Object Oriented Software Engineering Project

CS319 Project : Infinite Tale

Design Report

Erim Erdal

Halil İbrahim Azak

Can Özgürel

Mert Özerdem

Hüseyin Taşkesen

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

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.

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.

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.

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++.

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 lets 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.

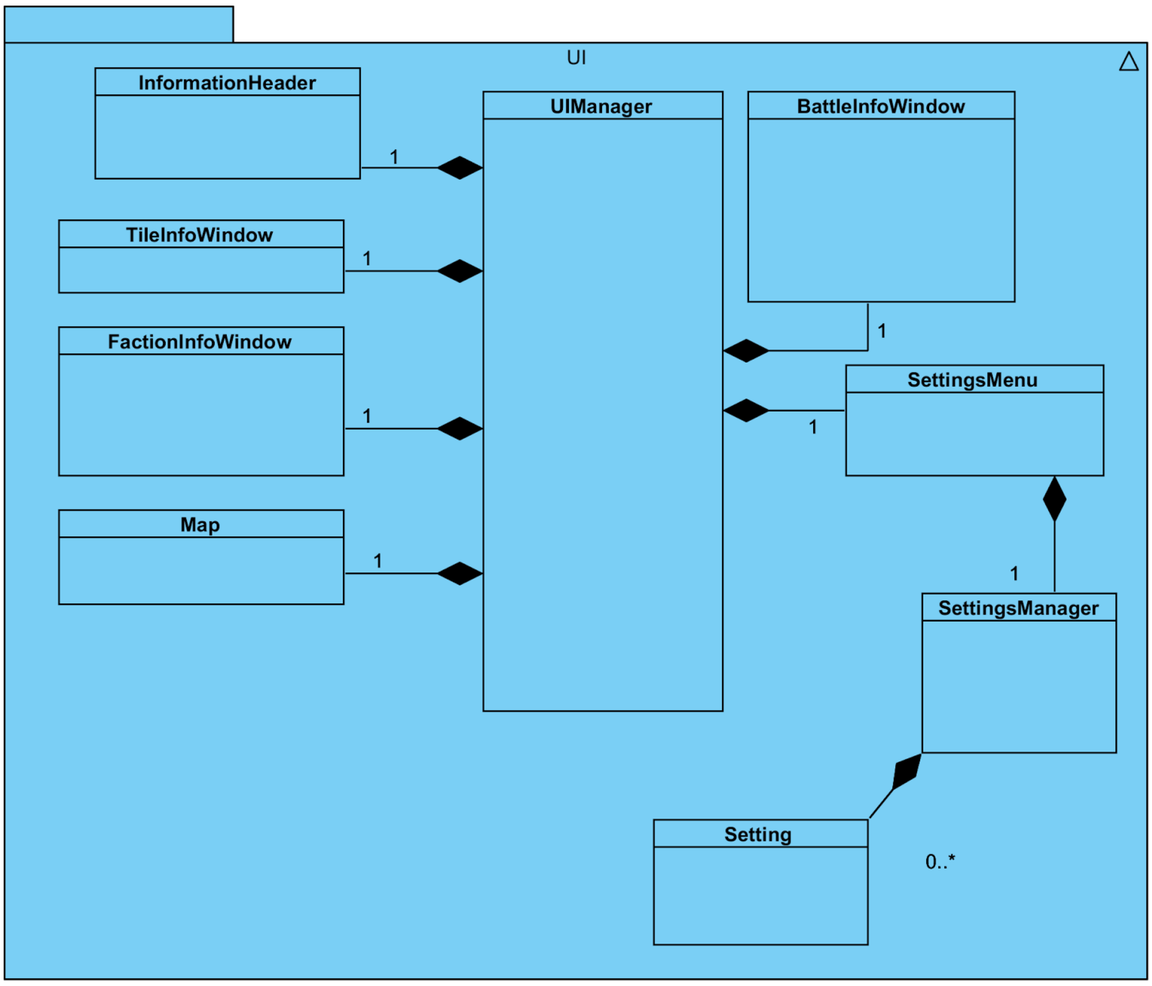


Figure - 1 (View Classes and UIManager)

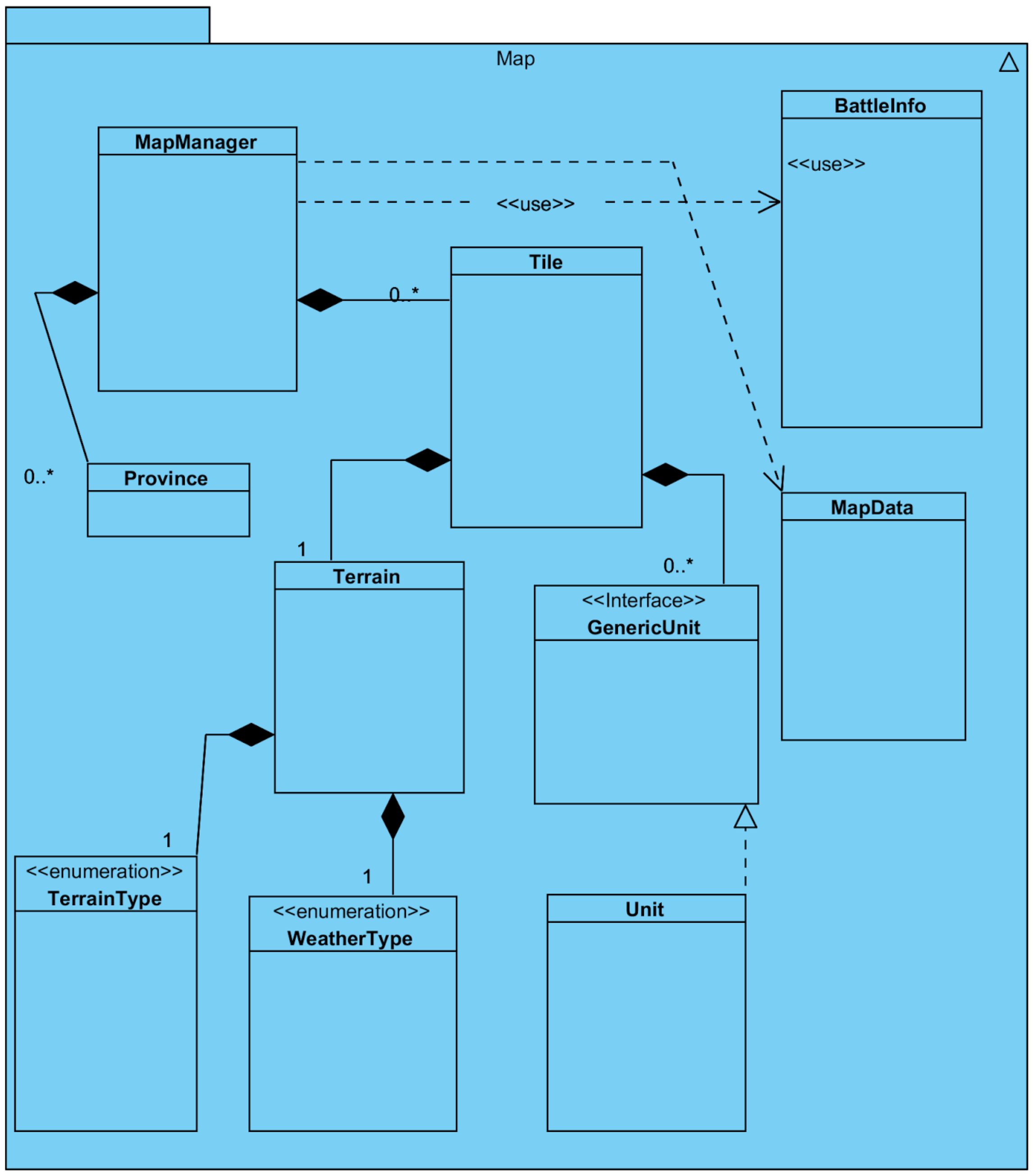


Figure - 2 (Model Classes and MapManager)

