****

**Object Oriented Software Engineering Project**

**Design Report**

**CS 319 Project: RISK: LOTR**

**Group 1J**

* **Miraç Vuslat Başaran**
* **Hazal Buruk**
* **Elena Çina**
* **Doğa Zeynep Germen**

**Intructor: Bora Güngören**

**Table of Contents**

1. **Introduction**
   1. **Purpose of The System**
   2. **Design Goals**
2. **Software Architecture**

**2.1 Subsystem Decomposition**

**2.2 Hardware/Software Mapping**

**2.3 Persistent Data Management**

**2.4 Access Control and Security**

**2.5 Boundary Conditions**

1. **Subsystem Services**

**1. Introduction**

**1.1 Purpose of the System**

Risk-LOTR is a desktop based game which we are designing with the primary aim to entertain people who are willing to play it. We were inspired by the traditional Risk table game, but we have adopted it and added some features from the famous Lord of Ring movies, hence the fans of these movies will enjoy playing this game even more. Additionally, developing this software system will help us understand and practice the object oriented programming concepts, gain experience on developing a real software system and improve our programming and team working skills.

**1.2 Design Goals**

* **Usability**

One of the main goals of our design is to develop a user friendly game. Menu will help users to access all the features of the game and through the help option user will be able to deliver information on tactics and logic of the game.

* **Performance**

*Response Time*

Risk-LOTR is an interactive game, hence we will be sure that the response time will not exceed a certain small threshold.

* **Well defined interfaces**

We aim to develop a game that will have well defined interface. All characters of the game such as different kind of units and factions. We will also provide animation for the execution of each phase of the game such as deployment, attacking and battle execution.

* **Extensibility**

We aim to build a system which can be updated without causing complication to the current system. It will be extensible in terms of its content, mechanics, interface and graphics. So, in the future we can enhance the system by adding additional new features.

* **Reliability**

Our goal is to build a reliable system which will not crush or give any run time errors, hence preventing players from any unpleasant experience.

* **Good Documentation**

We aim to well document all the work that we will do while developing this game.

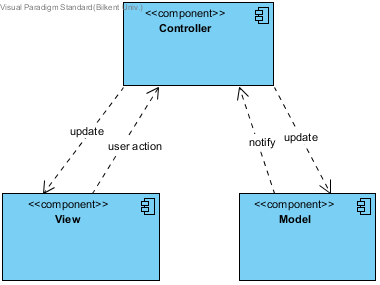
**2. Software Architecture**

This section includes a subsystem decomposition of our software project to make it understandable and easy to implement. Also, which hardware/software tools are needed and the database management is explained here. Finally, access control and security issues as well as boundary conditions are examined in their respective subsections.

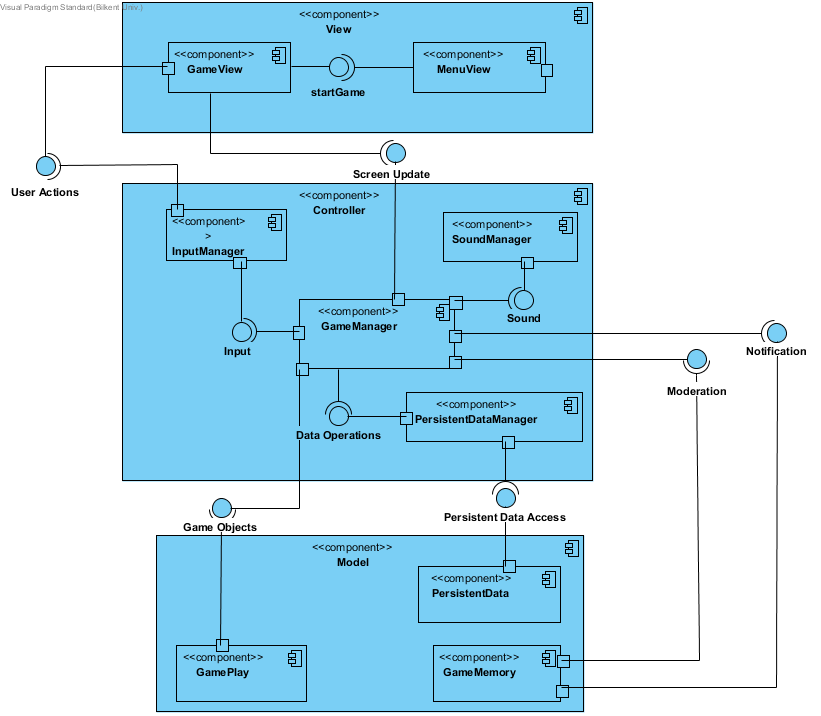
**2.1 Subsystem Decomposition**

RISK: LOTR, being a graphical strategy game, has a couple of interfaces to display the status of the game and to take user input as well as a complex logic that runs behind the scenes. To capitalize on this interface-logic separation, the game is designed on MVC (Model-View-Controller) pattern. This will help minimize coupling between main subsystems and maximize cohesion within those subsystems.

