CS319 - Object Oriented Software Engineering

Project Design Report

RISK

Group 2-B

Deniz Çalkan - 21703994

Irmak Akyeli- 21803690

Metehan Gürbüz - 21602687

Murat Sevinç - 21702603

Şebnem Uslu- 21802068

Yusuf Alpdemir - 21803035

1.Introduction

1.1 Purpose of the System

Risk is a multiplayer board game, which requires both luck and skill of the players. The purpose of our system is to create an online version of the Risk board game with brand new features while keeping the core features of the original game. The reason for adding new features is to make the game more strategic and more enjoyable for the players. To make the game more strategic, we want our game to focus more on the skills of the players rather than their luck.

1.2 Design Goals

1.2.1 Criteria

User-Friendliness

One of the first aspects that users notice about a software is GUI. So our Risk game will have a GUI that is aesthetically appealing to users and easy to navigate [1]. We will have a simple yet elegant GUI. For example, there won't be unnecessary buttons or labels on the screen and all buttons will have meaningful and simple titles such as ‘START’ button, ‘QUIT’ button, ‘HELP’ button and so on. Also, ‘HELP’ button will navigate the users throughout the game. The users will be able to press this button at any point of the game to get further information about how to play the game.

Another important aspect of a user-friendly software is whether the software can be easily installed and uninstalled [1]. Since we will be using Java to implement our game, users only need to have JRE installed on their computer to be able to play our game and they can easily remove our game by simply deleting the game file with the extension .jar.

Performance

A game with a good performance should have reasonable frame rate and short loading times. Our goal is to keep our game running with 60 FPS and keep our response time less than a second.

Portability

Our game will be portable to any operating system that supports JVM, since we chose to implement our game in Java and Java provides cross-platform desktop applications [2].

Reusability

Some of the classes in our game such as Dice and MainMenu classes can be used in other games without needing to completely change their implementation. These classes have core functionalities and can be easily used in other Java based software.

1.2.2 Trade-Offs

Memory vs Performance

We chose to implement our game in Java so we are not dealing with manual memory management because Java has an automatic garbage collector [3]. However, Java is relatively slow compared to other programming languages because Java code is compiled to an intermediary which is Java bytecode before it is converted to machine code [3].

Cost vs Portability

Since we don't have a budget, our game will be only available for desktop operating systems that support JVM. Moreover, our game won't be available for mobile operating systems such as IOS and Android.

Development Time vs Functionality

Since we have very limited time to implement our game, we aim our game to be able to perform core features of the original Risk game and some of our new features without making the game too complex [4].

Performance vs Security

Since risk is mostly an interactive game, our efforts are focused on the performance side. Also since in the game, there will be no sensitive data, security is not in the scope of our purpose.

1.3 Definitions, Acronyms and Abbreviations

Graphical User Interface (GUI): Computer program that interacts with the user through visual components [5].
Java Runtime Environment (JRE): Software layer that provides resources in order for Java programs to run. Also provides JVM [6].
Frame Per Second (FPS): Frequency of displayed images on the screen [7].
Java Virtual Machine (JVM): Virtual machine which the computer needs in order to run programs that are written in programming languages that are compiled to Java bytecode [8].

2. Software Architecture

2.1 System Decomposition

Our purpose in this section is decomposing our system into subsystems to reach the design goals of the project. During this process our purpose is to make the design of the project ideal which is with high coherence and low coupling.
Risk is an online game which includes many interactions with users. To provide **high performance** and to serve **many clients**, we have decided to choose **Client/Server architectural style**.

Risk is divided into Client and Server components. The Client component will consist of UsesInterface and UIController components. On the server side, Server component will consist of Data and DataController components.

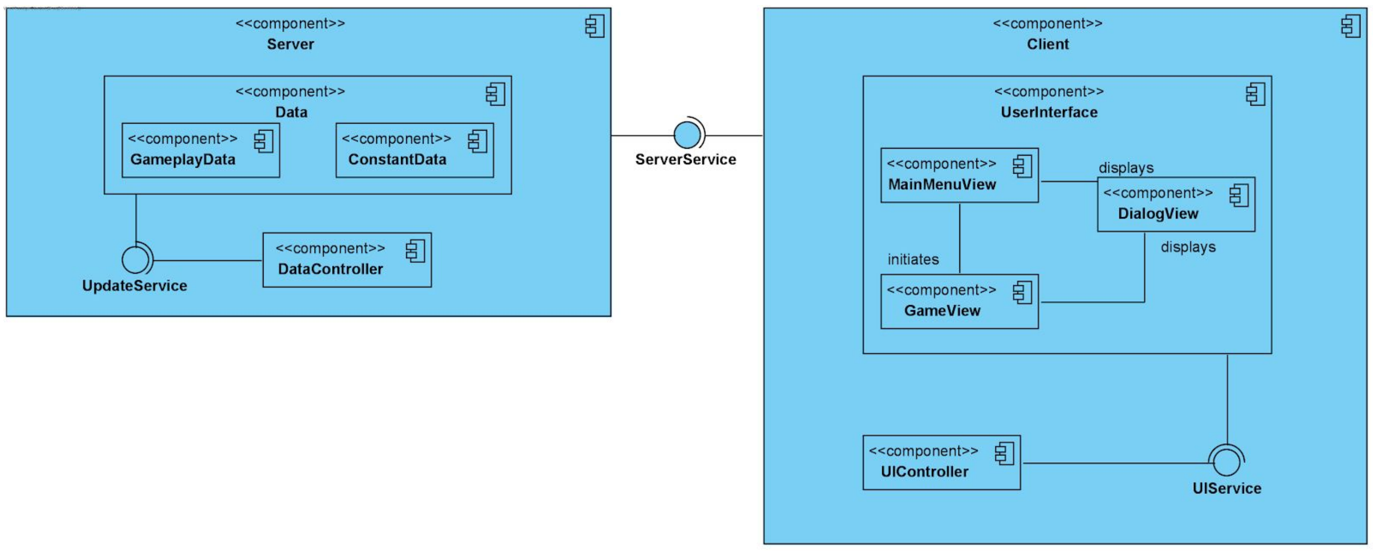

Figure 1: System Decomposition

2.2 Hardware/Software Environment and Dependencies

Our risk game will be implemented in Java Programming language. Therefore, a user will need to have JRE to be able to open the game. It will be downloadable as a jar file and will be easily uninstalled by just deleting the file. The server component will be installed in a **remote JSP Server** which is provided by Amazon EC2 (Amazon Elastic Compute Cloud, which provides computing in cloud). The communication with the server will be through **TCP/IP** as shown in the deployment diagram.

For hardware, a user will need a monitor to interact with visuals, a pc to install and enter the game, and an internet connection to reach the server which enables the user to play and interact with other players.

The game will run in any environment (Linux, Mac, Windows) that has JRE.

