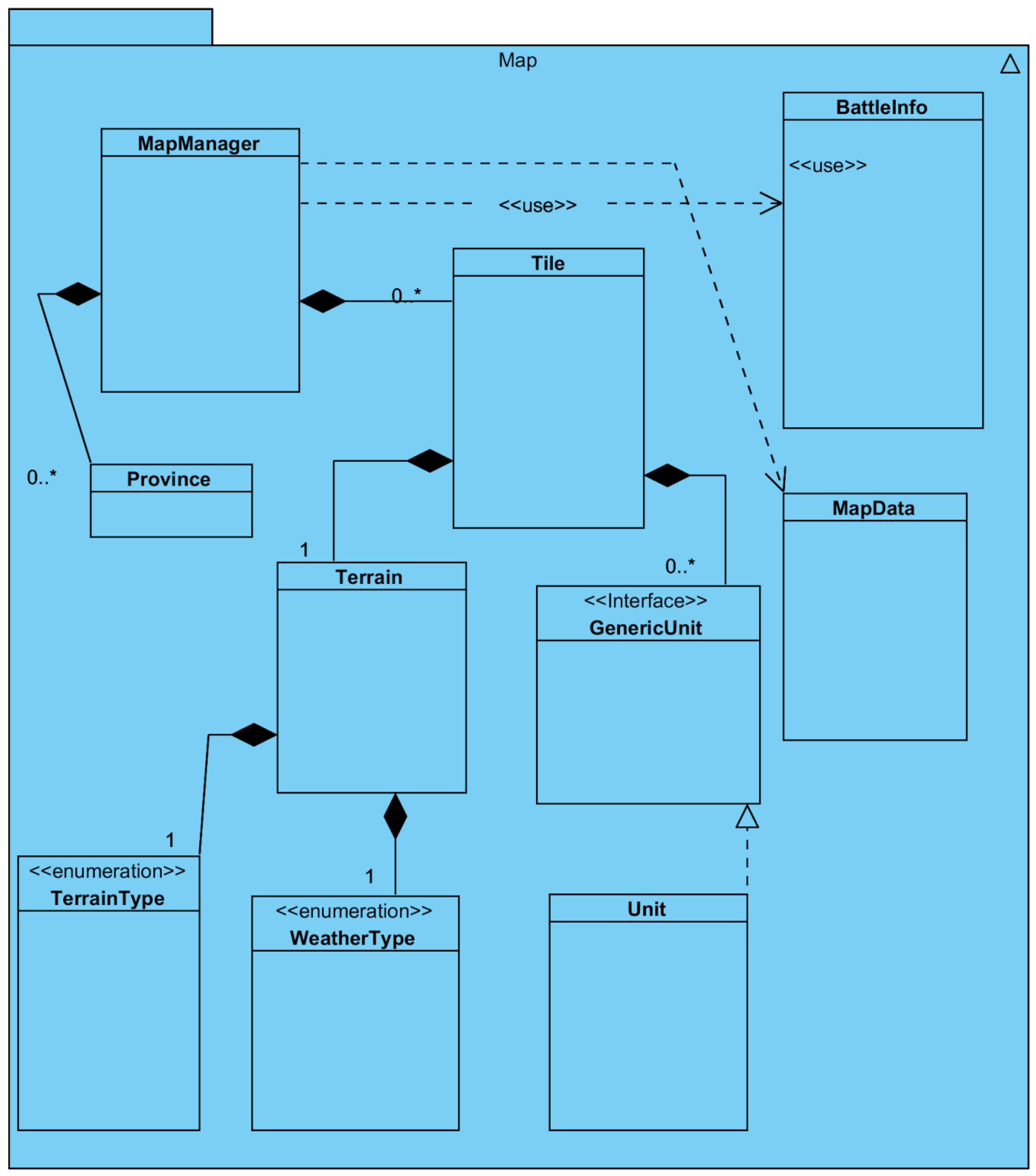


Figure - 2 (Model Classes and MapManager)

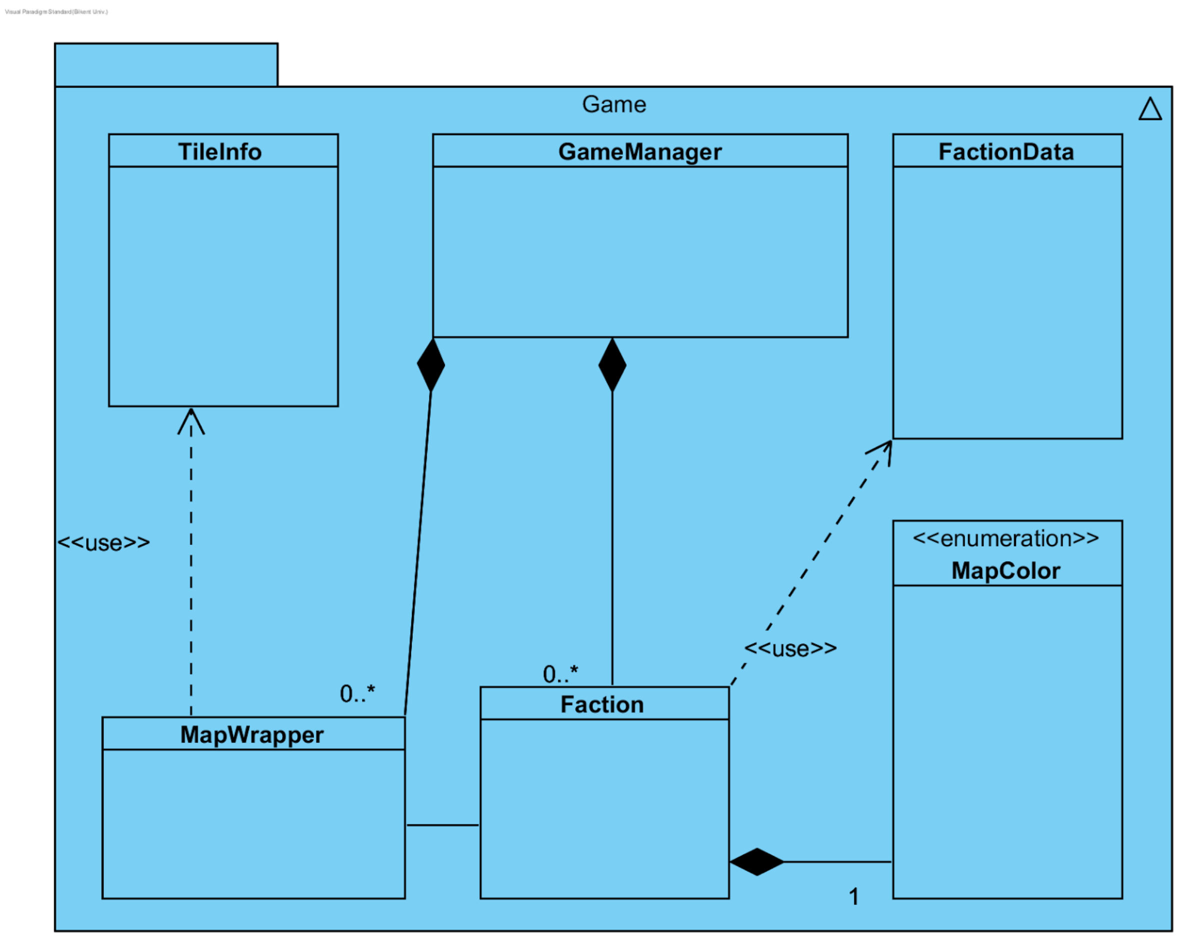


Figure - 3 (GameManager and its components)

Architectural Styles and MVC

Here we will briefly talk about other architectural styles and why our design choice was Model-View-Controller architectural style.

- Repository architectural style did not fit our design because we would have to store a structure called central repository. This style is often used in database dependent systems using the database as a central repository. We do not use a database system therefore we did not choose this style.

- Client/Server style is usually used in web based applications. Our game is a single-player offline game hence this style is irrelevant to our design.
- Peer-to-peer style is again a generalization of client server architecture, only service providers change.
- Pipe and Filter style has to divide inputs from user to different pipelines and filter them in parallel but our system does not require transformations of streams of data. We simply take the input and usually there is not many ways we can handle that input. This style is usually used in operating systems.
- Three Tier and Four Tier designs were the closest to MVC among others, actually we also made use of layering in our system but since these designs have storage layers in them and we do not require any huge amounts of storage, they are also unnecessary.

Why MVC?

Because MVC was the best fitting style among them all. Model (Map) subsystem in our design was developed in such a way that it does not depend on View and Controller subsystems. (UI and Game respectively.) This is also the reason why we could easily divide our design into a closed layer system. Also MVC is very object-oriented friendly.

2.1.a - Ball-and-socket

Again, in subsystem decomposition, it is important to focus enumerating the operations, their parameters and high-level behaviour. In order to help strengthen our design, we added provided and required interfaces with assembly connectors, also called ball-and-socket connectors (lollipop). Below is a ball-and-socket diagram for our design.