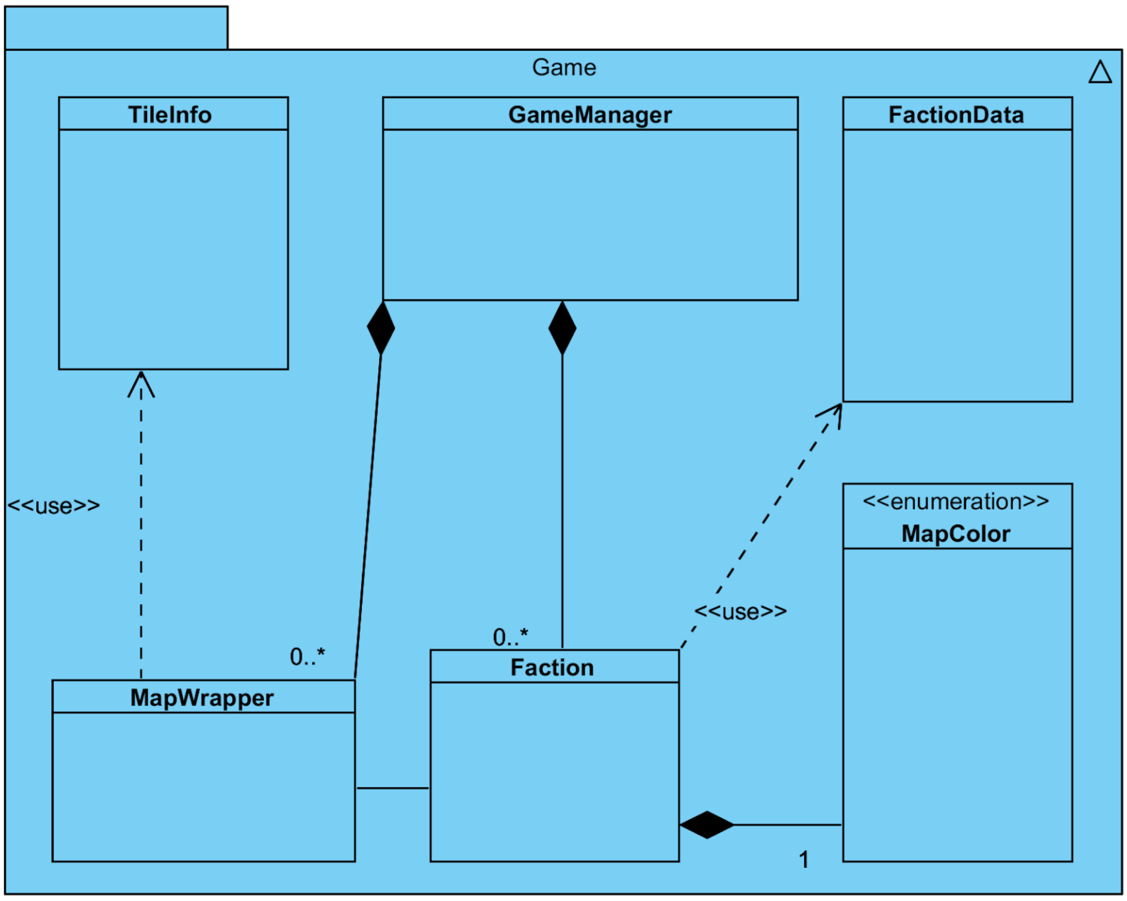


Figure - 3 (GameManager and its components)

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.

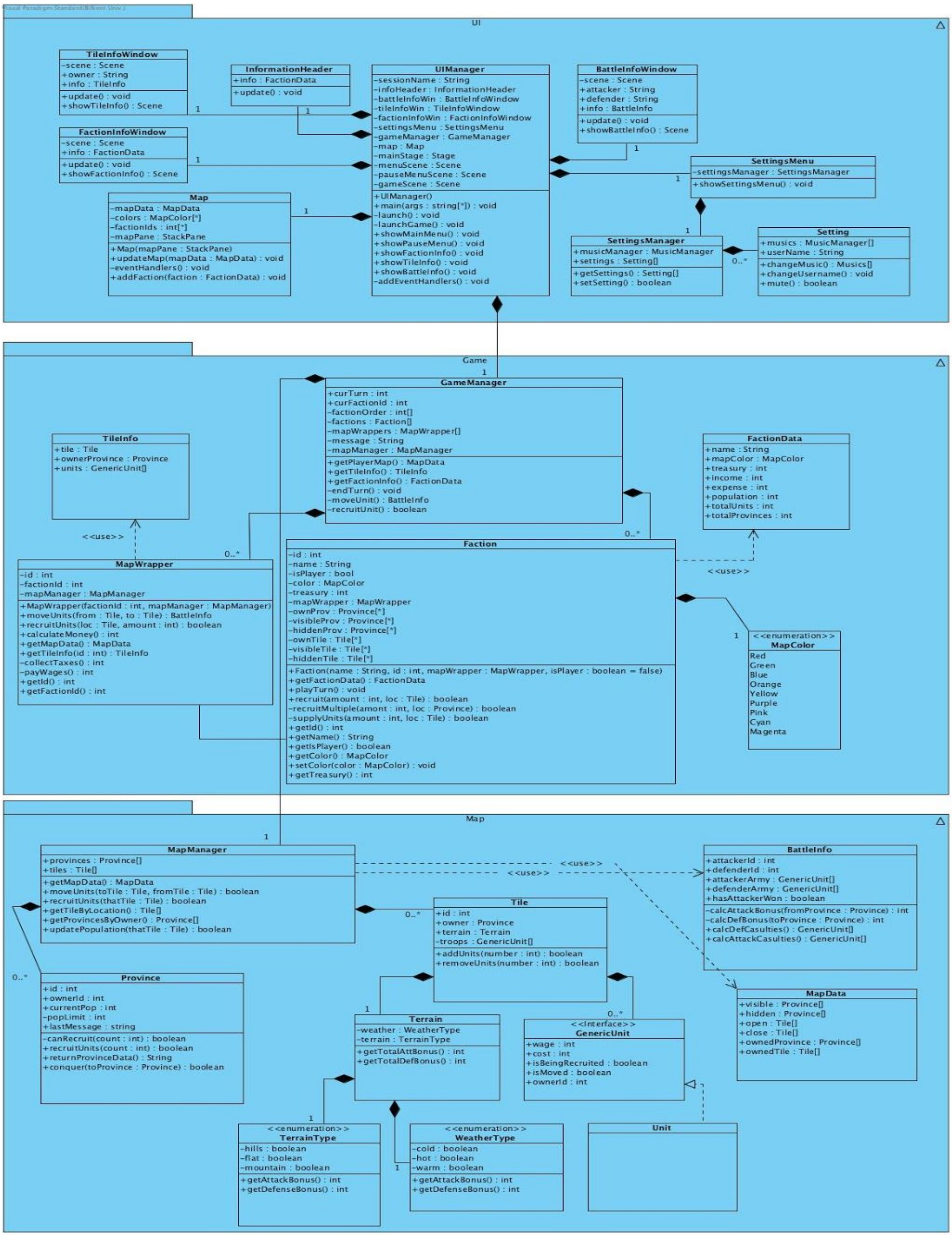
2.5 - Boundary Conditions

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.

3 - Subsystem Services

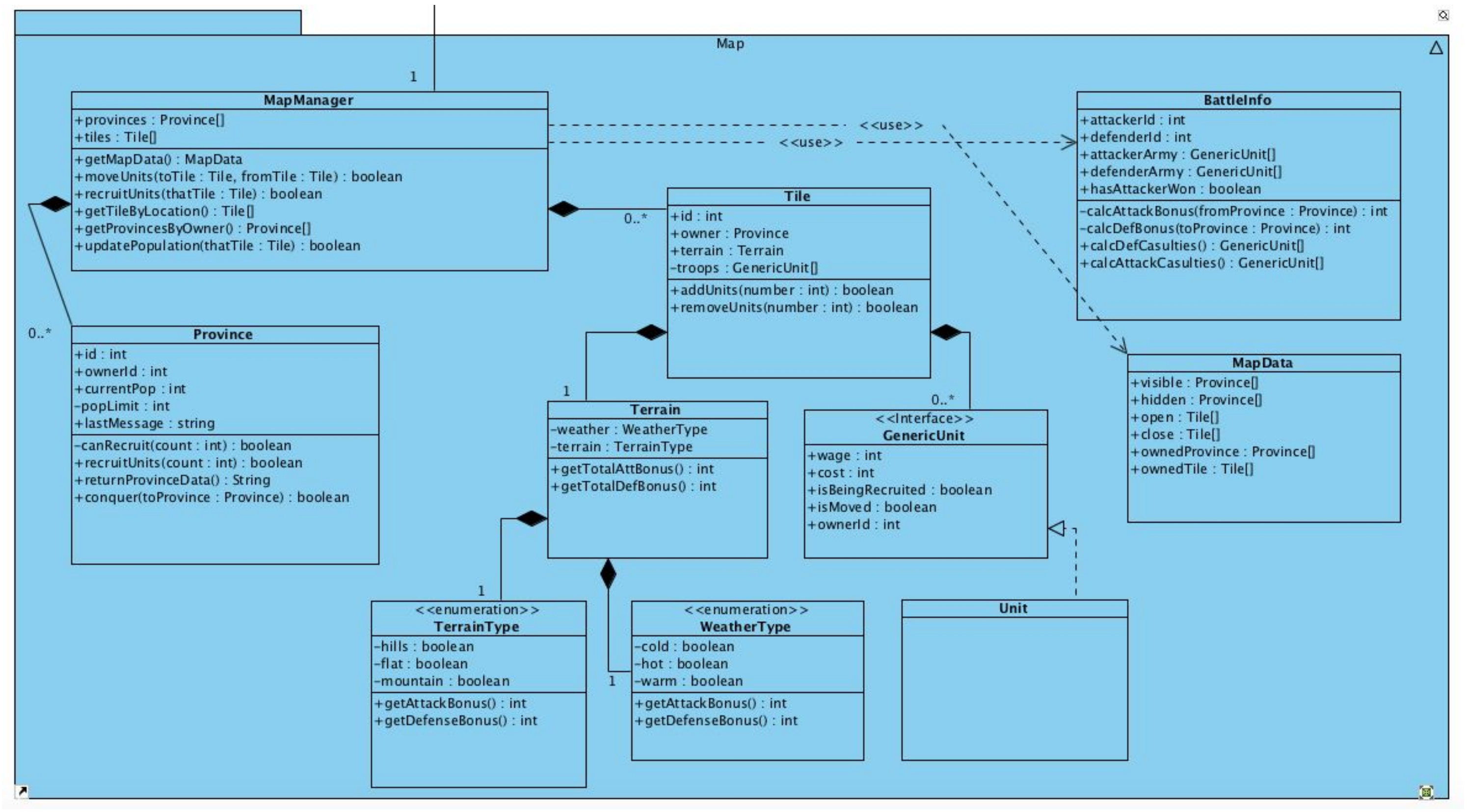
3.1 - Detailed Object Design

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.