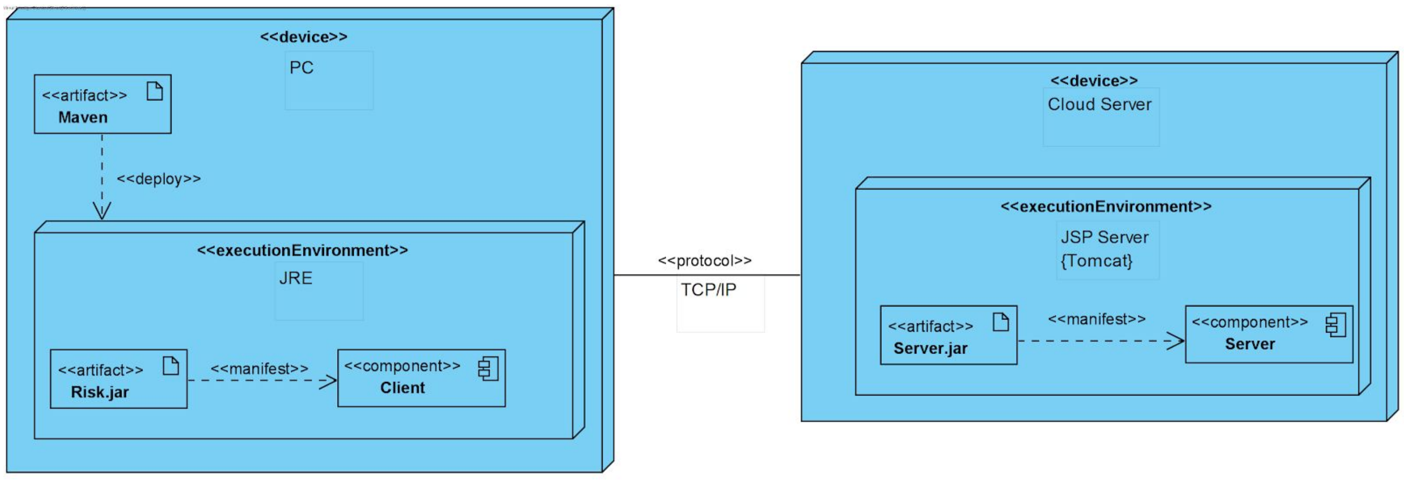

Figure 2:Deployment Diagram

2.3 Persistent Data Management

We store some necessary initialization and configuration data for our game engine in the form of JSON files such as player names, continents and regions. We use those configuration files to initialize our game engine on startup. However, we do not persist in any intermediate state of games. Therefore once the server is down any states corresponding to the running game are lost. Our game will not save any data for later circumstances. The images will be in .png format to enable transparency and sound files will be stored in .wav format.

2.4 Access Control and Security

Clients will constantly be interacting with our server through TCP/IP. External world will only access our encapsulated public methods therefore our sensitive data (such as structure of objects, important values) will be protected against external subjects. There will not be any authentication for users. When the app has opened, only a nickname is required. Hence, there will be no private or valuable data of users. So, security will not be our primary concern. Yet all the remaining security issues will be handled by the Server platform that we are using.

2.5 Boundary Conditions

2.5.1 Initialization

The moment a user opens the game, there will be a dialog asking for a nickname. After the input is provided, the app will wait until it creates a successful communication with the server, there will be a pop-up screen if the waiting time is more than 30 seconds; so, in the beginning stage a user must have an internet connection. Besides, in order to start the game, the client must have the image of the map, needed sound files, and default JSON file for the data of the map, menu, etc. All other communicative data will be provided by the server.

2.5.2 Termination

Users will be able to close the game by using the exit button on the corner. In the main menu there will also be another button to exit.

2.5.3 Failure

If an error occurs on the server side (an unexpected response or excessive response time, runtime error), the client side will close the current game and the user will be directed to the main menu.
If an error occurs on the client side, (unable to request, crash, connection loss); server side will not recieve and request and thus the current game will not be playable and the server will close the game, every client will be directed to the main menu.

3. Subsystem Services

3.1 Server Subsystem

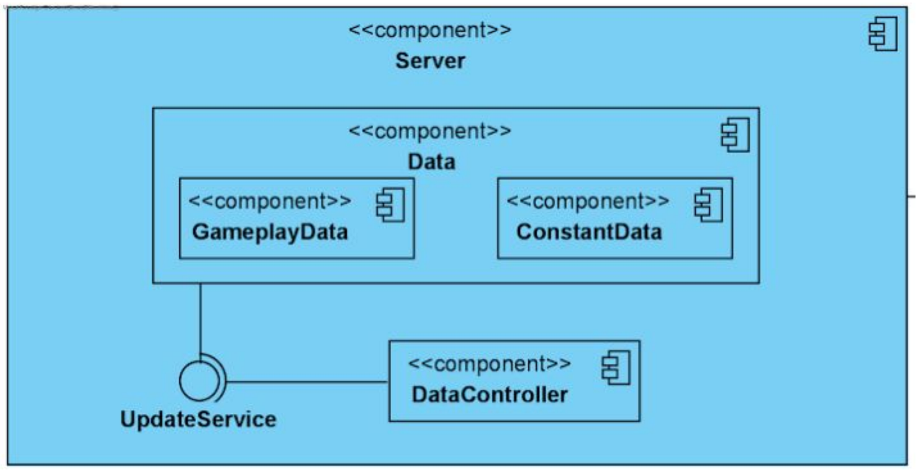

Figure 3: Server Subsystem

Server component consists of DataController component and Data component. ServerService enables the server to get requests from clients and respond back to clients after necessary processes.

Data component contains any data related to the game. ConstantData component basically holds data that does not change in a game, such as map configurations, player nicknames, and any related configuration data for the game. Whereas GameplayData component holds the data that constantly changes such as troop count on a region, room ID number etc. DataController is a component to communicate with necessary data. By this component, we are able to **boost the coherence** and **decrease coupling** of the system**.**

3.2 Client Subsystem

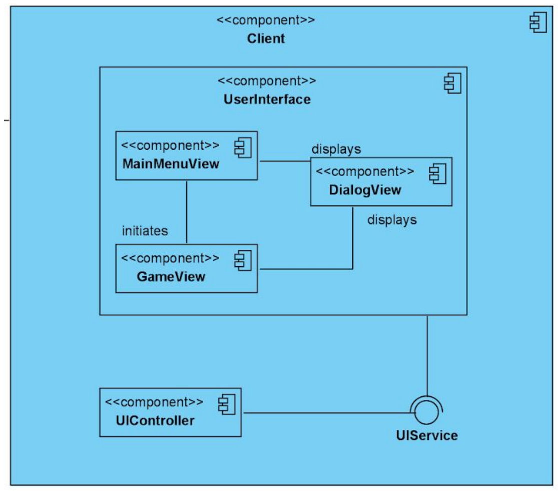

Figure 4: Client Subsystem

Client component consists of UIController and UserInterface components. UIController is able to communicate with both the user interface and the server. This again helps our system to have high coherence and low coupling. UIController basically adjusts the user interface accordingly to the responses from the server.