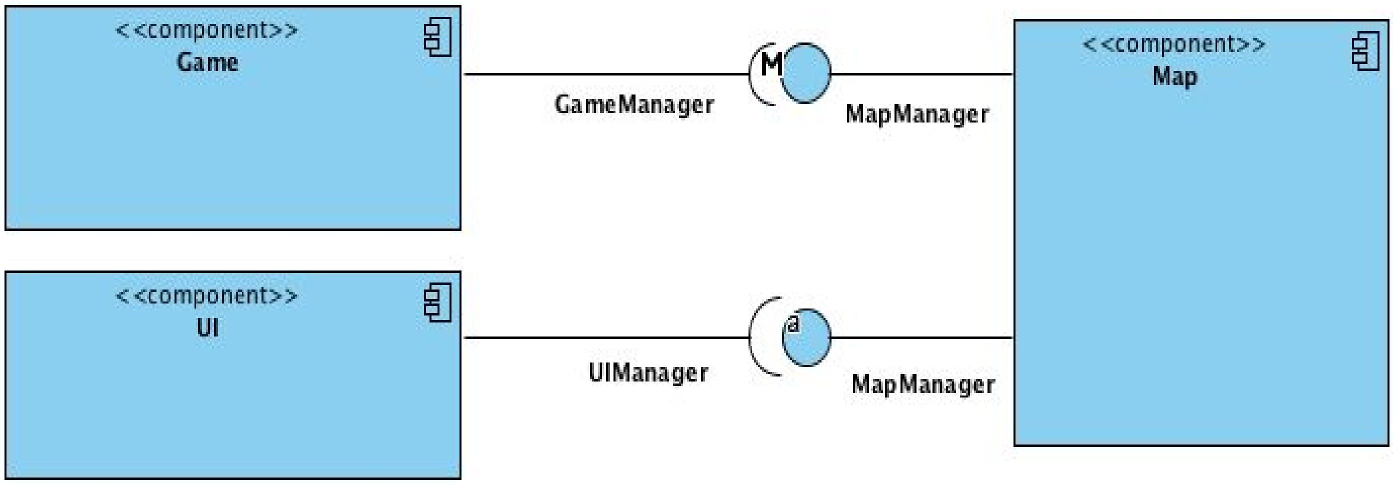


Figure - 4 (Ball-and-socket approach for subsystems)

Let's explain what this ball-and-socket tells us about. First of all, we understand that our Map subsystem is responsible for mainly feeding other subsystems data about the game. Of course this is not the only way that our subsystems interact, however this is the most commonly used one. Also we can say by looking at this diagram that our design makes use of façade pattern, because all these subsystems has a manager class in their own name which is responsible for the exchange of information.

2.1.b - Coupling - Cohesion

Coupling and cohesion are important aspects of subsystem design concepts which is why we separated a part for explaining them in our design.

Since ideal subsystem decomposition should minimize coupling and maximize cohesion, we tried achieving that. In our project, we minimized coupling which is dependency between two subsystems. As you could see in our detailed class diagram, coupling is very low hence we only have façade classes talking to each other handling input and output of the whole subsystem. This means between subsystems there is only 1 coupling class which is the façade class.

On the other hand, we maximized cohesion which is dependencies between classes in the same subsystem. Obeying Model View Controller helped us to bring together the classes that should work together with high cohesion. Again as you can see we have high cohesion, almost every class in the same subsystem exchanges and uses that information to achieve their goals.

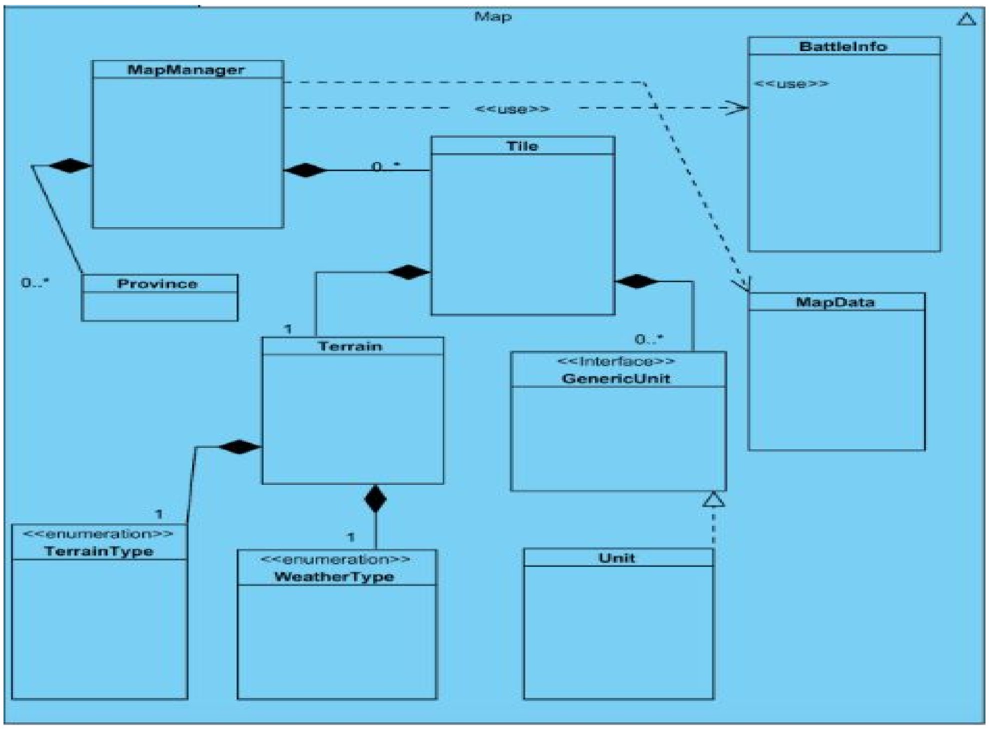


Figure - 5 (Determining Subsystems / Low Cohesion)

Now look at Figure 5, we can see that our Map subsystem actually does not have such a high cohesion. One might be tempted to think that this subsystem maybe should have been divided into two subsystems which has higher cohesion. But if you look closer, you can see that this is not possible. Every part of our subsystem has homogenous low cohesion which means dividing it into different subsystems would not actually mean anything, it would only bring unnecessary complexity. Also we obey the 7+2 rule of thumb of number of classes in a subsystem, hence we chose to not divide it. Moreover, purposes of classes perfectly fit together since they all represent a modelling of some object in game.

2.1.c - Layers and Partitions

Now we look at a hierarchical decomposition of our system. As you know, a layer is a grouping of subsystems providing related services. We have 3 layers for our 3 subsystems. Our system has a closed architecture. It consists of a single vertical slice. You can see our layer in below Figure - 6

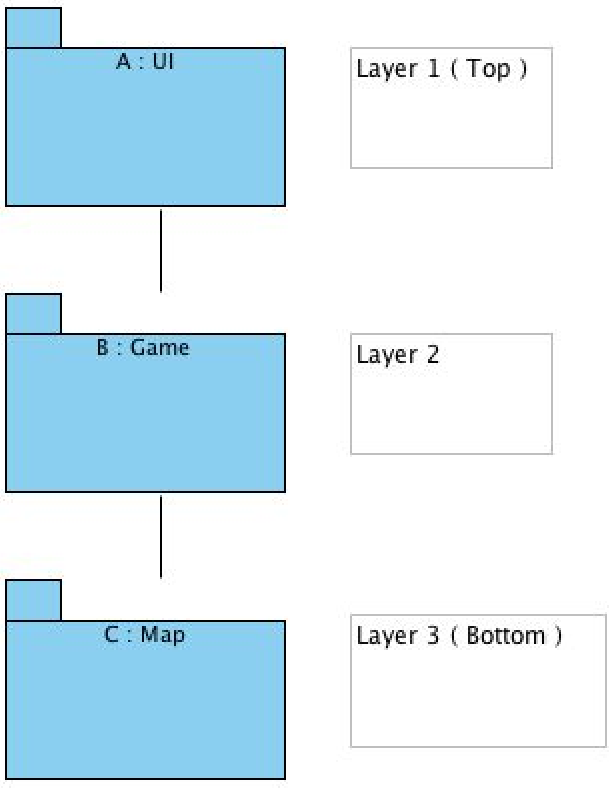
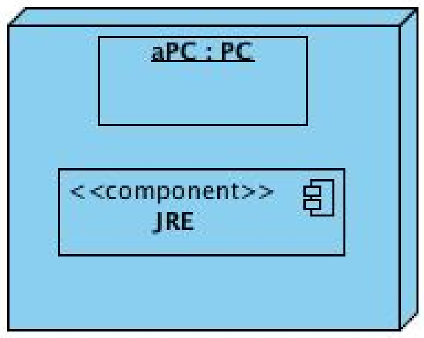


Figure - 6 (Layers and Partitions)

2.1.d - UML Deployment Diagram



UML deployment diagram shows physical devices or execution environments. As one can see we have a very simple UML deployment diagram because our game runs in Java Runtime Environment and will only work on devices that have it, in this case a PC.

- Our game does not have an internet connection and is single-player, that's the reason we do not have any WebServer physical components.
- Also no database required we only make use of local storage on text files, this means no Database physical component either.

One physical component will simply be enough.

2.2 - Hardware - Software Mapping

Since our game is written in Java it will require Java Runtime Environment, which can be found across many devices since Java is a widespread environment. Considering hardware requirements one needs to have a keyboard and a mouse, but mainly mouse will be used throughout the game. Game requires minimal amount of today's hardware in order to run smoothly, even if you do not have a considerably reasonable hardware you might enjoy the game. It would be nice to have a good CPU though since we are using JavaFX 32-bit graphics and it would give the user a much more enjoyable experience in gaming.