Visual Paradigm Standard(Bilkent Univ.)
UI
TileInfoWindow
-scene : Scene
+owner : String
+info : TileInfo
+update() : void
+showTileInfo() : Scene
InformationHeader
+info : FactionData
+update() : void
UIManager
-sessionName : String
-infoHeader : InformationHeader
-battleInfoWin : BattleInfoWindow
-tileInfoWin : TileInfoWindow
-factionInfoWin : FactionInfoWindow
-settingsMenu : SettingsMenu
-gameManager : GameManager
-map : Map
-mainStage : Stage
-menuScene : Scene
-pauseMenuScene : Scene
-gameScene : Scene
+UIManager()
+main(args : string[*]) : void
-launch() : void
-launchGame() : void
+showMainMenu() : void
+showPauseMenu() : void
+showFactionInfo() : void
+showTileInfo() : void
+showBattleInfo() : void
-addEventHandlers() : void
BattleInfoWindow
-scene : Scene
+attacker : String
+defender : String
+info : BattleInfo
+update() : void
+showBattleInfo() : Scene
FactionInfoWindow
-scene : Scene
+info : FactionData
+update() : void
+showFactionInfo() : Scene
SettingsMenu
-settingsManager : SettingsManager
+showSettingsMenu() : void
Map
-mapData : MapData
-colors : MapColor[*]
-factionIds : int[*]
-mapPane : StackPane
+Map(mapPane : StackPane)
+updateMap(mapData : MapData) : void
-eventHandlers() : void
+addFaction(faction : FactionData) : void
SettingsManager
+musicManager : MusicManager
+settings : Setting[]
+getSettings() : Setting[]
+setSetting() : boolean
Setting
+musics : MusicManager[]
+userName : String
+changeMusic() : Musics[]
+changeUsername() : void
+mute() : boolean
1
1
1
1
1
1
1
0..*
Game
1
GameManager
+curTurn : int
+curFactionId : int
-factionOrder : int[]
-factions : Faction[]
-mapWrappers : MapWrapper[]
-message : String
-mapManager : MapManager
+getPlayerMap() : MapData
+getTileInfo() : TileInfo
+getFactionInfo() : FactionData
-endTurn() : void
-moveUnit() : BattleInfo
-recruitUnit() : boolean
TileInfo
+tile : Tile
+ownerProvince : Province
+units : GenericUnit[]
FactionData
+name : String
+mapColor : MapColor
+treasury : int
+income : int
+expense : int
+population : int
+totalUnits : int
+totalProvinces : int
<<use>>
0..*
0..*
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, isPlayer : boolean = false)
+getFactionData() : FactionData
+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
<<use>>
MapWrapper
-id : int
-factionId : int
-mapManager : MapManager
+MapWrapper(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
1
<<enumeration>>
MapColor
Red
Green
Blue
Orange
Yellow
Purple
Pink
Cyan
Magenta
Map
1
MapManager
+provinces : Province[]
+tiles : Tile[]
+getMapData() : MapData
+moveUnits(toTile : Tile, fromTile : Tile) : boolean
+recruitUnits(thatTile : Tile) : boolean
+getTileByLocation() : Tile[]
+getProvincesByOwner() : Province[]
+updatePopulation(thatTile : Tile) : boolean
<<use>>
<<use>>
BattleInfo
+attackerId : int
+defenderId : int
+attackerArmy : GenericUnit[]
+defenderArmy : GenericUnit[]
+hasAttackerWon : boolean
-calcAttackBonus(fromProvince : Province) : int
-calcDefBonus(toProvince : Province) : int
+calcDefCasulties() : GenericUnit[]
+calcAttackCasulties() : GenericUnit[]
0..*
Tile
+id : int
+owner : Province
+terrain : Terrain
-troops : GenericUnit[]
+addUnits(number : int) : boolean
+removeUnits(number : int) : boolean
0..*
Province
+id : int
+ownerId : int
+currentPop : int
-popLimit : int
+lastMessage : string
-canRecruit(count : int) : boolean
+recruitUnits(count : int) : boolean
+returnProvinceData() : String
+conquer(toProvince : Province) : boolean
1
Terrain
-weather : WeatherType
-terrain : TerrainType
+getTotalAttBonus() : int
+getTotalDefBonus() : int
0..*
<<Interface>>
GenericUnit
+wage : int
+cost : int
+isBeingRecruited : boolean
+isMoved : boolean
+ownerId : int
MapData
+visible : Province[]
+hidden : Province[]
+open : Tile[]
+close : Tile[]
+ownedProvince : Province[]
+ownedTile : Tile[]
1
<<enumeration>>
TerrainType
-hills : boolean
-flat : boolean
-mountain : boolean
+getAttackBonus() : int
+getDefenseBonus() : int
<<enumeration>>
WeatherType
-cold : boolean
-hot : boolean
-warm : boolean
+getAttackBonus() : int
+getDefenseBonus() : int
1
Unit

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.

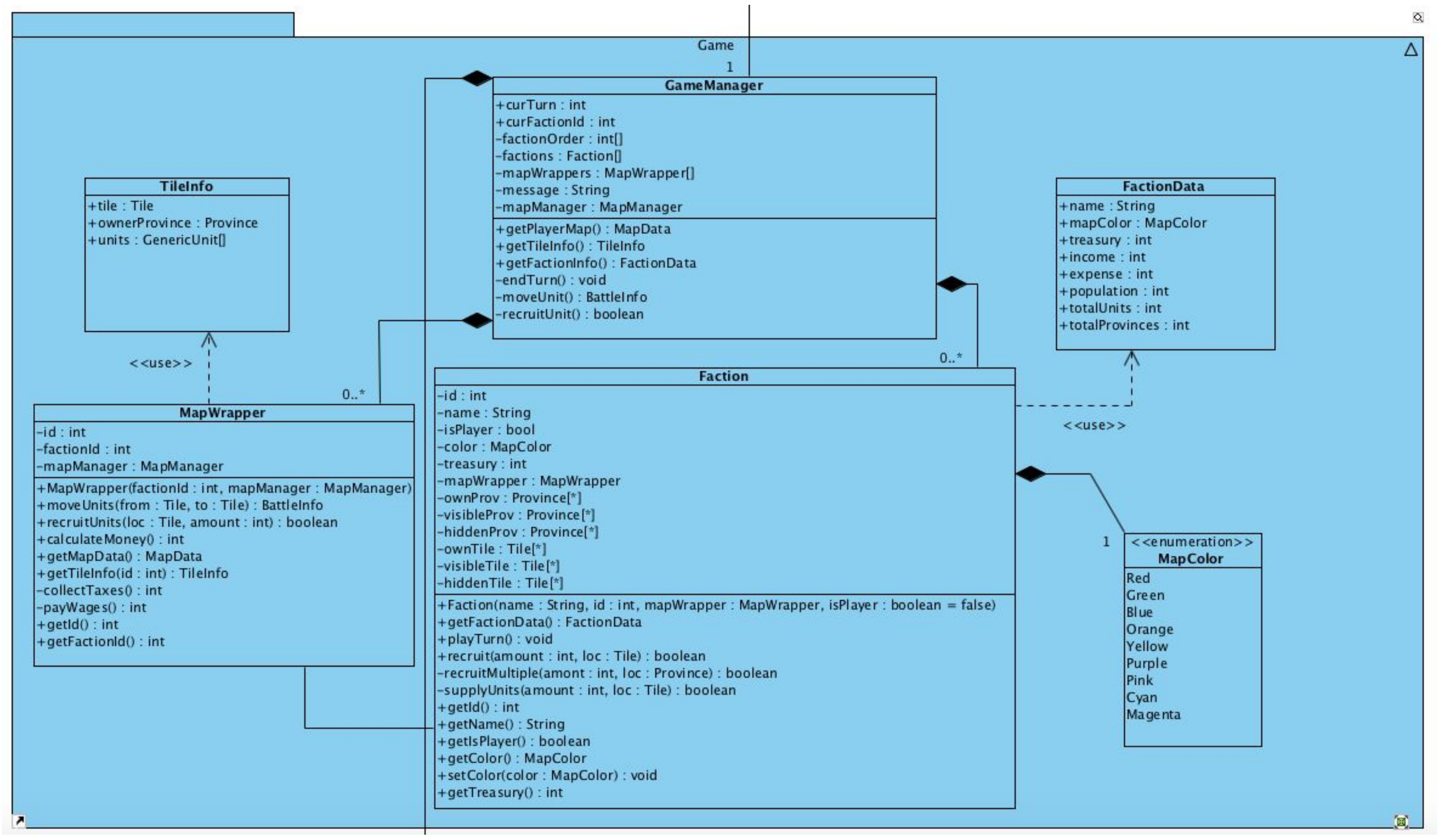


3.3 - View (UI) Subsystem

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.