3.2.1 UserInterface Subsystem

UserInterface will have different types of view components: MainMenuView, DialogView, and GameView.

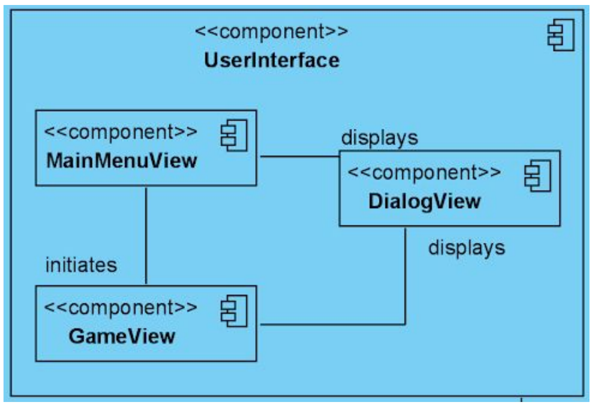

Figure 5: UserInterface Subsystem

- **MainMenuView:** This component will be responsible to display the main menu of the game.
- **DialogView:** In the game, there will be many cases where the user sees a dialog. This view component basically displays different types of dialogs such as nickname dialog (where we get a user's nickname), how to play dialog, and various warnings or errors like connection loss.
- **GameView:** This component is to provide the game screen to the interface. Main map of the game, regions, troop counts on regions, and animations are dealt by this component.

4. Low Level Design

4.1 Final Object Design

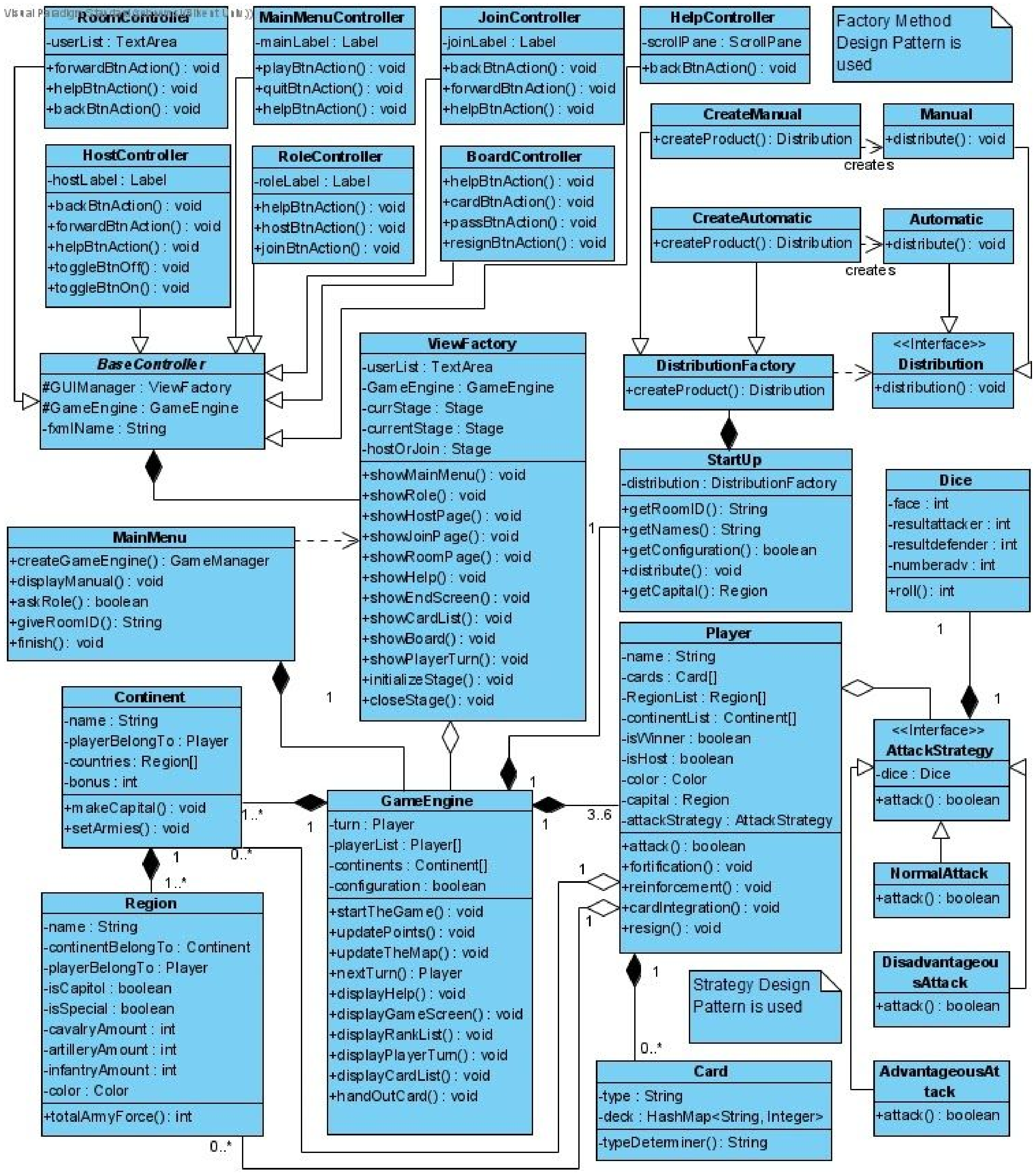

Figure 6: Final Object Design

4.2 Design Patterns

4.2.1 Facade Design Pattern

In the Application Layer, the GameEngine class is used as the Facade Design Pattern. By using Facade Design Pattern, the complex applications and interactions are masked to enhance the readability and usability. It provides the functionality and data for the controllers and views. The high level and unified interface offers an easy usage of the subsystems that it masks.

4.2.2 Factory Method Design Pattern

In the Application Layer, for the selection of the Distribution object to be created, the Factory Method Design Pattern is used. By using Factory Method Design Pattern, the problem of specifying the exact class of the object to be created is prevented. The StartUp class includes only one instance of the DistributionFactory which will initialize according to the choice of the host as either an automatic game or manual game. The children classes of the DistributionFactory class which act as the creators of the design pattern creates the related Distribution object whose role is the product in this design pattern. Each product implements the Distribution interface along with the distribute method which will differ between the product classes. After the creator children of the DistributionFactory creates one of them, the distribute method that the created Distribution object includes will be called in the StartUp so that the game begins with the configuration that the host chooses. Eventually, Factory Method Design Pattern allows us to use the desired distribute() method without specifying the exact class of it.

