CS 319 TERM PROJECT

Section 1 - Group 1I

Risk Game

Design Report - Iteration 2

Emin Adem Buran - 21703279

Onur Oruç - 21702381

Ömer Yavuz Öztürk - 21803565

Melike Demirci - 21702346

Yusuf Ziya Özgül - 21703158

Supervisor: Eray Tüzün

1. Introduction

1.1. Purpose of the System

Risk is a turn-based 2-D game. In the game, the purpose is to capture areas owned by other players and owning entire regions. The game can be played with two to four players. In this version of the risk game, the entire world map is divided into 42 regions. The difference between the original Risk Game and our version of Risk Game is that our version of the game includes some modes and extra futures explained below:

- Players can have money and money can be gained through gold mines.
- Gold mines will exist in some random regions.
- Troops in the game will have the motivation and the motivation of the troops will have effects on the results of wars.
- The motivation of troops can be increased by organizing events and these events can be organized by using money.
- There is a plague mode and in this mode, a plague can occur any time in the game and destroys the shoulders in the regions where it occurs.
- There is a commander mode and in this mode, each player will have one war hero and they will be able to move their heroes just like their soldiers. During battles, war heroes will increase the motivation level of nearby soldiers.
- There is a heat and snow mode and in this mode, weather conditions get in the way of armies. There will be different seasons and climates to determine the weather condition. According to the conditions, the way of soldiers to go to other counties may be closed.

With these additional features and modes, we aim to satisfy the player's expectations from a game.

To put it briefly, the purpose of the system is to entertain players by providing a user-friendly and high-performance Risk Game with some extra features, modes.

1.2. Design Goals

1.2.1. Maintainability

In our game, we have adopted three-layer architecture, which enables the developers to make changes or add new functionality to the software by layering the system. Subsystems of the game will have hierarchical structure so that it will be enough to modify related subsystem in order to make changes. Another aspect that we try to improve is readability. Although we made some design choices related to user interface libraries which decreases readability, we try to follow certain standards for commenting. Moreover, we try to organize files and folders in a way that eases the possible future developer's life.

1.2.2. Dependability

While implementing the user interface, we have limited some interactions for some users to avoid invalid user input. Thus, we will add an exception handling mechanism in order to increase the robustness of the game.

1.2.3. User - Friendliness

In our game user interface will be kept minimal in order to make the player's game experience simple and fun. Only the crucial information about the regions will be on the game map with simple design. Risk has a complex logic and rules of the game might be hard to remember. Thus, through the game players will be able to reach detailed explanations about the rules/concepts in how to play panel.

1.2.4. Portability

In order to increase portability of our game, we have chosen Java as the implementation language. Java is a platform independent language, in other words java compiled code can run on all operating systems. Also we will build executable such as jar in order to favor portability.

2. High-level software architecture

2.1. Subsystem decomposition

In order to satisfy our design goals, we decided to perform three-layer architecture style, which is a hierarchy of three layers. These three layers are Presentation, Application and Data Layers. Presentation layer includes boundary objects that deal with the user as fxml files and classes connected to them. Application layer contains manager classes and entities connected to them. Last layer is the data layer, which has txt files holding game data. In three-layer architecture, layering is opaque, in other words each layer can only call operations from layer below.

This closed architecture increases flexibility that we need for future implementations. For example, we aim to add different game maps in the future and separation between the interface layer and the application logic layer enables the development of different user interfaces for the same game logic. As it is represented in Figure1, the game has three subsystems that have a responsibility to invoke other systems in order to make the game maintainable during runtime.

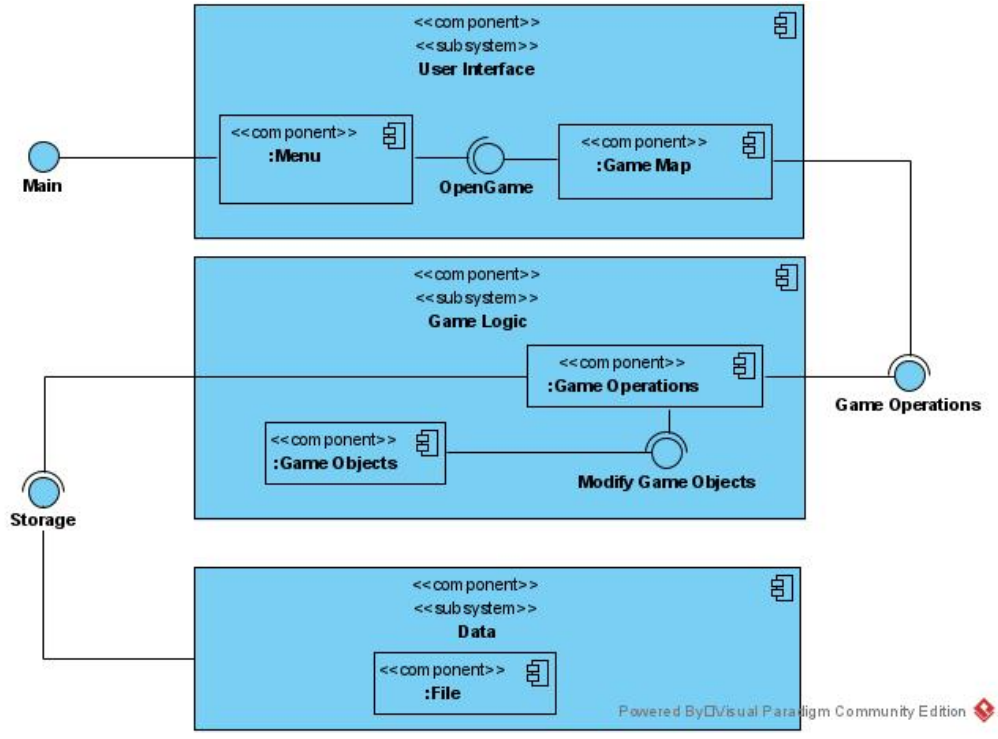

Figure 1: Risk Game Component Diagram

2.2. Hardware/software mapping

Our game will be implemented by Java programming language. So, as a software requirement, “Risk” will require Java Runtime Environment to be executed by users playing this game and “Risk” requires Java 11.0.9.

As a hardware requirement, “Risk” will not require any complex requirement. Users will control the menu and their movements during the game like attacking, fortifying and other movements by using only mouse and keyboard. So, user will play the game with minimal hardware. As a consequence of these two requirements, system requirements of “Risk” will be minimal. Only one computer with Java 11.0.9 installed will be enough to compile and run the game.

For storage of the game, we will use text files. Text files include the saved games’ data, and game map. Since “Risk” will be played on the same computer by users, “Risk” will not require a database or internet connection.

2.3. Persistent data management

Our game will not require any complex data storage system or database system. “Risk” will store game instances like a game map, and saved games in text files. Text files which are used for storing the game map will be created during the implementation stage and these files will not be modified. Players can save their games to continue it in the future. Information in the GameManager class which has Player, Region and Continent Entity classes is written in the text files when players want to save the game. When players want to load the game , the saved game data is read from the corresponding text files. If players want to save the game, text files will be instantiated and stored for later usage.

2.4. Access control and security

Risk Game will not use any network connection or database system. It will be an offline multiplayer game. There isn’t any threat to security so there is no need to design an access control system. However, some of the data required to launch the game will be stored in .txt files which can be changed or deleted by the user. These modifications or deletions might be detrimental so users will be advised to not do them.

2.5. Boundary conditions

Risk Game will be available to users by an executable .jar file, and it will not require an installation. After it is downloaded on a computer, the game files can be moved or can be copied and as long as they are not separated, the game will be ready for execution. Termination of the application can be done by using the exit button on the main menu. If users want to quit the application during an ongoing game, they are advised to pause the game and return to the main menu with or without saving and then quit it.

If any error about reading or writing the game files occurs, users can validate game files from the options menu. This operation will help fixing possible errors which might occur as a result of forced shut down.