The “View” subsystem deals with handling user interfaces and getting user input. “Model” subsystem handles the data, rules and the logic of the application. Finally, “Controller” subsystem gets input from the other subsystems and updates them.



Below, the MVC architecture is expanded to show the subsystems of these three components and their interactions.



**2.2 Hardware / Software Mapping**

**2.2.1 General**

This game will be implemented in Java programming language using the latest JDK (8u121). Because our program doesn’t include high graphics and intense physical operations while it’s running, most of the computers will be able to run it. All of the graphical and physical calculations will be implemented by using Java libraries. However, most of the graphics will be uploaded to the game after they designed with other graphic design programs. Because of excessing graphical content and excessing the short term memory may require a time, the operations may require a short time but we expect them to be fast enough so that the player cannot understand the require time.

Because every new game requires new conditions and different players as enemies, we won’t collect the previous scores in anywhere and because the game only requires a little amount of graphics, we don’t plan to use any database system in our game. Because the current states of the game will only requires a little amount of memory, it’s unnecessary to create a database system.

**2.2.2 Input / Output System**

User will need mouse clicks to select its action, finish the phases. However, the keyboard will be used only when the player enters her name or enters the number of the soldiers that she wants to buy or when she wants to escape the game. Otherwise, mostly the mouse will be used. The output will be monitored by monitor and the sound effects and music will come out via speakers. The only required hardware are them. There is no additional hardware system.

**2.2.3 Memory**

The PC memory will be enough to meet the requirements of the game.

**2.2.4 Processor**

The visual subsystems will only require GUI and some pictures designed with the help of photoshop, it will not require heavy computational power. So that, there will be no rendering process while running. Game engine will only make some basic calculations to pick which graphic or picture will be monitored in that specific time. It will check the new condition in every change and change the current output. So that, the PCs with single processors will do the required operations easily.

**2.3 Persistent Data Management**

The only required persistent data management will be the game base data like help script, some pictures to be monitored and they will be kept in documents or in .txt files. There will be nothing to keep persistent after playing the game because this game will not have a save button or high scores table.

**2.4 Access Control and Security**

Our game will not include an authentication system. Each player will have the same rights to play the game. Namely, no player has special rights. Each player will have equal access to the game. There will be no need for an authentication system, as there is no difference in restrictions for players.

**2.5 Boundary Conditions**

**Initialization**

CASE: RISK: LOTR will be initialized if user launches the program via .exe file.

**Termination**

CASE 1: RISK: LOTR can be terminated if user clicks Quit Game.

CASE 2: RISK: LOTR can be terminated if user presses ESC key on the keyboard while running the game on Windows or on other OSs.

**Failure**

CASE: If there appears a system failure or a hardware failure (.txt file not found, .txt file corrupted etc.), there is no way to prevent the crash of the program; in this case, user will lose all progress since our game does not have a progress-saving feature, in other words, user will not be able to continue the game from the point game crashed

**3. SUBSYSTEM SERVICES**

**3.1 View Subsystem**

View subsystem is the subsystem which conducts the graphical interfaces of the RISK-LOTR game. With the help of the other subsystems, view subsystem manages the user interfaces.

**3.1.1 Menu View**

Menu view is the first screen and the first activated subsystem of the RISK- LOTR game. It starts with showing the user a few buttons to choose what actions to do. After taking inputs from the user and send the inputs to the Input Manager, according to the posts of the Input Manager, Game Manager chooses which action to do which are placed in Controller Subsystem. According to the choice of the user and feedback of the Game Manager, Menu View subsystem leads to proper part of the Controller subsystem and program executes the requests of the user.

**3.1.2 Game View**

This subsystem is the responsible of most of the user interfaces which appears in the RISK-LOTR. According to the Input Manager's posts to Game Manager subsystem, Game Manager subsystem chooses the proper interface and send it to the Game View. Thus, Game View subsystem is changing by Game Manager and provides the user with the proper graphical interface.

