Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Term Project

CS 319 3G: RISK

Design Report

Group Members:

Işık Özsoy 21703160

Defne Betül Çiftci 21802635

Burak Yetiştiren 21802608

Alp Üneri 21802481

Mustafa Hakan Kara 21703317

Instructor: Eray Tüzün
Teaching Assistant(s): Elgun Jabrayilzade, Emre Sülün, Barış Ardıç

Progress/Final Report

Nov 29, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object-Oriented Software Engineering course CS319

.

Table of Contents

Introduction **5**
1.1 Purpose of the System 5
1.2 Design Goals 6
1.2.1 Reliability 6
1.2.2 Modifiability 6
1.2.3 Maintainability 6
1.2.4 Understandability 6
1.2.5 Adaptability 7
1.2.6 Reusability 7
1.2.7 Efficiency 7
1.2.8 Portability 7
1.2.9 Fault Tolerance 7
1.2.10 Backwards-Compatibility 7
1.2.11 Cost-Effectiveness 7
1.2.12 Robustness 8
1.2.13 Functionality 8
1.2.14 Usability 8
1.2.15 Readability 8
1.2.16 Security 8
1.2.17 Rapid Development 9
1.3 Trade-Offs 9
1.3.1 Functionality versus Usability 9
1.3.2 Cost versus Robustness 9
1.3.3 Efficiency versus Portability 9
1.3.4 Rapid Development versus Functionality 9
1.3.5 Cost versus Reusability 9
1.3.6 Backwards Compatibility versus Readability 10
1.3.7 Security versus Usability 10
1.4 Definitions, Acronyms, and Abbreviations 10
1.4.1 Java Virtual Machine (JVM) 10
1.4.2 Model View Controller (MVC) 10
1.4.3 Graphical User Interface (GUI) 10
1.4.4 Personal Computer (PC) 10
1.4.5 Java Runtime Environment (JRE) 10
1.5 Overview 11

Current Software Architecture **11**

Proposed System Architecture **11**
3.1 Overview 11

3.2 Subsystem Decomposition 12
3.2.1 UI Interface Management Subsystem 13
3.2.2 Game Logic Management Subsystem 13
3.2.3 File System Management Subsystem 13
3.3 Hardware/Software Mapping 13
3.4 Persistent Data Management 14
3.5 Access Control and Security 14
3.6 Global Software Control 14
3.7 Boundary Conditions 15
3.7.1 Initialization 15
3.7.2 Termination 15
3.7.3 Exception Handling 15

Subsystem Services 16
4.1 UI Interface Management Subsystem 16
4.1.1 Menu View Component 16
4.1.2 Map View Component 16
4.1.3 Rock-Paper-Scissors View Component 16
4.1.4 Pause Menu View Component 16
4.2 Game Logic Management Subsystem 17
4.2.1 Player Management Component 17
4.2.2 Map Management Component 17
4.3 File System Management Subsystem 18

Low Level Design 18
5.1 Object Design Trade-Offs 18
5.1.1 Memory Consumption versus Visual Quality and Complexity of the System 18
5.1.2 Rapid Development versus Code Navigability 18
5.1.3 Number of Options versus Simplicity 18
5.1.4 Functionality versus Understandability 19
5.2 Object Design Patterns 19
5.3 Final Object Design 19
5.3.1 Game Logic Subsystem 20
Player Class 23
Attributes 23
Constructors 24
Methods 24
Continent 24
Attributes 24
Constructor 25
Methods 25
Territory 25

Attributes 25
Constructor 25
Methods 25
Card 26
Attributes 26
Constructor 26
Methods 26
Cards 26
Attributes 26
Constructor 26
Methods 27
CardType (Enumeration) 27
5.3.2 User Interface Subsystem 28
MainMenuView 29
Attributes 29
Constructors 29
Methods 29
AddPlayersView 31
Attributes 31
Constructors 31
Methods 31
PlayerButton 32
Attributes 32
Constructors 33
Methods 33
GameView 34
Attributes 34
Constructors 34
Methods 34
ClickableTerritory 35
Attributes 35
Constructors 35
Methods 35
RPSGameView 36
Attributes 36
Constructors 37
Methods 37
Main 37
Attributes 37
Methods 37
MainMenuText 38

Attributes 38
Constructors 38
BeforeGameView 38
Attributes 38
Constructors 38
Methods 38
MainMenuButton 39
Attributes 39
Constructors 39
Methods 39
5.3.3 Data Access Subsystem 40
Data Manager 40
Constructors 40
Methods 40
5.4 Packages 41
Glossary 42
6.1 Territory 42
6.2 Continent 42
6.2.1 Target Continent 43
6.3 Cards 43
6.4 Player 43
6.5 Troops/Soldiers/Unit of Troops 43
6.6 Turn 43
6.7 Troop Allocation Phase 43
6.8 Attack Phase 44
6.9 Fortify Phase 44
6.10 Airport 44
6.11 Alliance 44
6.12 Rock-Paper-Scissors 44
6.13 Global Domination 44

Design Report

CS 319 3G: RISK

1. Introduction

This part will serve as the section where we outline the purpose of our system, go over each of the design goals talked about within the course contents and explain how our system will aim to meet them. We will also be discussing the trade-offs within these design goals, and which design goal in the trade-offs we will choose to favour. We will also be discussing introducing abbreviations which will be used for the remainder of the report, and conclude with an overview of the design of our system.

1.1 Purpose of the System

Our system will be the mapping of the RISK Global Domination table-top game to a digital system, with several added new functionalities who are outlined in our analysis report. Risk is a strategy board game aimed for two to six players of diplomacy, conflict, and conquest. The standard version of the game consists of a board in which a map of the world is depicted. The map is divided into forty-two territories which are grouped into six continents. Turns rotate among players in control of armies with which they attempt to capture territories via battles whose results are determined by dice rolls. The number of dice a player is able to attack or defend with is dependent on the number of soldiers that player controls in the applicable territories. The classic goal of the game is to occupy every territory on the board and achieve total global domination. Both formal and informal alliances may be formed during the course of the game.
Our project is based on RISK Global Domination, however some certain features will be added/removed, such as:

• We plan on allowing players to be able to leave territories in their control unguarded should they choose to.

• We plan on allowing players to set the goal of capturing a continent before the game starts and receive rewards should they be able to do so.

• We plan on implementing a rock-paper-scissors based combat system where more skill is involved as opposed to dice rolls.

• We plan on implementing an alliance mechanism where the two allied players will be allowed to go through their ally's territories to attack an opponent's territory. The allies will not be able to attack each other for the duration of their alliance.

• We plan on implementing an airplane mechanism. A player will be able to build an airport on his territories by paying a cost.

1.2 Design Goals

In this section, we will go over all of the design goals outlined within course contents, and discuss to what degree and how our system will achieve said design goals.

1.2.1 Reliability

Our game will be played using exclusively a mouse, with the exception of rounds of rock-paper-scissors. Therefore, we will not be expecting or evaluating keyboard strokes for the most part of the game, and will thus not have too many variable methods of input that we will have to deal with. During rounds of rock-paper-scissors, we will only be listening for the keys corresponding to either rock or paper or scissors, and will be ignoring the rest of the keystrokes, minimizing our surface where unexpected inputs may be received. We will not have the need to perform any input sanitization. We will also first expect the attacking player to input their choice of rock-paper-scissors and then the defending player, thus the timing of inputs will not be of issue.
We will also be testing our game for bugs at each stage of development in order to produce the most reliable and bug-free system that we can.

1.2.2 Modifiability

Our game will be implemented using the MVC approach, thus allowing for modifications within different subsystems while minimising the possibility of unexpected results within other systems. As we will not be changing the rules of the game we have set out previously, the main rules and the gameplay of the game will not be modifiable to a great extent.

1.2.3 Maintainability

Since when our project will be finished, we do not plan on maintaining our GitHub repository or the game as a whole, maintainability will not be an applicable design goal to our project.

1.2.4 Understandability

This will be one of the design goals that we as a team will be focusing on. Since Risk in essence is not a complicated game, we will be aiming to make our project as understandable as possible through the use of uncomplicated and clear user interface elements. We will as well not be introducing a large number of new features or functionalities in order to not drastically stray from the classic gameplay of the board game. We aim for anyone who is familiar with the board game to easily understand the new features we will have added and play the game as they would the tabletop game they are used to. We will be making the user interface as simple as possible, with indicators as to which phase the turn player is currently in as well as prompts for what they can do in that phase clearly visible on the screen. We will also be providing a help section which can both be accessed from the main menu and at any time via the pause menu, should players need additional help regarding the game itself or any of the new features we will have added.

1.2.5 Adaptability

Our game will be developed using object-oriented design principles, allowing us to modify separate objects individually depending on any new functionalities we may want to add to the game or any criticisms. We will also be using the MVC design pattern, allowing us even more flexibility when it comes to adaptability.

1.2.6 Reusability

Since we do not plan on reusing any of the assets after we turn our project in, this design goal is not applicable to our project to a great degree. However, we will be implementing a world map as part of our GUI which can then be used as part of any system that is based on a world map as well, allowing for the reuse of that component of our system.

1.2.7 Efficiency

We will be aiming to write a system that is efficient in the way that it allocates and deallocates resources, however since we will be using the Java programming language which features an automatic garbage collection system, we will not be doing additional optimizations regarding this design goal. We will also not be optimizing our software to run on different systems, and will instead rely on the portability the Java programming language offers.

1.2.8 Portability

Since we will exclusively be using the Java programming language, our game will be able to be played on any system that can run Java. Therefore, our game will be highly portable.

1.2.9 Fault Tolerance

We will be using the MVC design pattern when implementing our project, which helps in decomposing a system into several components. However, we will be expecting all of the components in our system to be functional at all times, thus our system will not be fault-tolerant.

1.2.10 Backwards-Compatibility

As our project does not have any previous iterations when it comes to implementation, this design goal is not applicable to our project.

1.2.11 Cost-Effectiveness

This is another design goal that we will be focusing on as a team. As our project does not have a cost in the classical sense (as in a monetary cost), we have decided to interpret this design goal as having to do with other limited resources that we have, such as our time. Since this project will be constituting 40% of our letter grade in a course that is 4 credits, we are planning on allocating a sizable chunk of our available time to its successful completion. We are as a group having meetings on the regular regarding task allocation and how we are doing when it comes to finishing the tasks that have been allocated to us. We have also assigned all of our functionalities a priority, and will have our work focused on achieving those with higher priority better and faster than those with lower priority. As such, should we as a group deem that a functionality with a lower priority is taking up more of our limited resources than we are willing to allocate to that functionality, we may abandon it and move on to better focus on implementing those of higher priority.

1.2.12 Robustness

We will be using several try-catch blocks to try and catch any exceptions that may arise during the running of our software, thus making our software as robust as possible. We will as well only be taking inputs via the mouse for the most part, and will therefore not have too many different types of input that we will have to worry about. We will also only be listening to the relevant keys during rounds of rock-paper-scissors, and will be ignoring all other keystrokes.

1.2.13 Functionality

We will be adding several new functionalities to our system, such as a rock-paper-scissors based combat system, an alliance system and the ability to build airports; as well as keeping all of the existing functionality from the classic tabletop game. Thus, our game will be highly functional with several new features added while keeping those of the classic game.

1.2.14 Usability

This will be another design goal that we will be focusing on as a team. We will be aiming to make our game as easy to play as possible, with an intuitive GUI that will be based on mostly input via the mouse. We will not be using the keyboard as an input mechanism except for rounds of rock-paper-scissors. This is because we believe it is easier for someone unfamiliar with PC systems to point and click using a mouse as opposed to using a keyboard. We will also be clearly displaying information such as which phase of their turn a player is in, whose turn it is, how many soldiers a player has on any territory and who controls what territory on the map at all times, increasing our ease of use.

1.2.15 Readability

We will be aiming to write our code in a way that is as readable as possible, following coding best practices and features copious amounts of comments. We will also aim to give our packages/subsystems/classes/variables et cetera meaningful names that can be easily traced to understand the functionality of our system.

1.2.16 Security

Since our game will not be using any network components, our attack surface will be incredibly small, making it so security will not be an issue. Our system will as well not be storing or dealing with any sort of sensitive data, thus leak of sensitive data will not be an issue either. We will also not be using a large amount of space on the disk, further minimizing our exposure.

1.2.17 Rapid Development

We will be aiming to develop our project in as rapid a manner as possible, maximising our cost effectiveness when it comes to the limited resource of time. As such, we may as a group decide to not implement any functionalities labeled as having low priority in our analysis report if we find that we are lacking in time or that it will simply be taking too much of our resources. We will be focusing on first putting out a system that will satisfy all of our higher priority functionalities and then will be implementing those of low priority later if we are able to do so.

1.3 Trade-Offs

In this section we will be addressing all of the design goal trade-offs outlined in our course contents, and will be declaring which of the two opposing design goals we will be giving more importance to as well as why we have decided to do so.

1.3.1 Functionality versus Usability

In this trade-off, we will be leaning more towards the usability side. This is because building a system that is easy and intuitive to use is a design goal that we will be focusing on as a team throughout the whole project. We do not wish to overly complicate neither the rules of our game nor our user interface so that it remains clear and easy to understand and to use.

1.3.2 Cost versus Robustness

As this project constitutes 40% of our letter grade in a course with 4 credits, we will not try to optimize our cost and will instead spend more time in order to make our project more robust if need be.