3. Low-level Design

3.1. Object design trade-offs

3.1.1. Usability vs. Readability

While implementing the game, we use JavaFX. In JavaFX, some extra CSS and FXML documents are needed to design GUI. Instead of using JavaFX, we could use some other API's such as Swing and AWT. However, using these API's make the game less usable for users because these API's offer less choice to programmers while making GUI design. On the other hand, FXML and CSS documents used in JavaFX makes the code of the game less readable for future developments.

3.1.2. Functionality vs. Performance

Our game allows players to play the game in different modes (e.g., commander, plague, and climate). This expands the functionality of the game. However, integrating the modes that players choose for the game and initializing the game that is a combination of different modes come with performance costs.

3.1.3. Dependability vs. Performance

While implementing the Risk Map design, we have disabled some options for some users to avoid invalid user input. For example, when it is the turn of a player, the regions which this player does not have become disabled for users not to interact with these regions (except Attack phase). While implementing this functionality, we need to iterate through all regions to disable regions that a user does not have and this situation decreases the performance of the game.

3.1.4 Maintainability vs Performance

While implementing some classes, we used integer or boolean arrays instead of using object arrays to decrease the memory use of our game. By decreasing the memory use, we increased the performance of the game. However, using boolean and integer arrays make our codes less readable for future developments. Also while using three-layer architectural style increases maintainability by allowing developers to change or add subsystems independently, it has a negative effect for performance.