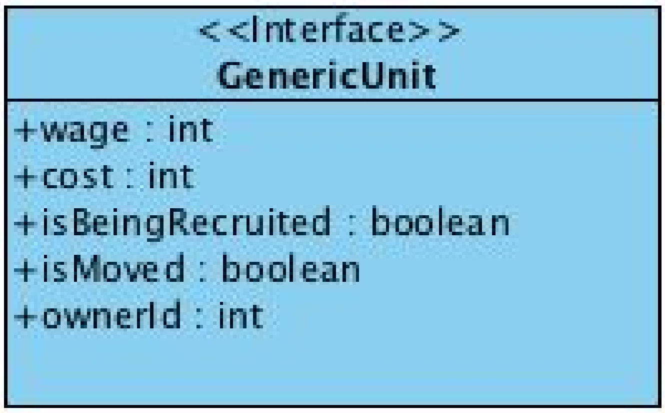
3.4 - Controller (Game Logic) Subsystem

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.

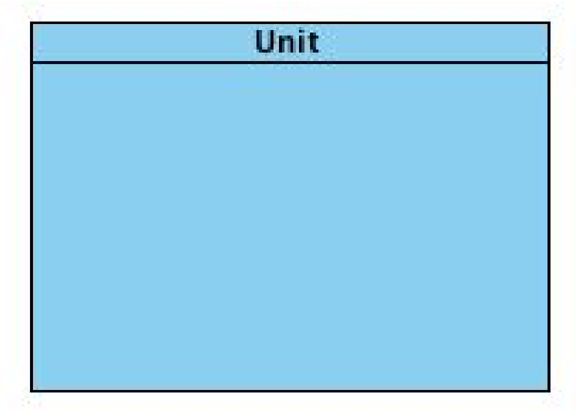
public boolean isBeingRecruited: Is a boolean value which denotes to the process of recruiting troops which takes time. If troops are in training phase, this value will be true.

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.

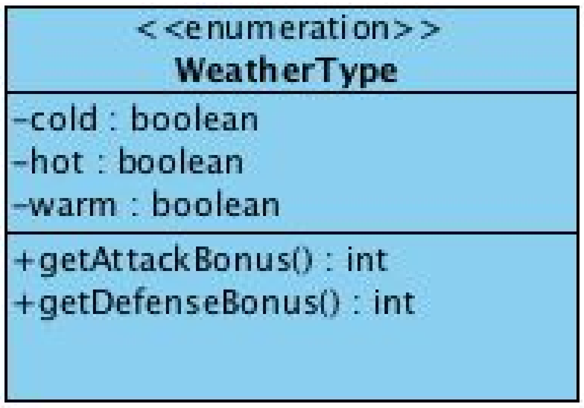
Reason why there is an interface like this will be clear in “Unit” Class’s description. A “Tile” Class can have zero or more “GenericUnit” but in a specific time a “GenericUnit” can only belong to one “Tile”. Also attributes are public because tiles will have to use these attributes for calculations.

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.

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.

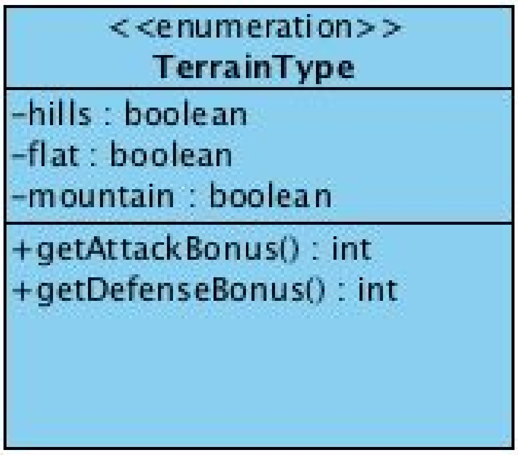
Methods:

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.

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.

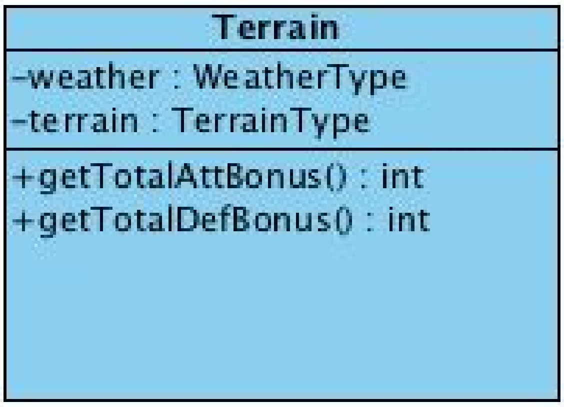
Methods:

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.

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.

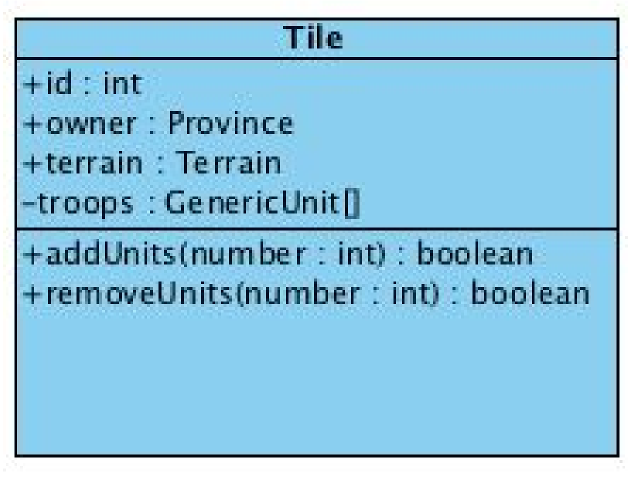
private TerrainType terrain: Enumerated terrain will give us the bonuses from terrain.

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.

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.

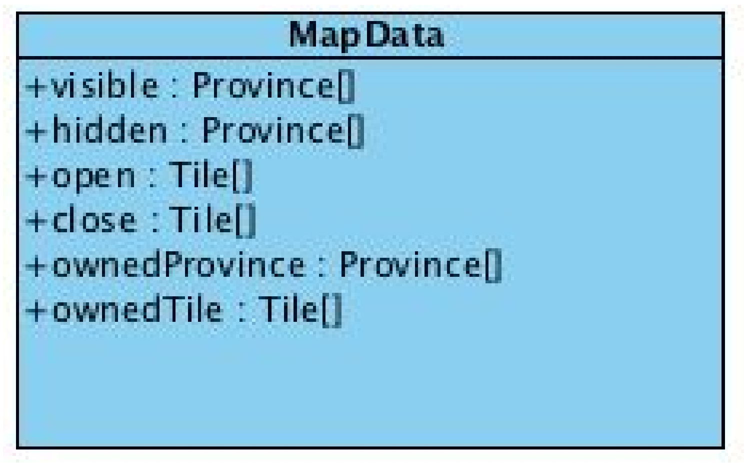
private GenericUnit[] troops: This will denote to the number of troops on the tile. It is private because the methods will handle changing the number of troops after a war or a migration. Hence there is no need for this attribute to be modifiable.

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.

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 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 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.

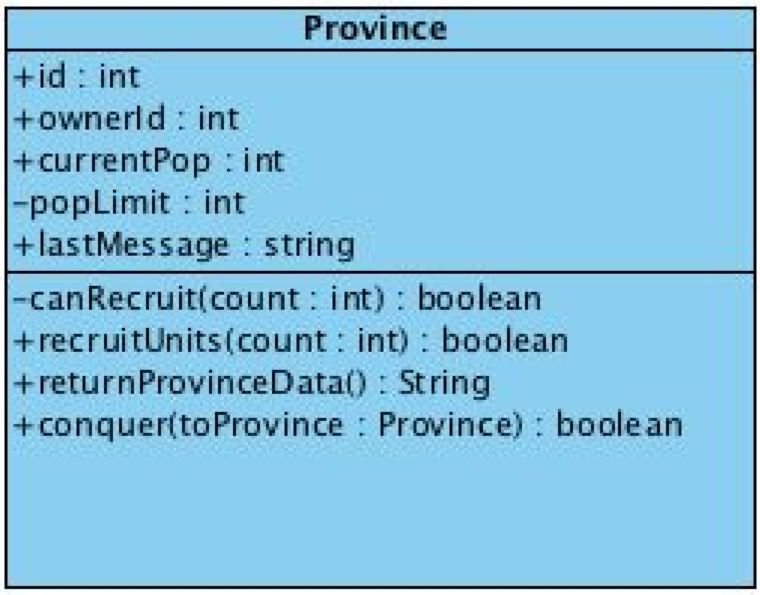
public Tile[] open: Stores an array of open tiles. This means that player is able to attack to that tile since no fog of war closes it

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

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

Province Class


Province
+id : int
+ownerId : int
+currentPop : int
-popLimit : int
+lastMessage : string
-canRecruit(count : int) : boolean
+recruitUnits(count : int) : boolean
+returnProvinceData() : String
+conquer(toProvince : Province) : boolean

“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.

