

CS 319
Object Oriented Software Engineering
2020 Fall



Design Report

Group 1-H

Kaan Atakan Array – 21703187

Rüzgar Ayan – 21801984

Emre Derman – 21703508

Kamil Kaan Erkan – 21601166

Cankat Anday Kadim – 21802988

Umut Ada Yürüten – 21802410

Table of Contents

1 Introduction	3
1.1 Purpose of the system	3
1.2 Design goals	3
1.2.1 End User Standard.....	3
1.2.2 Ease of Learning	3
1.2.3 Ease of Use	3
1.2.4 Maintenance Standard	4
1.2.5 Reliability	4
1.2.6 Modifiability	4
1.3 Trade-Offs	4
Usability-Functionality	4
Portability-Performance.....	4
1.4 Definitions, acronyms and abbreviations	4
1.5 Overview	5
2 Proposed Software Architecture.....	5
2.1 Subsystem Decomposition.....	5
2.2 Hardware/software mapping.....	6
2.3 Persistent data management	6
2.4 Access control and security	6
2.5 Global software control.....	6
2.6 Boundary conditions.....	6
Initialization	6
Termination.....	7
Failure.....	7
3 Subsystem Services	7
3.1 GUI Layer	7
GUI Layer	7
Input Layer.....	8
Application Layer	8
Storage Layer	8
4 Low level design	8
4.1 General Class Structure.....	8
4.2 Design Patterns	9

Strategy:	9
Singleton:.....	9
MVC:.....	9
4.3 Screen Classes	10
Screen.....	10
HowToPlayScreen.....	10
GameScreen	10
Menu	11
Main Menu	11
Options Menu	11
Pregame Menu	11
Pause Menu.....	11
4.4 Manager Classes	11
Game Manager	12
Storage Manager	12
Battle Manager	12
Input Manager	13
Settings Manager	13
Map Manager	13
4.5 Application Classes.....	13
Player Class.....	14
Faction Class	15
Map Class.....	15
Match Class	16
Region Class	17
Timer Class.....	17
Card Class	18
Mission Class.....	18

1 Introduction

RISK is the digital rendition of the age-old game with the same name with new concepts to spice up the gameplay aspects. RISK is designed with multiple players in mind with either human players or bots being the opposition. It is the aim of the players to dominate others and the world through means of diplomacy and conquest.

1.1 Purpose of the system

RISK features many systems to closely represent the feeling of playing the original board game as much as possible through the medium of computers. It is the aim of the game to drive the player to accomplish their mission, whether it be one of the many specified missions or world domination, through any means necessary. The game implements many of the core features that are present in the original board game such as additional army cards, different game modes such as conquest and secret mission. The entire game is played on a world map and the game features the necessary army movement and attack elements to present the opportunities the player can follow.

1.2 Design goals

RISK is judged with two groups in mind, the players and the developers thus our game will be designed around the needs of the aforementioned groups.

1.2.1 End User Standard

The player's understanding and familiarity with RISK and the design approaches we will follow to ensure such familiarity are as follows.

1.2.2 Ease of Learning

It is quite important for the user to pick up the game on his own and be able to learn every detail in RISK with one or at most two playthroughs. RISK implements a "How to Play" in both the main menu and the in-game "Pause Menu" to ensure that the player can always access help as he needs it. As well as that, the player will also be able to choose the level of difficulty that the bots will provide, granted there are any bots, to allow the user to ease himself into the mechanics of RISK.

1.2.3 Ease of Use

RISK is designed to be intuitive and easily accessible by the player without navigating menus or searching for a particular setting. This is accomplished by breaking everything down into its own menu or separating UI elements that might confuse the player about their effects.

1.2.4 Maintenance Standard

After the release, the development and the maintenance are crucial parts in development, as the game's longevity will only support its player retainability. The following implementations of this criteria are crucial for the implementation of this standard.

1.2.5 Reliability

RISK is designed in a way to run on any system without a drop in frame rates, without response lag exceeding draw time of a frame and without crashes. At the release version of the game there should be no game breaking bugs and the game should never crash as long as there isn't a problem with the files.

1.2.6 Modifiability

RISK offers two maps as default but either the team or the user will be allowed to add or edit maps to with patches. RISK will also allow customization of missions and new missions being added into the game. The game will allow the modification of the set number of factions and will allow the users to add new factions into the game. The extensive customizability that is offered in RISK is accomplished by structuring the aforementioned elements in easily accessible and modifiable groups.

1.3 Trade-Offs

Usability-Functionality

RISK restricts users in some ways that are not present in the tabletop version to achieve greater usability. This is also the case with the UI, as it will promote simplicity and will lack greater functionality for better user control.

Portability-Performance

RISK will be relying on Java Programming Language on all of its features. This means that the cross-platform maintainability and portability is quite high, however using Java means the use of Java Virtual Machine while running the game and there will be a negative performance effect compared to languages such as C. The lack of performance compared to other languages, however small it may be, will in turn grant us flexibility with development as the team won't have to port the game into other platforms.