3.2. Final object design

3.2.1. Object Diagram

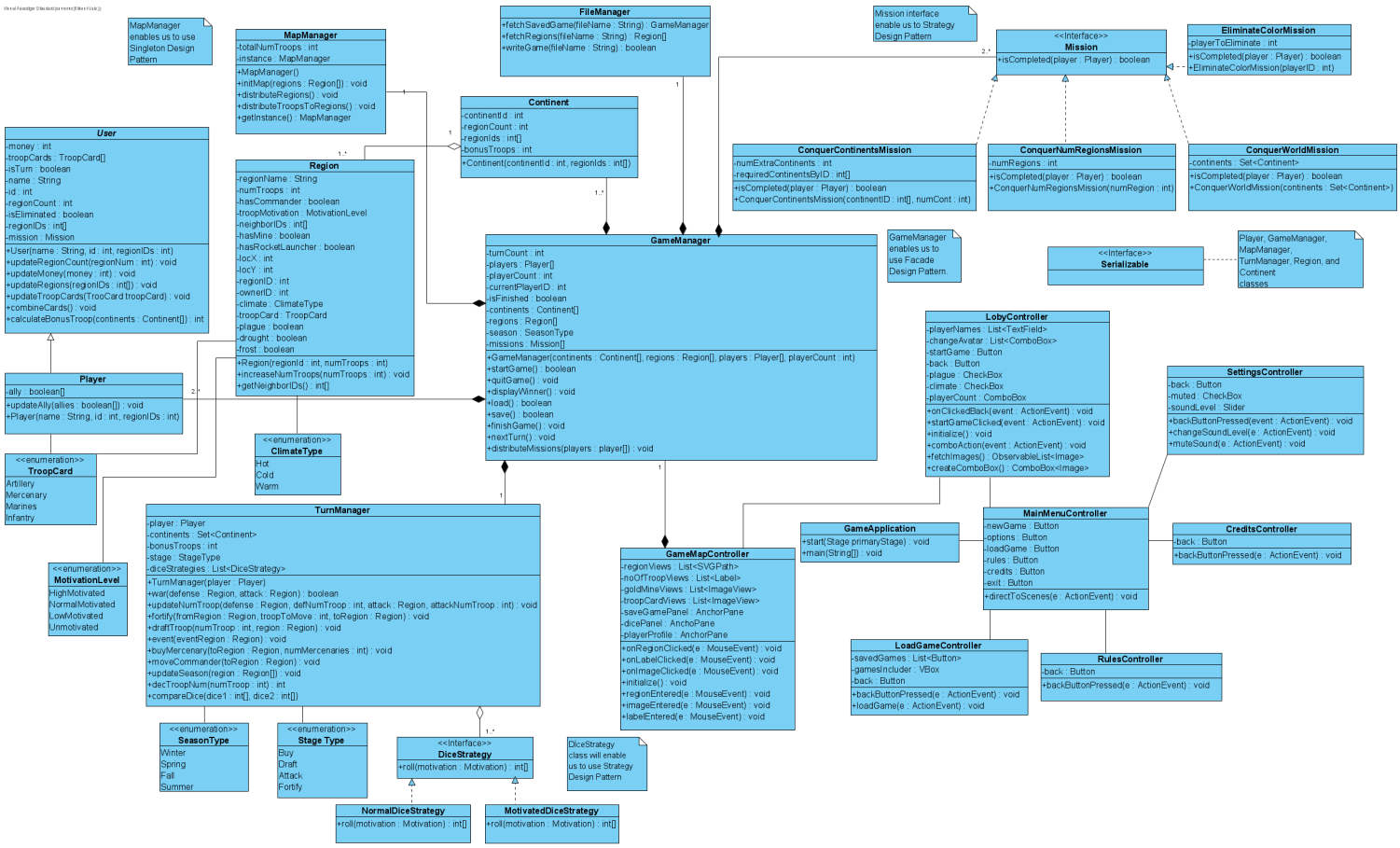

Figure 2: The whole object diagram of the game

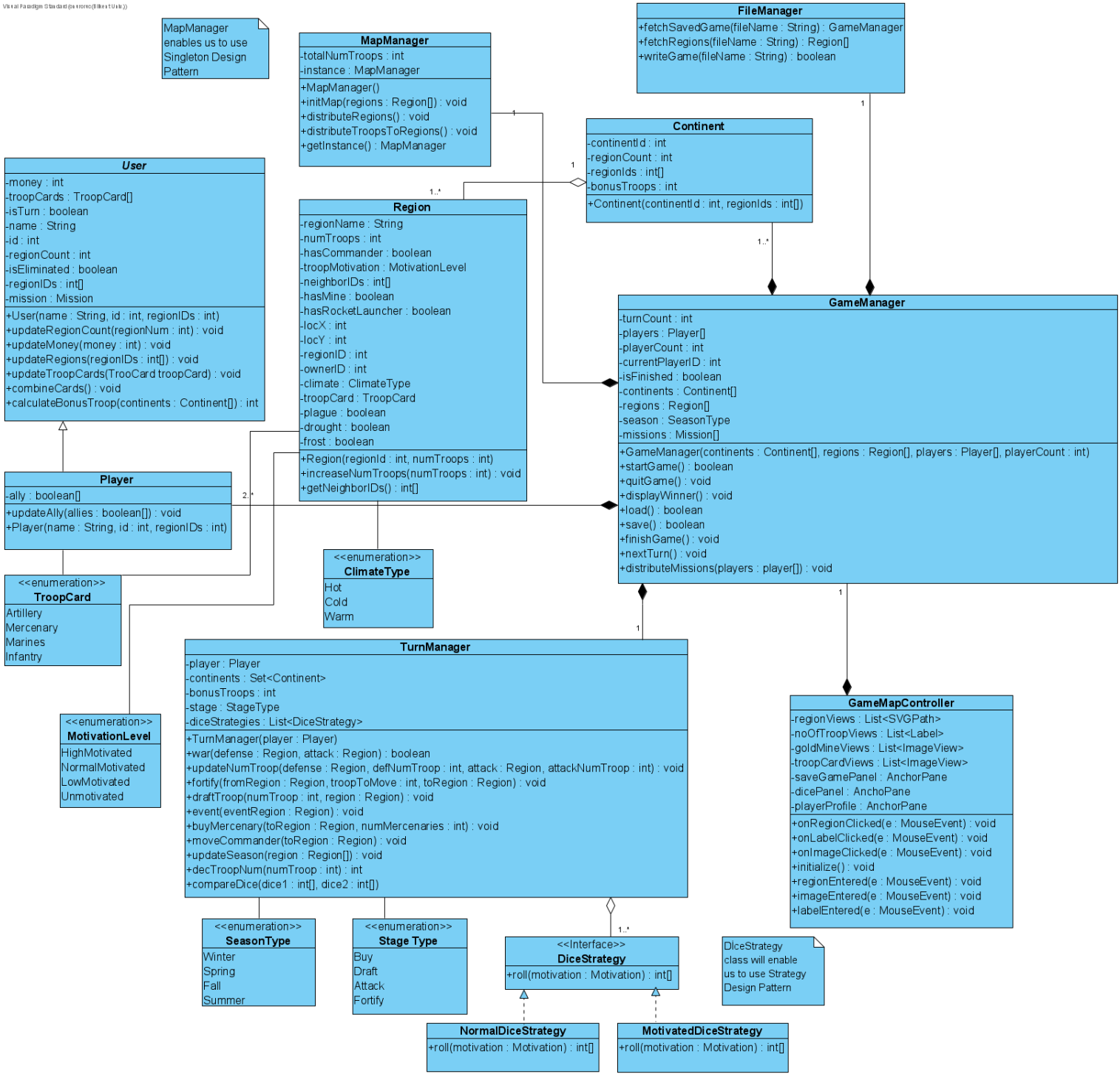

Figure 3: First divided part of the object diagram for readability

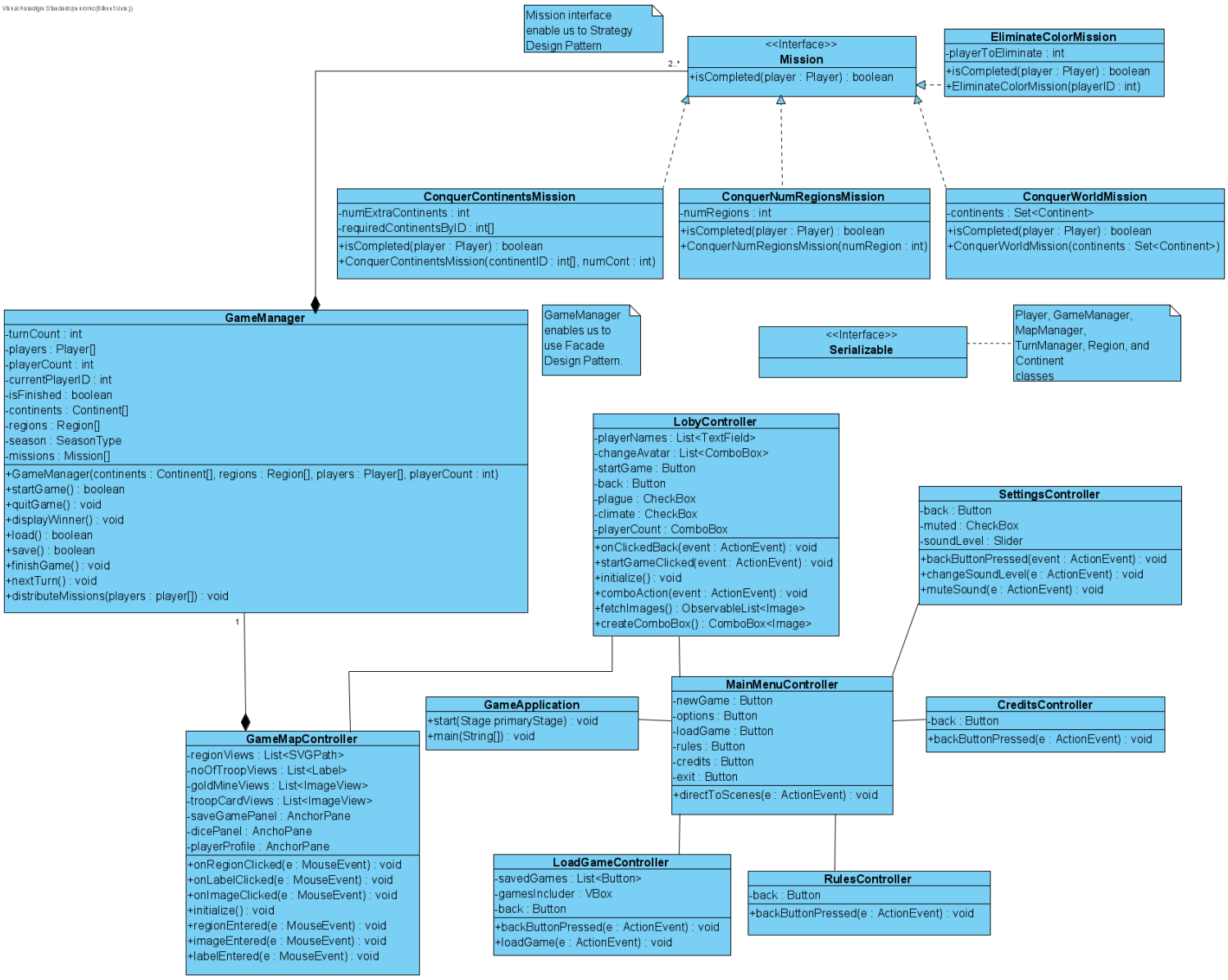

Figure 4: Second divided part of the object diagram for readability

3.2.2. Design Patterns

3.2.2.1. Strategy Design Pattern

Since the functionality of dice objects will change in run time based on the motivation level of the current region whose soldiers will attack, for the implementation of Dice and MotivatedDice classes, the strategy design pattern will be used.

3.2.2.2. Facade Design Pattern

Since many operations are done in the GameManager class, we used the Facade design pattern and divided operations into helper classes in order to simplify the code and to see what it is doing clearly.

3.2.2.3. Singleton Design Pattern

Since there will be one and only one instance of it, we implemented MapManager class using Singleton design pattern.

3.3. Packages

3.3.1. javafx.event.ActionEvent

This event type is widely used when a Button has been fired in order to handle event management.

3.3.2. javafx.fxml.FXMLLoader

This package loads an object hierarchy from an XML document. We will use this package to change scenes in a stage.

3.3.3. javafx.scene

This package is the container for all content such as panes, buttons,text fields, combobox in a scene graph.

3.3.4. javafx.stage

This package is the top level JavaFX container. The primary Stage is provided by the system. Additional Stage objects may be constructed by the application to add a new window to the program.

3.3.5. java.io.*

This package is used in systems for system input and output through data streams, serialization and the file system.

3.3.6. java.util

This package will provide ArrayList for some classes such as GameMapController to handle SVGPaths, labels, buttons and image views.

3.3.7. javafx.collections

This package contains the essential JavaFX collections and collection utilities. We will use this package when creating combo boxes with different content such as image view, string.

3.4. Class Interfaces

3.4.1. Class Interfaces of User Interface Layer

3.4.1.1. Game Map

3.4.1.1.1. GameMapController

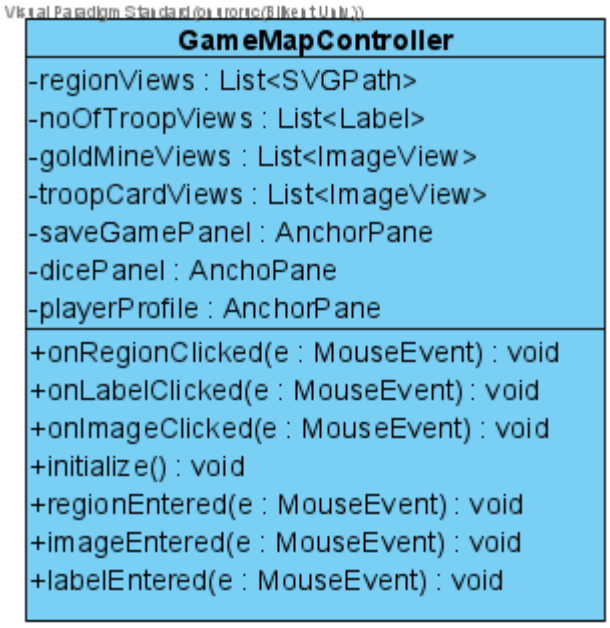

Figure 5: GameMapController Class with its attributes and operations

This class will contain GUI objects and manage the changes coming from the game map view and then update the game map view.