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

Methods:

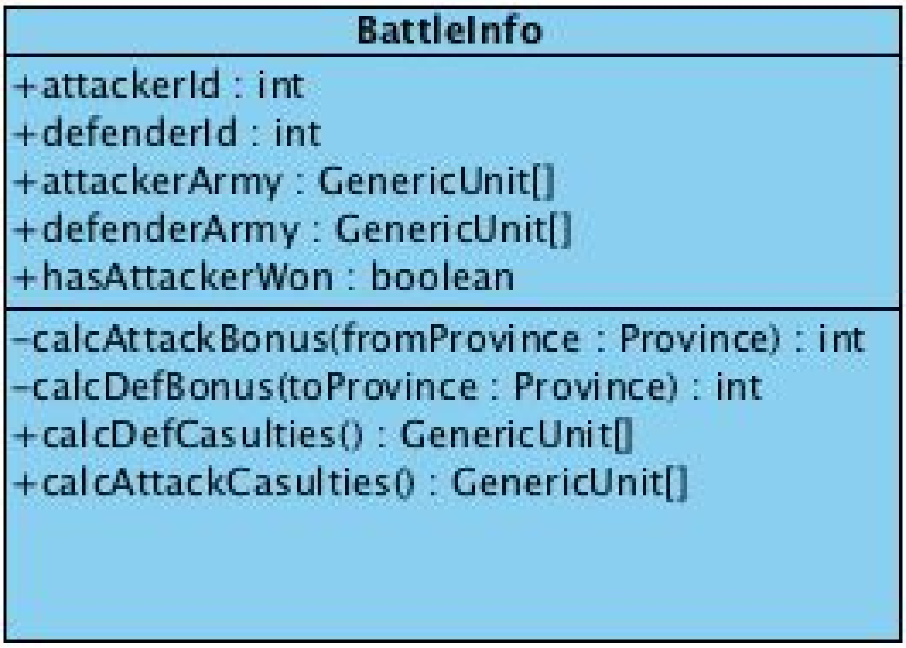
private boolean canRecruit(int count): This method checks if the owner of province can recruit the amount of units they want, returns true if it is possible. This method is a helper method of recruitUnits method, since recruitUnits will have to check if it is possible to recruit without reaching population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

BattleInfo Class


BattleInfo
+attackerId : int
+defenderId : int
+attackerArmy : GenericUnit[]
+defenderArmy : GenericUnit[]
+hasAttackerWon : boolean
-calcAttackBonus(fromProvince : Province) : int
-calcDefBonus(toProvince : Province) : int
+calcDefCasulties() : GenericUnit[]
+calcAttackCasulties() : GenericUnit[]

"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 GenericUnit[] 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.

public GenericUnit[] 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.

Methods:

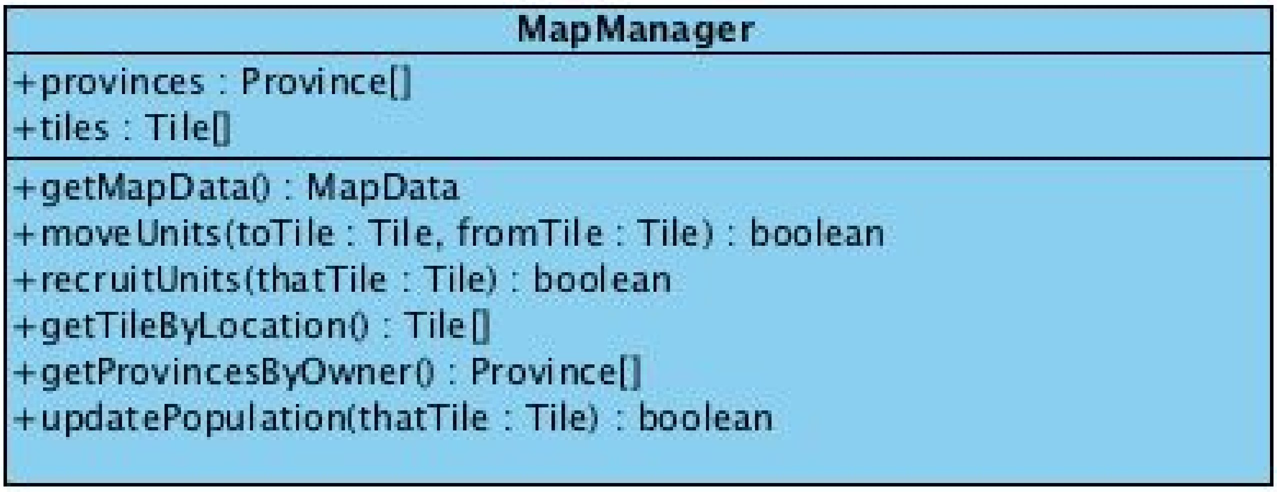
private int calcAttackBonus(Province fromProvince): This method calculates attack bonus of the attacker, it requires a province parameter because it will access to traits of a province, therefore a tile which will be the terrain and weather bonuses. Since troops do not have advantages over each other for now, only bonuses will be terrain bonuses.

private int calcDefBonus(Province toProvince): Same working principle with the method above but only for defender's point of view. They are private because these methods are helper methods, they will help calculating casualties and they return integer because it will represent a percentage value when calculating the casualties in calcDefCasulties and calcAttackCasulties.

public GenericUnit[] calcDefCasulties(): Calculates casualties of defender side. Has rather complex mathematical calculations inside which we determined would be a good simulation of the war with our team.

public GenericUnit[] calcAttackCasulties(): Calculates casualties for attacker side. This and above methods returns GenericUnit because in further iterations there might be different count of losses for different kinds of troops, which this method will also calculate. If attacker casualties reach the number of troops sent to invade, then the attacker will be considered to have lost the war. Also if defender casualties reach the number of troops waiting on the province, then attacker will have conquered the land to explain simply.

MapManager Class



```
MapManager
+provinces : Province[]
+tiles : Tile[]
+getMapData() : MapData
+moveUnits(toTile : Tile, fromTile : Tile) : boolean
+recruitUnits(thatTile : Tile) : boolean
+getTileByLocation() : Tile[]
+getProvincesByOwner() : Province[]
+updatePopulation(thatTile : Tile) : boolean
```

“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.

Methods:

public MapData getMapData(): Returns data of map in MapData object form in order to provide information and connection with other classes.

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.

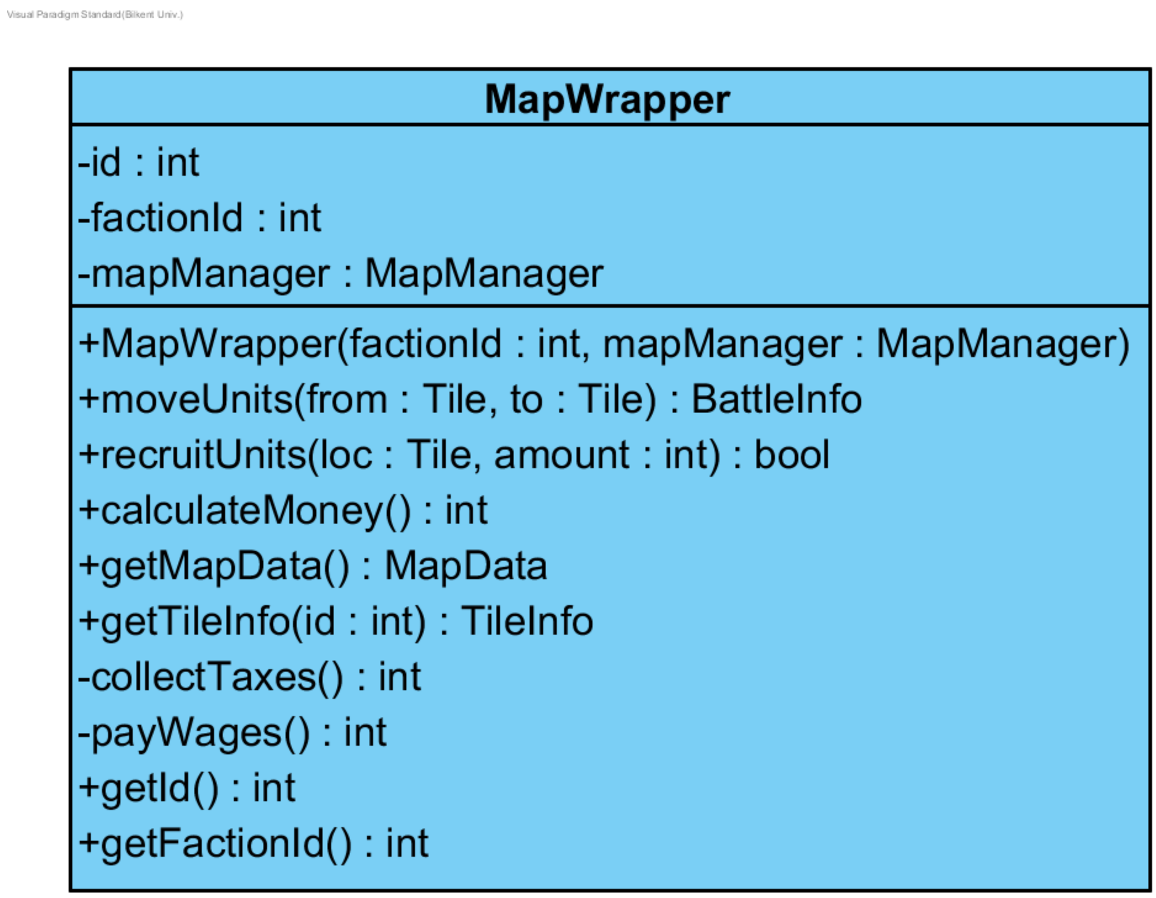
public boolean recruitUnits(Tile thatTile): This function takes a Tile parameter and tries to recruit some troops. Returns true if troops are successfully recruited, false if it is denied because of population limits. Works with using the Tile's addUnit function.