1.3.3 Efficiency versus Portability

We will be optimizing efficiency in this trade-off. Although we will be using Java which will be increasing our portability due to its platform-independent nature, we will be focusing on authoring a system that will be most efficient when it comes to issues like complexity and resource management.

1.3.4 Rapid Development versus Functionality

In order to maximize our cost-effectiveness, we may decide on not implementing some of the functionalities labeled low priority on our analysis report. Therefore, when it comes to this trade-off, we will be leaning more towards the rapid development design goal.

1.3.5 Cost versus Reusability

Since we are not planning to reuse any assets from this project in the future, we will be automatically leaning more towards optimizing the cost of our project when it comes to time et cetera as opposed to optimizing our reusability.

1.3.6 Backwards Compatibility versus Readability

Since we do not have any previous versions of our project, backwards compatibility is not a design goal that is applicable to our project. Therefore, we will automatically be optimizing our readability by providing appropriate comments and meaningful names. We will also be focusing on readability so that other members of our group can easily understand the pieces of code that we have written.

1.3.7 Security versus Usability

When it comes to this trade-off, we will be optimizing our usability as our game does not need to be made any more secure than it already is. Since our game will have no networking component, and as it will not store or deal with any sensitive data; our game is already as secure as realistically can be for any piece of software. Therefore, we will be optimizing our usability when it comes to our project.

1.4 Definitions, Acronyms, and Abbreviations

In this section we will be defining any acronyms and abbreviations that we will be using in the rest of our report.

1.4.1 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a virtual machine that allows for systems to run programs that were written in the Java programming language. Operating via Java bytecode, the JVM allows for the portability of Java.

1.4.2 Model View Controller (MVC)

A design pattern, Model View Controller design consists of splitting a system into three parts; Model, View, and Controller. The Model part consists of the actual entities and objects within the system, the View part consists of those objects having to do with user experience, and the Controller part consists of control objects.

1.4.3 Graphical User Interface (GUI)

A graphical user interface (GUI) is any user interface that allows users to interact with the system via the use of graphical icons as opposed to text-based or other types of user interfaces.

1.4.4 Personal Computer (PC)

A personal computer (PC) is a multi-purpose computer that is intended for personal use; whose size, capabilities, price and other attributes make it suited for this role.

1.4.5 Java Runtime Environment (JRE)

The Java Runtime Environment (JRE) is a software layer that runs on top of a computer’s operating system which provides the resources required in order for that computer to run a Java program.

1.5 Overview

In our project, the three design goals that we will mainly be focusing on are understandability, usability, and cost effectiveness. When it comes to trade-offs between design goals, we will be optimizing usability as opposed to functionality, robustness as opposed to cost, rapid development as opposed to functionality, cost as opposed to reusability, readability as opposed to backwards compatibility, and usability as opposed to security.

2. Current Software Architecture

This section is intentionally left blank, to be filled during Iteration 2 since we do not have a current system.

3. Proposed System Architecture

In this part of our report, we will be discussing the architecture of the system that we will be proposing for our project. We will briefly discuss an overview of the entire system; and will then delve into our subsystem decomposition, hardware/software mapping, persistent data management, access control and security, global software control, and our boundary conditions.

3.1 Overview

We will be using the MVC design pattern while decomposing our system into subsystems. Our game will not require any additional hardware to run. We will allow the user to save and continue their game through text files which will be parsed when they choose to load a game and will be created when they save their game. Our game does not have any security risks associated with it. The game will be played in the form of turns. We will have three boundary conditions; initialization, termination, and exception handling.

3.2 Subsystem Decomposition

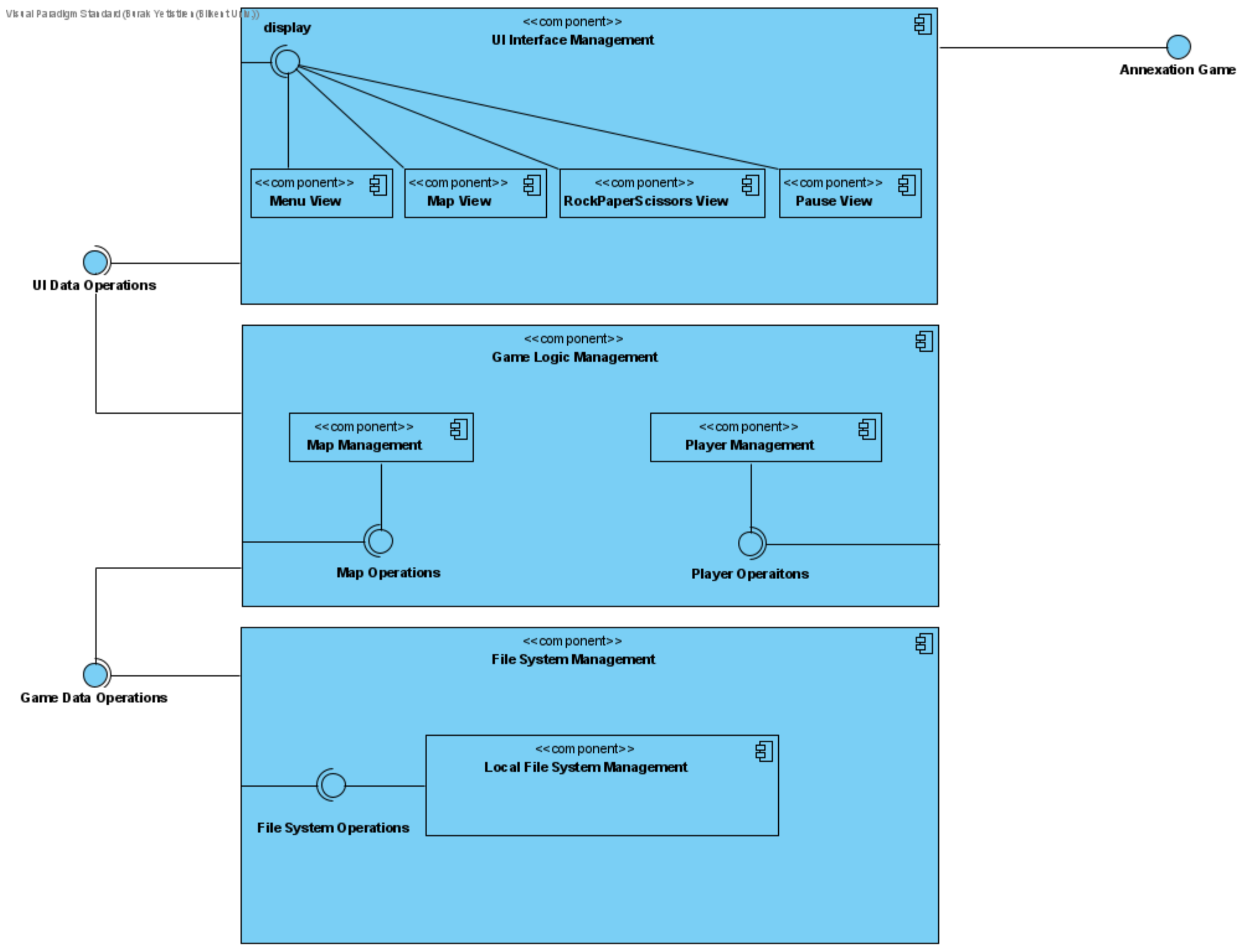


Figure 1: Component Diagram of the System

Our system will be divided into three subsystems, called the UI Interface Management, Game Logic Management, and the File System Management. We have decided to decompose our system into these three subsystems following a design approach inspired by both the MVC and the layered approach, however not sticking to either of them fully. We have essentially decomposed the system into a subsystem that implements both the Model and the Controller in the MVC approach as one (our Game Logic Management subsystem) and a separate View (our UI Interface Management subsystem). We have as well a third subsystem that will be handling the local save files which will be saved in the format of text files, and their parsing when loading a game from a save file.
The services associated with these subsystems will be elaborated on further on section 4. However, the following will be a quick summary rundown of the functionality of the three main subsystems.

3.2.1 UI Interface Management Subsystem

This is the subsystem that corresponds to the View subsystem in the MVC design approach. The main service this subsystem will provide is creating and managing the GUI of our game. It will be managing the GUI of the game in four different components, the Menu View, Map View, Rock-Paper-Scissors View, and the Pause View. These correspond to the four main GUI elements of our game.

3.2.2 Game Logic Management Subsystem

This is the subsystem that corresponds to both the Model and the Controller subsystems in the MVC design approach. We have decided not to separate the two as since our game and project are of a relatively compact scale, we chose to favor quick interaction between objects as opposed to having to establish a communication pattern between different subsystems for our Model and Controller subsystems. This subsystem will be responsible for mainly two services, which are the two components that form the gameplay of our implementation of RISK. Any operation within our game can essentially be thought of as either an operation having to do with players interacting among one another or players interacting with the map. Therefore, this subsystem will be decomposed into two components, Map Management and Player Management.

3.2.3 File System Management Subsystem

This is the subsystem that will be responsible for the creation of save files in the format of text files, and the parsing of said save files when continuing from a save file. We have decided to have this as a separate subsystem as we believe this subsystem does not necessarily fit into either of the categories within the MVC approach. Although it can be argued that this subsystem can be thought of as falling under the Control subsystem, since we have merged our Model and Control subsystems under Game Logic Management, we require a separate subsystem for file management. Since our game will only deal with local files, that will be the only component of this subsystem.

3.3 Hardware/Software Mapping

Our project will be implemented using the Java programming language, and as well using Java FX libraries. As such, our software will require the JRE in order to be run. As we will be using Java FX libraries, our project will require at least Java Development Kit version 8 to be run.

Our project does not require any other additional hardware on top of the standard keyboard / mouse set up to be run. For the most part, our game will be played using exclusively the mouse. The user will need to use the keyboard only during rounds of rock-paper-scissors. This will allow for our game to be played on virtually any PC with the necessary version of the JDK installed.

For storage, we will be using the hard disk space of the user to store our game files as well as our assets. No installation process will be required for our game to be launched. We will be using text files to store saved games as well as user defined settings.

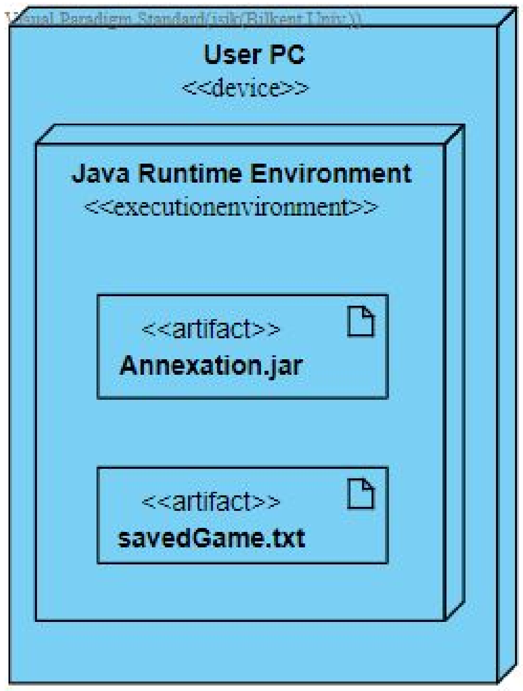


Figure 2: Deployment Diagram of the System

3.4 Persistent Data Management

Our game will not require any complex data storage system or database to be played. We will be giving the user the option to save their game and return to it later via text files that will be saved onto their hard drive which will be parsed to create their game when they decide to load it back. User settings will as well be controlled via a text file which will be changed when they change a setting via any of the menus. Our assets will as well be stored on the hard drive of the user.

3.5 Access Control and Security

As mentioned previously, we will not be using any network components for our game. Therefore, our attack surface will have been minimized. We will as well not be storing any sensitive data in our system, therefore no sensitive data leak exposure will be possible either. We will not take any measures to prevent users from accessing the text files where their save files or settings will be stored. We will not have an authentication system either, the users will be entering their names when they are configuring the game and these names will be stored in a text file for the duration of the game. Once the game is over, these text files will be deleted so as to not crowd the hard drive of the user with redundant game files. We will as well not be encrypting any of our files because we will not be dealing with any sensitive data.

3.6 Global Software Control

Our game will revolve in a turn-based fashion. Our system does not have any concurrent events, all our events will take place one after another in the form of actions within phases and phases within turns. For actions within turns, our system is going to be event-driven. We are not planning on using separate threads for each event.

3.7 Boundary Conditions

In this section of the report, we will be discussing the boundary conditions of our game. We have three boundary conditions; initialization, termination, and exception handling.

3.7.1 Initialization

Our game will not require any installation process in order to be playable. The system will be created whenever the executable is executed. Our executable will be a .jar file. This will as well allow our game to be more portable, easily being copied to different machines in a playable state.
When executing the jar file, the user will first be met with the main menu. Here, they will have the option to start a game, change settings, view credits, get help, or exit the game. If the users decide to start a game, they may either do so by creating a new game or continuing a game from a save file. If they choose to create a new game, they will have to input the number of players and the name of the players, as well as their color and their target continent. If they wish to continue from a save file, they will have to locate the .txt file serving as their save point. Afterwards, the game will start and the system will have been initialized.

3.7.2 Termination

The user may terminate the program at the main menu by clicking the exit button. Once they do so, the system will terminate. They may as well choose to exit a game to the main menu from the pause menu during gameplay and click the exit button, or exit directly from the pause menu. When they do so while the current game has not been concluded, the system will ask the user whether they wish to save their game or not. If they do not wish to do so, they will return to the main menu directly or the system will terminate, depending on which one they have chosen to do. If they wish to save their game however, they will be asked to browse to the directory in which they would like to create their save file in the form of a text document. If the user tries to click the close button from the window frame, the system will ask them if they wish to save or not if there is an active game present, and will terminate if there is no active game present.