Attributes:

- **private List<SVGPaths> regionViews:**This attribute contains the shape of each region.
- **private List<Label> noOfTroopViews:** This attribute contains labels which hold the number of troops in each region.
- **private List<ImageView> goldMineViews:** This attribute contains image views of gold mines in each region.
- **private List<ImageView> troopCardViews:** This attribute contains image views of troop cards in each region.
- **private AnchorPane saveGamePanel:**This attribute contains the save game panel and its visibility changes according to the stage of the game.
- **private AnchorPane dicePanel:**This attribute contains a dice panel and its visibility changes according to the stage of the game.
- **private AnchorPane playerProfile:**This attribute contains a panel which holds the situation of players and its visibility changes according to the stage of the game.

Functions:

- **public void onRegionClicked(MouseEvent e) :**This method handles events in SVGPaths in each region.
- **public void onLabelClicked(MouseEvent e):**This method handles events in labels in each region.
- **public void onImageClicked(MouseEvent e) :** This method handles events in image views in each region.
- **public void initialize() :**This method initializes the map according to the game management object.
- **public void regionEntered(MouseEvent e) :** This method handles events in SVGPaths when the mouse enters each region.
- **public void labelEntered(MouseEvent e) :** This method handles events in labels when the mouse enters each region.
- **public void imageEntered(MouseEvent e) :** This method handles events in image views when the mouse enters each region.

3.4.1.2. Menu

3.4.1.2.1. LobbyController

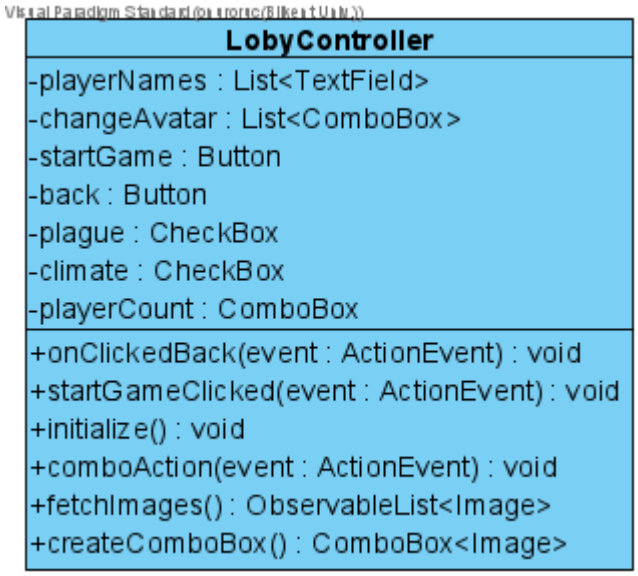

Figure 6: LobbyController Class with its attributes and operations

This class will contain GUI objects and manage the changes coming from the lobby view and then initializes the game.

Attributes:

- **private List<TextField> playerNames :** This attribute contains the nickname of players in text fields.
- **private List<ComboBox> changeAvatar :** This attribute contains the avatars of players in text fields.
- **private Button startGame :** This attribute will direct users to the game map.
- **private Button back :**This attribute will direct users to the main menu.
- **private CheckBox plague :** This attribute will determine whether the game will have plague mode or not.
- **private CheckBox climate :** This attribute will determine whether the game will have climate mode or not.
- **private ComboBox playerCount :** This attribute will determine the number of players in the game.

Functions:

- **public void onClickedBack(ActionEvent event) :** This method will direct users to the main menu by using FXML Loader.
- **public void startGameClicked(ActionEvent event) :** This method will direct users to the game map by using FXML Loader.
- **public void initialize() :** This method will initialize avatar combobox by fetching images.
- **public void comboAction(ActionEvent event) :**This method will change the visibility of avatar combobox and nick name text fields according to the number of players.
- **public ObservableList<Image> fetchImages() :** This method will fetch images for avatar comboboxes.
- **public ComboBox<Image> createComboBox() :**This method will create avatar comboboxes in the initialize method.

3.4.1.2.2. SettingsController

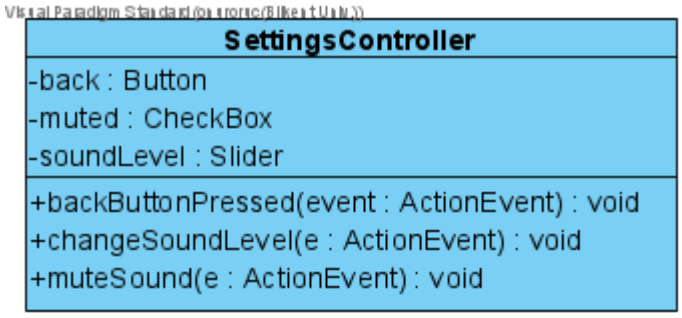

Figure 7: SettingsController Class with its attributes and operations

This class will contain GUI objects and manage the changes coming from the settings view and change the sound settings of the game.

Attributes:

- **private Button back :** This attribute will direct users to the main menu.
- **private CheckBox muted :** This attribute will determine whether the sound of the game is muted or not.

- **private Slider soundLevel :** This attribute will determine the sound level of the game.

Functions:

- **public void backButtonPressed(ActionEvent event) :**
- This method will direct users to the main menu by using FXML Loader.
- **public void changeSoundLevel(ActionEvent e) :**
- This method will change the sound level of the game according to the action event coming from the slider.
- **public void muteSound(ActionEvent e) :**
- This method will change the mutability of the game sound according to the action event coming from the checkbox.

3.4.1.2.3. MainMenuController

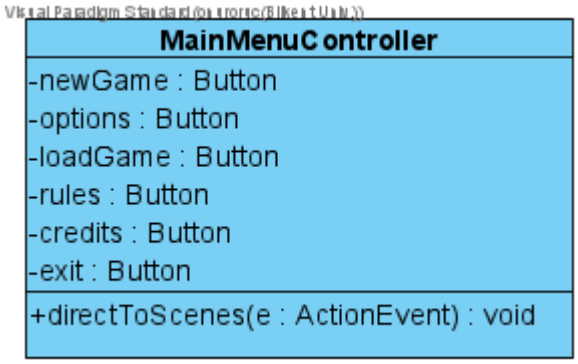

Figure 8:MainMenuController Class with its attributes and operations

This class will direct users to other scenes in the game such as load game, rules, options, lobby and credits.

Attributes:

- **private Button newGame :** This attribute will direct users to the lobby.
- **private Button options :** This attribute will direct users to the settings.
- **private Button loadGame :** This attribute will direct users to the load game.
- **private Button rules :** This attribute will direct users to the rules.

- **private Button credits :** This attribute will direct users to the credits.
- **private Button exit :** This attribute will direct users to the desktop of the pc.

Functions:

- **public void directToScenes(ActionEvent e) :** This method directs users to game scenes according to the action events coming from different buttons.

3.4.1.2.4. LoadGameController

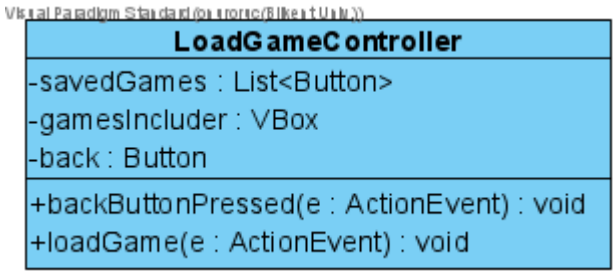

Figure 9: LoadGameController Class with its attributes and operations

This class will enable users to load saved games.

Attributes:

- **private ArrayList<Button> savedGames :** This attribute will contain saved games.
- **private VBox gamesIncluder :** This attribute will contain a collection of saved game buttons.
- **private Button back :** This attribute will direct users to the main menu.