2.3 - Persistent Data Management

First of all we are not using any databases since the amount of persistent data is minimal in our game design. We are storing our persistent data in files and folders without using any encryption algorithm since we want people to modify it depending on their interest. Background images and music could be easily changed by adding them to the game folder. Hard-disk drive is used to store all these information. We also removed the saving classes

2.4 - Access Control and Security

Since the game we are creating is single-player and must be downloaded to individual computers, we did not see any necessity to implement an authentication system in our game. But security of the program is controlled in a different way rather than authentication, we divided our logical elements to many different smaller components in order make it easy to track and debug, this will increase readability and also the robustness of our code which will make it harder to take down. Also only making the necessary parts public will make our code like a black-box which means one will not be able to modify the code as they want and will have to obey the obligatories of the system we created.

2.5 - Boundary Conditions

In case of losing all the tiles that player owns, game will return to main menu screen and player will granted the loser title. The same condition occurs if the player will successfully conquer all tiles possible than the game will again return to main menu and player will be granted as a winner. If user opens the program again while playing, one of the games will exit. Also if an unexpected event occurs, the game will safely exit without harming anything on the system.

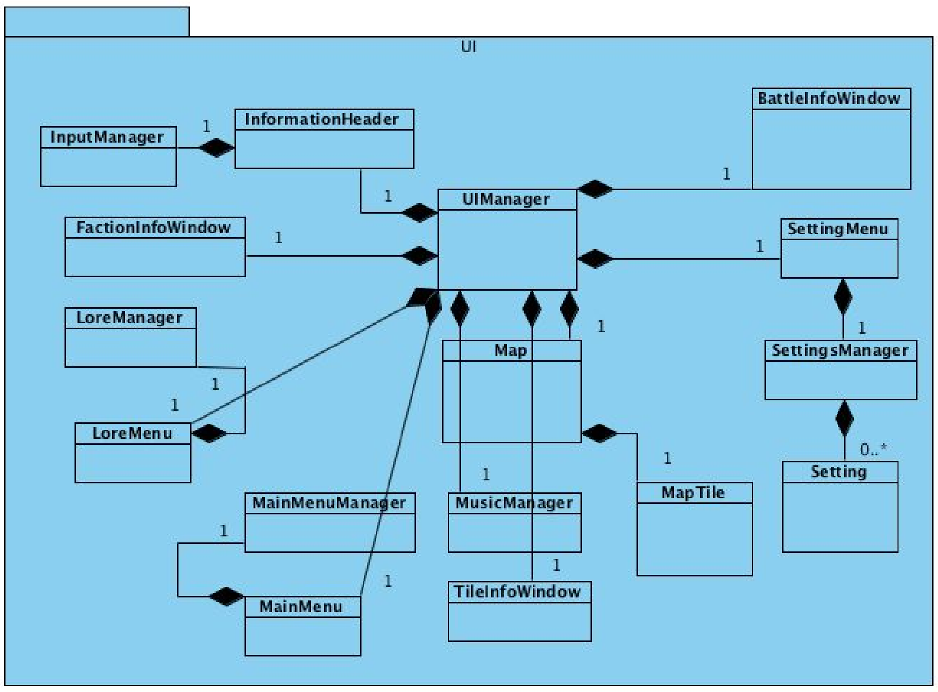
2.6 - Object Design Patterns

We used two design patterns in our project to achieve better understanding of our system, making it more readable and understandable by readers. In this section we will be giving examples and explaining why we used both of them.

2.6.a - Façade Design Pattern

As you know, Façade Design Pattern is a design pattern which helps us to hide complexity of the system and provides a huge but a single interface that user can interact with. It increases simplicity of the program by making that class handle all kinds of inputs and outputs in that subsystem.

We used three Façade classes for our three subsystems, namely UIManager, GameManager and MapManager. Pictures below explains this design pattern.



As you can see, all classes have either direct or indirect relationship with class UIManager which is our Façade Class, if you look at detailed object class diagram you can see that it handles all kind of inputs and outputs. Also it is an interface which is responsible for talking to other subsystems.

2.6.b - Singleton Design Pattern

Also we used singleton design pattern to increase protection of our classes. For instance, our Façade classes require only to be initialized one and not more, which brings in the singleton design pattern.

We have many managers in our game that handle different situations, all of them are created with singleton design pattern.

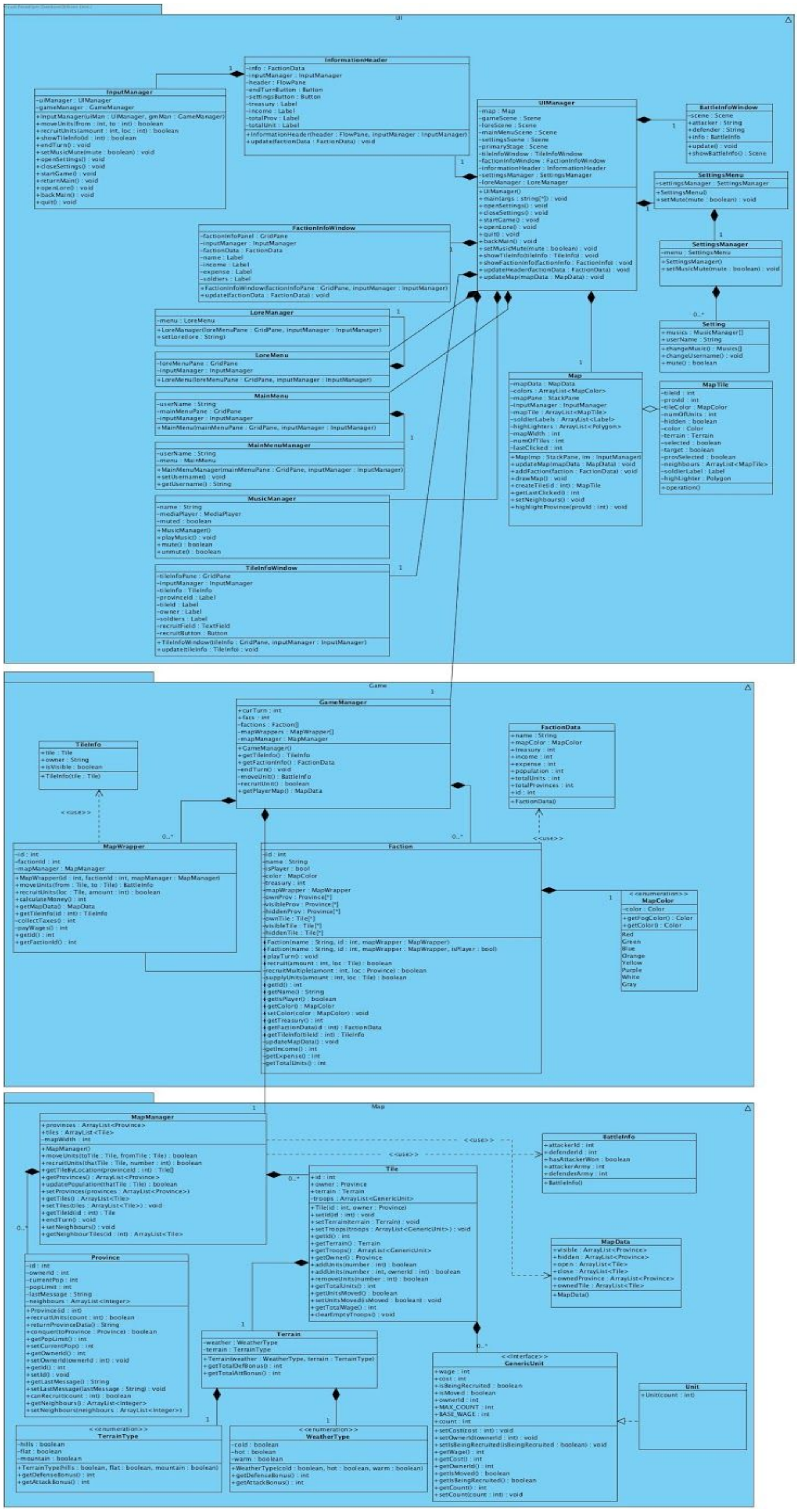
2.7 - Additional Design Requirements

- Two new design patterns added, Façade and Singleton.
- New Classes and better functionality added.
- UML Deployment Diagram added.
- Layers and Partitions added.
- Coupling-Cohesion explained.
- Made use of ball-and-socket, aka lollipops.
- Architectural style MVC and reasons explained.
- Design goals are detailed.
- Better class explanations added.
- Overall simplification of design report.