4.2.3 Strategy Design Pattern

In the Application Layer, for the selection of the attack method to be used, the Strategy Design Pattern is used. AttackStrategy interface is implemented by classes which define a family of attack algorithms. The Strategy Design Pattern helps to encapsulate each algorithm so that they can be interchangeable. The attack algorithm that will be used is defined by the children classes of AttackStrategy interface which are NormalAttack, DisadvantageousAttack, and AdvantageousAttack. The attack algorithm varies through them independently and the Player is able to use a different algorithm at each round. Eventually, Strategy Design Pattern allows us to bear a family of interchangeable and independent algorithms.

4.3 Layers

Our game has two main layers as discussed previously. The first layer is the Client layer that is visible by the user and the second is Server layer. The two layers always interact with each other and work together to give a good performance game. The client layer consists of user interfaces and GUI’s where the server layer consists of everything else.

4.3.1 Client Layer

In this layer we only transform the data obtained from the server to GUI and obtain the new information from the client. As our game is mostly operated on a server, this layer’s main objective is to create a communication between the user and the server layer. The biggest part of the client layer is the controller classes that are implementing façade design patterns to access all the controllers from the base controller. View factory, main menu and controller classes do every necessary interaction with the players and communicate with the server layer to implement the game.

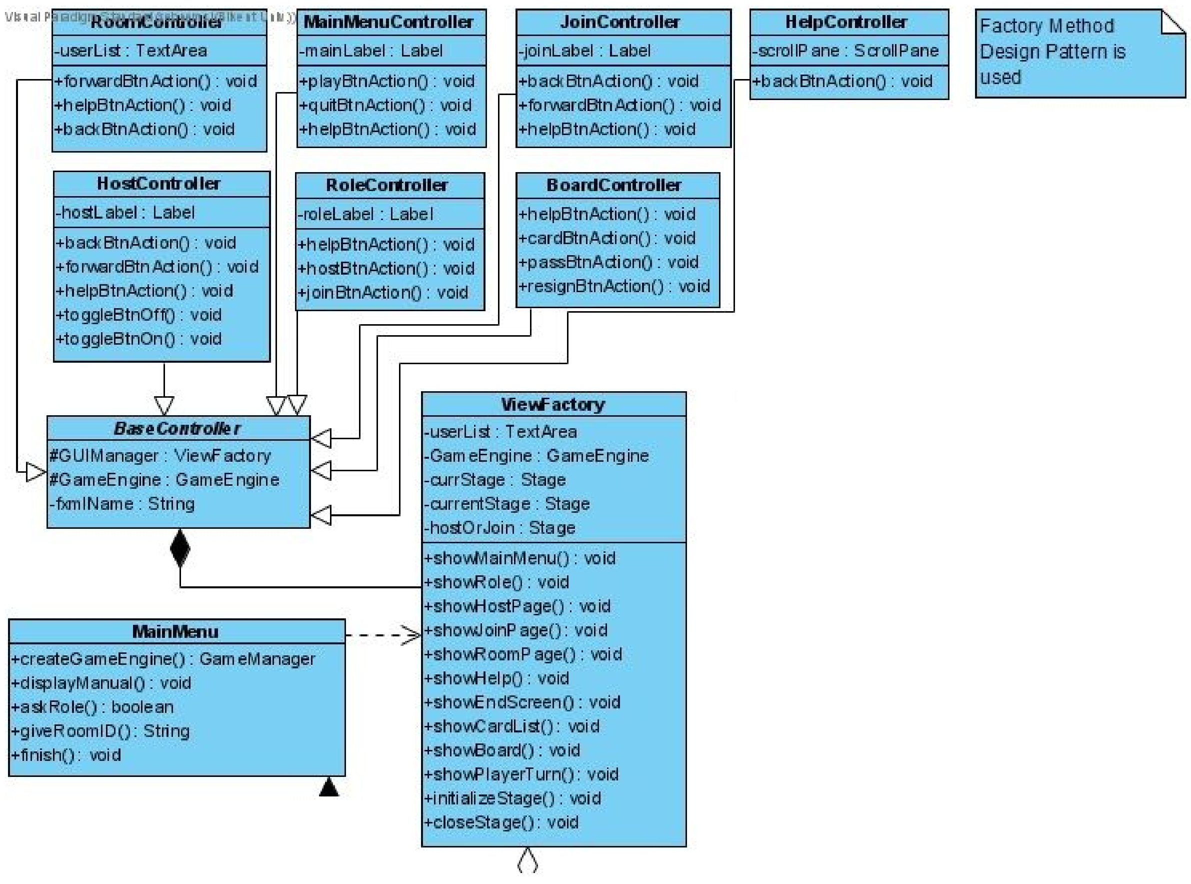

Figure 7: Client Layer

4.3.2 Server Layer

This layer is the most complicated layer and therefore three different design patterns are implemented on different areas. The façade design pattern can be seen through that almost all the classes are accessed through GameEngine and with this implementation we prevent further coupling and make it more controlled. However, this also reduces performance as we need to go through multiple classes in order to access some classes. The attack phase is a controller with a strategy pattern to enable different types of attack options and there is also a factory implementation for the distributions. This layer operates solely on the server and all the interactions with the users are held through the client layer.