public Tile[] getTileByLocation(): Returns all Tile's.

public Province[] getProvincesByOwner(): 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.

MapWrapper Classes


Visual Paradigm Standard(Bilkent Univ.)
MapWrapper
-id : int
-factionId : int
-mapManager : MapManager
+MapWrapper(factionId : int, mapManager : MapManager)
+moveUnits(from : Tile, to : Tile) : BattleInfo
+recruitUnits(loc : Tile, amount : int) : bool
+calculateMoney() : int
+getMapData() : MapData
+getTileInfo(id : int) : TileInfo
-collectTaxes() : int
-payWages() : int
+getId() : int
+getFactionId() : int

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.

Constructors:

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.

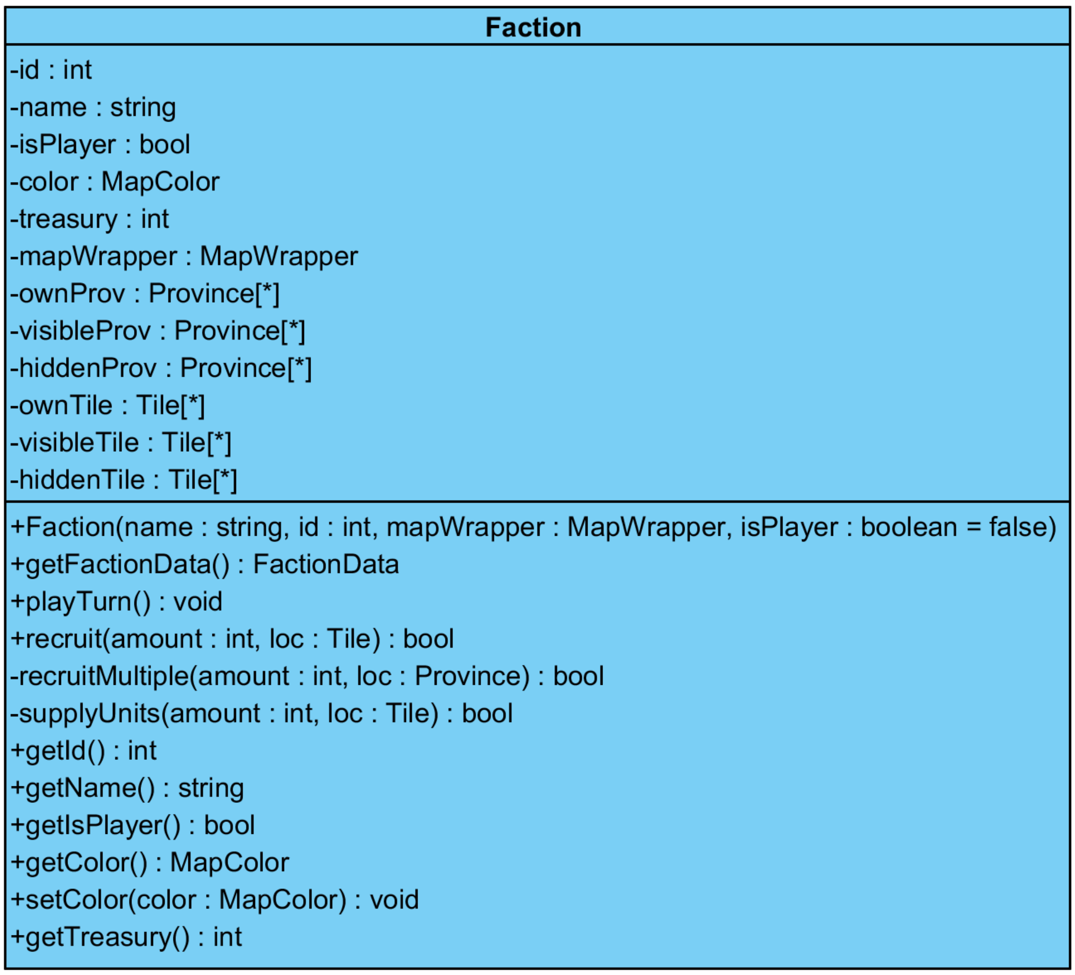
private int collectTaxes(): Calculates and returns total amount of taxes.

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

-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, isPlayer : boolean = false)
+getFactionData() : FactionData
+playTurn() : void
+recruit(amount : int, loc : Tile) : bool
-recruitMultiple(amount : int, loc : Province) : bool
-supplyUnits(amount : int, loc : Tile) : bool
+getId() : int
+getName() : string
+getIsPlayer() : bool
+getColor() : MapColor
+setColor(color : MapColor) : void
+getTreasury() : int

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.

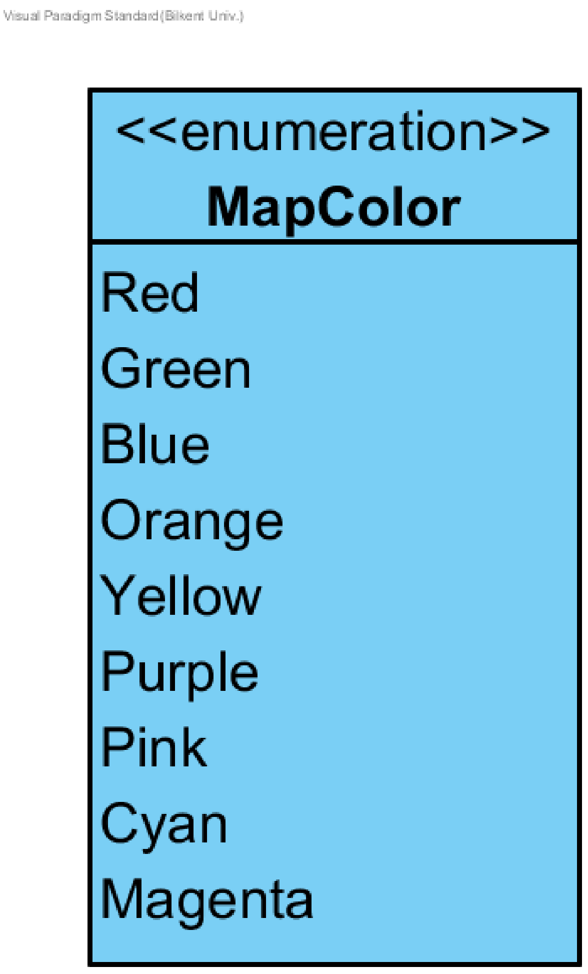
public bool getIsPlayer(): Getter function for the isPlayer attribute.

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).

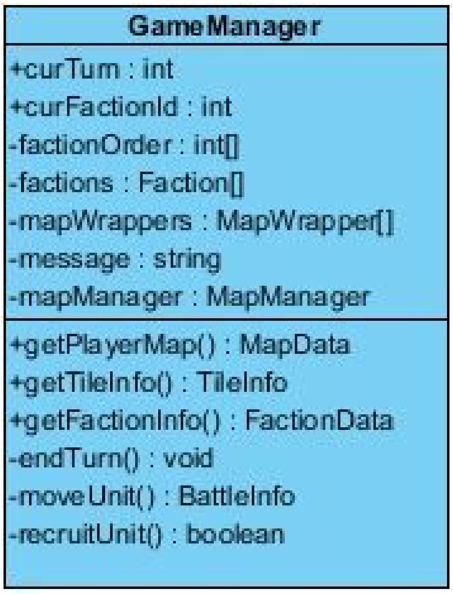
public int getTreasury(): Getter function for the treasury attribute.

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 curFactionId: The faction that has to play the current turn.

private int[] factionOrder: Turn order of the factions in the game.

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

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

private string message: Message to be shown on the console.

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.