3.7.3 Exception Handling

If during trying to load a game the user chooses a file format that is not .txt, or chooses a file that is not in accordance with our format, the system will not accept that file and will not start the game. This will be accomplished with a key that will be present at the very beginning in the first line of the text files which will serve as save files. The game will not have an autosave feature, it will be up to the user to save their game when/if they wish to do so. Therefore, should there be an unexpected circumstance such as a power outage during the game, users will have the ability to start from their previous save point if they have decided to save their game at any time, or else all their data regarding that particular game will be lost. The more technically adept user may always create a .txt file that will be serving as a save file according to our specifications and continue their game from there.

4. Subsystem Services

In this section, we will be discussing in further detail the services of the subsystems we have outlined under section 3.2. We will delve into further discussion regarding the services of our subsystems.

4.1 UI Interface Management Subsystem

This will be the subsystem that will be responsible for the GUI of our game. It will be further decomposed into four separate components; them being Menu View, Map View, Rock-Paper-Scissors View, and Pause Menu View. These components correspond to the four types of GUI screens that will be present in our game.

4.1.1 Menu View Component

This component will be responsible for creating the GUI regarding our menus, and transitions between said menu screens depending on the input from the user. Our game will feature several main screens, among them being the Main Menu Screen, the Launch Game Screen, the Credits Screen, the Help Screen, and the Settings Screen. This component will be responsible for the creation of said screens when the user decides to access them, and the transition between them.

4.1.2 Map View Component

This component will be responsible for the main gameplay screen of our game, the Map Screen. Most of our gameplay will revolve around the Map Screen, in which players will be able to see all of the territories as well as which player controls which territory and how many units of troops are on any territory. From this screen, the turn player will be able to select one of their territories to launch an attack from and to what other territory. All phases of a player’s turn, except for rounds of rock-paper-scissors, will be done on this screen. This screen will therefore be responsible for the creation of the Map Screen and player interaction with said screen.

4.1.3 Rock-Paper-Scissors View Component

This component will be responsible for the Rock-Paper-Scissors overlay that will come on top of the Map Screen when a player engages another player in combat. It will be responsible for displaying the images of rock, paper, and scissors on the screen and as well display the key mappings to each option of rock/paper/scissors. This overlay will be active once a player attacks another player’s territory, and will disappear once the battle has concluded.

4.1.4 Pause Menu View Component

At any point during gameplay, players will have the option to press the ESC key in order to pause the game. This component will be responsible for the creation of the Pause Screen which they will be seeing once they pause their game. From this screen, players will have the option to quit to either the main menu or their desktop, view the help page or the credits, or change any settings. This component will be responsible for the creation of this screen and as well the transitions between the screens via the buttons on the Pause Screen.

4.2 Game Logic Management Subsystem

This is the subsystem that will be encompassing the Model and Controller subsystems of the MVC design approach. It will feature two components, Player Management and Map Management. This is because any operation found within our gameplay can either be thought of as players interacting with each other or players interacting with the map. The first component will be taking care of the former functionalities while the second component will be responsible for the latter.

4.2.1 Player Management Component

This component will be responsible for the functionalities having to do with players interacting among one another. These encompass mainly two different functionalities, the first having to do with our alliance system feature, and the second having to do with our combat. This component will mainly be responsible for the implementation of these two categories of features. It will be responsible for handling our alliance system, which includes functionalities such as proposing an alliance to an player who is not already part of one's alliance, accepting or declining alliance proposals, and the granting of several rights to allied players, such as being able to attack territories going over an ally's territory, as well as several restrictions such as not being able to attack any territories controlled by an ally.
This component will as well be responsible for taking care of our combat system, which includes functionalities such as a player declaring combat from a territory under their control to a neighboring territory that is not under their or an ally's control. This component will as well be the place where our rock-paper-scissors based combat system will be implemented.

4.2.2 Map Management Component

This component will be responsible for the functionalities having to do with players interacting with the map. These encompass mainly two different functionalities, the first having to do with the troop allocation phase and the second having to do with the fortification phase. Troop allocation phase will be the first phase of a player's turn, in which the number of soldiers that they will receive will be computed using the number of territories and continents under their control, and as well checking whether they have managed to conquer their target continent or not. This part of the component will have to do with the calculation of how many soldiers they will receive, as well as the placement of those soldiers onto territories under their control. It will also manage their ability to build an airport and the functionality of swapping cards for additional units of troops.
This component will as well be responsible for the fortification phase of a player's turn. This will be the last phase of a player's turn, coming after their combat phase, in which they will have the ability to redistribute their troops, moving any amount that they wish from one of their territories to another one under their control; provided that there is a bridge of neighboring territories controlled by either themselves or an allied player in between.

4.3 File System Management Subsystem

This is the subsystem that will have to do with the save functionality of our game. It will be responsible for the creation of the text file that will be serving as the save file when users decide to save their game whilst exiting, and will as well be responsible for the parsing of said text files to create their game once they decide to continue from their save file. Since our game will only be dealing with local files, this subsystem will only have a single component called Local File System Management which will be responsible for all its functionality.

5. Low Level Design

This is the part of our report in which we will be providing the final object design of our system. We will do this in the form of class diagrams and explanations for all of the attributes and functions of each individual class. We will as well be discussing object design trade-offs and design patterns under this section of our report, as well discussing the package structure of our system.

5.1 Object Design Trade-Offs

In this part we will address the object design trade-offs outlined in our lectures and discuss which concept we will lean more towards as a group.

5.1.1 Memory Consumption versus Visual Quality and Complexity of the System

In this trade-off, we will be leaning more towards the visual quality and complexity of the system side. We will not be optimizing our system to minimize the use of memory space to a great degree, instead relying more on the automatic garbage collection system found in Java to take care of allocating and deallocating memory space for us. As we will not be using images taking up a great deal of memory space anyways, we will be able to get by without optimizing our use of memory.

5.1.2 Rapid Development versus Code Navigability

In this trade-off, we will be leaning more towards the rapid development side of the spectrum. Although we will not try to sacrifice on our code navigability and readability to a great degree, as we are constrained by deadlines for our project we will be leaning more towards rapid development and try to produce a system with the expected functionality first; as opposed to a system whose code is easily navigable.

5.1.3 Number of Options versus Simplicity

As we will be optimizing our usability and understandability as opposed to functionality, we will be leaning more towards the simplicity side on this trade-off. We believe this design choice of ours is in line with the spirit of the tabletop game as well, which is a quite simplistic game in its essence. Our adaptation of the game to a digital environment will as well try to be as faithful to this interpretation as possible, and our goal is to create a game that is intuitive to understand in both user interface and design of game.

5.1.4 Functionality versus Understandability

In this trade-off, we will be leaning more towards the understandability side of the spectrum. As explained above, we will be aiming to create a simplistic game that does not include excessive complex features but is rather easy to understand the concepts of and intuitive to play.

5.2 Object Design Patterns

We are planning on using both the decorator and the strategy design patterns within our project. However, further elaboration on this subject is left for iteration 2 once we have covered the material more thoroughly.

5.3 Final Object Design

This is the part of our report where we will be providing the final object design of our system. We will do so splitting our class diagrams into our three individual subsystems for the sake of clarity. Under this section, we will first provide the reader with the class diagram corresponding to either of our subsystems, followed by detailed explanation regarding the attributes and functions of each class found within the subsystem.

5.3.1 Game Logic Subsystem

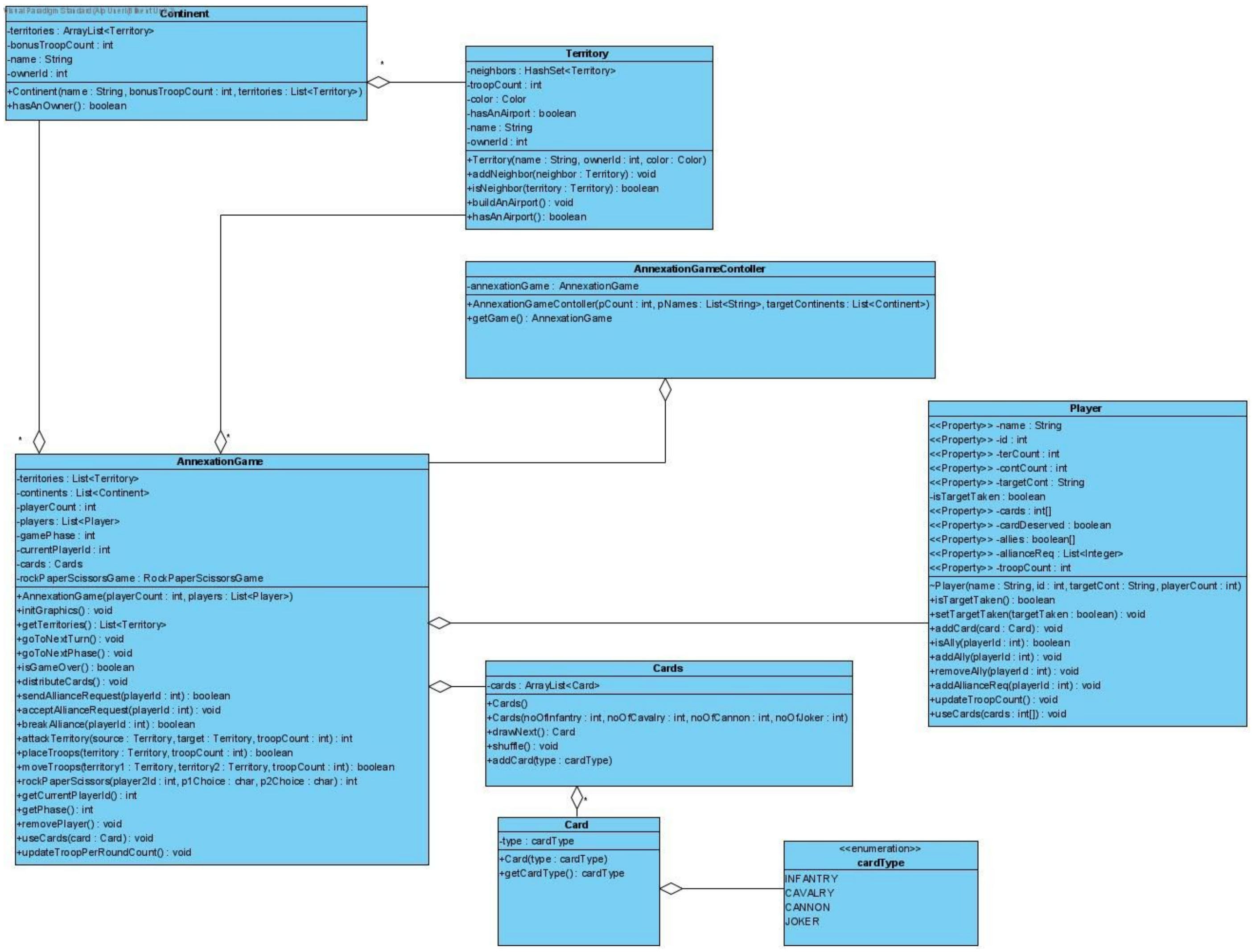
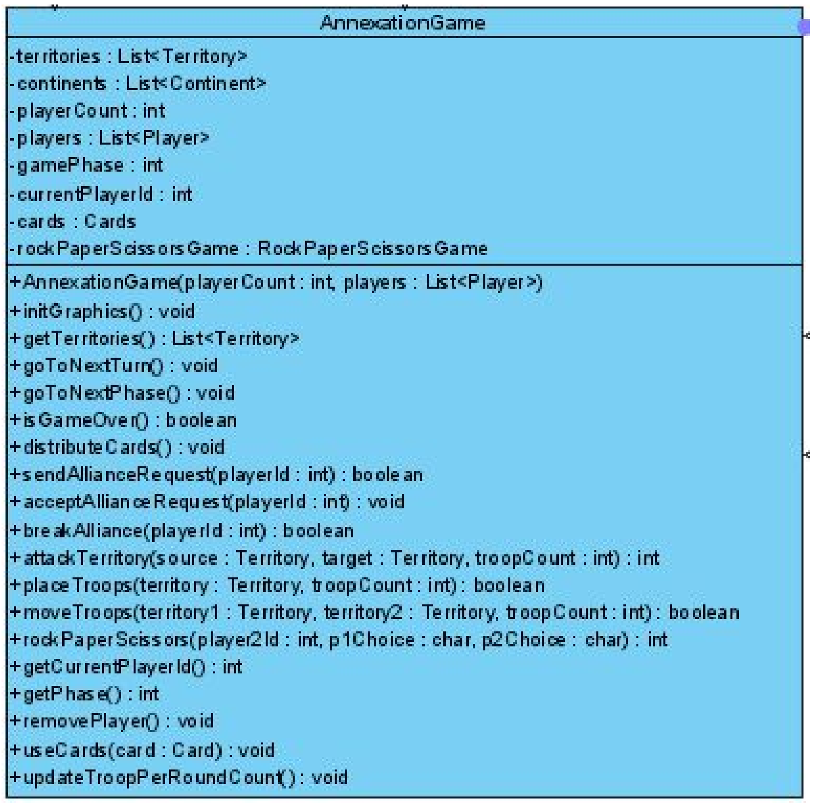