**3.2 Controller Subsystem**

This subsystem is a subsystem which manages and conducts the RISK- LOTR game's working logic and also handles the logical background of the auxiliary subsystems by taking inputs from user, processing it and sending the feedback to the proper subsystems.

**3.2.1 Input Manager**

Input Manager is the subsystem which is responsible of taking inputs from the user and sending it to the Game Manager subsystem to process the input. After Game Manager processes the information which comes from the Input Manager, it executes the proper actions and creates links between subsystems.

**3.2.2 Sound Manager**

Sound Manager subsystem is the responsible subsystem to manage the sound effects of the RISK-LOTR game. It allows the user change the sound levels by using proper interface. According to Game Managers references, Sound Manager subsystem is activated.

**3.2.3 Game Manager**

Game Manager subsystem is the most important subsystem of the RISK-LOTR game. This subsystem handles all the logical background of the program. It refers proper subsystems, calls proper functions and decides all the proper actions which game logic requires. Also it conducts the game by processing inputs which comes from the user and executing the actions by linking the other proper subsystems. We can consider this subsystem as the brain of the RISK-LOTR game. Thus, this subsystem can access and activate all the other subsystems directly or indirectly.

**3.2.4 Persistent Data Manager**

Persistent Data Manager is the subsystem which is activated after Game Manager calls it. After it is activated, it accesses the Model Subsystem's Persistent Data component and makes the Persistent Data collects the persistent data.

**3.3 Model Subsystem**

Model Subsystem includes the components which hold the persistent or transient data in the proper memory.

**3.3.1 Game Play**

Game Play subsystem holds the general data which is same for every different game play. The Game Play parts hold the information which includes the rules about game and affects the game play. For example, information such as army units' features is held in the Game Play subsystem and affects the logic of the game. Thus, Game Manager accesses the Game Play subsystem and reaches the data about specific features of the game and manages the game play according to these data.

**3.3.2 Game Memory**

Game Play subsystem holds the information of the current game which is not persistent such as which area holds how many soldiers, which player has how many coins and areas, etc. The dependency between Game Memory and Game Manager is reciprocal. After processing the logical calculations and game rules, Game Manager sends the transient data, which is suitable for the current session of the game, to the Game Memory. Also Game Manager subsystem needs the notifications of the Game Memory to provide logical and suitable game play for the player. This subsystem uses RAM to store the data.

**3.3.3 Persistent Data**

Persistent Data subsystem holds the lasting data. Game Manager can reach this part via Persistent Data Manager. Persistent Data Manager can change and reach the data which is held by Persistent Data subsystem. Permanent features of the game such as saved games are held in this subsystem and naturally in the hard disk.