Functions:

- **public void backButtonPressed(ActionEvent event) :** This method will direct users to the main menu by using FXML Loader.

3.4.1.2.5. RulesController

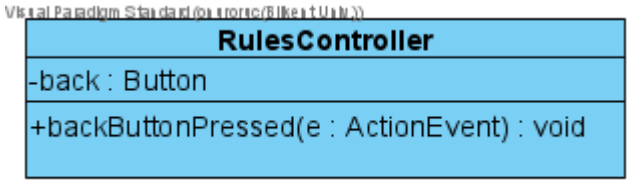

Figure 10: RulesController Class with its attributes and operations

Attributes:

- **private Button back :** This attribute will direct users to the main menu.

Functions:

- **public void backButtonPressed(ActionEvent event) :** This method will direct users to the main menu by using FXML Loader.

3.4.1.2.6. CreditsController

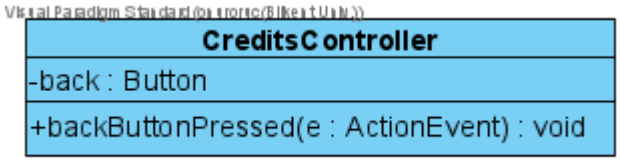

Figure 11: CreditsController Class with its attributes and operations

Attributes:

- **private Button back :** This attribute will direct users to the main menu.

Functions:

- **public void backButtonPressed(ActionEvent event) :** This method will direct users to the main menu by using FXML Loader.

3.4.2. Class Interfaces of Game Logic Layer

3.4.2.1. Game Operations

3.4.2.1.1. GameManager

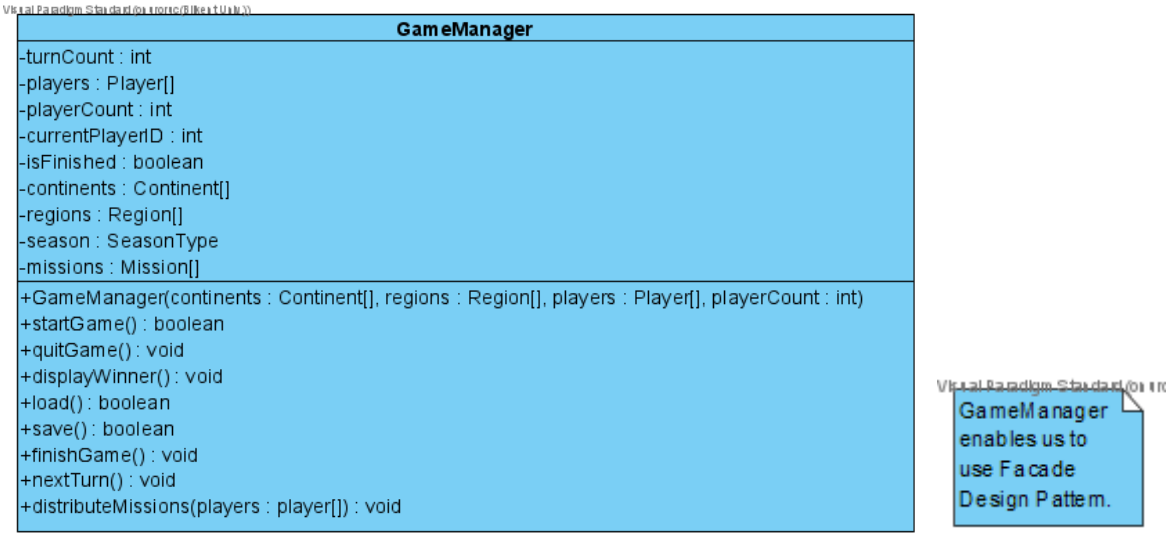

Figure 12: GameManager class with its attributes and operations

This is the class where players, continent, regions, number of players are initialized. This class will utilize the Facade design pattern by using GameManager, MapManager, and FileManager.

Attributes:

- **private int turnCount**: It will keep track of in which turn the game is.
- **private int playerCount:** It will store how many players will play the game.
- **private Player players[]:** It will be the size of playerCount and will store Player objects.
- **private Continent continents[]:** It will store the continent of the map.
- **private Region regions[]:** It will store the regions of the map.
- **private int currentPlayerID:** Every player will have an id and this attribute will store the current player's id. This id will be used to display corresponding components in the UI.

- **private boolean isFinished:** This attribute will keep track of the game stage. It will be assigned to “false” when the game is over (i.e., when a player wins or a player quits the game).

Constructor(s):

- **public GameManager(Continent[] continents, Region[] regions, Player[] players, int playerCount):**
- The constructor will initialize how many players will play the game, and create Player objects accordingly. It will initialize the continents and regions of the map.

Functions:

- **public boolean startGame():** This function will start the game by using a MapManager object to initialize the game map.
- **public void quitGame():** This function will manage quit requests.
- **public void displayWinner():** This function will be responsible for displaying the winner (if exists) at the end of the game.
- **public boolean load():** This function will load a chosen game that is previously saved.
- **public boolean save()**: This function will be responsible for saving the game process during a game.
- **public void nextTurn():** This function will be responsible for turn management between players. It will allow the next player to play when the previous player’s turn is done.

3.4.2.1.2. TurnManager

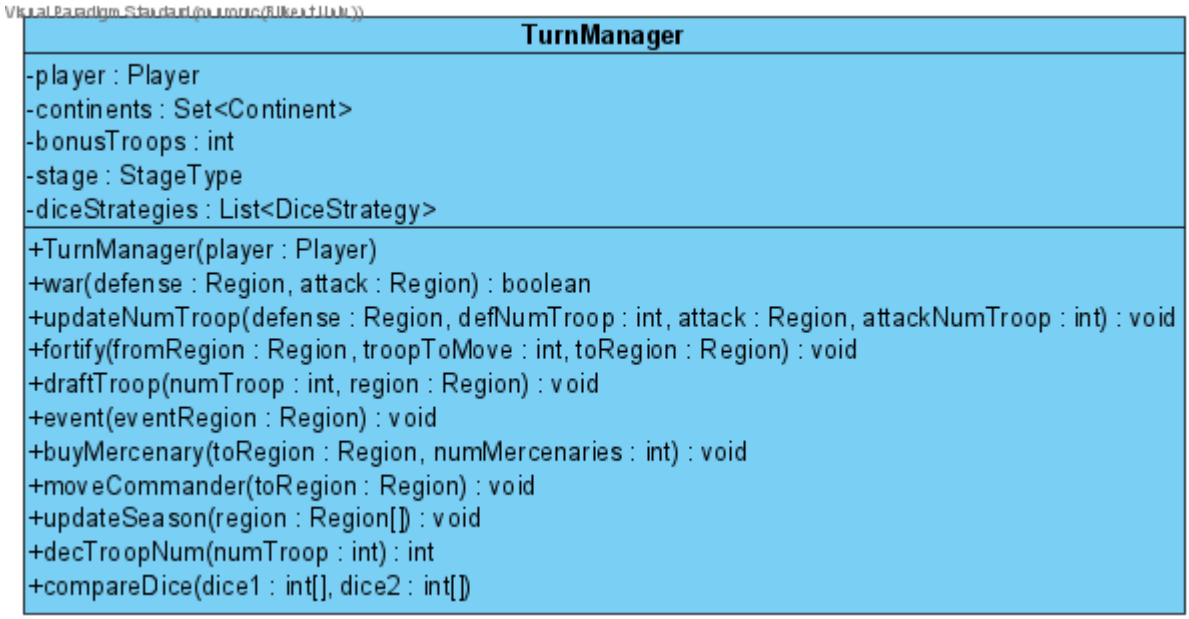

Figure 13: TurnManager class with its attributes and operations

This class will manage, for each player, the actions that can be taken in a single turn.