Continent
-territories : ArrayList<Territory>
-bonusTroopCount : int
-name : String
-ownerId : int
+Continent(name : String, bonusTroopCount : int, territories : List<Territory>)
+hasAnOwner() : boolean
Territory
-neighbors : HashSet<Territory>
-troopCount : int
-color : Color
-hasAnAirport : boolean
-name : String
-ownerId : int
+Territory(name : String, ownerId : int, color : Color)
+addNeighbor(neighbor : Territory) : void
+isNeighbor(territory : Territory) : boolean
+buildAnAirport() : void
+hasAnAirport() : boolean
AnnexationGameContoller
-annexationGame : AnnexationGame
+AnnexationGameContoller(pCount : int, pNames : List<String>, targetContinents : List<Continent>)
+getGame() : AnnexationGame
Player
<<Property>> -name : String
<<Property>> -id : int
<<Property>> -terCount : int
<<Property>> -contCount : int
<<Property>> -targetCont : String
-isTargetTaken : boolean
<<Property>> -cards : int[]
<<Property>> -cardDeserved : boolean
<<Property>> -allies : boolean[]
<<Property>> -allianceReq : List<Integer>
<<Property>> -troopCount : int
~Player(name : String, id : int, targetCont : String, playerCount : int)
+isTargetTaken() : boolean
+setTargetTaken(targetTaken : boolean) : void
+addCard(card : Card) : void
+isAlly(playerId : int) : boolean
+addAlly(playerId : int) : void
+removeAlly(playerId : int) : void
+addAllianceReq(playerId : int) : void
+updateTroopCount() : void
+useCards(cards : int[]) : void
AnnexationGame
-territories : List<Territory>
-continents : List<Continent>
-playerCount : int
-players : List<Player>
-gamePhase : int
-currentPlayerId : int
-cards : Cards
-rockPaperScissorsGame : RockPaperScissorsGame
+AnnexationGame(playerCount : int, players : List<Player>)
+initGraphics() : void
+getTerritories() : List<Territory>
+goToNextTurn() : void
+goToNextPhase() : void
+isGameOver() : boolean
+distributeCards() : void
+sendAllianceRequest(playerId : int) : boolean
+acceptAllianceRequest(playerId : int) : void
+breakAlliance(playerId : int) : boolean
+attackTerritory(source : Territory, target : Territory, troopCount : int) : int
+placeTroops(territory : Territory, troopCount : int) : boolean
+moveTroops(territory1 : Territory, territory2 : Territory, troopCount : int) : boolean
+rockPaperScissors(player2Id : int, p1Choice : char, p2Choice : char) : int
+getCurrentPlayerId() : int
+getPhase() : int
+removePlayer() : void
+useCards(card : Card) : void
+updateTroopPerRoundCount() : void
Cards
-cards : ArrayList<Card>
+Cards()
+Cards(noOfInfantry : int, noOfCavalry : int, noOfCannon : int, noOfJoker : int)
+drawNext() : Card
+shuffle() : void
+addCard(type : cardType)
Card
-type : cardType
+Card(type : cardType)
+getCardType() : cardType
<<enumeration>>
cardType
INFANTRY
CAVALRY
CANNON
JOKER
*
*
*
*

Figure 3: Game Logic Subsystem Class Diagram

AnnexationGame



```
AnnexationGame
-territories : List<Territory>
-continents : List<Continent>
-playerCount : int
-players : List<Player>
-gamePhase : int
-currentPlayerId : int
-cards : Cards
-rockPaperScissorsGame : RockPaperScissorsGame
+AnnexationGame(playerCount : int, players : List<Player>)
+initGraphics() : void
+getTerritories() : List<Territory>
+goToNextTurn() : void
+goToNextPhase() : void
+isGameOver() : boolean
+distributeCards() : void
+sendAllianceRequest(playerId : int) : boolean
+acceptAllianceRequest(playerId : int) : void
+breakAlliance(playerId : int) : boolean
+attackTerritory(source : Territory, target : Territory, troopCount : int) : int
+placeTroops(territory : Territory, troopCount : int) : boolean
+moveTroops(territory1 : Territory, territory2 : Territory, troopCount : int) : boolean
+rockPaperScissors(player2Id : int, p1Choice : char, p2Choice : char) : int
+getCurrentPlayerId() : int
+getPhase() : int
+removePlayer() : void
+useCards(card : Card) : void
+updateTroopPerRoundCount() : void
```

Figure 4: The AnnexationGame Class

Attributes

- **private HashSet<Territories> territories:** Holds all the territories in the game.
- **private List<Continent> continents:** Holds all the continents in the game.
- **int playerCount:** Holds the number of player.
- **private List<Player> players:** Holds the players.
- **private int gamePhase:** Holds phase the game is in.
- **private int currentPlayerId:** Holds the ID of the player who is currently playing.
- **private Cards cards:** Holds the card deck that is created at the beginning of the game.
- **private RockPaperScissorsGame rockPaperScissorsGame:** Holds an instance of RockPaperScissorsGame class.

Constructors

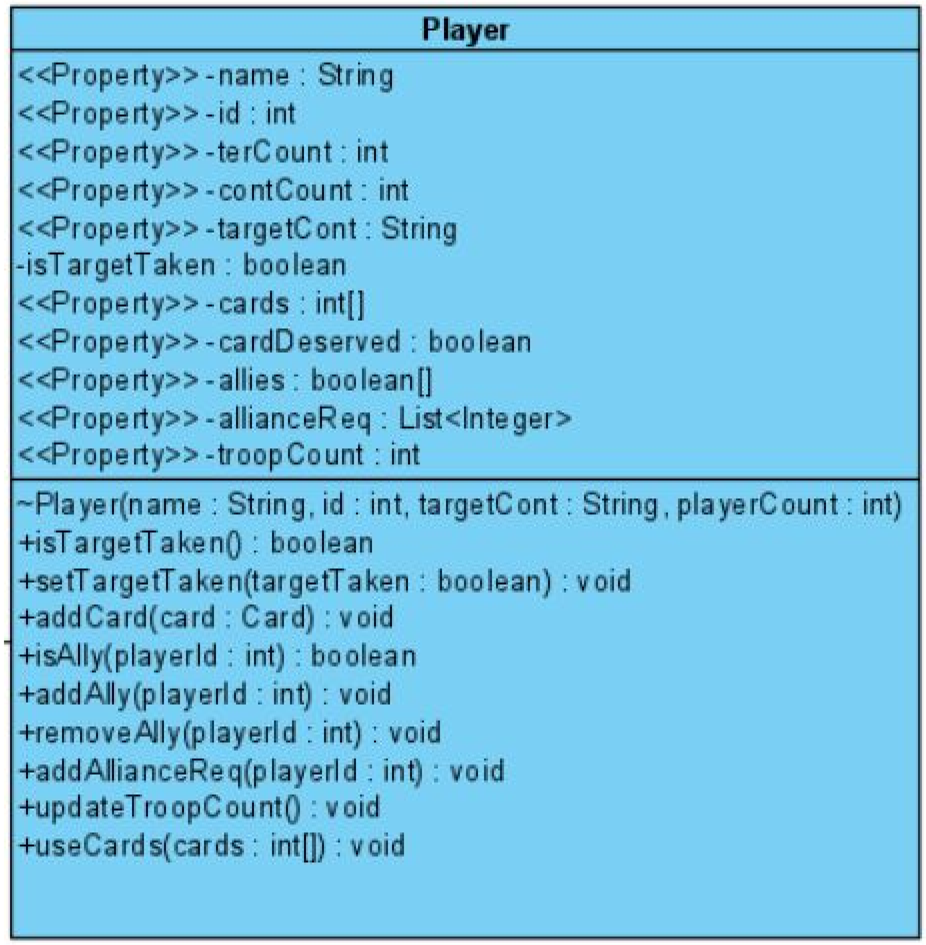
- **public AnnexationGame(int playerCount, List<Player> players):** Sets playerCount and players to its parameters, creates all the territories, continents, players and cards, instantiates a RockPaperScissorsGame and finally assigns 1 to gamePhase

Methods

- **public List<Territory> getTerritories:** Returns all the territories in the game.

- **public void initGraphics():** Initializes graphics.
- **public void goToNextTurn():** Sets currentPlayerId to the next player’s ID.
- **public void goToNextPhase():** Sets gamePhase to the next phase.
- **public boolean isGameOver():** Checks whether all the continents have the same owner. If so, returns true. Otherwise, returns false.
- **public void distributeCards():** If the current player deserved a new card, calls cards.drawNext() method and passes it to players.get(currentPlayerId).addCard(card) method as a parameter.
- **public boolean sendAllianceRequest(int playerId):** Sends an alliance request from the current player to the player whose ID is given by calling players.get(playerId). addAllianceRequest(currentPlayerId) method.
- **public void acceptAllianceRequest(int playerId):** Forms an alliance between the current player and the player whose ID is given by calling players.get(currentPlayerId).addAlly(playerId) and players.get(playerId).addAlly(currentPlayerId) functions. Of course,an alliance to be formed, both players must not be eliminated.
- **public boolean breakAlliance(int playerId):** Breaks an alliance as per the request of one of the allies or when one of the allies is eliminated.
- **public int attackTerritory(Territory source, Territory target, int troopCount):** Attacks the target territory with troopCount troops in the source territory. The success of the attack is determined according to the Rock, Paper, Scissors game that will be played between the current player and the owner of the target territory. Returns 0, if the attack is impossible (e.g. the source territory does not belong to the current player), Returns 1, if the attack is successful. Otherwise, returns 2.
- **public boolean placeTroops(Territory territory, int troopCount):** If the current player tries to place their new troops to one of their own territory, calls territory.setTroops(territory.getTroops() + players.get(currentPlayerId).getTroopPerRoundCount()) methods and returns true. Otherwise, returns false.
- **public boolean moveTroops(Territory territory1, Territory territory2, int troopCount):** If the territory1 and territory2 belongs to the current player and territory1 has more than one troop, moves troopCount troops from territory1 to territory2 and returns true. Otherwise, returns false.
- **public int rockPaperScissors(int player2Id, char p1Choice, char p2Choice):** Calls rockPaperScissorsGame.play(p1Choice, p2Choice) method and returns its value.
- **public int getCurrentPlayerId():** Returns currentPlayerId.
- **public int getPhase():** Returns gamePhase.
- **public void removePlayer():** Checks if there is a player who owns no territory and if such a player exists, removes him/her from the players list.
- **public void useCards(Card card):** If the current player uses his/her cards, calls players.get(currentPlayerId).useCards(card.getCardType()) method.
- **public void updateTroopPerRoundCount():** Calls players.get(currentPlayerId). updateTroopPerRoundCount() method.

Player Class



```
Player
<<Property>> -name : String
<<Property>> -id : int
<<Property>> -terCount : int
<<Property>> -contCount : int
<<Property>> -targetCont : String
-isTargetTaken : boolean
<<Property>> -cards : int[]
<<Property>> -cardDeserved : boolean
<<Property>> -allies : boolean[]
<<Property>> -allianceReq : List<Integer>
<<Property>> -troopCount : int
~Player(name : String, id : int, targetCont : String, playerCount : int)
+isTargetTaken() : boolean
+setTargetTaken(targetTaken : boolean) : void
+addCard(card : Card) : void
+isAlly(playerId : int) : boolean
+addAlly(playerId : int) : void
+removeAlly(playerId : int) : void
+addAllianceReq(playerId : int) : void
+updateTroopCount() : void
+useCards(cards : int[]) : void
```

Figure 5: The Player Class

Attributes

- **private String name:** Holds the name of the player.
- **private int id:** Holds the id of the player.
- **private int terCount:** Holds the count of the territory that the player owns.
- **private String targetContinent:** Holds the continent name which is selected at the beginning of each game by players.
- **private boolean isTargetTaken:** Holds true if the player owns all the territories in the target continent, otherwise false.
- **private int[] cards:** Holds the number of the cards with respect to their types. In index 0, it has the number of infantry cards, in index 1, the number of cavalry cards, in index 2, the number of cannon cards and in index 3, the number of joker cards.
- **private boolean cardDeserved**: Holds true if the player will take a card, otherwise false.
- **private boolean[] allies:** Holds true if the player is ally ,otherwise false. The indexes represent players with respect to their ids.

- **private List<Integer> allianceReq:** Holds the id of the players that send alliance requests.
- **private int troopCount:** Holds the number of soldiers that player can allocate at the beginning of each turn.
- **private String color:** Holds the player's chosen color in a HEX code format.

Constructors

- **Player(String name, int id, String targetCont, int playerCount):** Creates a player object by using the given parameters.

Methods

- **public boolean isTargetTaken():** Return true if the player owns all the territories in the target continent, otherwise false.
- **public void setTargetTaken(boolean targetTaken):** Sets the property targetTaken to the given boolean value.
- **public void addCard(Card card):** Adds the drawn card to the cards' of the player, increases the number of cards with respect to its type.
- **public void addAlly(int playerId):** Sets the index with the playerId of the allies array as true. The player becomes an ally with the given player.
- **public void removeAlly(int playerId):** Sets the index with the playerId of the allies array as false. The player is not an ally of the given player from now.
- **public void addAllianceReq(int playerId):** Adds the playerId to the list called allianceReq.
- **public void updateTroopCount():** Calculates the troop count with respect to the attributes called terCount and isTargetTaken and sets the result to troopCount property.
- **public void useCards(int[] cards):** Decreases the number of cards with respect to the values in the given array. The values in each index are subtracted from the player's cards with respect to their indexes.