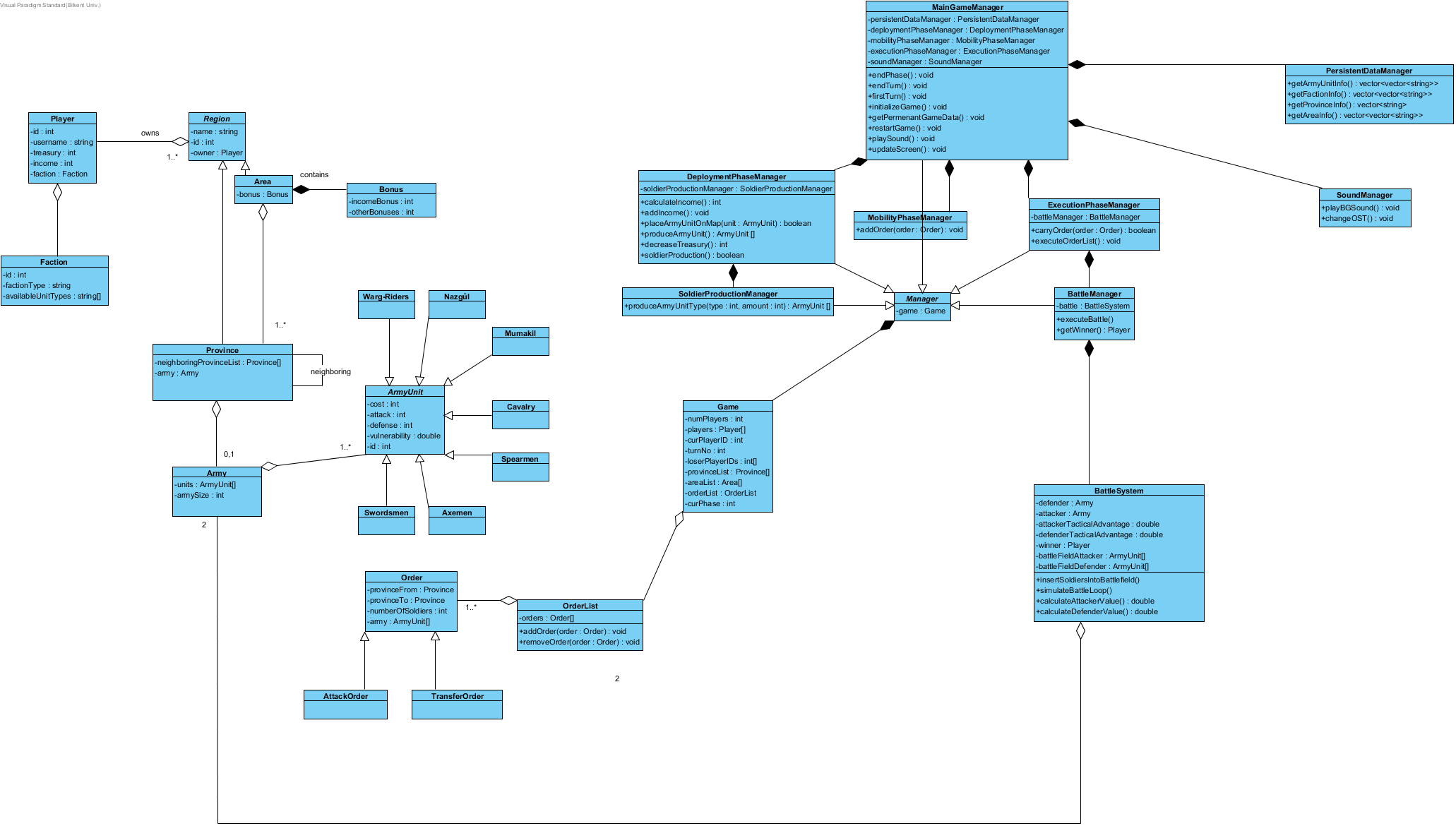
**4. Low-Level Design**

**4.1 Object Design Trade-offs**

**DOĞA🡪FILL HERE WITH 3-4+ TRADEOFFS**

**4.2 Final Object Design**

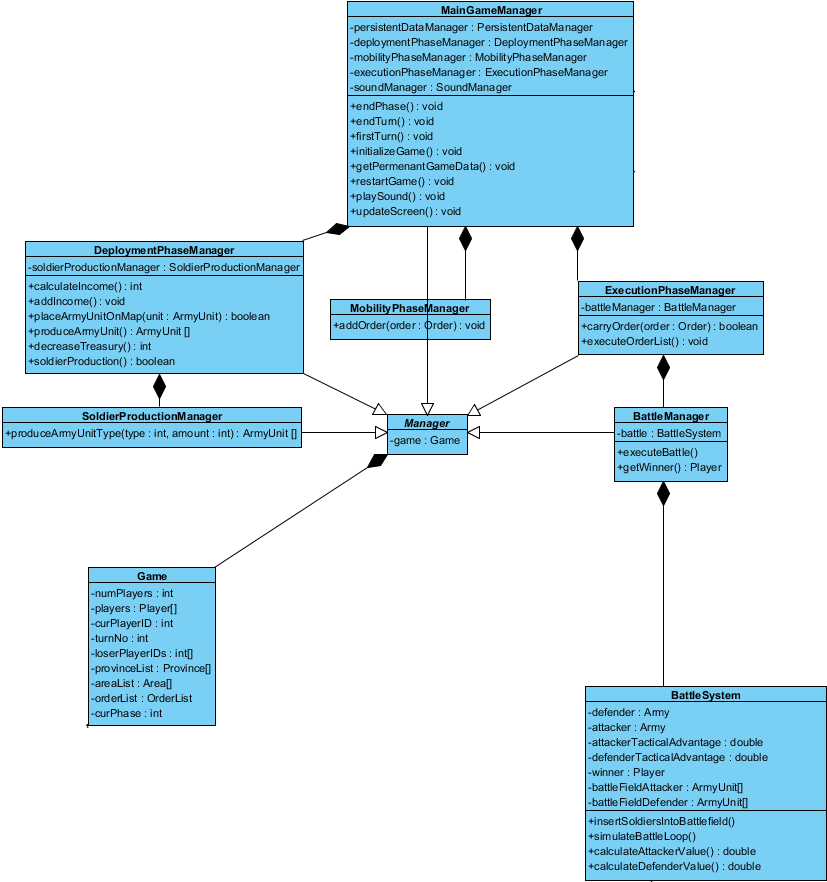
Below is the final object design diagram for the project.