Attributes:

- **private Player player:** This attribute will store the information of the player whose turn is on.
- **private Set<Continent> continents:** This property will be used to check whether the current has a continent(s).
- **private int bonusTroops:** This will be the extra troops that will be given to a player according to combination of his/her cards (if any), and according to the continents that s/he owns (if any).
- **private Stage StageType:** It will be used to manage the stages (buy, draft, attack, fortify) in a turn.
- **private List<DiceStrategy> diceStrategies:** This property will be used to create different mission strategies that will be distributed to players at the beginning of the game.

Constructor(s):

- **public TurnManager (Player player):** A player object will be passed into TurnManager.

Functions:

- **public void buyMercenary(Region toRegion, int numMercenaries)**: This function will allow players to buy mercaniers (if they have enough money) in the “buy” stage.
- **public void event(Region eventRegion)**: This function will perform events in a chosen region by a player and update the motivation level of the region accordingly.
- **public void draftTroop (int numTroop, Region region)**: This function will be used to place soldiers in a region chosen by a player in the “draft” stage.
- **public boolean war(Region defense, Region attack)**: This function will perform the calculations to decide whether the player who is attacking will win or not. If the attacker wins the battle, the function will return true.
- **public void fortify (Region fromRegion, Region toRegion)**: This function manages the relocation of troops in the “fortify” stage.
- **public void moveCommander(Region toRegion):** This function, in the “fortify stage”, will allow a player to move his/her commander independent of the soldiers in the current region.
- **public void updateTroopNum (Region defense, int defNumTroop, Region attack, int attackNumTroop)**: This method will be used to update the number of troops in the region of the player who defended that region, and the number of troops in the region of the player who attacked.
- **public void updateSeason(Region[] regions)**: This function will update the season of the regions based on the turn count.
- **public int decTroopNum(int numTroop):** This function will be used to decrease the number of troops in a region during a turn.
- **public compareDice(int[] dice1, int[]dice2):** This function will be used to compare dice results. dice1 is for attacker, and dice2 is for defender.

3.4.2.1.3. MapManager

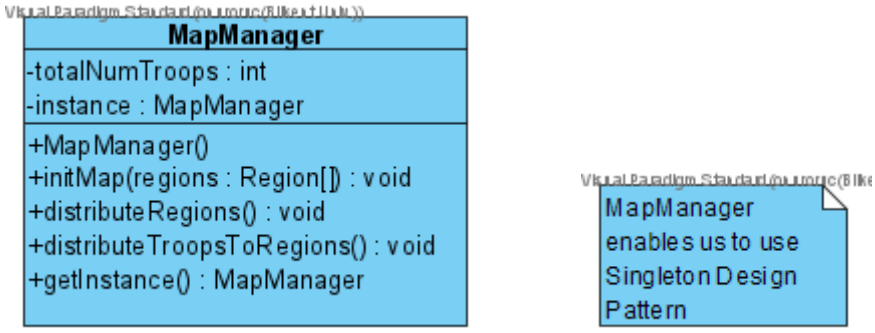

Figure 14: MapManager class with its attributes and operations

This class will be responsible for initializing the game map, and distributing the regions to players at the beginning of the game. Since this class will have a single instance during the whole game, we decided to apply a singleton design pattern.

Attributes:

- **private Region regions[]**: It will store the whole regions in the map.
- **private Player players[]:** It will store the players that are created in GameManager.
- **private Continent continents[]:** It will store the players that are created in GameManager
- **private int totalNumTroops:** It will store how many troops in total will be distributed at the beginning of the game.

Constructor(s):

- **public MapManager (Player[] player, Region[] regions, Continent[] continents)**
- The continents, regions and players that are initialized in the GameManager will be passed into the constructor.

Functions:

- **public void initMap (Region[] regions):** This function will add gold mines, troop cards, and climates to some regions.
- **public void distributeRegions():** This function will distribute regions to players at the beginning of the game.
- **public void distributeTroopsToRegions():** This function will distribute troops to regions. Moreover, it will initialize where the commander will be placed at the beginning of the game for each player.

3.4.2.2. Game Objects

3.4.2.2.1. *User*

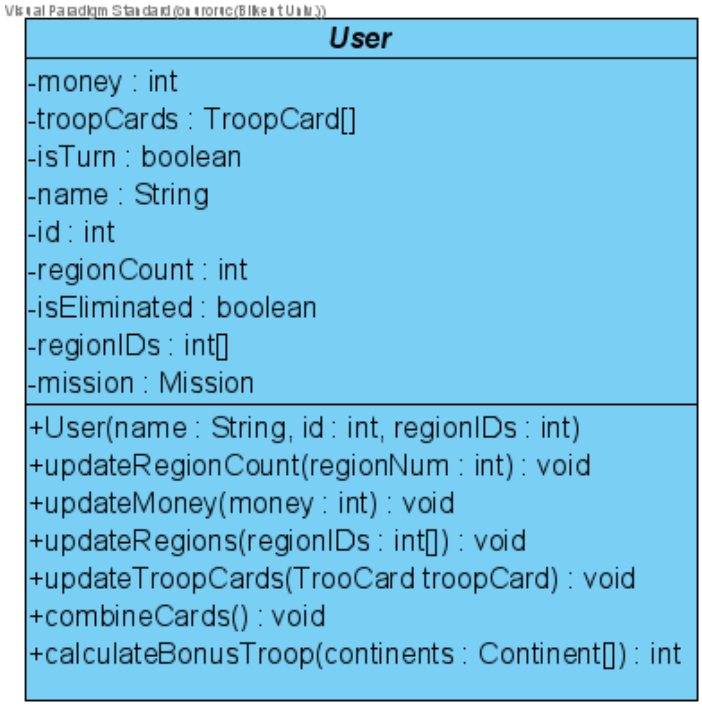

Figure 15: Abstract *User* class with its attributes and operations

We do not have bots in the game. However, it can be added in the future. Therefore, we decided to create a *User* class which is extended by Player class and which may be implemented by a possible bot class.

Attributes:

- **private TroopCard troopCards[]:** This attribute will store the troop cards that a player has.
- **private int money:** This attribute will store money of a player.
- **private boolean isTurn:** This attribute will be used to manage whose turn is on in the game.
- **private String name**: This attribute will store the player name.
- **private int id**: Every player will have a unique id.
- **private int regionCount**: Number of regions that a player owns.
- **private boolean isEliminated**: It will be used to check whether a player is eliminated from the game.

- **private int regionIDs**: It will store the region IDs that a player owns.
- **private Mission mission:** It will store the mission assigned at the beginning of the game.

Constructor(s):

- **public User(String name, int[] regionIDs):** It will initialize the player's name and the regions s/he owns. It will also give starting money to the player.

Functions:

- **public void updateRegionCount (int regionNum):** It updates the number of regions the player owns.
- **public void updateMoney(int money):** It will update the money that a player owns.
- **public void updateRegions(int[] regionIDs):** It will update the regions that a player owns based on the regionIDs given as a parameter.
- **public void updateTrooCards(TroopCard troopCard):** It will update the troop cards that a player has.
- **public void combineCards():** It will be used to combine troop cards (if possible). There are two possible combinations. One option is to combine three different troop cards. The other option is to combine three cards that are of the same type.
- **public int calculateBonusTroop(Continent[] continents):** This function will check whether a player has a continent(s). If there is a continent(s), it will increase the number of bonus troops according to the "bonusTroops" attribute of the continent object.

3.4.2.2.2. Player

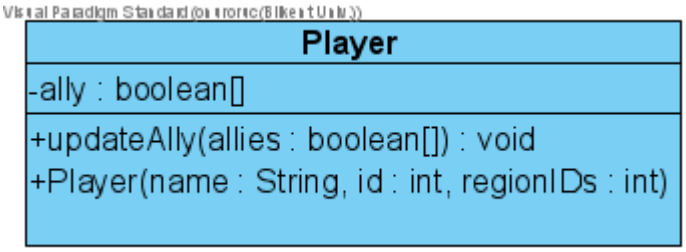

Figure 16: Player class which extends user class with its attributes and operations.