3 - Subsystem Services

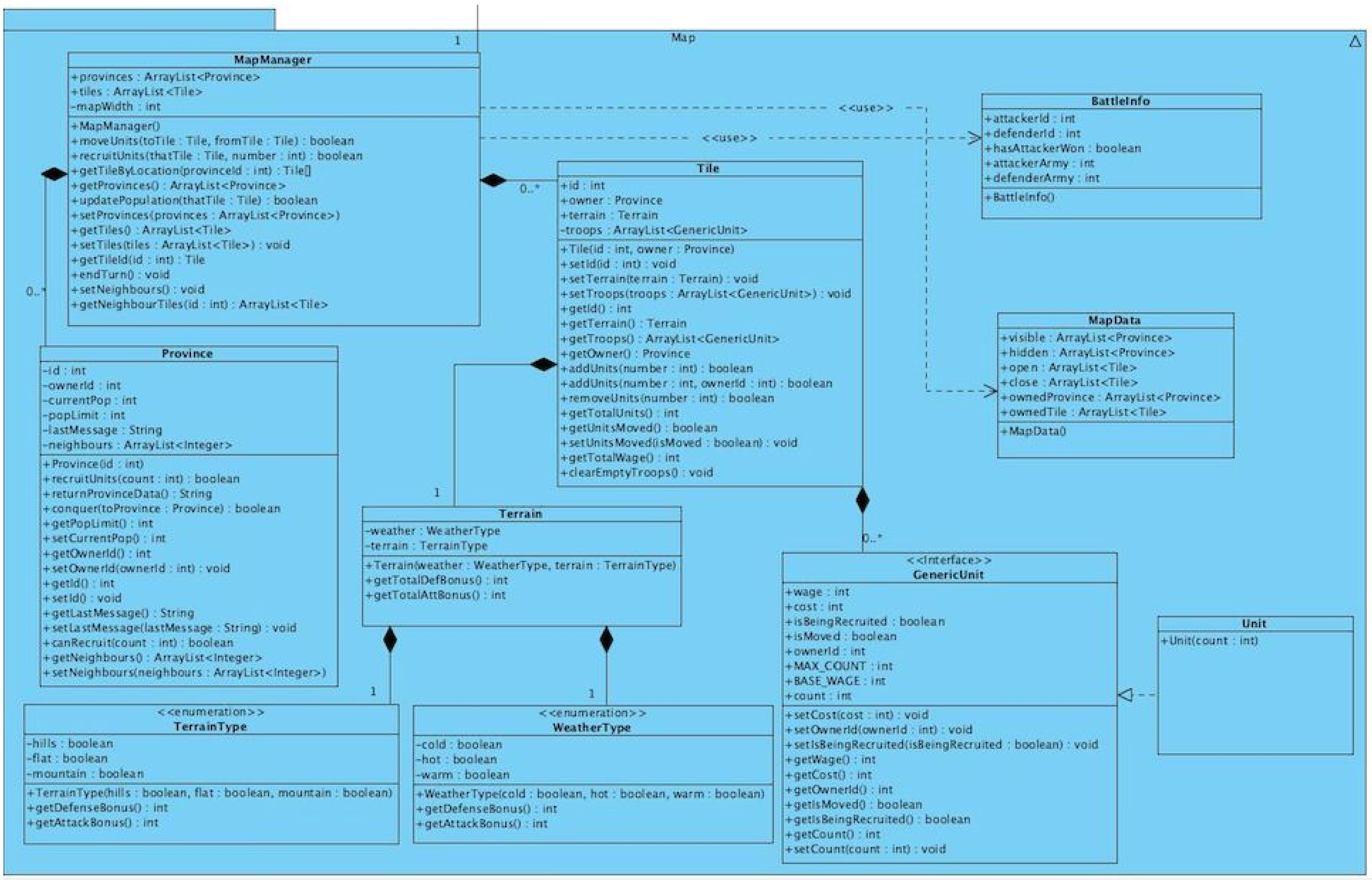
3.1 - Detailed Object Design

In this section, the UML diagram of classes can be seen in detail. The game has three subsystems and this subsystems are connected to each other through UIManager, GameManager and MapManager controller classes.


UI
InformationHeader
InputManager
UIManager
BattleInfoWindow
SettingsMenu
SettingsManager
Setting
FactionInfoWindow
LoreManager
LoreMenu
MainMenu
MainMenuManager
MusicManager
TileInfoWindow
Map
MapTile
Game
GameManager
TileInfo
FactionData
MapWrapper
Faction
<<enumeration>> MapColor
<<use>>
Map
MapManager
BattleInfo
Tile
MapData
Province
Terrain
<<interface>> GenericUnit
Unit
<<enumeration>> TerrainType
<<enumeration>> WeatherType

3.2 - Model (Map) Subsystem

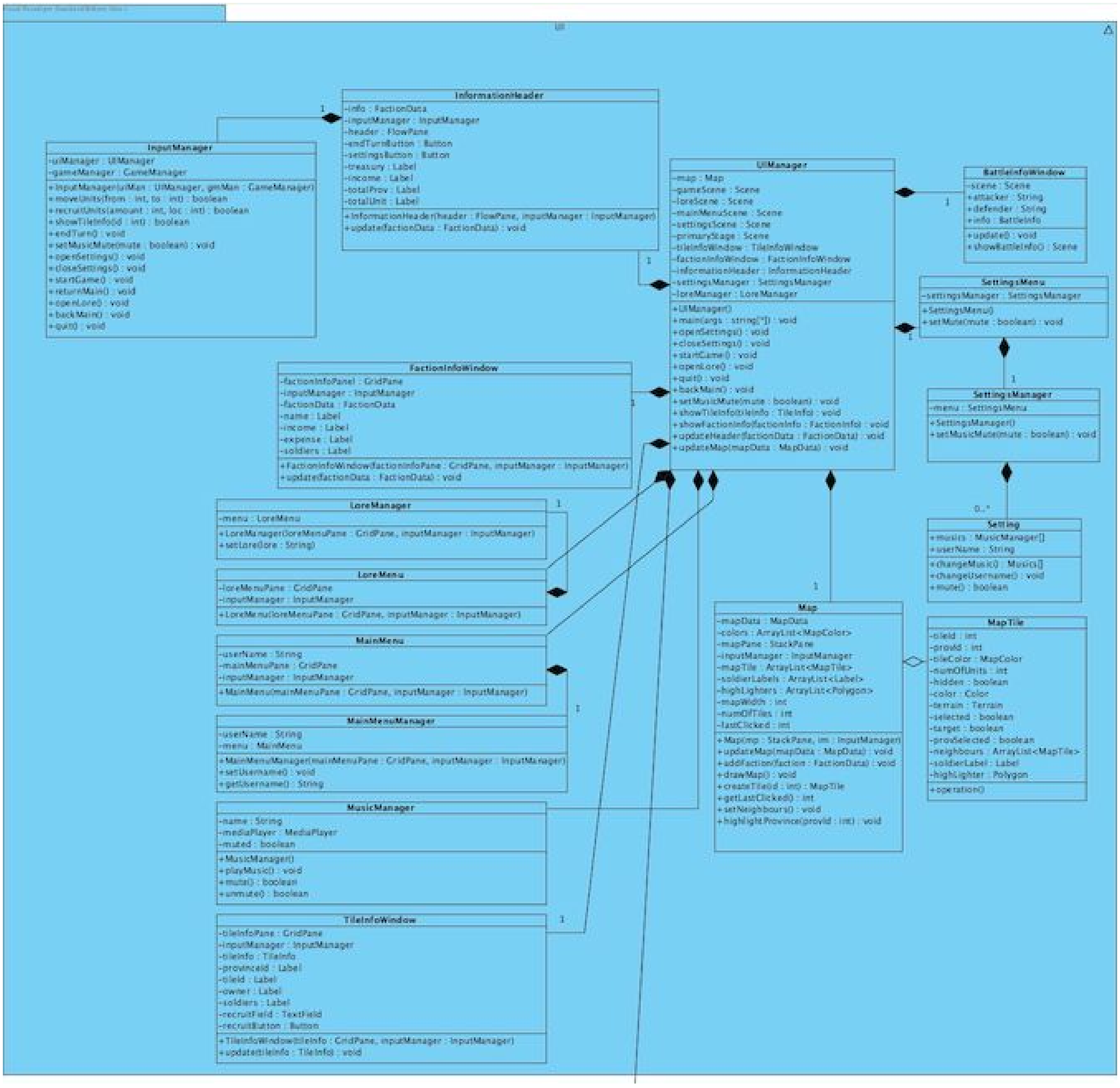
Model subsystem consists of model classes and a controller class (MapManager). These classes hold information about the status of the game. Required information can be reached through MapManager class but modifiable only by the object itself that holds information.



Map
1
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>
0..*
<<use>>
<<use>>
BattleInfo
+attackerId : int
+defenderId : int
+hasAttackerWon : boolean
+attackerArmy : int
+defenderArmy : int
+BattleInfo()
0..*
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>)
1
Terrain
-weather : WeatherType
-terrain : TerrainType
+Terrain(weather : WeatherType, terrain : TerrainType)
+getTotalDefBonus() : int
+getTotalAttBonus() : int
0..*
<<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)
1
1
<<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