In order to be able to examine classes, we partitioned it into subsections.

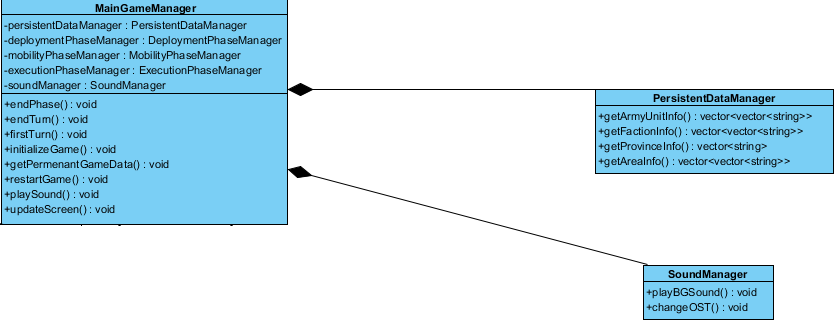
**4.2.1 GameManager Classes**

Here, one can see the classes of the subsystem GameManager.



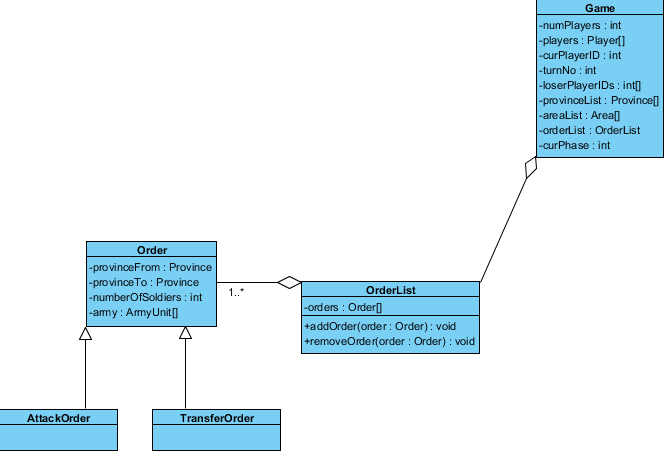
**4.2.2 PersistentDataManager and SoundManager Classes**

Here, one can see the PersistentDataManager and SoundManager classes that make up the PersistentDataManager and SoundManager subsystems, respectively. There is a composition relation between them and the MainGameManager class.



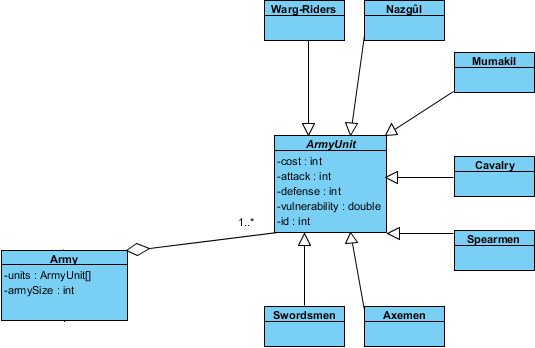
**4.2.3 Game and Order Classes**

Here, one can see Game, OrderList, Order, AttackOrder and TransferOrder classes. They are all part of GameMemory subsystem.



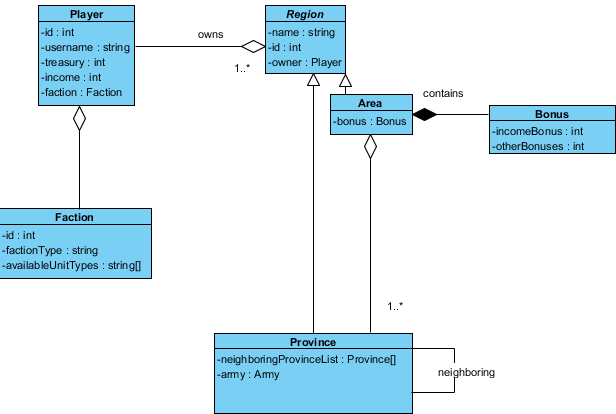
**4.2.4 Army Classes**

Here, one can see Army and ArmyUnit classes. Also, the child classes of ArmyUnit class are included here. They are also a part of GameMemory Subsystem.