Player is a type of User. It has the following properties additional to *User* class.

Attributes:

- **private boolean ally[]:** This boolean array will be used to store a player's allies.

Constructor(s):

- **Player(String name, int id, int regionIDs):** It will initialize the name, id and region IDs that player has.

Functions:

- **public void updateAlly(boolean[] allies):** It will be used to break alliance or make alliance with other players.

3.4.2.2.3. TroopCard

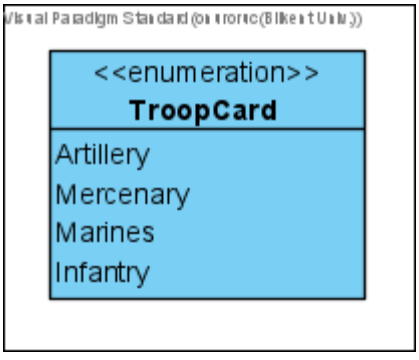

Figure 17: TroopCard enumeration class with its enums

This is an enum class that consists of four enums (Artillery, Mercenary, Marines, Infantry). It will be used to manage troop card related events in the game.

3.4.2.2.4. Region

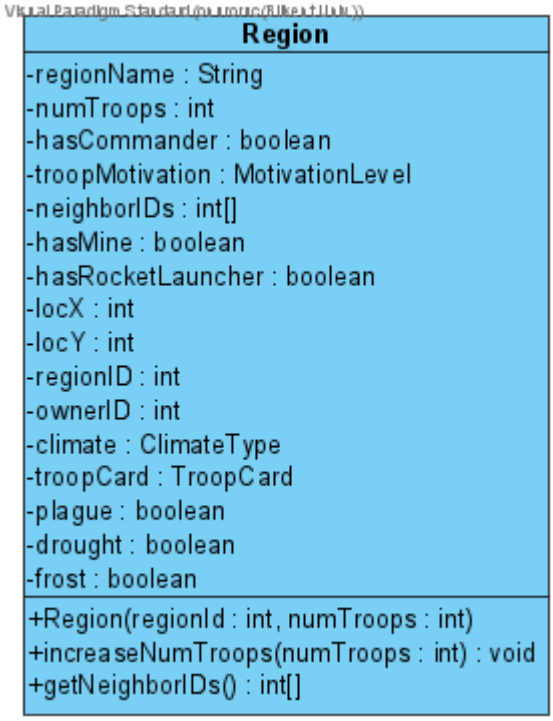

Figure 18: Region class with its attributes and operations

Attributes:

- **private String regionName:** It will store the name of the region.
- **private int numTroops:** It will store the number of troops in the region.
- **private boolean hasCommander**: It will store whether the region has a commander.
- **private MotivationLevel troopMotivation:** Each region will have a motivation level such as HighMotivated, NormalMotivated, LowMotivated, and Unmotivated.
- **private int neighborIDs[]**: It will be responsible for storing the IDs of neighbor regions.
- **private boolean hasMine**: It will store whether the region has a gold mine
- **private boolean hasRocketLauncher:** It will store whether the region has a rocket launcher.
- **private int locX**: X location of the region in the map.
- **private int locY**: Y location of the region in the map.
- **private int regionID**: Each region will have a unique id.
- **private int ownerID**: This is the player's id who owns this region.
- **private ClimateType climate**: Climate type of the region.

- **private TroopCard troopCard**: Troop card belonging to the region. This troop card will be displayed to the players in the map.
- **private boolean plague:** This will be used to keep track of whether a plague is active in the region.
- **private boolean dought:** This will be used to keep track of whether a drought is active in the region.
- **private boolean frost:** This will be used to keep track of whether a frost is active in the region.

Constructor(s):

- **public Region(int regionId, int numTroops):** It will initialize the region id, and the number of troops that will be in the region at the beginning of the game.

Functions:

- **public void increaseNumTroops(int numTroops):** It will update the number of troops that a region has.
- **public int[] getNeighborIDs():** It will return the neighbor region's IDs as an integer array.

3.4.2.2.5. ClimateType

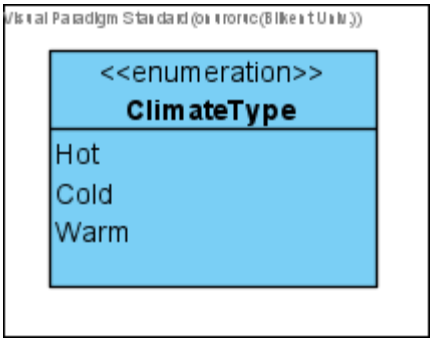

Figure 19: ClimateType enumeration class with its enums

This is an enum class that consists of three enums (Hot, Cold, Warm). It will be used to manage climates of regions.

3.4.2.2.6. Continent

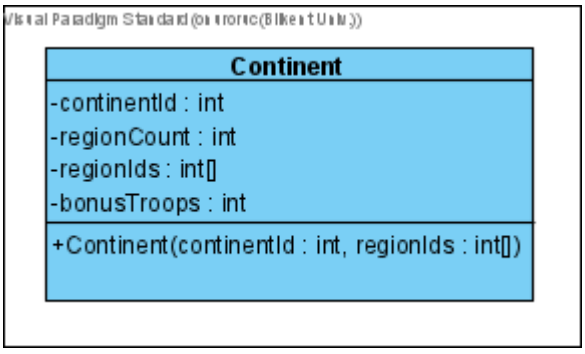

Figure 20: Continent class with its attributes and operations

Attributes:

- **private int continentId:** Every continent will have a unique id.
- **private int regionCount:** The number of regions that a continent has.
- **private int[] regionIds:** The region IDs that a continent has.
- **private int bonusTroops:** Number of the bonus cards that will be given to a player in case that the player owns this continent.

Constructor(s):

- **public Continent(int continentId, int[] regionIds):** The constructor will initialize the continent id, and the region IDs that form this continent.

3.4.2.2.7. SeasonType

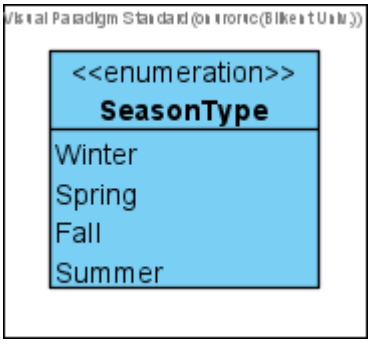

Figure 21: SeasonType enumeration class with its enums

This is an enum class that consists of four enums (Winter, Spring, Fall, Summer). It will be used to manage seasons in the game.

3.4.2.2.8. StageType

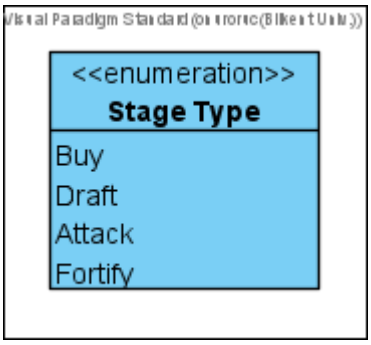

Figure 22: StageType enumeration class with its enums

This is an enum class that consists of four enums (Buy, Draft, Attack, Fortify). It will be used to manage stages in a single turn in the game.

3.4.2.2.9. DiceStrategy

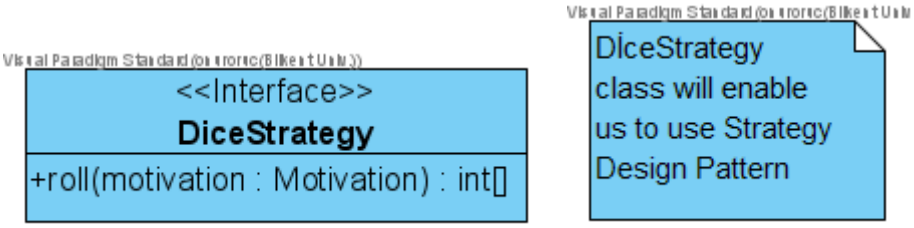

Figure 23: DiceStrategy interface with its single operation roll()

This is an interface required for the strategy pattern that we will apply to NormalDiceStrategy and MotivatedDiceStrategy classes.