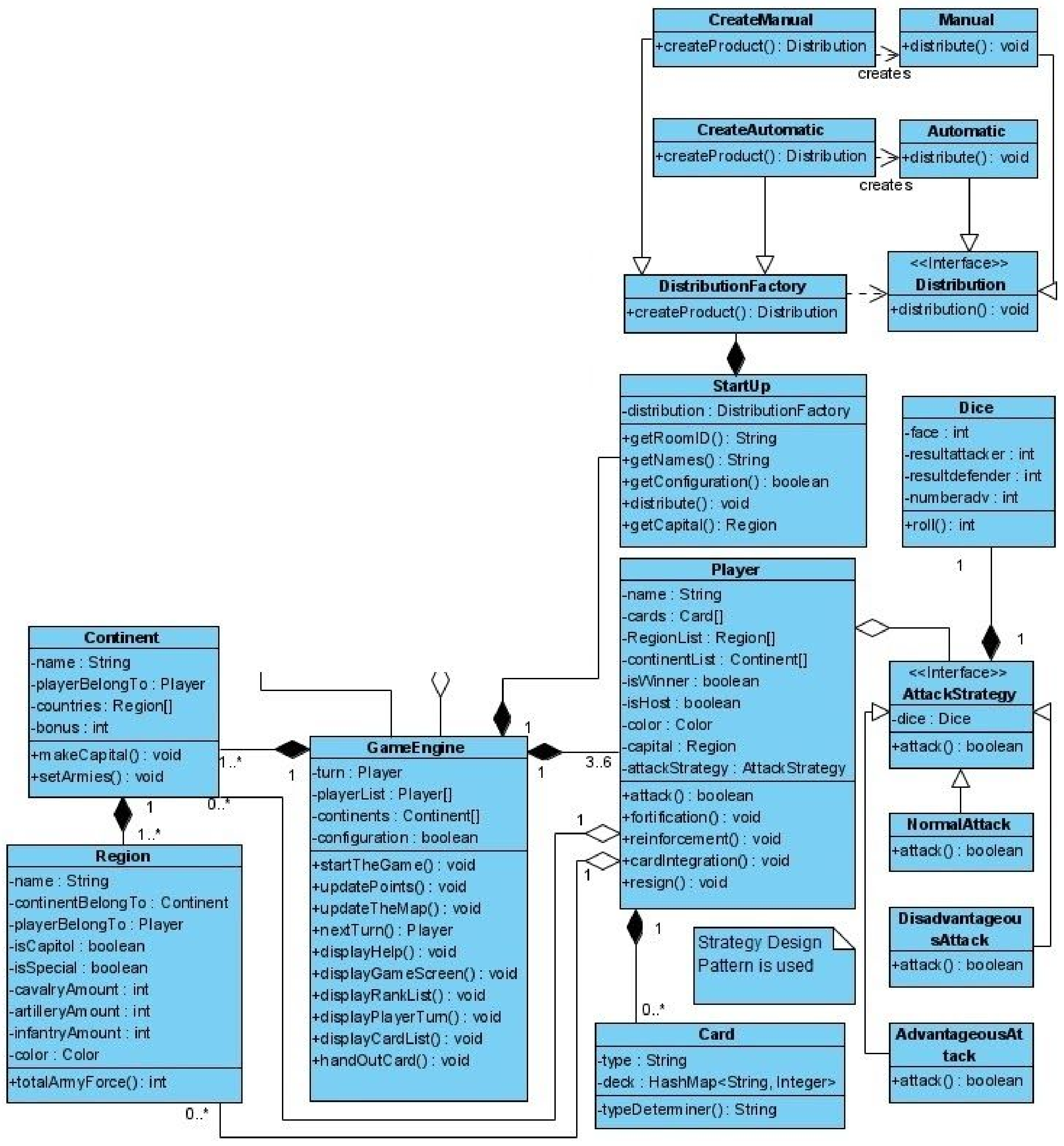

Figure 8: Server Layer

4.4 Packages

4.4.1 Internal Packages

- **Server Package**

  a) ServerController Package:
  This package consists of classes that are responsible for the management of the server.

  b) Data Package:
  Data package is responsible for static and dynamic data updates during the game.

- **Client Package**

  a) ClientController Package:
  ClientController manages client based processes.

  b) View Package:
  The View package deals with UI updates, processes. It is responsible for UI management.

4.4.2 External Packages

We use JavaFx in our project to conduct game related UI's and many other operations. As JavaFx is not included on JDE we needed to download it and included it in our game. With javafx.scene we do layouts and use javafx.event to handle objects and interactions. With many other utilities in the javaFx package it is an essential to our project.

4.5 Class Interfaces

4.5.1 Client Layer Class Interfaces

MainMenu Interfaces

MainMenu class handles the beginning of the game. It takes the input from the user for the options: Manuel, Play, and Exit. In order to offer the user this selection in a proper interface, it calls ViewFactory. If the user selects to play, it calls GameEngine.

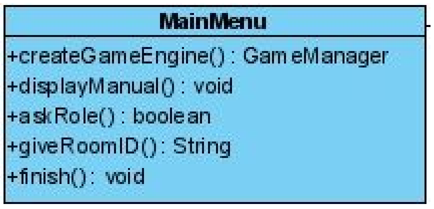

Figure 9: MainMenu Class

Operations:

- **public GameEngine createGameEngine():** It creates a GameEngine object to start the game.
- **public void showManual():** It calls displayHelp() from ViewFactory to display the Help screen.
- **public boolean askRole():** It calls displayHostOrJoin() from ViewFactory and takes the submitted choice of the user and returns it.
- **public String giveRoomID():** It attains an unique Room ID and returns it.
- **public void finish():** It stops the execution.

ViewFactory Interfaces

ViewFactory class has been established due to the necessity that emerged for the interaction between the players. ViewFactory provides the user interface by calling the relevant method. It is called by MainMenu and GameEngine.

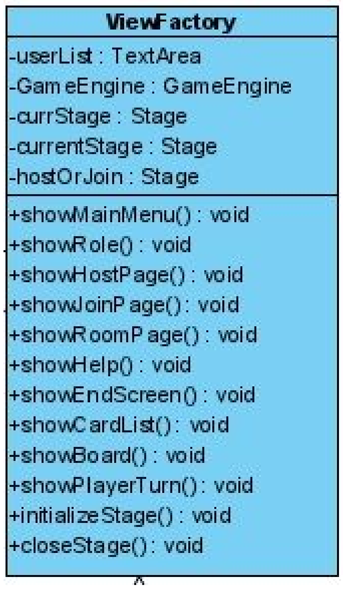

Figure 10: ViewFactory Class

Operations:

- **public void showMainMenu():** It displays the user interface of Main Menu.
- **public void showRole():** It displays the user interface of the option screen between the host or a participant.
- **public void showHostPage():** It displays the user interface of Host screen where the host selects the game mode and enters the player name.

- **public void showJoinPage():** It displays the user interface of Joiner screen where the participant enters the player name and the game code.
- **public void showRoomPage():** It displays the user interface of the Room screen where the players wait for others and the host can add more people.
- **public void showHelp():** It displays the user interface of Help screen.
- **public void showEndScreen():** It displays the user interface of Finish Screen where the winner is shown.
- **public void showCardList():** It displays the user interface of the Card List screen where the collected card of the player is seen above the integration request button.
- **public void showBoard():** It displays the user interface of Game Screen.
- **public void showPlayerTurn():** It displays the user interface of Whose Turn screen.
- **public void initializeStage(BaseController controller, String fxmlName):** It initializes a stage with the given controller and fxml file whose name is the parameter.
- **public void closeStage(Stage stage):** It closes the given stage.
- **public Stage getCurrentStage():** Returns the current stage.
- **public Stage getHostOrJoin():** Returns whether the player is in the join stage or host stage.

RoomController Interfaces

RoomController class has been established to provide functions to buttons of the Room window. RoomController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

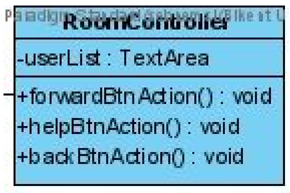

Figure 11: RoomController Class

Attributes:

- **private TextArea userList:** The purpose of this label is to provide the class an attribute that we can extract the window it belongs to and most importantly show for other players the rest of the participants of the game.

Operations:

- **void forwardBtnAction():** It calls the showBoard() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.
- **void backBtnAction():** Returns to the join stage if the player is joining the game and returns to the host stage if the player is the host.
- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

RoleController Interfaces

RoleController class has been established to provide functions to buttons of the Role Choosing window. RoleController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

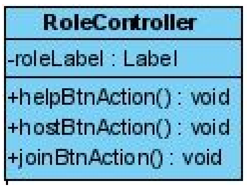

Figure 12: RoleController Class

Attributes:

- **private Label roleLabel:** The purpose of this label is to provide the class an attribute that we can extract the window it belongs to.