****

**4.2.5 Player and Region Classes**

Here, one can see classes related to Player and Region. They are also a part of GameMemory Subsystem.



**4.2.6 MenuView Classes**

**HAZAL🡪INSERT DIAGRAM HERE**

**4.2.6 GameView Classes**

**HAZAL🡪INSERT DIAGRAM HERE**

**4.3 Packages**

**4.3.1 View Package**

**HAZAL🡪INSERT EXPLANATION HERE**

**4.3.1.1 MenuView Subsystem**

**HAZAL🡪INSERT EXPLANATION HERE**

**4.3.1.2 GameView Subsystem**

**HAZAL🡪INSERT EXPLANATION HERE**

**4.3.2 Controller Package**

**DOĞA🡪INSERT EXPLANATION HERE**

**4.3.2.1 InputManager Subsystem**

This subsystem is responsible for the managing of inputs. The View Package actually takes care of getting inputs from the user. Then, these inputs are transferred to GameManager Subsystem where they are evaluated.

**4.3.2.2 GameManager Subsystem**

All classes related with game management are in this package. This package has 8 classes in total which will be implemented to control the game

**4.3.2.2 SoundManager Subsystem**

**DOĞA🡪INSERT EXPLANATION HERE**

**4.3.2.2 PersistentDataManager Subsystem**

**DOĞA🡪INSERT EXPLANATION HERE**

**4.3.3 Model Package**

**4.3.3.1 PersistentData Subsystem**

**DOĞA🡪INSERT EXPLANATION HERE**

**4.3.3.2 GameMemory Subsystem**

**DOĞA🡪INSERT EXPLANATION HERE**

**4.4 Class Interfaces**

**4.4.1 View Package**

**HAZAL🡪INSERT EXPLANATION HERE**

**4.4.1.1 MenuView Subsystem**

**HAZAL🡪INSERT THIS SUBSYSTEMS CLASS DIAGRAMS ONE BY ONE HERE. THEN, FOR EACH OF THEM, EXPLAIN THEM, THEIR ATTRIBUTES AND FUNCTIONS.**

**4.4.1.2 GameView Subsystem**

**HAZAL🡪INSERT THIS SUBSYSTEMS CLASS DIAGRAMS ONE BY ONE HERE. THEN, FOR EACH OF THEM, EXPLAIN THEM, THEIR ATTRIBUTES AND FUNCTIONS.**

**4.4.2 Controller Package**

**4.4.2.1 InputManager Subsystem**

There are no classes in this subsystem as the View Subsystems send inputs to GameManager Subsystem.

**4.4.2.2 GameManager Subsystem**

**4.4.2.2.1 Manager**

All the other manager classes extend from Manager class.

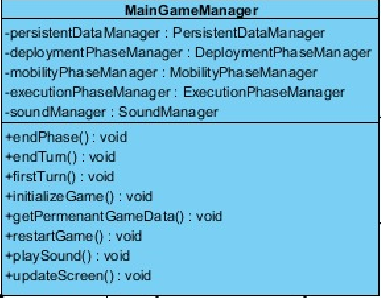


**Attribute:**

***game*:** Game object that every manager class has access.

**4.4.2.2.2 MainManagerGame**

Deals with delegation of user inputs to other manager classes, and more general methods, extends manager class.



**Attributes:**

**PersistentDataManager:** is a PersistentDataManager object which will be used to manage the persistent date.

**deploymentPhaseManager:** is a deploymentPhaseManager object which will be used to manage the deployment phase.

**mobilityPhaseManager:** is aMobilityPhaseManager object which will be used to manage the mobility phase.

**executionPhaseManager:** is a executionPhaseManager object which will be used to manage the execution phase.

**soundManager:** isa SoundManager object used to produce the different sounds.

**Methods**

***endPhase:*** since the game is composed of different phases, this method will terminate each phase.

***endTurn:*** make function delegations inside this method.

***firstTurn:*** this method initializes the first term of a specific player

***initializeGame:*** starts the game by creating a new game object.

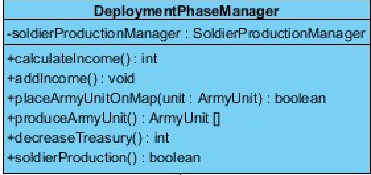
***getPermanentGameData:*** handles the load of permanent data into system