Continent

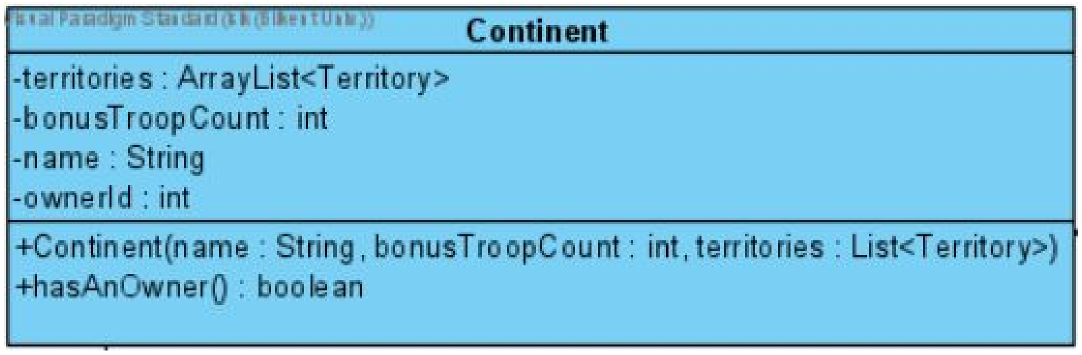

Continent
-territories : ArrayList<Territory>
-bonusTroopCount : int
-name : String
-ownerId : int
+Continent(name : String, bonusTroopCount : int, territories : List<Territory>)
+hasAnOwner() : boolean

Figure 6: The Continent Class

Attributes

- **private String name:** Holds the name of the continent.
- **private int ownerId:** Holds the id of the player, who is the owner of the continent, if any.

- **private ArrayList<Territory> territories:** Holds Territoires present in the continent.
- **private int bonusTroopCount:** Holds the number of bonus troops that may be given to the possible owner of the continent.

Constructor

- **Continent(String name, int bonusTroopCount, ArrayList territories):** Creates a continent object with the given parameters.

Methods

- **public boolean hasAnOwner():** Returns a boolean value regarding if the continent has an owner or not.

Territory

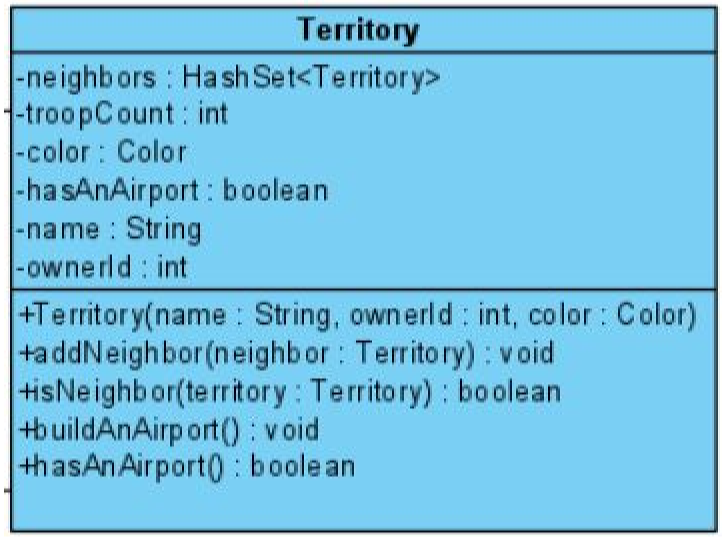

Territory
-neighbors : HashSet<Territory>
-troopCount : int
-color : Color
-hasAnAirport : boolean
-name : String
-ownerId : int
+Territory(name : String, ownerId : int, color : Color)
+addNeighbor(neighbor : Territory) : void
+isNeighbor(territory : Territory) : boolean
+buildAnAirport() : void
+hasAnAirport() : boolean

Figure 7: The Territory Class

Attributes

- **private HashSet<Territory> neighbors:** Holds the neighbor territories of the territory.
- **private int troopCount:** Holds the number of troops present in the territory.
- **private Color color:** Holds the color of the territory, which will be displayed on the map.
- **private boolean hasAnAirport:** Holds a boolean value if the territory has an airport or not.
- **private String name:** Holds the name of the territory.
- **private int ownerId:** Holds the id of the player, who is the owner of the territory, if any.

Constructor

- **Territory(String name, int ownerId, Color color):** Creates a territory object with the given parameters.

Methods

- **public void addNeighbor(Territory neighbor):** Adds a given territory to the neighbors HashSet.
- **public boolean isNeighbor(Territory territory):** Returns true if the given territory is a neighbor of the territory, if not it returns false.
- **public void buildAnAirport():** Builds an airport to the territory, by changing the boolean variable to true, which indicates if the territory has an airport or not.
- **public boolean hasAnAirport():** Returns true if the given territory has an airport, if not it returns false.

Card

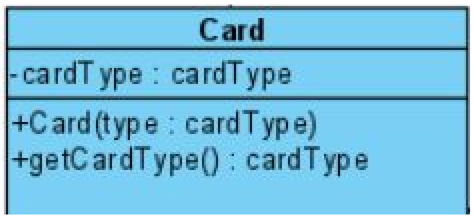


Figure 8: The Card Class

Attributes

- **private CardType cardType:** Holds the type of the Card.

Constructor

- **Card(CardType type):** Creates a Card object with the given type.

Methods

- **public CardType getCardType():** Returns the type of the Card.

Cards

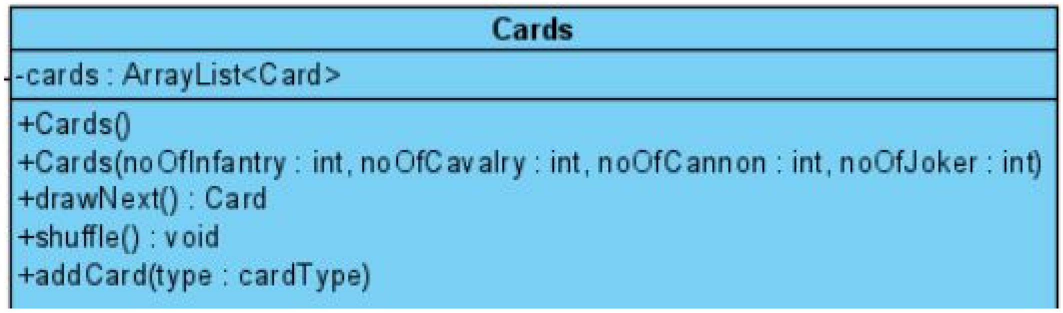

Cards
-cards : ArrayList<Card>
+Cards()
+Cards(noOfInfantry : int, noOfCavalry : int, noOfCannon : int, noOfJoker : int)
+drawNext() : Card
+shuffle() : void
+addCard(type : cardType)

Figure 9: The Cards Class

Attributes

- **private ArrayList<Card>:** Holds the list of Cards.

Constructor

- **Cards():** Creates Cards with different types of Cards with the default count.

Methods

- **Public void shuffle():** It shuffles the deck.
- **addCard(Card card):** It adds the Card to the end of Cards.
- **public Card drawNext():** Returns the first Card in the Cards

CardType (Enumeration)

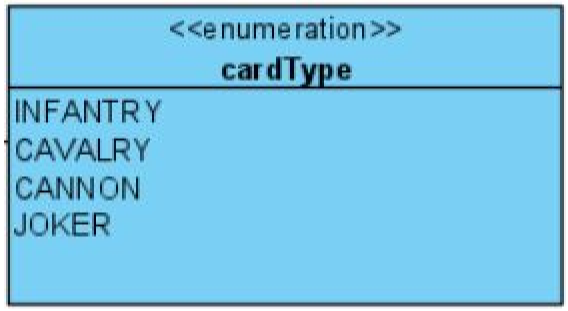
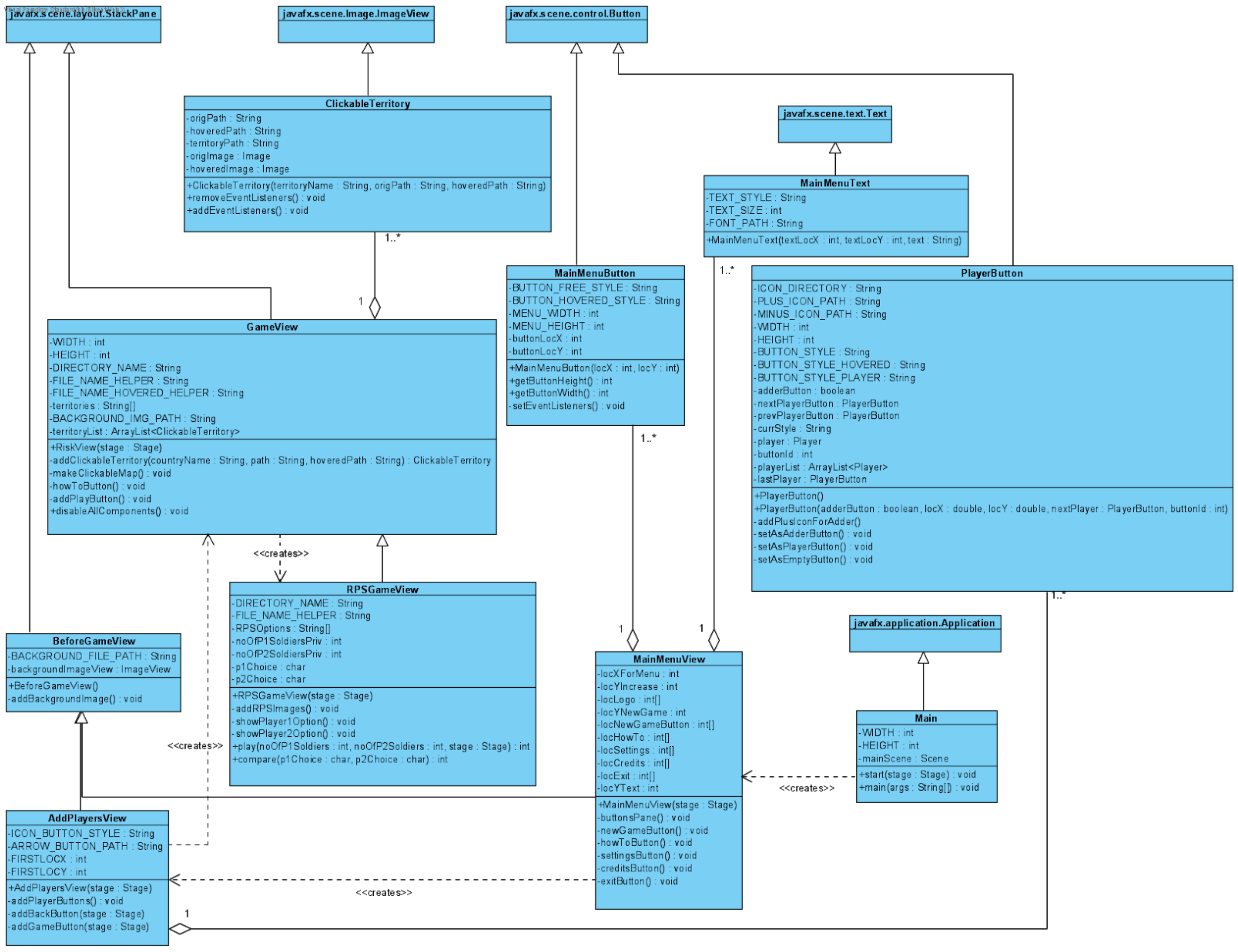


Figure 10: The CardType Enumeration

These enums determine the type of Cards.