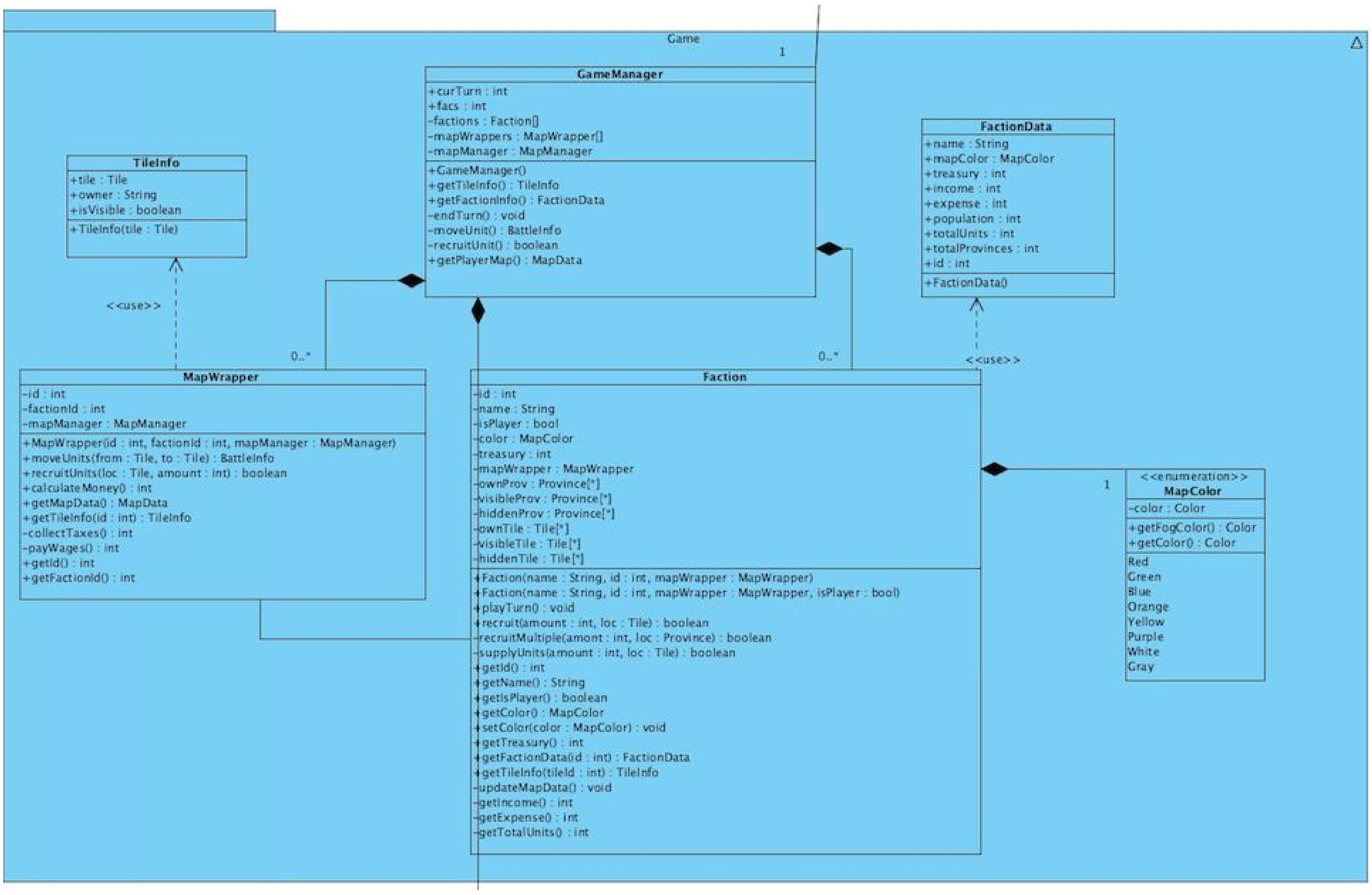
3.3 - View (UI) Subsystem

View subsystem is responsible for the presentation of the game. It consists of view classes and a controller class (UIManager). View classes acquire required information through UIManager class.



3.4 - Controller (Game Logic) Subsystem

Controller subsystem consists of UIManager, MapManager, GameManager and components of GameManager. Controller classes establishes connections between model and view classes and controls the general flow of the game.



3.5 - Detailed Description of Classes

GenericUnit Class

"GenericUnit" is an interface about the troops in the game. Just like a usual interface, "GenericUnit" is responsible of being a parent to all the units in the game and it carries characteristics from all units. It is parent of "Unit" Class and will be used by "Tile" Class. Its attributes are rather straightforward.

Attributes:

public int wage: Is an integer attribute which denotes to the wage the troops will be requiring.

public int cost: Is also an integer which denotes the training cost, which is the gold player needs to use in the start in order to create the soldier.

public boolean isBeingRecruited: Is a boolean value which denotes to the process of recruiting troops which takes time. If troops are in training phase, this value will be true.

public boolean isMoved: Is a boolean value changed constantly with other control classes, checks if the unit is being moved or moving. Returns true if that GenericUnit is on the move.

public int ownerId: Is an int attribute denoting which emperor does that specific unit belongs to.

public static int MAX_COUNT: This is a constant used to denote the maximum number of troops.

public static int BASE_WAGE: This integer constant denotes the base wage amount.

Methods:

public void setCost(int cost): This method helps us to set the costs of troops.

public void setOwnerId(int ownerId): This method helps setting owner's id which will later on help to determine whom does the tiles and troops belong to.

public void setIsBeingRecruited(boolean isBeingRecruited): This is a boolean function which helps us to determine whether we are recruiting any soldiers in that specific turn. Helps us to manage controlling other things that emperor can perform.

public int getWage(): Returns the wages required for troops.

public int getOwnerId(): Returns id of owner to perform user specific tasks later.

public boolean getIsMoved(): A boolean function which returns if troops are moved. Helps in calculations of battle and sending troops to emperor's own tiles.

public int getCount(): A function which returns total number of troops.

public void setCount(int count): A function which helps us to set total number of troops in a Tile, which will help in initialization phase and should not be used anywhere else because in game phase we will be using recruit function.

Reason why there is an interface like this will be clear in "Unit" Class's description. A "Tile" Class can have zero or more "GenericUnit" but in a specific time a "GenericUnit" can only belong to one "Tile". Also attributes are public because tiles will have to use these attributes for calculations.

Unit Class

"Unit" is a class which represents a single specific type of troop. It has no attributes neither methods because it is child of "GenericUnit" Class. The reason of this class being child of "GenericUnit" and this simply being an empty class is because in this iteration there are no different unit types. Our group is considering to add different unit types in game to increase variety but it is postponed for now. In other iterations probably this Class will denote to a specific type in many, and there will be some other classes like this with their own attributes, children of "GenericUnit" Class.

Constructor:

public Unit(int count): This constructor helps us to initialize a given number of units of Unit type.

WeatherType Class

“WeatherType” is an enumeration of class “Terrain”. Since in game, each Tile will belong to a Terrain, Tile has the Terrain attribute. Terrains have different weather types and terrain types which change the advantages and disadvantages in battle.

Attributes:

private boolean cold: Is a boolean value which will be true if terrain has cold weather.

private boolean hot: Is a boolean value which will be true if terrain has hot weather.

private boolean warm: Is a boolean value which will be true if terrain has warm weather.

Methods:

public int getAttackBonus(): This method will return an integer denoting the amount of bonus after making calculations about the attacker’s bonus depending on the weather in terrain.

public int getDefenseBonus(): This method will return an integer denoting the amount of bonus after making calculations about the defender’s bonus depending on the weather in terrain.

Constructor:

public WeatherType(boolean cold, boolean hot, boolean warm): Initializes a weather type, cold hot or warm.

We have public getter methods and private values of weather because we do not want them to be modifiable after being set. Also there are several reasons to use enumeration. Especially type safety is our main concern here because compiler will enforce type safety in particular enum types.

TerrainType Class

“TerrainType” is yet another enumeration of class “Terrain”. Terrains have different ground conditions which also has their own attack and defense bonuses.

Attributes:

private boolean hills: Is a boolean value which will be true if terrain has hills as a surface, hill is in between flat and rocky surfaces.

private boolean flat: Is a boolean value which will be true if terrain has flat surface.

private boolean mountain: Is a boolean value which will be true if terrain has rocky surface.

Methods:

public int getAttackBonus(): This method will return an integer denoting the amount of bonus after making calculations about the attacker’s bonus depending on the ground condition in terrain.

public int getDefenseBonus(): This method will return an integer denoting the amount of bonus after making calculations about the defender’s bonus depending on the ground condition in terrain.

Constructor:

public TerrainType(boolean hills, boolean flat, boolean mountain): This constructor constructs a specific terrain type.

Same reasons of creating enumeration and creating private attributes with public getters with WeatherType still applies here.

Terrain Class

“Terrain” is the class where the enumerations “TerrainType” and “WeatherType” are used. “Terrain” class is used by “Tile” class since the game logic suggests that each tile in the game is on a terrain, therefore “has a” terrain. Terrain Class will simply use enumerations to calculate total bonuses.

Attributes:

private WeatherType weather: Enumerated weather will give us the bonuses from weather.

private TerrainType terrain: Enumerated terrain will give us the bonuses from terrain.

Methods:

public int getTotalAttBonus(): This method returns the total attack bonus calculated by weather and terrain in integer form.

public int getTotalDefBonus(): This method returns the total defense bonus calculated by weather and terrain in integer form.

Constructor:

public Terrain(WeatherType weather, TerrainType terrain): Depending on given weather type and terrain type, creates a particular terrain.

Tile Class

“Tile” is the class representing tiles in game. A tile can be thought of a piece of land without an owner. Tile will be in constant communication with “Province” class and “MapManager” class which will be explained later to determine the owner of that specific tile. A Tile can have zero or more generic units on it, whereas it must have one terrain instance only. Also a terrain could not exist on itself and can not be a part of another class. Similarly a tile could not be created without having an instance of a terrain. Hence we have aggregation between them.