***restartGame:*** restart game by creating a new game object

***playSound:*** depending on users input plays the corresponding sound

***updateScreen*:** thismethod handles the necessary changes to be done on the screen

**4.4.2.2.3 DeploymentPhaseManager**



**Attributes:**

**SoldierProductionManager:** is a soldierProductionManager object that all functions of this class can use.

**Functions:**

***calculateIncome:*** calculates income of each turn of each player and returns it as an integer.

***addIncome:*** this method add incomes of each specific player at the end of each turn

***placeArmyUnitOnMap:*** this method adds the army unit on the map

***produceArmyUnit:*** produce an army unit by creating a new army unit object

***decreaseTreasury:*** decreases treasury amount of the player.

***soldierProduction:***  produces a soldier by creating a solder object.

**4.4.2.2.4 MobillityPhaseManager**



**Function:**

***addOrder:*** adds an Order object

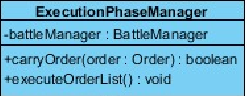
**4.4.2.2.5 SoliderProductionManager**



**Function:**

**produceArmyUnitType** : produces Army units; by taking the type and amount of unit to be produced.

**4.4.2.2.6 ExecutionPhaseManager**



**Atributes:**

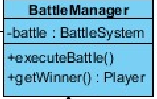
**BattleManager:** is a battleManager object that can be used from all the functions of the class

***Functions:***

***carryOrder:*** returns a Boolean which indicates if the order is carried or not.

***executeOrderList:*** executes the list of all actions that user had ordered

**4.4.2.2.17 BattleManager**



**Atributes:**

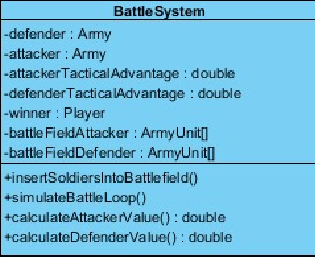
**battle:** is a BattleSystem object that can be used by functions of this class

**Functions:**

***executeBattle:*** This method handles all actions to be performed during execute stage

***getWinner:*** This method returns a Player object which is the winner and which is chose based on executeBattle output

**4.4.6.8 BattleSystem**



**Attributes:**

**defender**: is an Army object; as the name is self-explanatoryits role is defending

**attacker**: is an Army object; as the name is self-explanatoryits role is attaching

**attackerTechnicalAdvantage**: is a double which keeps the number of technical advantages of attacker

**attackerTechnicalAdvantage:** is a double which keeps the number of technical advantages of defender

**winner:** is a player object which keeps the winner

**battleFieldAttacker:** is an army unit as name indicates is the attacker in the battlefield

**battleFieldDrfender:** is an army unit and as name indicates is the defender in the battlefield

**Functions:**

**insertSoldersIntoBattlefield:** this method adds different kind of solders in the battlefield

**simulateBattleLoop:** In accordance with the attacks and transfers that the player performs this function simulates the corresponding battle

**calculateAttackerValue:** calculates the points gathered from the attacker

**calculateDefenderValue:** calculates the points gathered from the defender

**4.4.2.2 SoundManager Subsystem**

**DOĞA🡪INSERT THIS SUBSYSTEMS CLASS DIAGRAM HERE. THEN, EXPLAIN IT, ITS ATTRIBUTES AND FUNCTIONS.**

**4.4.2.2 PersistentDataManager Subsystem**

**DOĞA🡪INSERT THIS SUBSYSTEMS CLASS DIAGRAM HERE. THEN, EXPLAIN IT, ITS ATTRIBUTES AND FUNCTIONS.**

**4.4.3 Model Package**

**4.4.3.1 PersistentData Subsystem**

This subsystem does not have any classes. This is because the PersistentData Subsystem is data stored on the hard drive. The persistent data, which is crucial information regarding the game such as army unit information or province information, is stored as “.txt” files. So, PersistentData Subsystem is a collection of “.txt” datas.

**4.4.3.2 GameMemory Subsystem**

**4.4.3.2.1 Player**

Player Class that is central to the game and is unique for each player of the game.

****

**Attributes**

**id:** This is an integer representing the id of each player. It is unique.

**username:** This is a string representing the name of the player.

**treasury:** This is an integer representing amount of gold the player currently holds in her treasury.

**income:** This is an integer representing the current income/turn of the player.

**faction:** This is a Faction object that represents the Faction the Player belongs to.

**4.4.3.2.2 Faction**

Faction class that is owned by the Player class.