5.3.2 User Interface Subsystem



javafx.scene.layout.StackPane
javafx.scene.image.ImageView
javafx.scene.control.Button
javafx.scene.text.Text
ClickableTerritory
-origPath : String
-hoveredPath : String
-territoryPath : String
-origImage : Image
-hoveredImage : Image
+ClickableTerritory(territoryName : String, origPath : String, hoveredPath : String)
+removeEventListeners() : void
+addEventListeners() : void
MainMenuText
-TEXT_STYLE : String
-TEXT_SIZE : int
-FONT_PATH : String
+MainMenuText(textLocX : int, textLocY : int, text : String)
1..*
1
GameView
-WIDTH : int
-HEIGHT : int
-DIRECTORY_NAME : String
-FILE_NAME_HELPER : String
-FILE_NAME_HOVERED_HELPER : String
-territories : String[]
-BACKGROUND_IMG_PATH : String
-territoryList : ArrayList<ClickableTerritory>
+RiskView(stage : Stage)
-addClickableTerritory(countryName : String, path : String, hoveredPath : String) : ClickableTerritory
-makeClickableMap() : void
-howToButton() : void
-addPlayButton() : void
+disableAllComponents() : void
MainMenuButton
-BUTTON_FREE_STYLE : String
-BUTTON_HOVERED_STYLE : String
-MENU_WIDTH : int
-MENU_HEIGHT : int
-buttonLocX : int
-buttonLocY : int
+MainMenuButton(locX : int, locY : int)
+getButtonHeight() : int
+getButtonWidth() : int
-setEventListeners() : void
1..*
PlayerButton
-ICON_DIRECTORY : String
-PLUS_ICON_PATH : String
-MINUS_ICON_PATH : String
-WIDTH : int
-HEIGHT : int
-BUTTON_STYLE : String
-BUTTON_STYLE_HOVERED : String
-BUTTON_STYLE_PLAYER : String
-adderButton : boolean
-nextPlayerButton : PlayerButton
-prevPlayerButton : PlayerButton
-currStyle : String
-player : Player
-buttonId : int
-playerList : ArrayList<Player>
-lastPlayer : PlayerButton
+PlayerButton()
+PlayerButton(adderButton : boolean, locX : double, locY : double, nextPlayer : PlayerButton, buttonId : int)
-addPlusIconForAdder()
-setAsAdderButton() : void
-setAsPlayerButton() : void
-setAsEmptyButton() : void
1..*
<<creates>>
RPSGameView
-DIRECTORY_NAME : String
-FILE_NAME_HELPER : String
-RPSOptions : String[]
-noOfP1SoldiersPriv : int
-noOfP2SoldiersPriv : int
-p1Choice : char
-p2Choice : char
+RPSGameView(stage : Stage)
-addRPSImages() : void
-showPlayer1Option() : void
-showPlayer2Option() : void
+play(noOfP1Soldiers : int, noOfP2Soldiers : int, stage : Stage) : int
+compare(p1Choice : char, p2Choice : char) : int
1..*
1
1
javafx.application.Application
BeforeGameView
-BACKGROUND_FILE_PATH : String
-backgroundImageView : ImageView
+BeforeGameView()
-addBackgroundImage() : void
MainMenuView
-locXForMenu : int
-locYIncrease : int
-locLogo : int[]
-locYNewGame : int
-locNewGameButton : int[]
-locHowTo : int[]
-locSettings : int[]
-locCredits : int[]
-locExit : int[]
-locYText : int
+MainMenuView(stage : Stage)
-buttonsPane() : void
-newGameButton() : void
-howToButton() : void
-settingsButton() : void
-creditsButton() : void
-exitButton() : void
Main
-WIDTH : int
-HEIGHT : int
-mainScene : Scene
+start(stage : Stage) : void
+main(args : String[]) : void
<<creates>>
<<creates>>
AddPlayersView
-ICON_BUTTON_STYLE : String
-ARROW_BUTTON_PATH : String
-FIRSTLOCX : int
-FIRSTLOCY : int
+AddPlayersView(stage : Stage)
-addPlayerButtons() : void
-addBackButton(stage : Stage)
-addGameButton(stage : Stage)
<<creates>>
1

Figure 11: The User Interface Subsystem Class Diagram

MainMenuView

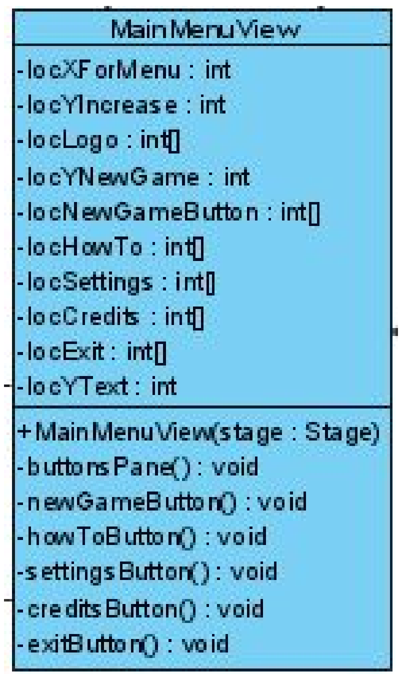


Figure 12: MainMenuView Class

Attributes

- **final private int   LOCXFORMENU:** Holds the initial X location for Menu items as an integer.
- **final private int   LOCYINCREASE:** Holds how much the Y location will increase for each menu item as an integer.
- **final private int[] LOCLOGO:** Holds the X and Y locations of the logo as an integer array containing two integers, as X and Y locations for the logo are different from the other menu items.
- **final private int   LOCYNEWGAME:** Holds the Y location of the button for New Game Button, which is the first button amongst buttons.
- **final private int   LOCYTEXT:** Holds the Y location of the first MainMenuText item, as MainMenuText objects are separate from the MainMenuButton objects in the game interface for styling reasons.
- **final private String LOGO_STYLE:** The JavaFX integrated CSS style for the Logo that is held as a String that will be set using the setStyle() method.

Constructors

- **MainMenuView (Stage stage):** Calls the buttonsPane() method to set all of the buttons to the pane.

Methods

- **private void buttonsPane (Stage stage):** Calls each of the following methods inside it that will create MainMenuButton objects along with their corresponding MainMenuText objects which will be set calling the MainMenuText(int locX, int locY, String text) constructor. Each text of the “String text” variable of the constructor will be as follows: “New Game”, “How to Play”, “Settings”, “Credits”, “Exit”. Then groups each of the MainMenuButton and MainMenuText objects together as a javafx.scene.Group object using its Group.getChilden().add() method to then call the parent’s addGroup(Group g) method to add all of these to the pane.

- **private MainMenuButton newGameButton (Stage stage):** Creates a new MainMenuButton object using the LOCXFORMENU and LOCYNEWGAME variables as its X location and Y location parameters. Then sets a setOnMousePressed() method to set an action for when the mouse interacts with the button as an interaction that creates a new GameView object, constructs a new javafx.scene.Scene object with this GameView object, then calls setScene() method for the stage variable inside the parameter.
- **private MainMenuButton howToButton (Stage stage):** Creates a new MainMenuButton object using the LOCXFORMENU and LOCYNEWGAME + LOCYINCREASE variables as its X location and Y location parameters. Then sets a setOnMousePressed() method to set an action for when the mouse interacts with the button as an interaction that creates a new HowToView object, constructs a new javafx.scene.Scene object with this HowToView object, then calls setScene() method for the stage variable inside the parameter.
- **private MainMenuButton settingsButton (Stage stage):** Creates a new MainMenuButton object using the LOCXFORMENU and LOCYNEWGAME + 2 * LOCYINCREASE variables as its X location and Y location parameters. Then sets a setOnMousePressed() method to set an action for when the mouse interacts with the button as an interaction that creates a new SettingsView object, constructs a new javafx.scene.Scene object with this SettingsView object, then calls setScene() method for the stage variable inside the parameter.
- **private MainMenuButton creditsButton (Stage stage):** Creates a new MainMenuButton object using the LOCXFORMENU and LOCYNEWGAME + 3 * LOCYINCREASE variables as its X location and Y location parameters. Then sets a setOnMousePressed() method to set an action for when the mouse interacts with the button as an interaction that creates a new CreditsView object, constructs a new javafx.scene.Scene object with this CreditsView object, then calls setScene() method for the stage variable inside the parameter.
- **private MainMenuButton exitButton (Stage stage):** Creates a new MainMenuButton object using the LOCXFORMENU and LOCYNEWGAME + 4 * LOCYINCREASE variables as its X location and Y location parameters. Then sets a setOnMousePressed() method to set an action for when the mouse interacts with the button. Inside the method, the function that is set for the event is only the close() method for the stage object within the parameters of the function.

AddPlayersView

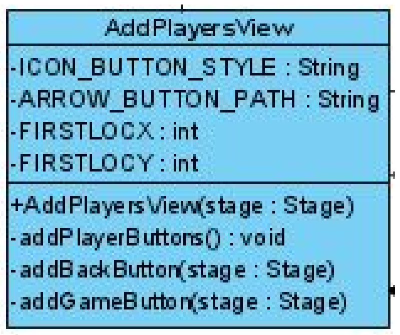

AddPlayersView
-ICON_BUTTON_STYLE : String
-ARROW_BUTTON_PATH : String
-FIRSTLOCX : int
-FIRSTLOCY : int
+AddPlayersView(stage : Stage)
-addPlayerButtons() : void
-addBackButton(stage : Stage)
-addGameButton(stage : Stage)

Figure 13: AddPlayerView Class

Attributes

- **private final String ICON_BUTTON_STYLE:** Holds the sty
- **private final String ARROW_BUTTON_PATH:** Holds the path for the stylistic arrow image object as a String.
- **private final int BETWEEN_TWO_PLAYER_BUTTONS:** Holds how many pixels will be between two different PlayerButtn objects.
- **private final int firstLocX**
- **private final int firstLocY**

Constructors

- **AddPlayersView(Stage stage):** Calls the parent constructor, calls the addPlayerButtons(), addBackButton(stage) and addGameButton(stage) methods.

Methods

- **private void addPlayerButtons():** Creates six
- **private void addBackButton(Stage stage):**
- **private void addGameButton(Stage stage):**

PlayerButton

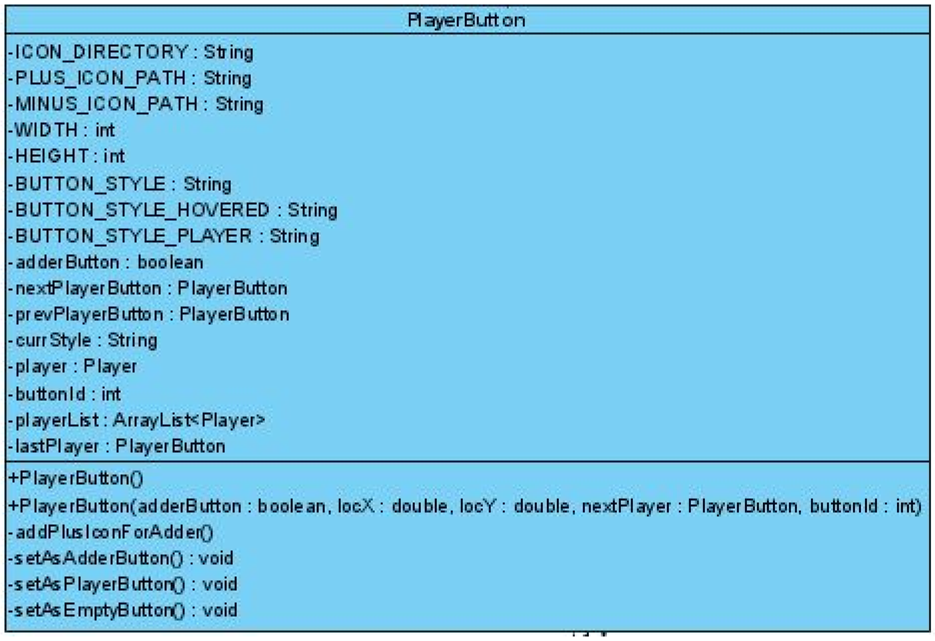

PlayerButton
-ICON_DIRECTORY : String
-PLUS_ICON_PATH : String
-MINUS_ICON_PATH : String
-WIDTH : int
-HEIGHT : int
-BUTTON_STYLE : String
-BUTTON_STYLE_HOVERED : String
-BUTTON_STYLE_PLAYER : String
-adderButton : boolean
-nextPlayerButton : PlayerButton
-prevPlayerButton : PlayerButton
-currStyle : String
-player : Player
-buttonId : int
-playerList : ArrayList<Player>
-lastPlayer : PlayerButton
+PlayerButton()
+PlayerButton(adderButton : boolean, locX : double, locY : double, nextPlayer : PlayerButton, buttonId : int)
-addPlusIconForAdder()
-setAsAdderButton() : void
-setAsPlayerButton() : void
-setAsEmptyButton() : void

Figure 14: PlayerButton Class

Attributes

- **final private String ICON_DIRECTORY:** The directory where minus and plus icons are held.
- **final private String PLUS_ICON_PATH:** Holds the path for the plus object, also using the ICON_DIRECTORY as a helper.
- **final private String MINUS_ICON_PATH:** Holds the path for the minus object, also using the ICON_DIRECTORY as a helper.
- **final private int WIDTH:** Maximum width of the button.
- **final private int HEIGHT:** Maximum height of the button.
- **final private String BUTTON_STYLE:** A style definition for the button's empty in a CSS form held in a String object.
- **final private String BUTTON_STYLE_HOVERED:** A style definition for when the button is hovered on state in a CSS form held in a String object.
- **final private String BUTTON_STYLE_PLAYER:** A style definition for when the button is a button that holds Player information in a CSS form held in a String object.
- **private boolean adderButton:** A boolean to hold whether this PlayerButton is an adder button or not.
- **private PlayerButton nextPlayerButton:** The player button next to this one, null if no next button exists.
- **private PlayerButton prevPlayerButton:** The player button that is previous to this one, null if no previous button exists.
- **private String currStyle:** Current style of the button in a CSS form held in a String object.
- **private Player player:** Holds the Player object associated with the current PlayerButton.

- **private int buttonId:** Holds the id of the button.
- **private static ArrayList<Player> playerList**
- **private static PlayerButton lastPlayer:** A static variable to hold which button is currently the last player button, i.e. the adder button if not all of the player button spaces are occupied, and the last button if all of them are.

Constructors

- **PlayerButton():** A placeholder constructor, does nothing except for to call for the parent constructor.
- **PlayerButton(boolean adderButton, double locX, double locY, PlayerButton nextPlayerButton, int buttonId):** Sets X and Y locations of the button using locX and locY, sets the adderButton boolean to the variable with the same name, sets the nextPlayerButton as well as sets the nextPlayerButton's prevPlayerButton as this, sets the current style as BUTTON_STYLE, and if adderButton is true then calls addIcon() and setAsAdderButton() methods.