Attributes:

public static int id: Each tile will have a unique integer id hence we can store all tiles in an integer array and access them easier. In a similar sense we can call id as names of the tiles. It is public because we want it to be visible to other classes in the same component.

public Province owner: Tiles are neutral in their natural form. Belonging to a province will make them non-neutral. Since neutral lands will have simpler cases rather than belonged lands, it is logical for tiles to have provinces as owners. It is public because owner can be changed by other classes in the same component.

public static Terrain terrain: Each tile has a specific terrain which changes their behaviour during wars. We have created a terrain object to denote to this situation and tile will have an instance of it. It is static because neither weather nor terrain will change as the game progresses.

private GenericUnit[] troops: This will denote to the number of troops on the tile. It is private because the methods will handle changing the number of troops after a war or a migration. Hence there is no need for this attribute to be modifiable.

Methods:

public boolean addUnits(int number): This method returns a boolean value if the given parameter number of units can be added to this tile, also adds the number of troops to the tile.

public boolean removeUnits(int number): This method tries to remove the given number of units from the tile, if successful returns true as a boolean value.

public void setId(int id): Sets the id of the Tile. This function is used to denote belongness of that Tile.

public void setTerrain(Terrain terrain): This function is used to place a specific Tile into a Terrain, which will help us determine its attack and defence bonuses in war situations.

public void setTroops(ArrayList<GenericUnit> troops): This function is obviously used for setting troops in a particular Tile. Should only be used during initialization phase because addUnits method will be used after setting troops for that Tile.

public int getId(): Returns the id of that Tile.

public Terrain getTerrain(): Returns the Terrain that Tile belongs to. This helps us determining attack and defence bonuses and calculations in war.

public ArrayList<GenericUnit> getTroops(): Returns the troops belonging to that Tile. This function is also used in battles but this time for troop and loss calculations.

public Province getOwner(): Returns the owner of the Tile in Province form. Helps to determine visible and invisible tiles.

public boolean addUnits(int number, int ownerId): This is an extended form of addUnits, difference is that it helps to determine whom to add the troops if Tile is not specific. Used in general troop calculations.

public int getTotalUnits(): Returns number of total units.

public boolean getUnitsMoved(): Returns if the units are moved in that specific turn.

public void setUnitsMoved(): When units are moved in that turn, sets units as moved so that they can not move in that very same turn, after a turn this method is called again to reset moved.

public int getTotalWage(): Returns total wage requirements of troops in that tile.

public void clearEmptyTroops(): Clears unnecessary empty troops.

Constructor:

public Tile(int id, Province owner): This is a constructor for initializing a Tile with an owner and id.

MapData Class

“MapData” is basically a data storage class for simplification purposes. It stores data about map’s visibility which will determine an important aspect of the game which is called “fog of war”. It means that some of the map will not be able for the user to march through since it is not yet discovered. Discovery of tiles are made by moving an army to the closest tiles to that tile.

Attributes:

public ArrayList<Province> visible: Stores an array of visible provinces. This means that when visible provinces are clicked, player will know to which empire that tile belongs to.

public ArrayList<Province> hidden: Stores an array of hidden provinces. This means that player will not be able to gather any information about the belonging of the tile.

public ArrayList<Tile> open: Stores an array of open tiles. This means that player is able to attack to that tile since no fog of war closes it

public ArrayList<Tile> close: Stores an array of closed tiles which means player will not be able to attack it since fog of war closes it. Player will need to explore the closest tile nearest to these tiles if they want to take some action against it.

public ArrayList<Province> ownedProvince: Lists owned provinces. This attribute is created to help AI gather faster information with sacrificing some memory.

public ArrayList<Tile> ownedTile: Lists owned tiles. This attribute is created to help AI gather faster information with sacrificing some memory.

Constructor:

public MapData(): Initializes a MapData object which will store data about map.

Province Class

“Province” is the class which determines which tile belongs to whom. It is one of the core classes of this subsystem because it carries the conquer function within it. Conquering is the main object of this game which makes this class very important. Main duty of the class is to determine who is the owner and calculate what condition is the population limit is in, also it gives data about the province to “MapManager” class which will use it to do further actions with other classes.

Attributes:

public int id: Just like tiles have id which represent their names, Provinces will also use id’s which will also correspond to their names in the sake of simplicity of actions taken between methods.

public int ownerId: Each conqueror has an id which will help the game to determine if the Province is an enemy province or allied province. Also it has responsibility in conquer method. One is not able to conquer an allied land obviously.

public int currentPop: This variable stores the current population and is changed when troops take over the province or troops are trained.

private static int popLimit: This variable stores the top population limit a province can achieve. Is static and private because it helps canRecruit function to determine if one can recruit the amount of soldiers they want to, also once the maximum value of population is determined it can not be changed.

public String lastMessage: This is a variable for returning information to other classes.

public ArrayList<Integer> neighbours: An arraylist storing neighbours by their id’s.

Other attributes are public because they will be used mostly by MapManager class which is in the same subsystem.

Methods:

private boolean canRecruit(int count): This method checks if the owner of province can recruit the amount of units they want, returns true if it is possible. This method is a helper method of recruitUnits method, since recruitUnits will have to check if it is possible to recruit without reaching population capacity. It is private because it will only be used by recruitUnits method.

public boolean recruitUnits(int count): This method recruits the wanted number of units to the province. Public because it will work in a synchronously fashion with Tile class.

public String returnProvinceData(): Method returns data of the province, such as population and owner in a String form.

public boolean conquer(Province toProvince): This is considerably one of the most important methods in the game which starts the process of conquering another land. It will work with BattleInfo and MapManager to handle this complexity. Will return true if user is able to conquer the land successfully.

public int getPopLimit(): Returns population limit.

public int setCurrentPop(): Sets a new population limit. Should be only used in initialization phase because population capacities can not change over time in course of game.

public int getOwnerId(): Returns id of owner of that province.

public void setOwnerId(int id): Sets id of owner. Might change in war situations when someone conquers a place, also is used in initialization phase.

public int getId(): Returns id of province.

public void setId(int id): Sets id of province.

public String getLastMessage(): Returns the last message of the province. Used to help in showing information of provinces. Especially useful when determining labels in the game.

public void setLastMessage(String lastMessage): Changes the last message which consists information about gold, troops and population limits.

public ArrayList<Integer> getNeighbours(): Returns neighbours of that province, helps to see if visible or invisible also if enemy or ally provinces.

public void setNeighbours(ArrayList<Integer neighbours): Sets the neighbours of that Province, is used when conquering and initialization.

BattleInfo Class

“BattleInfo” class is the class where calculations about the clash will be made. This class is also a core class of this subsystem because since the main operation of this game is to battle your way out others and this class is the single class which will determine the future of events. This class will exchange information with Province and MapManager frequently which are in the same subsystem.

Attributes:

public int attackerId: This attribute helps us determine who is the attacker. Either the enemy will be the attacker or the player.

public int defenderId: Similar like the above attribute, only now determines the defender.

public int attackerArmy: After determining who is attacking and who is defending, BattleInfo stores the information of attacker’s army, how many troops it includes in order to start clashing armies. It is GenericUnit[] because if there were more than one troop types, which there is not in the first iteration, simply storing int would be insufficient. Also this value is public because we want it to be modified by MapManager class. Then we would need complex methods in BattleInfo in order to determine which unit is stronger to other and other dependencies but this simplicity is just enough for now. Note: It is not in GenericUnit[] anymore because there is still only one type of troop.

public int defenderArmy: Similar like the above attribute, only for the defender's army is stored in this one.

public boolean hasAttackerWon: Is the boolean attribute which will change after methods calculate the winner, is set to true if attacker succeeds to conquer.

Constructor:

public BattleInfo(): Initializes a BattleInfo class to hold battle information.

MapManager Class

"MapManager" is the main class of this subsystem, handling all map operations, exchanging information constantly with Province, BattleInfo and Tile classes. Class also is the only class that regulates operations with other subsystems, exchanging information with them. Handles the population update, taking information from tiles and provinces also moving and recruiting units.

Attributes:

public Province[] provinces: Holds all the provinces and makes it public because Provinces are meant to be reached by other subsystems.

public Tile[] tiles: Stores tiles in public form because MapManager exchanges informations with other classes in different subsystems.

private int mapWidth: Stores width of the map.

Methods:

public boolean moveUnits(Tile toTile, Tile fromTile): The main function which regulates movement of units, takes two Tile parameters first one is to determine which tile troops are moving and second one is to determine where these troops are going. It returns true if troops are able to reach the destination, false otherwise.

public boolean recruitUnits(Tile thatTile): This function takes a Tile parameter and tries to recruit some troops. Returns true if troops are successfully recruited, false if it is denied because of population limits. Works with using the Tile's addUnit function.

public ArrayList<Tile> getTileByLocation(int provinceId): Returns all Tile's.

public ArrayList<Province> getProvinces(): Finds owners of Provinces and returns them in a Province array.

public boolean updatePopulation(Tile thatTile): Takes parameter of a specific Tile and updates its population in end of every turn. Population needs to increase if troops are successfully recruited.

public void setProvinces(ArrayList<Province> provinces): Sets provinces during initialization.

public ArrayList<Tile> getTiles(): Gets tiles and their owners to do calculations about map.

public void setTiles(ArrayList<Tile> tiles): Sets tiles, used only in initialization.

public Tile getTileId(int id): Returns id of that Tile, helps to determine owner of the tile in that specific turn.

public void endTurn(): Since our game is turn based, this method ends turn after operations are done on the map.

public void setNeighbours(): Sets neighbours, uses functions of lower level classes with the same name. Map Class has the same function with specified functionality.

public ArrayList<Tile> getNeighbourTiles(int id): Returns neighbouring tiles in order to do calculations about map, determining if neighbouring tile is an enemy tile or an ally tile, or visibility of that tile.