****

**Attributes**

**id:** This is an integer representing the id of each Faction. It is unique.

**factionType:** This is a string representing the name/type of the Faction.

**availableUnitType:** This is a string array consisting of the name of the unit types that are possible to produce for this Faction.

**4.4.3.2.3 Region**

This is an abstract class. Area and Province classes extend Region abstract class.

****

**Attributes**

**name:** This is a string representing the name of each Region.

**id:** This is an integer representing the id of each Region. It is unique.

**owner:** This is a Player object representing the owner of the Region. If it is null, there is no current owner.

**4.4.3.2.4 Area**

Extends Region Class.

****

**Attributes**

**bonus:** This is Bonus object representing the Bonus corresponding to the Area.

**4.4.3.2.5 Bonus**

Is owned by Area Class.

****

**Attributes**

**incomeBonus:** This is an integer representing the income bonus of the Bonus object.

**otherBonus:** This is an integer representing the other sorts of bonuses of the Bonus object.

**4.4.3.2.6 Province**

Extends Region Class.

****

**Attributes**

**neighboringProvinceList:** This is an array of Province objects that are neighbors of this instance of Province.

**army:** This is an Army object representing the Army that is currently on this province. If army is null, it means that there are no ArmyUnits (soldiers) on this province.

**4.4.3.2.7 Army**

Holds ArmyUnit objects.

****

**Attributes**

**units:** This is an array of ArmyUnit objects that are a part of this army.

**armySize:** This is an integer representing the number of ArmyUnits in this army.

**4.4.3.2.8 ArmyUnit**

An abstract class that is the parent of different types of ArmyUnits.

****

**Attributes**

**cost:** This is an integer representing the cost of producing the ArmyUnit.

**attack:** This is an integer representing the attack value of the ArmyUnit.

**defense:** This is an integer representing the defense value of the ArmyUnit.

**vulnerability:** This is a double representing the vulnerability value of the ArmyUnit. If it is high, it is easier for this Unit to be destroyed in battle.

**Id:** This is an integer representing the id of the ArmyUnit. It is unique.

**4.4.3.2.9 Swordsmen**

Extends ArmyUnit.

****

**4.4.3.2.10 Axemen**

Extends ArmyUnit.

****

**4.4.3.2.11 Spearmen**

Extends ArmyUnit.

****

**4.4.3.2.12 Cavalry**

Extends ArmyUnit.

****

**4.4.3.2.13 Mumakil**

Extends ArmyUnit.

****

**4.4.3.2.14 Nazgul**

Extends ArmyUnit.

****

**4.4.3.2.15 Warg-Riders**

Extends ArmyUnit.

****

**4.4.3.2.16 OrderList**

Holds Order objects.

****

**Attributes**

**orders:** This is an array of Order objects that are to be carried out in the execution phase of the current turn.

**Methods**

**addOrder:** This method adds an order to the OrderList. It takes an Order as a parameter.

**removeOrder:** This method removes an order from the OrderList. It takes an Order as a parameter.

**4.4.3.2.17 Order**

An abstract class that is the parent of TransferOrder and AttackOrder.

****

**Attributes**

**provinceFrom:** This is a Province object representing the province from which the mobility order takes place.

**provinceFrom:** This is a Province object representing the province that is the target of the mobility order.

**army:** This is an Army object representing army that is to take place in this order.

**4.4.3.2.18 TransferOrder**

Extends Order Class.

****

**4.4.3.2.19 AttackOrder**

Extends Order Class.

****

**4.4.3.2.20 Game**

Holds all the relevant information about the current Game.

****

**Attributes**

**numPlayers:** This is an integer representing the number of players playing the game.

**players:** This is an array of Player objects that represent the players playing the game.

**curPlayerID:** This is an integer representing the id of the player who is currently playing her turn.

**turnNo:** This is an integer representing the turn count.

**loserPlayerIDs:** This is an integer array representing the id’s of the players who lost the game.

**provinceList:** This is an array of Province objects representing the provinces on the map.

**areaList:** This is an array of List objects representing the areas on the map.

**orderList:** This is an OrderList object representing the orders to be executed.

**curPhase:** This is an integer representing the current phase of the turn. A value of 1 means deployment phase, 2 means mobility phase and 3 means execution phase.