Operations:

- **void hostBtnAction():** It calls the showHostPage() from ViewFactory to open the Host window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.
- **void joinBtnAction():** It calls the showJoinPage() from ViewFactory to open the Join window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.
- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

JoinController Interfaces

The JoinController class has been established to provide functions to buttons of the Join window. JoinController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

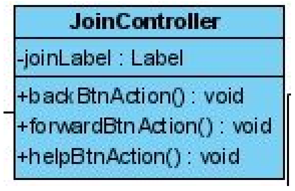

Figure 13: JoinController Class

Attributes:

- **private Label joinLabel:** The purpose of this label is to provide the class an attribute that we can extract the window it belongs to.

Operations:

- **void backBtnAction():** It calls the showRole() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void forwardBtnAction():** It calls the showRoomPage() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

HostController Interfaces

HostController class has been established to provide functions to buttons of the Host window. HostController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

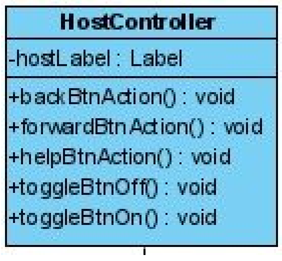

Figure 14: HostController Class

Attributes:

- **private Label hostLabel:** The purpose of this label is to provide the class an attribute that we can extract the window it belongs to.

Operations:

- **void backBtnAction():** It calls the showRole() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void forwardBtnAction():** It calls the showRoomPage() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void toggleBtnOn():** Sets game configuration as automatic.

- **void toggleBtnOff():** Sets game configuration as not automatic.

MainMenuController Interfaces

MainMenuController class has been established to provide functions to buttons of the Main Menu window. MainMenuController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

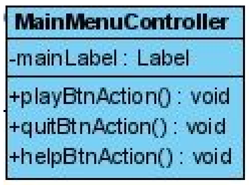

Figure 15: MainMenuController Class

Attributes:

- **private Label mainLabel:** The purpose of this label is to provide the class an attribute that we can extract the window it belongs to.

Operations:

- **void playBtnAction():** It calls the showRole() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void quitBtnAction():** The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

HelpController Interfaces

HelpController class has been established to provide functions to buttons of the Help window. HelpController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

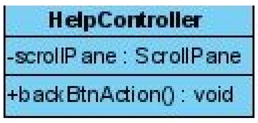

Figure 16: HelpController Class

Attributes:

- **private ScrollPane scrollPanel:** The purpose of this ScrollPane is to provide the class an attribute that we can extract the window it belongs to but more importantly, it allows the user to paddle throughout the screen which enhances user friendliness. With the help of the scroll pane, a long text can be screened with a proper size in an area whose size is a couple of screens.

Operations:

- **void backBtnAction():** It calls the showMainMenu() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

BoardController Interfaces

BoardController class has been established to provide functions to buttons of the Game window. BoardController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

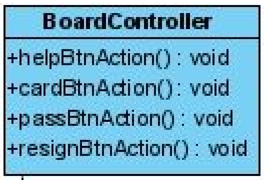

Figure 17: BoardController Class

- **void helpBtnAction():** It calls the showHelp() from ViewFactory to open the next window. The function accesses the window of the action requested by the class label and closes the window by close() function of ViewFactory.

- **void passBtnAction():** It enables the user to switch states of attack, fortification, and reinforcement.

- **void resignBtnAction():** It enables the player to quit the game gracefully by resigning.

- **void cardBtnAction():** It enables the user to open the card list pop-up that s/he possesses.

BaseController Interfaces

BoardController class has been established to provide functions to buttons of the Game window. BoardController manages the attributes and actions related to the window that it controls. It extends the abstract class BaseController and uses its properties to call the view functions.

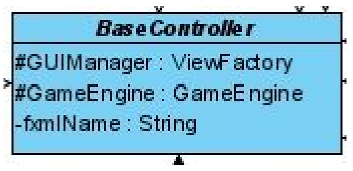

Figure 18: BaseController Class

Attributes:

- **protected GameEngine GameEngine:** The GameEngine attribute allows the children classes of BaseController to access model classes which are located after the GameEngine bridge. Since we wish children classes to use this attribute, it is classified as protected.

- **protected ViewFactory ViewFactory:** The ViewFactory attribute functions as a bridge from the code to the user. The children classes of BaseController can access the view functions where located at ViewFactory via ViewFactory instance. Since we wish children classes to use this attribute, it is classified as protected.

- **private String fxmlName:** The fxmlName is created due to the necessity of passing the related fxml file's name to the stage in the ViewFactory so that each window would open its unique user interface.

4.5.2 Server Layer Class Interfaces

GameEngine Interfaces

GameEngine class is the main controller of the game. It is created when the MainMenu calls it after the player selects the "Play" option. Other attributes of the game such as Player, Continent, Card, and StartUp are attached to it and created if GameEngine calls them. Furthermore, GameManager calls ViewFactory so that the changes upon the game can be reflected to the player through the user interface that ViewFactory provides.

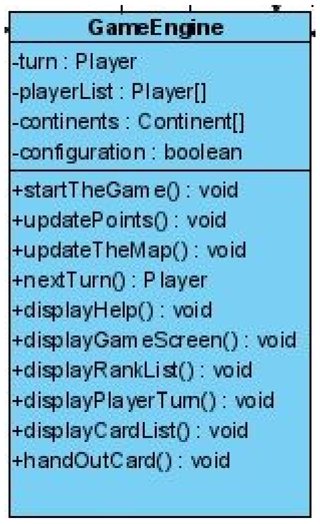

Figure 19: GameEngine Class

Attributes:

· **private Player[] playerList:** Holds the list of Players on the game.

· **private boolean configuration:** Holds the game mode.

· **private Continent[] continentList:** Holds the list of continents in a way of a map.

· **private Player turn:** Holds the Player that is currently playing.

Operations:

· **public void startTheGame():** It initiates the game by initializing attributes.

· **public void updatePoints():** It updates the points of each Player.

· **public void updateTheMap():** It updates the map according to the latest movements.

· **public Player nextTurn():** It passes to the next Player, calls displayPlayerTurn() from ViewFactory, and returns to the Player who will play in the next round.

· **public void displayHelp():** It calls displayHelp() from ViewFactory.

· **public void displayGameScreen():** It calls updateTheMap() and updatePoints() within itself and calls displayGameScreen() from ViewFactory.

· **public void displayRankList():** It calls updatePlayerRank() from ViewFactory to show the ranking bar.

· **public void displayPlayerTurn():** It calls displayPlayerTurn() from ViewFactory to show Whose Turn screen.