MapWrapper Class

MapWrapper class is a wrapper class for MapManager class. MapWrapper controls access to map information. It limits visibility of provinces and tiles for its owner faction that the faction normally shouldn’t be able to see. MapWrapper class establishes connection between Faction class and MapManager class. This connection always consists id of faction and this way operations made by Faction class stays under control and it is customized to the faction.

Attributes:

private int id: Id of the class.

private int factionId: Id of the related faction. This attribute is set once at creation and used at all times.

private MapManager mapManager: A reference to the MapManager object used through out the game.

Constructor:

public MapWrapper(int factionId, MapManager mapManager): MapWrapper class has only one constructor which takes id of the related faction and a reference to the MapManager object used in the game. Constructor creates an id for itself using the faction id and a random integer value.

Methods:

public BattleInfo moveUnits(Tile from, Tile to): Used to move units in one tile to another. Takes the starting tile and destination tile as parameters. If for some reason (like requesting faction is not the owner of the troops in the starting tile) the move is not a valid operation, returns null. If the move results in a battle, returns a BattleInfo object with battle information. If the move does not result in a battle returns a BattleInfo object with attacker id of -1 (not a valid faction id).

public bool recruitUnits(Tile loc, int amount): Checks if the given amount of unit can be recruited in the given tile and recruits if it is. Returns true if recruited and false otherwise.

public int calculateMoney(): Calculates change of the money of the related faction. Calls collecTaxes() and payWages() methods to do that. Returns the result.

public MapData getMapData(): Takes a MapData object from MapManager and returns it after customizing specifically for the related faction.

public TileInfo getTileInfo(int id): Creates and returns a TileInfo object considering informations the related faction can know about the tile with the given id.

private int collectTaxes(): Calculates and returns total amount of taxes.

private int payWages(): Calculates and returns total amount of wages.

public int getId(): Getter function for the id attribute.

public int getFactionId(): Getter function for the factionId attribute.

Faction Class

Faction class represents factions in the game. It stores information and has operations related to a faction. Faction class also has AI related operations that are used to control AI factions. An instance of Faction class created for each faction in the game at the beginning of the game and used through all game.

Attributes:

private int id: Id of the faction.

private string name: Name of the faction.

private bool isPlayer: True if this object is the player faction. False otherwise.

private MapColor color: Color of the faction that owned tiles are shown on the map.

private int treasury: Treasury of the faction.

private MapWrapper mapWrapper: MapWrapper object of this faction.

private Province[] ownProv: Array that holds owned provinces.

private Province[] visibleProv: Array that holds visible enemy provinces.

private Province[] hiddenProv: Array that holds hidden enemy provinces.

private Tile[] ownTile: Array that holds owned tiles.

private Tile[] visibleTile: Array that holds visible enemy tiles.

private Tile[] hiddenTile: Array that holds hidden enemy tiles.

Constructors:

public Faction(string name, int id, MapWrapper mapWrapper, bool isPlayer = false): Name and id of the faction is required when creating an instance of the Faction class. Related MapWrapper object must also be passed as the parameter. isPlayer attribute can only be set at creation. Constructor defines other attributes with default values.

Methods:

public FactionData getFactionData(): Creates and returns a FactionData object that holds information about this faction.

public void playTurn(): This method calculates final treasury value at first. Then updates attributes that holds information about the map. After that if the faction is not the player faction, AI controls and plays the turn for the faction.

public bool recruit(int amount, Tile loc): Calls recruitUnits method of the MapWrapper object with the given information. Returns the returned value.

private bool recruitMultiple(int amount, Province loc): Calculates best locations to recruit units to supply the given Province with the given amount of units and makes corresponding recruitUnits method calls. Returns true if all calls return true and false

Otherwise.

private bool supplyUnits(int amount, Tile loc): Decides which units to move to supply given tile with at least given amount of units and makes corresponding moveUnits method calls. Returns true if enough amount of units moved and false otherwise.

public int getId(): Getter function for the id attribute.

public string getName(): Getter function for the name attribute.

public bool getIsPlayer(): Getter function for the isPlayer attribute.

public MapColor getColor(): Getter function for the color attribute.

public void setColor(MapColor color): Setter function for the color attribute. Set the attribute only if its null (default value).

public int getTreasury(): Getter function for the treasury attribute.

MapColor Class

MapColor is an enumeration class that the possible colors factions can be represented on the map are listed.

GameManager Class

GameManager class is the gamestate controller of Infinite Tale. It gets information about the current state of the game from the MapWrapper and the MapManager classes, controls the turn order of the game and keeps track of the factions that the player and the computer belong to. This way it formats the game info into a controllable state and supplies the UIManager with this information.

Attributes:

public int curTurn: The current turn number.

public int facs: The faction that has to play the current turn.

private Faction[] factions: The factions in the game.

private MapWrapper[] mapWrappers: The map wrappers that are being controlled by the game manager.

private MapManager mapManager: The map manager to get the map info from.

Methods:

public MapData getPlayerMap(): Gets the info of the map which is being controlled by the map wrappers and the map manager.

public TileInfo getTileInfo(): Returns the information about the chosen tile. Uses the map manager to do so.

public FactionData getFactionInfo(): Returns information about the faction that has the current turn.

private void endTurn(): Ends the current turn.

private BattleInfo moveUnit(): Moves the selected units to the selected tile.

private boolean recruitUnit(): Recruits units on the selected tile. This action is limited by the player’s resources and the population cap.

Constructor:

private static GameManager(): Helps to initialize a singleton manager object.

FactionData Class

FactionData is the class where we keep all the information about a certain faction. These include the name of the faction, the color by which the faction is represented on the map, total resources in hand, total income, total expenses, the current population, the current number of movable units and the total number of provinces that are being controlled by the faction.

Attributes:

public string name: The name of the faction.

public MapColor mapColor: The color by which the faction is represented on the map.

public int treasury: Total amount of resources of the faction.

public int income: The income of the faction which is the amount of resources that the faction earns at the end of every turn.

public int expense: The outcome of the faction which is the amount of resources that the faction has to pay at the end of every turn.

public int population: The current total population of the faction which includes peasants as well as the military units.

public int totalUnits: The current number of military units.

public int totalProvinces: The total number of provinces that have been conquered by the faction.

public int id: Corresponds to id as explained before.

Constructor:

public FactionData(): Initializes a FactionData object.

TileInfo Class

TileInfo is the class that holds the information about a given tile. It merges the regular information about a certain tile with the information of its current holder and the province that it belongs to.

Attributes:

public Tile tile: The regular tile object that contains common information for all tiles.

public String owner: Stores owner in a String version.

public boolean isVisible: A boolean value which checks if the Tile is visible, checks fog of war basically.

Constructor:

public TileInfo(Tile tile): This constructor generates information about a Tile.

UIManager Class

UIManager controls UI and also responsible for the initialization of the game. Main method to run the game is in UIManager and starts launching of the game. If the user starts the game from main menu, UIManager creates all necessary classes and controls initialization of the game. During the game, UIManager establishes connection between UI classes and other game classes, taking the role of controller in a MVC structure.

Attributes:

private InformationHeader informationHeader: Holds a reference to the object that controls information header.

private TileInfoWindow tileInfoWindow: Holds a reference to the object that controls tile information window.

private FactionInfoWindow factionInfoWindow: Holds a reference to the object that controls faction information window.

private Map map: Holds a reference to the Map object that is responsible for creation and modification of game map.

private Scene gameScene: Main scene game will be shown.

private Scene loreScene: Lore scene in game will be shown.

private Scene mainMenuScene: Main menu in game will be shown.

private Scene settingsScene: Settings scene in game will be shown.

private Scene primaryStage: Which scene will be shown is chosen here.

private SettingsManager settingsManager: Manager of settings scene, will control what will be shown in settings menu in the game.

private LoreManager loreManager: Manager of lore scene, will control what will be shown in lore menu in the game.

Constructors:

public UIManager(): Constructor of UIManager class just defines some properties with default values.

Methods:

public void main(String[] args): Main method that will be called at program initialization.

public void showFactionInfo(): Shows the faction info window.

public void showTileInfo(): Shows the tile info window.

public void openSettings(): Opens the settings menu.

public void closeSettings(): Closes the settings menu.

public void startGame(): This function starts the game with using main menu scene.

public void openLore(): This function activates when lore menu is opened. Uses lore manager and lore scene.

public void quit(): This function quits from the game, closes it. Calls system.exit function to handle this.

public void backMain(): This function is used to return back to main. Used in SettingsMenu to go back to main screen.

public void setMusicMute(boolean mute): This function is used to mute the music, used again in settings menu to stop music playing.

public void updateHeader(FactionData factionData): This function is used to update information which is shown ingame labels, right label and up label, they show total information of any resource.

public void updateMap(MapData mapData): This function is used to update the visualization of Map shown in game.

Map Class

Map class holds information about the map the player will see and controls the UI elements to show the map.