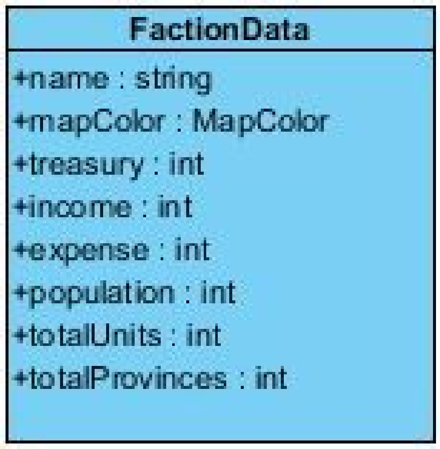
public FactionData getFactionInfo(): Returns information about the faction that has the current turn.

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.

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.

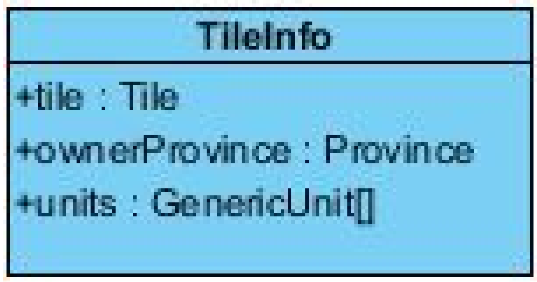
public int expense: The outcome of the faction which is the amount of resources that the faction has to pay at the end of every turn.

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.

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.

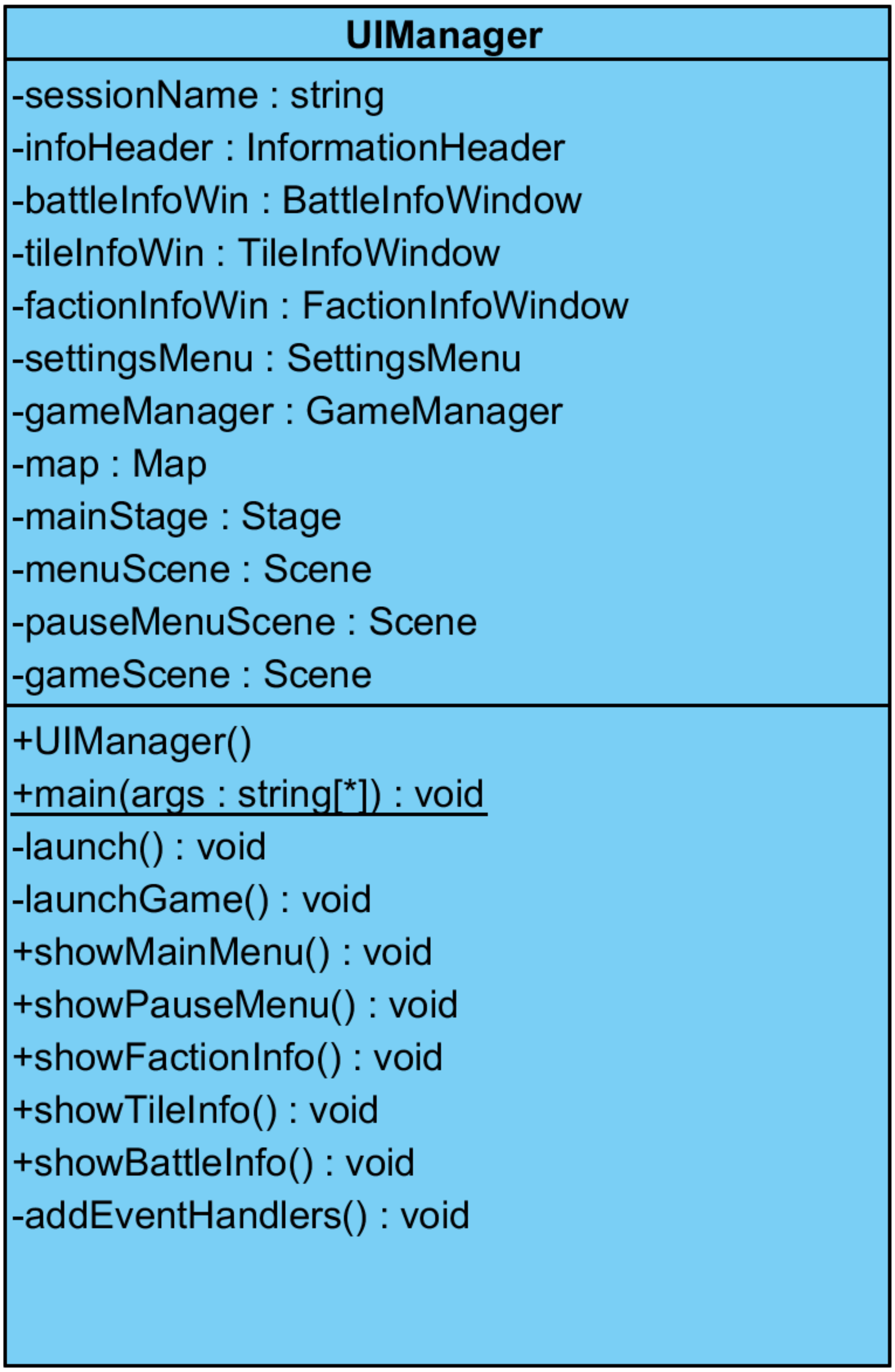
Attributes:

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

public Province ownerProvince: The province which the tile is currently inside of.

public GenericUnit[] units: All units that are currently on the chosen tile.

UIManager Class


UIManager
-sessionName : string
-infoHeader : InformationHeader
-battleInfoWin : BattleInfoWindow
-tileInfoWin : TileInfoWindow
-factionInfoWin : FactionInfoWindow
-settingsMenu : SettingsMenu
-gameManager : GameManager
-map : Map
-mainStage : Stage
-menuScene : Scene
-pauseMenuScene : Scene
-gameScene : Scene
+UIManager()
+main(args : string[*]) : void
-launch() : void
-launchGame() : void
+showMainMenu() : void
+showPauseMenu() : void
+showFactionInfo() : void
+showTileInfo() : void
+showBattleInfo() : void
-addEventHandlers() : void

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 String sessionName: Name of the current session. Holds the name of the save if the game is loaded from a save game.

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

private BattleInfoWindow battleInfoWin: Holds a reference to the object that controls battle information window.

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

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

private SettingsMenu settingsMenu: Holds a reference to the object that controls settings menu.

private GameManager gameManager: Holds a reference to the GameManager object of the game.

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

private Stage mainStage: Main stage used by the game.

private Scene menuScene: Scene that main menu will be shown.

private Scene pauseMenuScene: Scene that pause menu will be shown.

private Scene gameScene: Main scene game will be shown.

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.

private void launch(): Launches the game for first time and shows main menu.

private void launchGame(): Launches game. At first creates all necessary game and map objects. After that creates UI objects and updates with necessary information.

public void showMainMenu(): Shows the scene that has main menu.

public void showPauseMenu(): Shows the scene that has pause menu.

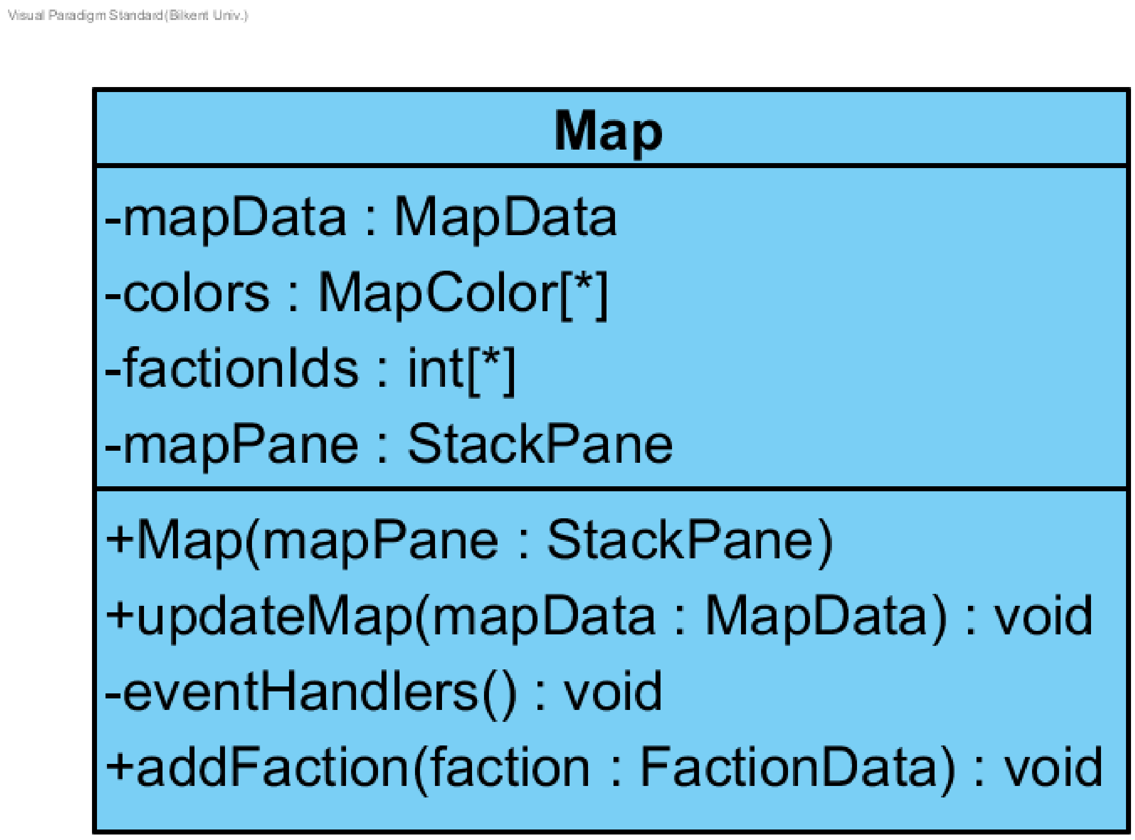
public void showFactionInfo(): Shows the faction info window.