- **public void displayCardList():** It calls displayCardList() from ViewFactory to show Card List window.
- **public void handOutCard(Player aPlayer):** It attains cards to the earning player.

DistributionFactory Interfaces

The DistributionFactory class was designed for the soldier distribution options. Classes that extend this class creates the related class option which determines the distribution method functionality which differs as automatically or manually, and returns the created class instance.

DistributionFactory
+createProduct() : Distribution

Figure 20: DistributionFactory Class

Operations:

- **public Distribution createProduct():** It waits to be overwritten by the children classes.

CreateAutomatic Interfaces

CreateAutomatic class was designed for the automatically soldier distribution options. It calls and creates an instance of Automatic class and returns the object of it to be used in the StartUp class.

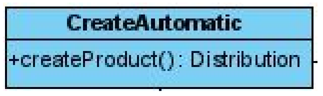

Figure 21: CreateAutomatic Class

Operations:

- **public Distribution createProduct():** It creates and returns an object of Automatic class.

CreateManual Interfaces

CreateManual class was designed for the manually soldier distribution options. It calls and creates an instance of Manual class and returns the object of it to be used in the StartUp class.

CreateManual
+createProduct() : Distribution

Figure 22: CreateManualClass

Operations:

· **public Distribution createProduct():** It creates and returns an object of Manual class.

Distribution Interfaces (Interface)

Distribution interface was designed for the soldier distribution options. Classes that implement this interface determine the distribution method functionality which differs as automatically or manually. The interface usage gives clean and less complicated codes for us.

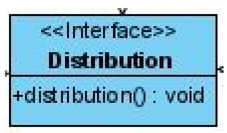

Figure 23: Distribution Interface

Operations:

· **public void distribute():** It waits to be overwritten by the children classes.

Automatic Interfaces

Automatic class was designed for the automatic soldier distribution option. If the host selects the "On" on the toggle switch, StartUp will call DistributionFactory to create the Automatic Class instance which will result in the distribute() method at the StartUp class to distribute soldiers automatically.

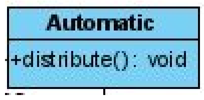

Figure 24: Automatic Class

Operations:

· **public void distribute():** It initially distributes army forces automatically.

Manual Interfaces

The Manual class was designed for the manual soldier distribution option. If the host selects the "Off" on the toggle switch, StartUp will call DistributionFactory to create the Manual Class instance which will result in the distribute() method at the StartUp class to distribute soldiers manually.

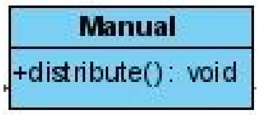

Figure 25: ManualClass

Operations:

- **public void distribute(Region region):** It initially distributes army forces manually for the given Region.

StartUp Interfaces

StartUp class was designed for the beginning functions of the game. If the player selects the "Play" option, GameEngine will call StartUp to determine player names, configuration status and the initialization of the game.

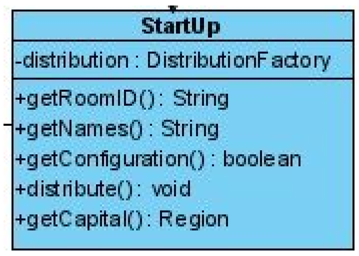

Figure 26: Startup Class

Attributes:

- **private distribution DistributionFactory:** Holds the distribution choice.

Operations:

- **public void distribute(Region toRegion):** It initially distributes army forces to the Region which is given.
- **public String getNames():** It gets the player name from the user and returns it.
- **public boolean getConfiguration():** It gets the game mode from the host and returns it.
- **public String getRoomID():** It gets the room ID from the participant and returns it.
- **public Region getCapital():** It gets the selection of capital Region from the participant and returns it.

Card Interfaces

Card class represents a Card that will be drawn within the game. Cards can provide cavalry, infantry... etc. Thus, a card should have a property that indicates the type. The type of the card should be random so that the player cannot count the cards to determine what will come. typeDeterminer() provides a random card to be created. Because this method is only used when the card is created which is in the constructor, the method will stay private.

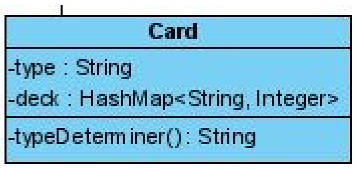

Figure 27: Card Class

Attributes:

· **private String type:** Holds the type of the Card.

· **private HashMap<String, Integer> deck:** Holds the latest deck with the keys as card types and values as the available number of cards belong to that type.

Operations:

· **private String typeDeterminer():** It randomly determines the type of the card within the constructor.

Player Interfaces

Player class will be created when the Game Manager calls it. Thus, the existence of the Player is dependent on the GameEngine. If there is no game to play, there will be no player. The player has multiple properties as well as methods. Player will have the attributes of a name, a Region list to track down the conquered countries, the continent list so that at the next round, the player can receive a bonus for conquering an entire continent, a boolean to determine whether the exact player is the winner one or not. Lastly, the rank of the player.

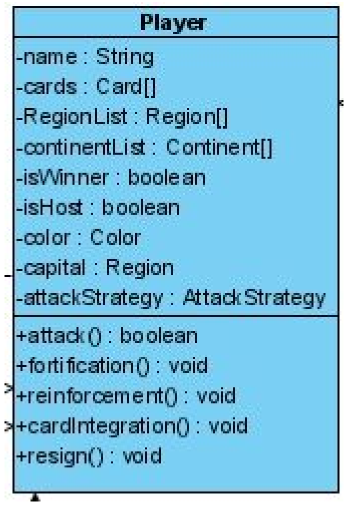

Figure 28: Player Class

Attributes:

- **private String name:** Holds the information of name.
- **private Card[] cards:** Holds a list of Cards that Player possesses.
- **private Region[] RegionList:** Holds a list of Countries that Player invaded.
- **private Continent[] continentList:** Holds a list of Continents that Player invaded.
- **private boolean isWinner:** Holds whether the information of the Player is the winner.
- **private boolean isHost:** Holds whether the information of the Player is the host.
- **private Color color:** Holds the color of the Player.

Operations:

- **public void attack(Region attackerRegion, Region attackedRegion):** It performs the attacking action with the given parameters.
- **public void fortification(Region selectedRegion, Region soldiersAdded):** It performs fortification with the given parameters.
- **public void reinforcement(Region fromRegion, Region toRegion):** It performs reinforcement with the given parameters.
- **public void cardIntegration():** It integrates the card and updates the card list accordingly.
- **public void resign():** It enables the Player to quit the game gracefully.

AttackStrategy Interfaces (Interface)

The AttackStrategy interface was designed to be able to use strategy design patterns in our attacking component since there are different types of attacks. Payer class is the class that uses this strategy.