1.4 Definitions, acronyms and abbreviations

MVC – Model View Controller

(G)UI – (Graphical) User Interface

1.5 Overview

RISK is trying to be a faithful representation of the original board game and also present new features into the game to make it more up-to-date with the 21st century. In accordance with the original game, our rendition is designed around a player base of any age and any culture.

2 Proposed Software Architecture

2.1 Subsystem Decomposition

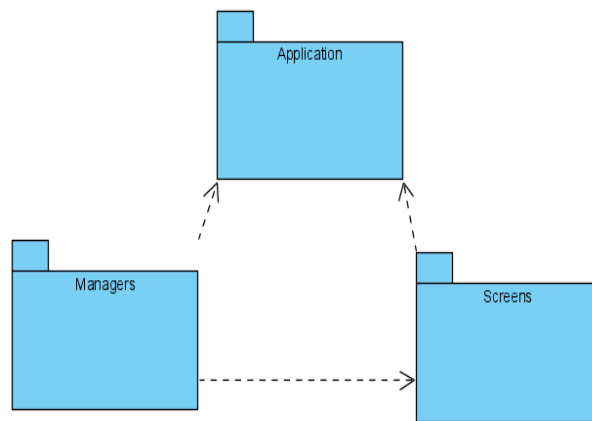


Figure 1: Subsystem Decomposition

For the decomposition of the system, we have decided to use a similar approach to the Model-View-Controller (MVC) pattern since it was a good fit for our project and our needs. We have divided the system to three subsystems: Application, Managers and Screens. This decomposition is not as strict as MVC, but they are very similar.

This subsystem decomposition was already compatible with our initial class diagram designs and it will make it easier to extend them since it is very flexible. This flexibility is important since we are planning to add many new features to the game and since extensibility is one of the nonfunctional requirements.

The Application subsystem will contain the classes that are carrying data, the Screens subsystem will contain the visual components and the Managers subsystem will contain the classes that provide the communication between the model and view classes.

2.2 Hardware/software mapping

Game will be running on a single computer that must at least have a mouse and Java Runtime Environment. Keyboard is not necessary, but some actions that are done with a mouse can also be done with a keyboard. Internet connection will not be necessary. Since the game is mostly static, other hardware requirements of the computer will also be minimal.

2.3 Persistent data management

In our game, players can save the game to not lose the current game session and load from the saved game to continue the previous game session. To be able to save and load the saved game, persistent data storage will be needed. Using this data storage, Match and Settings classes will be stored. To address this need, we will use a basic file system instead of a database system. Since this is a turn-based game that is played on a single computer, there will be only one reader and writer to the file. Therefore, for our case, instead of dealing with the complexity of the database system, using a basic file system will be more appropriate.

2.4 Access control and security

In our game, each player has the same role and one has no privilege over others. Also, since this is a turn-based game that is played on a single computer and therefore, does not require internet connection and since saved game data does not contain any private information, our game will not require access control or security measures.

2.5 Global software control

In our game, there is a GameController class that handles the flow of events in the game. Also, since the game depends on mouse events, software control is chosen to be event-driven.

2.6 Boundary conditions

Initialization

First run:

Risk will be provided as an executable .jar file and it will only require Java Runtime Environment to be installed on the player's computer.

Start up:

The player will be prompted with an option to either “Load Game” or to start a “New Game” (Analysis Report Figure 9.). If the player chooses to load a saved game, saved game data will be read from a .rsk file and the game will be initialized accordingly.

Termination

The player will be able to quit the game session using the “Exit” button (Analysis Report Figure 9.) on the Pause Menu. After clicking on the “Exit” button, the player will be prompted with an option to “Save Game”. If the player chooses to “Save Game”, game data will be written into a .rsk file. After this, the player will be returned to the “Main Menu”. In the “Main Menu”, the player will be able to terminate the game using the “Exit” button. The player will also be able to terminate the game using the close window button of the operating system. After this, the player will be prompted with an option to “Save Game”.

Failure**Corrupted Persistent Data:**

During the start up, if the player chooses to “Load Game” and if the saved data is corrupted or invalid, a failure might occur. When corrupted data is encountered, the system stops reading the file and displays an error message.

Missing files:

If the game files are missing, because files are quarantined or removed by antivirus software or deleted by the player accidentally, a failure might occur. Game displays an error pop-up and then terminates the program.

3 Subsystem Services

3.1 GUI Layer

GUI Layer

GUI layer will consist of JavaFx components like buttons, bars and textfields. It will also contain the graphical components that are created by us such as our maps. Will will mostly use built-in JavaFx components, since JavaFx is reliable and stable it will make our implementation easier.

Input Layer

This layer is together with our GUI layer since the user input is directed to our GUI layer. This layer will handle the user input through the mouse listeners provided by JavaFX library

Application Layer

This layer will provide us with the game logic, it will implement the game mechanics through the components that we will be created.