Methods

- **private void addIcon():** Sets the graphic of the button to the ImageView constructed with the Image object constructed by accessing PLUS_ICON_PATH.
- **private void setAsAdderButton():** Sets lastPlayer static variable as this, sets current style as BUTTON_STYLE, sets adderButton to true and calls addIcon(). Also calls mouse event handlers which do these:
  - setOnMousePressed(): Creates a new Player object that it will then add to the playerList static variable, get the player's color to set as this button's style color, calls setAsPlayerButton() for this class and calls the setAsAdderButton() method for the nextPlayer if nextPlayer is not null, if it is then sets lastPlayer as this.
  - setOnMouseEntered(): Sets the current style as BUTTON_STYLE_HOVERED.
  - setOnMouseExited(): Sets the current style as what currStyle variable is now equal to.
- **private void setAsPlayerButton():** Sets graphic to null, sets adderButton variable to false, sets currStyle to BUTTON_STYLE_PLAYER + player's color which is in the form of a HEX code in a String. Sets event handlers as follows:
  - setOnMousePressed(): Removes the button from the playerList using the buttonId of the current class, then shifts each PlayerButton object to the place of the previous one. If the lastPlayer object is not an adderButton, meaning that it is the sixth player which is now occupied, it calls setAsAdderButton() for itself. If lastPlayer is any other button however, then it calls setAsEmptyButton() for lastPlayer. Then it checks whether the prevPlayerButton object of lastPlayer is null, calls setAsAdderButton() for it if not and sets lastPlayer as this if it is null.
  - setOnMouseEntered(): Sets the current style as BUTTON_STYLE_HOVERED.
  - setOnMouseExited(): Sets the current style as what currStyle variable is now equal to.

- **private void setAsEmptyButton():** Empties all of the event listeners, sets graphic to null, sets adderButton variable to false, and sets currStyle to BUTTON_STYLE to then set the button style to currStyle.

GameView

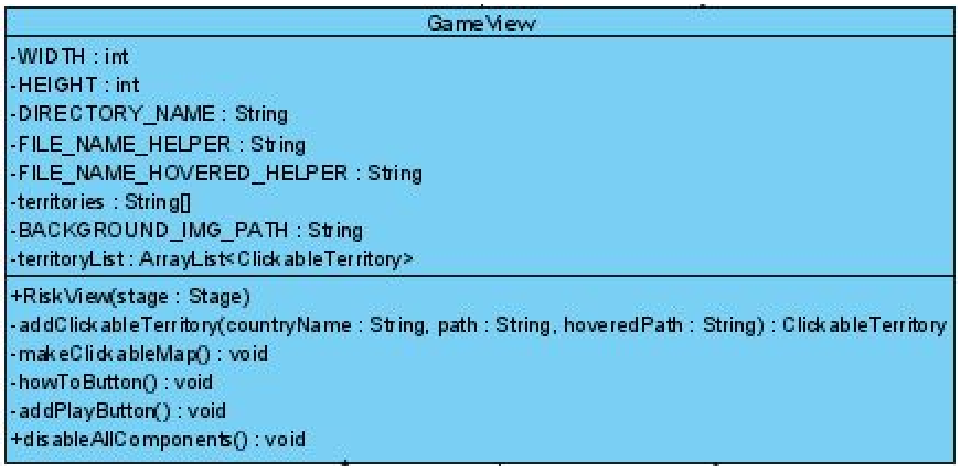

GameView
-WIDTH : int
-HEIGHT : int
-DIRECTORY_NAME : String
-FILE_NAME_HELPER : String
-FILE_NAME_HOVERED_HELPER : String
-territories : String[]
-BACKGROUND_IMG_PATH : String
-territoryList : ArrayList<ClickableTerritory>
+RiskView(stage : Stage)
-addClickableTerritory(countryName : String, path : String, hoveredPath : String) : ClickableTerritory
-makeClickableMap() : void
-howToButton() : void
-addPlayButton() : void
+disableAllComponents() : void

Figure 15: GameView Class

Attributes

- **final int WIDTH:** The maximum width for the pane.
- **final int HEIGHT:** The maximum height for the pane.
- **final String DIRECTORY_NAME:** Holds the directory for where images are located as a String.
- **final String FILE_NAME_HELPER:** Holds a String object as "_bw.png" as all the files have the same extension to them.
- **final String FILE_NAME_HOVERED_HELPER:** Holds a String object as "_hovered_bw.png" as all the hovered image files have the same extension to them.
- **final String[] TERRITORIES:** Holds a String array consisting of all of the territory names.
- **final String BACKGROUND_IMG_PATH:** Holds the directory for where images are located as a String.
- **private List<ClickableTerritory> territoryList:**

Constructors

- **public GameView(Stage stage):** Creates a new territoryList object as an empty ArrayList of ClickableTerritory objects, sets preferred width and height of the pane as final variables WIDTH and HEIGHT, then calls the makeClickableMap() and addPlayButton() methods consecutively.

Methods

- **private void makeClickableMap():** Calls the parent's setOnMousePressed(), setOnMouseReleased(), setOnMouseEntered(), setOnMouseExited() methods with their parameter functions as empty, removing any event done after any mouse interaction with the territories.

- **private void bindMapToPaneSize(ImageView imageView):**
- **private void addPlayButton(Stage stage):**
- **public void disableAllComponents():** Calls the removeEventListeners() method for each clickableTerritory object inside the territoryList ArrayList.

ClickableTerritory

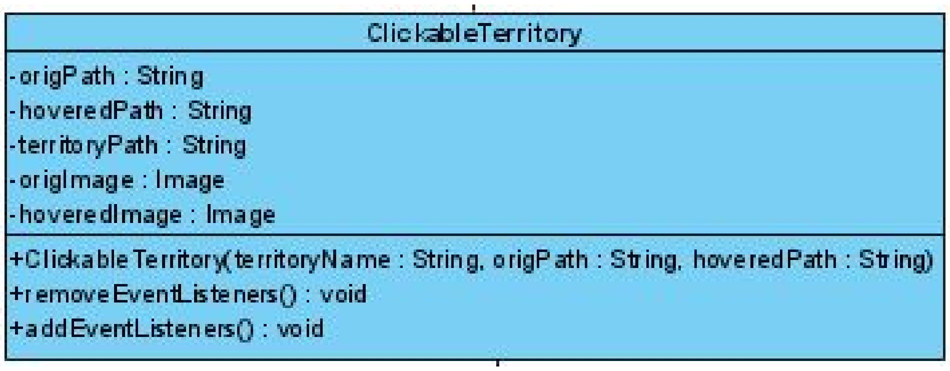

ClickableTerritory
-origPath : String
-hoveredPath : String
-territoryPath : String
-origImage : Image
-hoveredImage : Image
+ClickableTerritory(territoryName : String, origPath : String, hoveredPath : String)
+removeEventListeners() : void
+addEventListeners() : void

Figure 16: ClickableTerritory Class

Attributes

- **private String origPath:** Holds the path for the original image as a String object.
- **private String territoryPath:** Holds a temporary path object that is equal to what origPath is originally that will stay the same, as origPath will change during the course of the program.
- **private String hoveredPath:** Holds the path for the hovered image as a String object.
- **private Image origImage:** Holds a javafx.scene.image.Image object that is the original grayscale image that is constructed using the origPath variable.
- **private Image hoveredImage:** Holds a javafx.scene.image.Image object that is the hovered version of the image of territory that is constructed using the hoveredPath variable.
- **private boolean clicked:** Holds a boolean value based on whether the territory had been previously clicked or not.

Constructors

- **public ClickableTerritory(String territoryName, String origPath, String hoveredPath):** Creates a ClickableTerritory object constructed with the given parameters, assigning the class's origPath and hoveredPath attributes to the parameters of the constructor with the same name. Then constructs new Image objects using the given path variables to then assign them origImage and hoveredImage objects for origPath and hoveredPath variables, respectively. While constructing these Image objects, uses a try-catch system to catch any exceptions if there are any for there is the possibility of an invalid File IO Exception. Then calls the parent's setImage() method, using the origImage variable as a parameter. Finally, the constructor calls the addEventListeners() method to initialize the mouse listeners during the construction process.

Methods

- **public void removeEventListeners():** Calls the parent's setOnMousePressed(), setOnMouseReleased(), setOnMouseEntered(), setOnMouseExited() methods with their parameter functions as empty, removing any event done after any mouse interaction with the territories.
- **public void addEventListeners():** Calls the parent's setOnMousePressed(), setOnMouseReleased(), setOnMouseEntered(), setOnMouseExited() methods with their instructions as parameters as functions for each. For all of these, the value clicked is first checked and the method is exited if so and continued if not. If the method continues for setOnMousePressed(), setImage() of parent class is called to set the Image of ClickableTerritory to hoveredImage. For setOnMouseReleased(), after the if statement that checks for clicked variable, origImage variable is set to hoveredImage, making the setImage(origImage) call from the constructor set the Image of ClickableTerritory permanently as a clicked image. The methods setOnMouseEntered() and setOnMouseExited() set the image as hoveredImage and origImage respectively, after the check for clicked variable.

RPSGameView

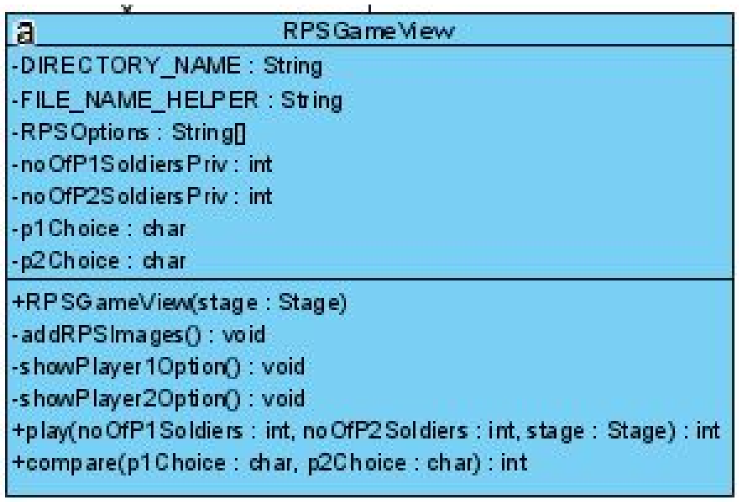

RPSGameView
-DIRECTORY_NAME : String
-FILE_NAME_HELPER : String
-RPSOptions : String[]
-noOfP1SoldiersPriv : int
-noOfP2SoldiersPriv : int
-p1Choice : char
-p2Choice : char
+RPSGameView(stage : Stage)
-addRPSImages() : void
-showPlayer1Option() : void
-showPlayer2Option() : void
+play(noOfP1Soldiers : int, noOfP2Soldiers : int, stage : Stage) : int
+compare(p1Choice : char, p2Choice : char) : int

Figure 17: RPSGameView Class

Attributes

- **final private String DIRECTORY_NAME:** Holds string including the directory of the rock-paper-scissors icons.
- **final private String FILE_NAME_HELPER:** Holds string including the common part of the file names of the rock-paper-scissors icons.
- **final private String[] RPSOptions:** String array that holds strings "rock", "paper" and "scissors".

- **private int noOfP1SoldiersPriv:** Holds the number of soldiers used in the war by the first player.
- **private int noOfP2SoldiersPriv:** Holds the number of soldiers used in the war by the second player.
- **private char choice p1Choice:** Holds the char value that is entered by the first player.
- **private char choice p2Choice:** Holds the char value that is entered by the second player.

Constructors

- **public RPSGameView(Stage stage):** Creates RPSGameView object with the given parameter.

Methods

- **public void showPlayer1Option:** Shows player 1 the letters / numbers corresponding to either rock , paper or scissors which are ”A”, “S” and “D”.
- **public void showPlayer2Option:** Shows player 2 the user the letters corresponding to either rock , paper or scissors which are ”1”, “2” and “3”.
- **public int play(int noOfP1Soldiers, int noOfP2Soldiers, Stage stage):** Method that takes soldier counts and stage as arguments. Gets keyboard input from the players one by one and compare them. Continues until one of the players loses all his / her soldiers. Returns 1 if the winner is player 1, 2 if the winner is player 2.
- **private int compare(char p1Choice, char p2Choice): Compares the choices of the players and compares them according to the rock-paper-scissors rules.** Returns 1 if the winner is player 1, 2 if the winner is player 2 and -1 if none of them wins.

Main

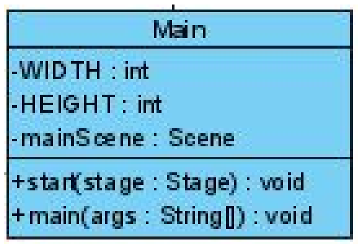


Figure 18: Main Class

Attributes

- **final private int WIDTH:** The immutable value that sets the width of the screen.
- **final private int HEIGHT:** The immutable value that sets the height of the screen.
- **Scene mainScene:** The scene object that the GUI will be set on.

Methods

- **public void start(Stage stage):** Creates the necessary objects for the GUI, overrides the start method in the Application class. Also features a try catch block for exception handling.

- **public static void main(String[] args):** Launches the game with the appropriate arguments.

MainMenuText

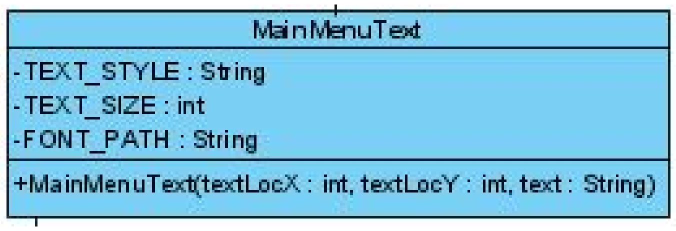


Figure 19: MainMenuText Class

Attributes

- **final private String TEXT_STYLE:** Sets the text style for the main menu.
- **final private static int TEXT_SIZE:** Sets the font size for the main menu.
- **final private static String FONT_PATH:** Sets the path for the font for the text in the main menu.