Attributes:

private MapData mapData: Holds the information about the player map.

private MapColor[] colors: Holds the map colors of the factions.

private StackPane mapPane: Holds a StackPane object for the map.

private InputManager inputManager: Holds an InputManager to be used in managing user inputs.

private ArrayList<MapTile> mapTile: Holds an ArrayList of MapTiles for each individual tile of the map.

private ArrayList<Label> soldierLabels: Holds the Labels for the soldiers on the tiles.

private ArrayList<Polygon> highLighters: Highlighters to be used to show the chosen tiles.

private int mapWidth: Holds the width of the map.

private int numOfTiles: Holds the number of tiles on the map.

private int lastClicked: Holds the number of the last clicked tile.

Constructors:

public Map(StackPane mapPane): Map class has only one constructor that takes the pane object that the map will be drawn on.

Methods:

public void updateMap(MapData mapData): Updates the map information and redraws map.

public void addFaction(FactionData faction): Adds id and color of the given faction.

public void drawMap(): Draws the map using the current class components.

public int getLastClicked(): Returns the last clicked tile.

public void setNeighbours(): Sets the neighbouring tiles.

public void highLightProvince(int provId): Highlights the province for which the chosen tile belongs to.

public MapTile createTile(int id): Creates a tile to use in the construction of the whole map.

SettingsManager Class

"SettingsManager" class is a class that sets the changes which has been modified by the users.

Attributes:

private SettingsMenu menu: Holds the SettingsMenu to be used by the user.

Methods:

public void setMusicMute(boolean mute): Mutes or unmutes the music.

Constructor:

public SettingsManager(): SettingsManager has one constructor which does not take any parameters.

SettingsMenu Class

"SettingsMenu" class manages the Settings section in the Menu.

Attributes:

private SettingsManager settingsManager: Use the settingsManager object to get the required information from SettingsManager class.

Method:

public void setMute(boolean Mute): This method is used by UI to show the Settings in the Menu.

Constructor:

public SettingsMenu(): SettingsMenu has one constructor which does not take any parameters.

Setting Class

Shows the sections that user can make changes.

Attributes:

public MusicManager musics[]: Musics which are ready to play in the game.

public string userName: The username of the player.

Methods:

public void changeMusic(): Change the music that has been playing in the game.

public void changeUsername(): Change the username of the player to distinguish the different players.

public boolean mute(): Return true if the player wants to mute the sound, if not; keep false.

MusicManager Class

Attributes:

private String name: Holds the name of the music to be played.

private MediaPlayer mediaPlayer: Holds the MediaPlayer object to be used in playing the music selected by the user.

private boolean muted: Holds the boolean value showing whether the music has been muted or not.

Methods:

public void playMusic(music): Method that plays the music which user has selected.

public boolean mute(): Stops the currently selected music from playing.

public boolean unmute(): Starts playing the music again if it was muted before.

Constructor:

private static MusicManager(): MusicManager has one constructor which does not take any parameters.

TileInfoWindow Class

Attributes:

private Label owner: Tile’s Owner’s name.

private TileInfo tileInfo: TileInfo object to display in TileInfoWindow.

private GridPane tileInfoPane: Holds the pane to display the info about the tile on.

private InputManager inputManager: Holds the input manager of the TileInfoWindow.

private Label provinceId: Holds the Label for the ID of the province that the chosen tile belongs to.

private Label soldiers: Holds the Label for the soldiers on the chosen tile.

private TextField recruitField: Holds the TextField where the user is expected to specify the amount of soldiers to be recruited.

private Button recruitButton: Holds the Button to finalize the recruitment operation with the specified amount of soldiers taken from the recruitField.

Constructor:

public tileInfoWindow(GridPane tileInfo, InputManager inputManager): Scene object is returned to be displayed on TileInfoWindow.

Methods:

public void update(TileInfo tileInfo): Repaints TileInfoWindow scene.

FactionInfoWindow Class

Attributes:

private FactionData factionData: FactionData object to display in TileInfoWindow.

private GridPane factionInfoPanel: GridPane to draw the FactionInfoPanel on.

private InputManager inputManager: User input manager for the FactionInfoWindow.

private Label name: Name of the chosen Faction.

private Label income: Current total income per turn of the chosen Faction.

private Label expense: Current total expenses per turn of the chosen Faction.

private Label soldiers: Current total amount of soldiers owned by the chosen Faction.

Methods:

public void update(): Repaints FactionInfoWindow scene.

Constructor:

public FactionInfoWindow(GridPane factionInfoPane, InputManager inputManager): FactionInfoWindow has one constructor which takes two parameters that are the GridPane object to draw the FactionInfoPanel on and the InputManager object to be used in managing the user inputs respectively.

BattleInfoWindow Class

Attributes:

private Scene scene: Contents of BattleInfo is displayed by using this scene.

public String attacker: Attacker's name.

public String defender: Defender's name.

public BattleInfo info: BattleInfo object to display in BattleInfoWindow.

Methods:

public Scene showBattleInfo(): Scene object is returned to be displayed on BattleInfoWindow.

public void update(): repaints BattleInfoWindow scene.

InformationHeader Class

Attributes:

public FactionData info: Contains player's faction data.

Methods:

public void update(): repaints InformationHeader scene.

Constructor:

public InformationHeader(FlowPane header, InputManager inputManager): InformationHeader has one constructor which takes two parameters that are the FlowPane for the header to be displayed on and the InputManager to manage the user inputs respectively.

LoreManager Class

Attributes:

private LoreMenu menu: The menu for the information about the lore to be displayed with.

Methods:

public setLore(String lore): Sets the lore to the given String.

Constructor:

public LoreManager(GridPane loreMenuPane, InputManager inputManager): LoreManager has one constructor that takes two parameters that are the GridPane for the LoreMenu to be displayed on and the InputManager for the user inputs respectively.

MapTile Class

Attributes:

private int tileId: ID of the MapTile.

private int provId: ID of the Province that the MapTile belongs to.

private MapColor tileColor: Color of the MapTile chosen from the MapColor enumeration.

private int numOfUnits: Number of units on the MapTile.

private boolean hidden: Shows whether the MapTile is affected by the Fog of War or not.

private Color color: Generic color for all MapTiles.

private Terrain terrain: The terrain specialty of the MapTile that gives attackers/defenders different advantages/disadvantages.

private boolean selected: Shows whether the MapTile is currently selected by the user or not.

private boolean target: Shows if the MapTile is being targeted by the user as a possible attack location.

private ArrayList<MapTile> neighbours: ArrayList for all MapTiles that are connected in any direction to this MapTile.

private Label soldierLabel: Amount of soldiers on a specific MapTile are shown with Labels on the MapTiles.

private Polygon highLighter: Brightly colored Highlighter used to cover the perimeter of the chosen MapTile to let the user know that the Highlighted MapTile is the chosen one indeed.

Methods:

public void operation(): All operations to be handled that were done on the MapTile in the last turn. Updates every information of the MapTile.

InputManager Class

Attributes:

private UIManager uiManager: UIManager object of the game to be used to get user inputs.

private GameManager gameManager: GameManager object of the game to be used to get operations to be performed on the currently running game.

Methods:

public boolean moveUnits(int from, int to): Used to move the units from the given tile to the given tile. The tiles must be neighbours for this method to work.

public boolean recruitUnits(int amount, int loc): Recruits the specified amount of units on the MapTile that sits on the given location

public boolean showTileInfo(int id): Shows information about the MapTile that is on the given location.

public void endTurn(): Finalizes every user operation performed in the current turn and advances the game to the next one.

public void setMusicMute(boolean mute): Mutes or unmutes the music depending on the information taken from the user with the help of the MusicManager.

public void openSettings(): Opens the SettingsMenu.

public void closeSettings(): Closes the SettingsMenu.

public void startGame(): Starts the game by initializing every class.

public void returnMain(): Returns to the main menu.

public void openLore(): Opens the lore menu.

public void backMain(): Goes back to the main menu.

public void quit(): Quits the game.

Constructor:

public InputManager(UIManager uiMan, GameManager gmMan): InputManager has one constructor that takes two parameters which are the UIManager object for the user interface of the game and the GameManager object for the main game operation manager.

LoreMenu Class

Attributes:

private GridPane loreMenuPane: GridPane object to display the LoreMenu on.

private InputManager inputManager: InputManager to manage the user inputs given on the LoreMenu window.

Methods:

Constructor:

public LoreMenu(GridPane loreMenuPane, InputManager inputManager):

LoreMenu has one constructor that takes two parameters which are the GridPane object to display the LoreMenu on and the InputManager object to manage user input respectively.

MainMenu Class

Attributes:

private String username: The username chosen to be used by the current user.

private GridPane mainMenuPane: GridPane object for the MainMenu to be displayed on.

private InputManager inputManager: InputManager object to manage user input on the MainMenu window.

Methods:

Constructor:

public MainMenu(GridPane mainMenuPane, InputManager inputManager):

MainMenu has one constructor which takes two parameters that are the GridPane object for the MainMenu to be displayed on and the InputManager object to manage the user input respectively.

MainMenuManager Class

Attributes:

private String userName: The username chosen to be used by the current user.

private MainMenu menu: MainMenu object to be managed.

Methods:

public void setUsername(String userName): Sets the username of the user to the String value given by the user.

public String getUsername(): Returns the username chosen to be used by the current user.

Constructor:

public MainMenuManager(GridPane mainMenuPane, InputManager inputManager): MainMenuManager has one constructor which takes two parameters that are the GridPane object to display the MainMenu on and the InputManager object to manage the user input respectively.