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

public void showBattleInfo(): Shows the battle info window.

private void addEventHandlers(): Adds required event handlers to the UI elements.

Map Class


Visual Paradigm Standard (Bilkent Univ.)
Map
-mapData : MapData
-colors : MapColor[*]
-factionIds : int[*]
-mapPane : StackPane
+Map(mapPane : StackPane)
+updateMap(mapData : MapData) : void
-eventHandlers() : void
+addFaction(faction : FactionData) : void

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 int[] factionIds: Holds the ids of the factions. Faction ids and colors are synchronized.

private StackPane mapPane: Holds the pane object that the player map will be drawn on.

Constructors:

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

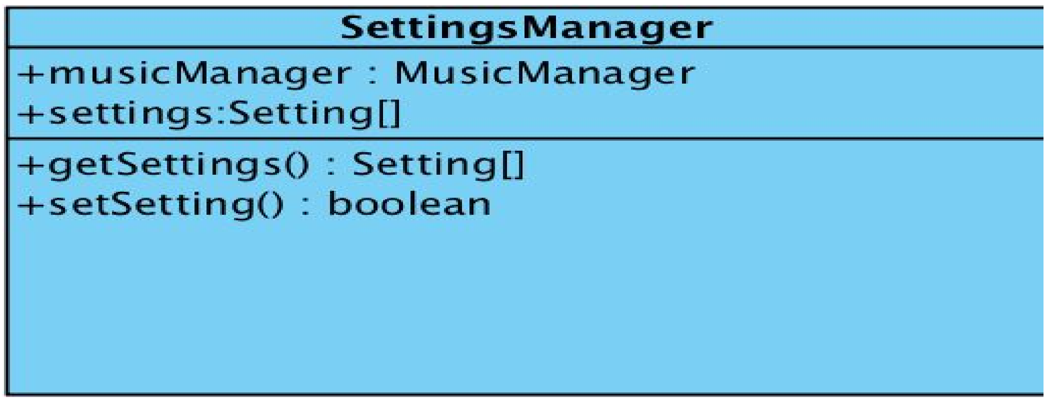
Methods:

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

private void eventHandlers(): Adds required event handlers to UI elements.

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

SettingsManager Classes



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

Attributes:

private Setting[] settings: Array that holds the setting information.

private MusicManager musicManager: Object from "MusicManager" class to play the music.

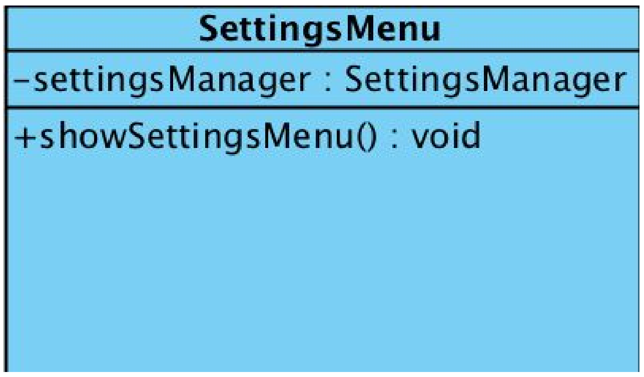
Methods:

private Setting[] getSettings(): Return the settings information in an array.

private bool setSetting(): Is a boolean value that returns true if the settings are

set, false otherwise.

Settings Menu


SettingsMenu
-settingsManager : SettingsManager
+showSettingsMenu() : void

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

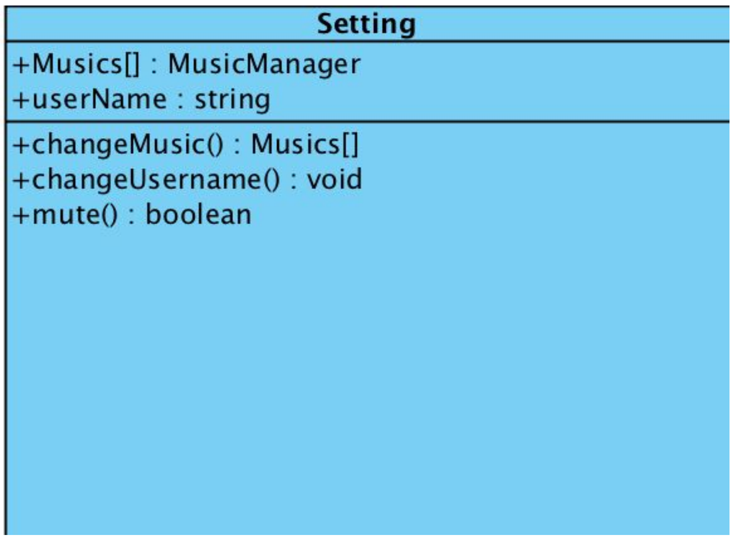
Attributes:

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

Method:

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

Setting



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.

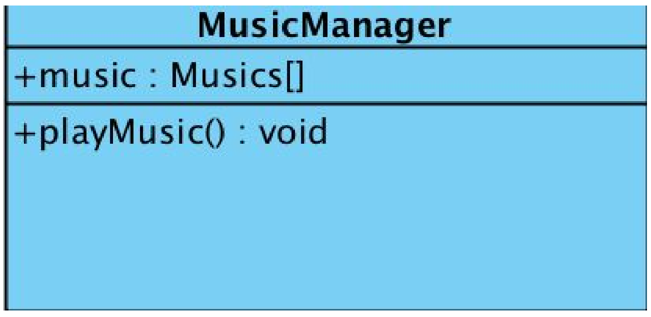
Methods:

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.

Music Manager Class



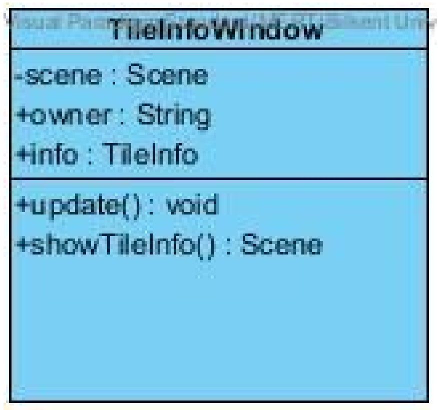
Attributes:

public Musics[] music: Array that holds the musics that are ready to play. Music is a variable that user wants to play.

Methods:

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

TileInfoWindow Class



Attributes:

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

public String owner: Tile's Owner's name.

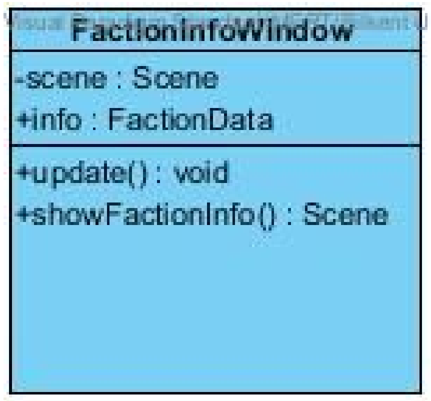
public TileInfo info: TileInfo object to display in TileInfoWindow.

Methods:

public JFrame showTileInfo(): Scene object is returned to be displayed on TileInfoWindow.

public void update(): repaints TileInfoWindow scene.

FactionInfoWindow Class



Attributes:

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

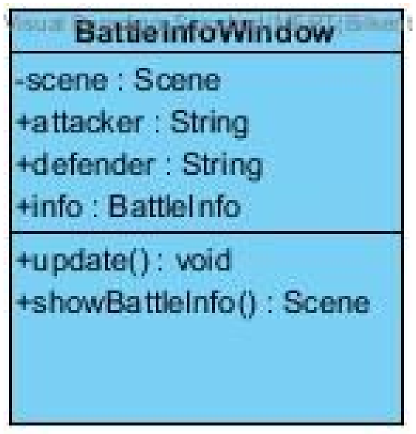
public FactionData info: FactionData object to display in TileInfoWindow.

Methods:

public Scene showFactionData(): JFrame object is returned to be displayed on FactionInfoWindow.

public void update(): repaints FactionInfoWindow scene.

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.

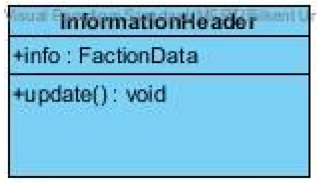
public BattleInfo info: BattleInfo object to display in BattleInfoWindow.

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.