Constructors

- **MainMenuText(int textLocX, int textLocY, String Text):** Calls the constructor of the super class, which is Text. Sets fills to white and style to the attribute TEXT_STYLE. Loads the font in a try-catch block to handle possible exceptions.

BeforeGameView

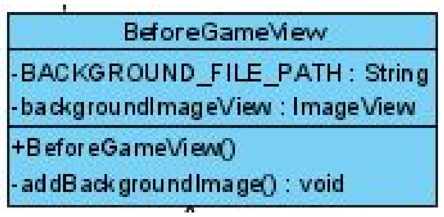

BeforeGameView
-BACKGROUND_FILE_PATH : String
-backgroundImageView : ImageView
+BeforeGameView()
-addBackgroundImage() : void

Figure 20: BeforeGameView Class

Attributes

- **private String BACKGROUND_FILE_PATH:** Set the file path for the background image.
- **public ImageView backgroundImageView:** The ImageView object for the background.

Constructors

- **BeforeGameView():** Creates the ImageView object for the background using the file path specified in the attributes.

Methods

- **private void addBackgroundImage():** Adds the background image to the class and binds it to the width and height of the screen so images do not distort when resizing the screen.
- **public void addGroup(Group g):** Adds Group g to the object.

MainMenuButton

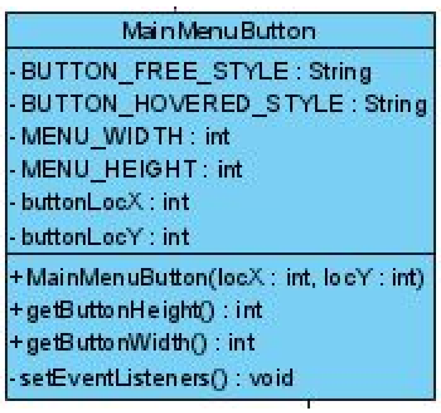


Figure 21: MainMenuButton Class

Attributes

- **final private String BUTTON_FREE_STYLE:** Variable to hold the style of a button that is not hovered over.
- **final private String BUTTON_HOVERED_STYLE:** Variable to hold the style of a button that is hovered over.
- **final private int MENU_WIDTH:** Sets the width of the menu.
- **final private int MENU_HEIGHT:** Sets the height of the menu.
- **public int menuButtonCount:** Variable to hold the number of buttons on the menu.
- **private int buttonLocX:** Variable to hold the x-location of the button.
- **private int buttonLocY:** Variable to hold the y-location of the button.

Constructors

- **public MainMenuButton(int locX, int locY, int count):** Sets the x-location and the y-location variables to the values given in the arguments, and the preferred height and width with the values specified in the attributes. Sets the number of buttons on the menu to the value given in the arguments. Sets event listeners and the X and Y layouts of the button.

Methods

- **public int getButtonWidth():** Returns the value of the width of the button.
- **public int getButtonHeight():** Returns the value of the height of the button.
- **public int getButtonLocX():** Returns the value of the x-location of the button.
- **public int getButtonLocY():** Returns the value of the y-location of the button.

- **private void setEventListeners():** Sets the event listeners so that the button changes styles when it is pressed, hovered over, or is idle using the values found in the attributes.

5.3.3 Data Access Subsystem

The Data Access Subsystem only consists of DataManager class. It is planned that the current condition of the Game will be either written into the .txt file or read from the .txt file.

Data Manager

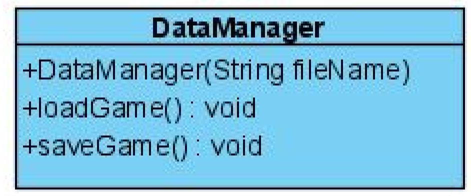


Figure 22: DataManager Class

Constructors

- **DataManager(String fileName):** Creates a DataManager class which will be used either to load the game data from the file with the given name or save the game to the file with the given name.

Methods

- **public void loadGame():** Loads the data of the previous game to continue playing from the text file by parsing.
- **public void saveGame():** Saves the current game into a text file to be able to continue next time.

5.4 Packages

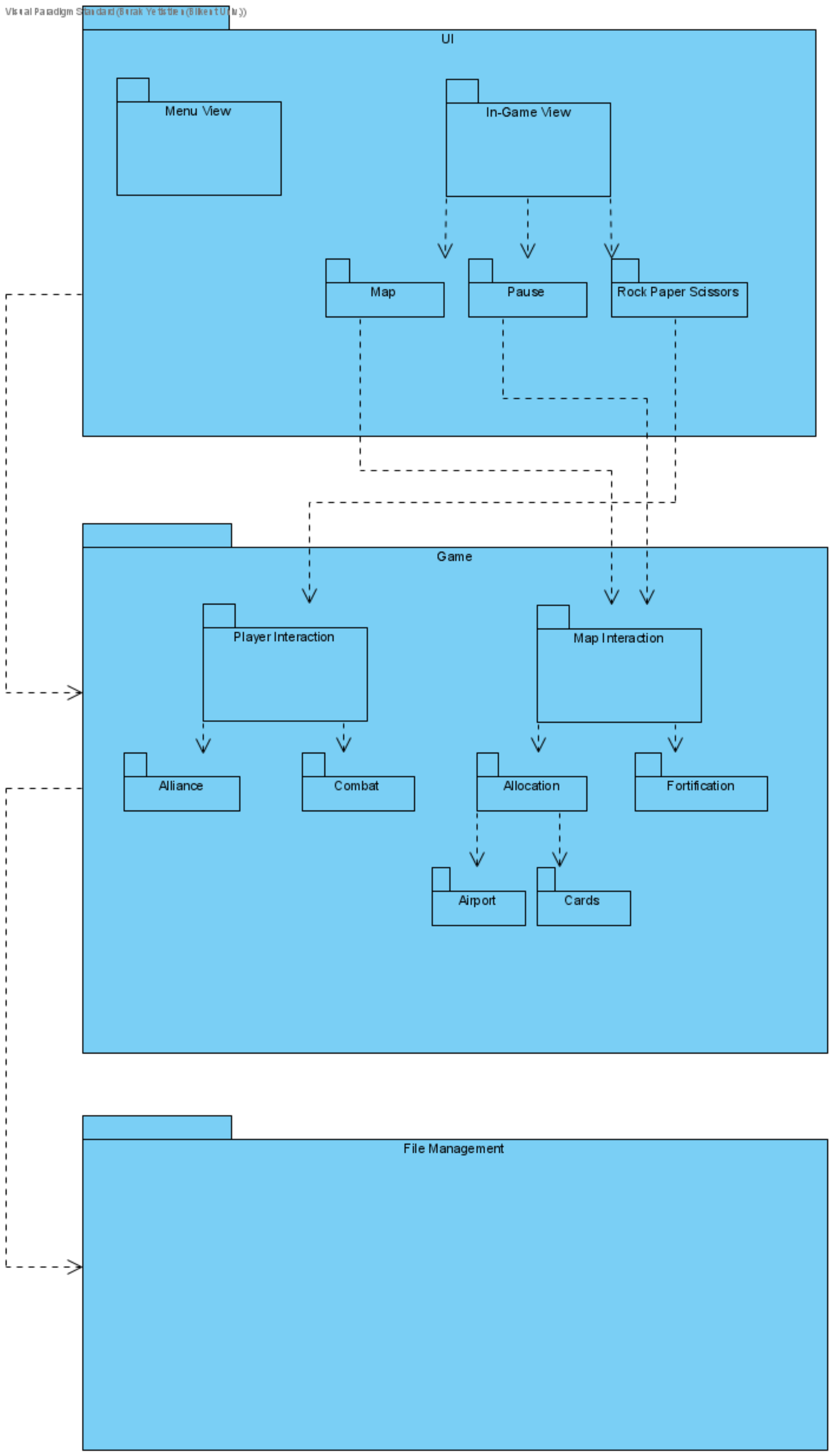


Figure 23: Package Diagram of the System

Our system will be decomposed into packages in accordance with our subsystem decomposition. We will have three main packages, them being the UI Package, the Game Package, and the File Management Package. The UI package will further contain packages having to do with the user interface of the game. It will have two subpackages, named Menu View and In-Game View. The Menu View package will contain the classes necessary for the creation of the menu screens of our game and the transitions between them. The In-Game View package will be further divided into three subpackages, called the Map package, the Pause package, and the Rock-Paper-Scissors package. These will contain the classes necessary for the creation of the screens that will be used during the gameplay of our game.
The Game package will contain the classes which will form the logic and the entities of our game. It will be divided into two subpackages called the Player Interaction package and the Map interaction package. The Player Interaction package will contain classes having to do with parts of our game that include players interacting with one another, and will be further decomposed into Alliance and Combat packages, the two main features of our game that have player to player interaction. The Map Interaction package will contain classes having to do with functionalities of our game that include players interacting with the map. It will be further decomposed into two subpackages called the Allocation package and the Fortification package, the two phases of a player’s turn that require the player interacting with the map. The Allocation package will further be separated into two packages called Airport and Cards, which will have to do with the airport building feature and the card exchanging feature of our game respectively. The Fortification package will not be decomposed into further packages.
The File Management package will likewise not be decomposed into further packages, and will include classes having to do with our saving and continuing from a save file features. It will contain classes that will generate the text file serving as the save file when the user wishes to save their game, and will as well contain a class that will parse through said text file serving as the save file when the player wants to launch a game from a save file.

6. Glossary

In this section will be definitions as well as explanations of specific terms having to do with the tabletop game we are implementing, RISK Global Domination, as well as terms having to do with the new features that we have added.

6.1 Territory

A territory is any individual region on the map of the game. A player wins once they conquer all of the territories within the board. Territories are painted in the color of the player controlling them on the map to show who they belong to, with as well a number in the middle of them showing the number of troops present on that territory. A territory will also be painted grey with a zero indicating no troops are present if no player is currently holding control of the territory.

6.2 Continent

A continent is a collection of territories. Continents range from small one such as Australia to large ones such as Asia. Once a player is able to conquer an entire continent for more than a turn, they will receive additional troops at the start of their turn, the number of which will be proportional to the size of the continent.

6.2.1 Target Continent

Players will be able to declare target continents at the start of the game. This will allow them to declare which continent they will be aiming to conquer before the game, and if they are able to conquer their target continent they will receive additional troops in addition to the ones they will have already received for conquering said continent. Two players may declare the same target continent at the start of the game. Sometimes referred to as “Goal Continent” as well.

6.3 Cards

If a player is able to conquer at least one territory during their turn, at the end of their turn they will be awarded a random card. These cards will be of any of the four following types: Infantry, Cavalry, Cannon, or Joker. If a player has three cards of matching type in their possession (with the joker cards being able to be substituted for any other type), they will be able to turn those cards in to receive a number of troops.

6.4 Player

A player will be the individual in control of an army, waging battles in order to achieve global domination and win the game. Each player will have their own starting territories, the same number of starting troops, as well as a color that will be used to mark territories currently under their control. A player will lose the game and be eliminated if they lose control of all of their territories and thus run out of soldiers, and will win the game by achieving global domination.

6.5 Troops/Soldiers/Unit of Troops

These all refer to the same thing, the troops/soldiers/unit of troops a player has on any territory is the number of deployable units that they have on that territory which they can use to attack any neighboring territories or defend that territory should another player decide to attack there.

6.6 Turn

Our game is turn-based, with play revolving around turns of players. A turn of a player is separated into three distinct phases; the troop allocation phase, the attack phase, and the fortification phase. At the beginning of their turn, players will also have the option to propose an alliance to any other player that is not already an ally, or secede from already formed alliances.

6.7 Troop Allocation Phase

This phase is the first phase of a player’s turn. In this phase players will decide where to place the troops that they will receive at the start of their turn. They will be asked to allocate all of the troops that they have received, and will be able to do so in any manner that they wish.

6.8 Attack Phase

This will be the second phase of a player’s turn. In this phase players will be able to attack neighboring territories of any non-allied other player. A player will have to have at least two units of troops present on any territory that they wish to launch an attack from. Players will also have the option to skip this phase entirely if they decide not to attack any territory during that turn.

6.9 Fortify Phase

This will be the third and last phase of a player’s turn. In this phase players will be able to choose a number of soldiers from one of their territories and transfer those soldiers to any other of their territories, provided that those two territories are connected, with either the player’s own or allied players’ territories in between the two territories.

6.10 Airport

Players will have the option to exchange five units of troops with an airport on any territory in which they have at least six soldiers. Doing so will place an airport on that territory, allowing players to attack territories of up to three units away from the territory in which an airport is present. If a territory with an airport is conquered by a different player, the airport will be removed; presumed to have been destroyed during the battle.

6.11 Alliance

Players will have the ability to form alliances with each other, provided that there are more than two players in the game. Once two players are allied, they will be able to freely cross each other’s borders to either attack a territory or during their fortification phase. Alliances can be broken during the start of either player's turn, and allying players will not be able to attack each other for the duration of their alliance.

6.12 Rock-Paper-Scissors

Once a player declares combat on another player, battles will take place in the form of rounds of rock-paper-scissors. After each round, the player who has lost the round will lose a single unit of troops. The battle will be over once the defending side has run out of troops or all the attacking troops have been defeated.

6.13 Global Domination

When a player is able to conquer all of the territories on the map, they will have achieved global domination. This is the winning condition for any player, the game ends once a player has achieved global domination and that player will be declared the winner of the game.