Function:

- **public int[] roll (Motivation motivation):** This function is responsible for the calculations of dice according to the given motivation.

3.4.2.2.10. NormalDiceStrategy

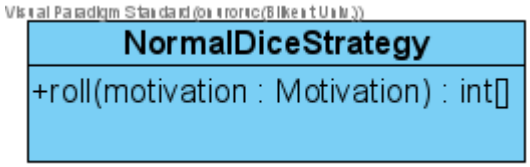

Figure 24: NormalDiceStrategy class with its operation

This is a class that implements the DiceStrategy interface to implement roll() function.

Function:

- **public int[] roll (Motivation motivation):** This function is responsible for the calculations of unmotivated dice.

3.4.2.2.11. MotivatedDiceStrategy

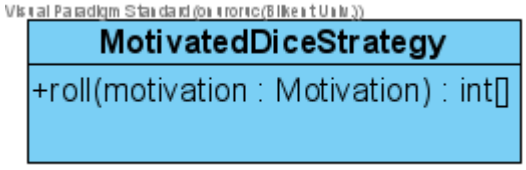

Figure 25: MotivatedDiceStrategy class with its operation

This is a class that implements the DiceStrategy interface to implement roll() function.

Function:

- **public int[] roll (Motivation motivation):** This function is responsible for the calculations of motivated dice according to the given motivation.

3.4.2.2.12. MotivationLevel

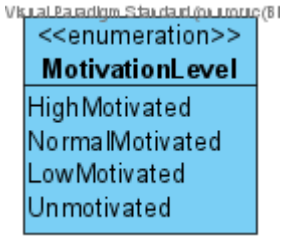

Figure 26: MotivationLevel enumeration class with its enums

This is an enum class that consists of four enums (HighMotivated, NormalMotivated, LowMotivated, Unmotivated). It will be used to manage motivation levels of regions.

3.4.2.2.13. Mission

Figure 27: Mission interface with its method

The function of this class will be used to check whether a player completed the mission. Since different missions can be created and given to players at the beginning of the game, the function to check the missions will also change. Since this difference happens in run time, we decided to use a strategy design pattern.

Function:

- **public boolean isCompleted(player: Player):** This function takes a player object to to check whether the mission has been completed by utilizing the player object's properties.

3.4.2.2.14. EliminateColorMission

Figure 28: EliminateColorMission class with its attributes and operations

This class will be responsible for the mission related to eliminating a specific player.

Attributes:

- **private int playerToEliminate:** It stores the player ID that will be eliminated.

Constructor(s):

- **EliminateColorMission (int playerID):** This will be used to initialize the player to be eliminated for this mission.

Functions:

- **public boolean isCompleted (Player player):** This function takes a player object to to check whether the mission has been completed by utilizing the player object's properties.

3.4.2.2.15. ConquerWorldMission

Figure 29: ConquerWorldMission class with its attributes and operations

This class will be responsible for the mission related to eliminating all players.

Attribute(s):

- **Set<Continent> continents[]:** This will be used to check whether the player who is responsible for this mission has all the continents in the game.

Constructor(s):

- **ConquerWorldMission(Set<Continent> continents):** This will be used to initialize the continents attribute which will be used to check whether the player who is responsible for this mission has all the continents in the game.

Functions:

- **public boolean isCompleted (Player player):** This function takes a player object to to check whether the player has all continents in the game.

3.4.2.2.16. ConquerNumRegionsMission

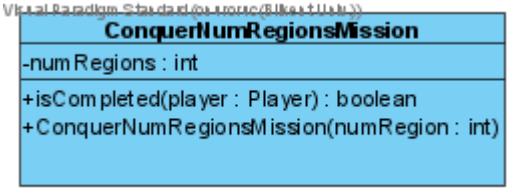

Figure 30: ConquerNumRegionsMission class with its attributes and operations

This class will be responsible for the missions that require the player to have a minimum number of regions to win.

Attribute(s):

- **int numRegions**: This represents the minimum number of regions that the player must have in order to win.

Constructor(s):

- **ConquerNumRegionsMission(int numRegions):** This constructor will be used to set numRegions variable.

Functions:

- **public boolean isCompleted (Player player):** This function takes a player object to to check whether the player satisfies the winning condition, which is to have a minimum number of regions.

3.4.2.2.17. ConquerContinentsMission

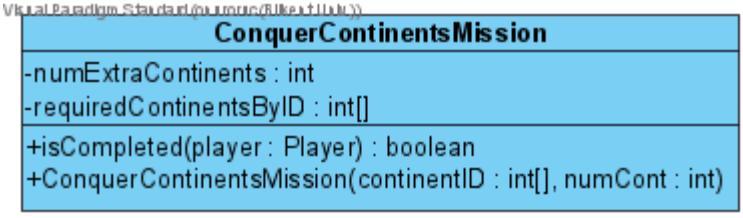

Figure 31: ConquerContinentsMission class with its attributes and operations

This class will be responsible for the missions that require the player to have some specific and some unspecific continents. (Like “Conquer Europe, Australia, and one other continent” mission)

Attribute(s):

- **int numExtraContinents**: This represents the minimum number of extra continents that the player must have in order to win.

- **int[] requiredContinentsByID:** This array holds the IDs of the continents that the player must have in order to win.

Constructor(s):

- **ConquerContinentsMission(int[] reqContByID, int numExtraContinents):** This constructor will be used to set properties of the class.

Functions:

- **public boolean isCompleted (Player player):** This function takes a player object to to check whether the player satisfies the winning condition, which is having some continents conquered.

3.4.3. Class Interfaces of Data Layer

3.4.3.1. FileManager

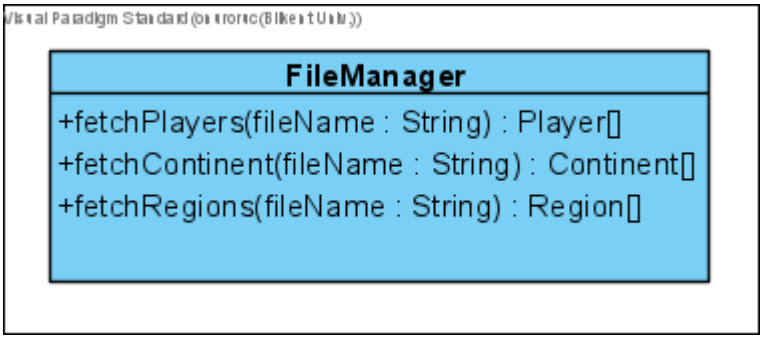

Figure 32: FileManager class with its three operations

This class will be responsible for fetching the player, continent, and region object data from text files.

Functions:

- **public void fetchPlayers(String fileName):** This function will fetch player data and create player objects.
- **public void fetchContinent(String fileName):** This function will fetch continent data and create continent objects.
- **public void fetchRegions(String fileName):** This function will fetch region data and create region objects.

Improvement Summary

When we were writing the second iteration of the report, we made some corrections based on the feedback for Design Report Iteration 1. While making corrections, we have added new things to this report. First of all, we have revised our object class model and we modified some classes in this model and added some new classes. Also, we have added some design patterns to our object class models which are Facade and Singleton Design Patterns. In the final class diagram, we have added DiceStrategy and Mission interfaces to implement Strategy Design Pattern. Then, we have changed our class interfaces accordingly. In the final class diagram, we have added User abstract class. This class has been added considering that a new user which is a bot might be added to the game in the future. Finally, we have eliminated an object design trade-of, because three layers architectural style does not affect the performance of the system.

4. Glossary & References

[1] Diagrams have been designed using the Visual Paradigm desktop application. Available: https://www.visual-paradigm.com/