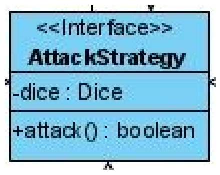

Figure 29: AttackStrategy Interface

Attributes:

- **Dice dice:** Dice to be rolled.

Operations:

· **public boolean attack():** Implementation will be done in child classes.

NormalAttack Interfaces

Normal attack class was created to perform attack strategy when there is no advantage for either side (attacker and attacked).

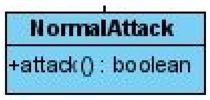

Figure 30: NormalAttack Class

Operations:

· **public boolean attack():** Attack method calls Dice class' roll method with the right advantage indicator.

DisadvantageousAttack Interfaces

Disadvantageous attack class was created to perform attack strategy when there is an advantage for the attacked side.

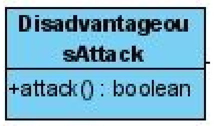

Figure 31: Disadvantageous Class

Operations:

· **public boolean attack():** Attack method calls Dice class' roll method with the right advantage indicator.

AdvantageousAttack Interfaces

Advantageous attack class was created to perform attack strategy when there is an advantage for the attacker side.

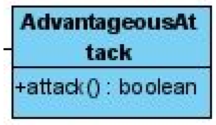

Figure 32: AdvantageousAttack Class

Operations:

· **public boolean attack()**: Attack method calls Dice class' roll method with the right advantage indicator.

Region Interfaces

The creation of the Region class is dependent on the existence of the Continent class. If only if the Continent class is ever created, a list of Region objects will be created. Region class has 8 attributes: the name of the Region, the owner of the Region, the continent that the Region is within, is it a capital or is it a special city like Rome, and lastly, the numbers of three different kinds of soldiers deployed in the Region. There is only one method that this class contains which calculates the total army force according to the amount of the different types of soldiers.

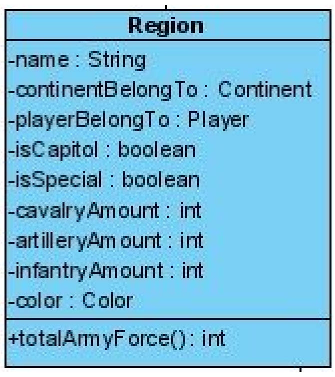

Figure 33: Region Class

Attributes:

· **private String name:** Holds the name of the Continent.

· **private Continent continentBelongTo:** Holds the Continent that Continent in.

· **private Player playerBelongTo:** Holds the Player that invaded the Region.

· **private boolean isCapitol:** Holds the information of whether the Region is a capitol.

· **private boolean isSpecial:** Holds the information of whether the Region is a special.

· **private int cavalryAmount:** Holds the number of cavalries deployed in the Region.

· **private int artilleryAmount:** Holds the number of artilleries deployed in the Region.

· **private int infantryAmount:** Holds the number of infantries deployed in the Region.

· **private Color color:** Holds the current color of the Region.

Operations:

· **public int totalArmyForce():** It calculates the total army force in the Region.

Continent Interfaces

The creation of the Continent class is dependent on the existence of the GameEngine class. If only if GameEngine class is ever created, a list of Continent objects will be created. Continent class has 3 attributes: the name of the continent, the owner of the continent, and the list of countries that the continent includes within. This class works as the board of the game which is the map.

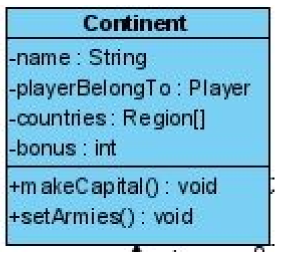

Figure 34: Continent Class

Attributes:

- **private String name:** Holds the name of the Continent.
- **private Player playerBelongTo:** Holds the invader Player of the Continent.
- **private Region[] RegionList:** Holds a list of Countries that the Continent encapsulates.
- **private int bonus:** Holds the bonus point that Player will get if s/he conquers the continent.

Dice Interfaces

The existence of the Dice class is dependent on the AttackStrategy interface, since if there is no attacking there is no need for the dice. Dice class takes two army and an advantage indicator and returns the army lost from a side after roll function.

Attributes:

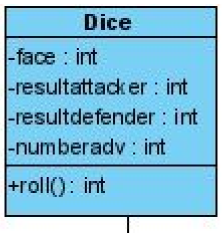

Figure 35: Dice Class

- **private final int face:** Holds the value of the dice which is 100 in our game.
- **private int resultattacker:** Holds the value of the result for the attacker Player.

- **private int numberadv:** Holds the advantage for either side as an indicator.
- **private int resultdefender:** Holds the value of the result for the defender Player.

Operations:

- **public int roll(attackingArmy, attackedArmy, advantage):** Takes the number of each side of the army and the indicator advantage showing if one side has advantage or not and returns the army lost after a single attack.

Glossary and References

[1] Wallen, J., 2010. *10 Things That Make Software User-Friendly*. [online] TechRepublic. Available at: <https://www.techrepublic.com/blog/10-things/10-things-that-make-software-user-friendly/> [Accessed 26 November 2020].

[2] En.wikipedia.org. 2020. *Cross-Platform Software*. [online] Available at: <https://en.wikipedia.org/wiki/Cross-platform_software> [Accessed 26 November 2020].

[3]McKenzie, C., 2019. *Is Java Slow? Compared To C++, It's Faster Than You Think*. [online] TheServerSide.com. Available at: <https://www.theserverside.com/opinion/Is-Java-slow-Compared-to-C-its-faster-than-you-think#:~:text=In%20fact%2C%20when%20compared%20against,features%20of%20its%20underlying%20architecture.> [Accessed 28 November 2020].

[4] Kanat-Alexander, M., 2008. *The Goals Of Software Design » Code Simplicity*. [online] Code Simplicity. Available at: <https://www.codesimplicity.com/post/the-goals-of-software-design/#:~:text=From%20the%20purpose%20of%20software,of%20software%20design%20should%20be%3A&text=To%20design%20systems%20that%20can,be%E2%80%93as%20helpful%20as%20possible.> [Accessed 26 November 2020].

[5] Levy, S., 2018. *Graphical User Interface | Computing*. [online] Encyclopedia Britannica. Available at: <https://www.britannica.com/technology/graphical-user-interface> [Accessed 26 November 2020].

[6] Ibm.com. 2020. *What Is The JRE (Java Runtime Environment)?*. [online] Available at: <https://www.ibm.com/cloud/learn/jre> [Accessed 29 November 2020].

[7] En.wikipedia.org. 2020. *Frame Rate*. [online] Available at: <https://en.wikipedia.org/wiki/Frame_rate#:~:text=Frame%20rate%20(expressed%20in%20frames,and%20be%20expressed%20in%20hertz.> [Accessed 26 November 2020].

[8]En.wikipedia.org. 2020. *Java Virtual Machine*. [online] Available at: <https://en.wikipedia.org/wiki/Java_virtual_machine> [Accessed 27 November 2020].