Storage Layer

This layer will handle the storage management, it will read the game data from storage unit to load the game and it will write the game data to the storage when the user wants to save the game state.

4 Low level design

4.1 General Class Structure

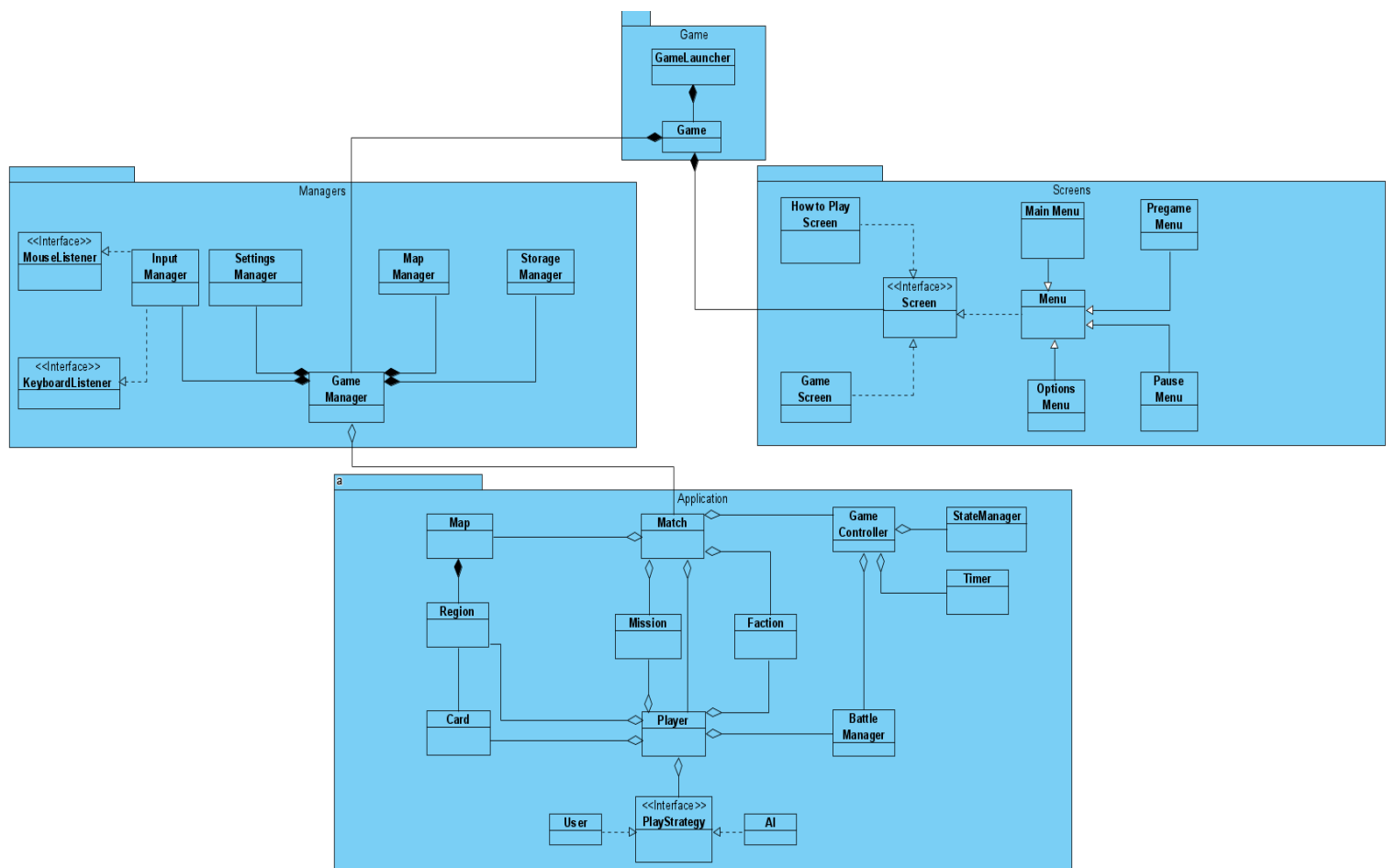


Figure 2: General Class Structure

Above diagram shows the general structure of the classes, next, each part of these will be explained in detail.

4.2 Design Patterns

Strategy:

To enable different algorithms to be used by the different difficulty levels of AI and to be able to change these in the runtime, we have added Strategy Design Pattern to the “**Player**” class which defines how he will play. This will both make it easy to add and configure the difficulty levels and make the code more readable.

Singleton:

We have used the Singleton Design Pattern in the “**Game**” class. Singleton is applicable here, because obviously, Game class should have only one instance at the runtime as it controls the game window. Other than that, this will also make it easy for other classes to access the “**GameManager**” which is connected to the Game.

MVC:

In our game we used MVC design pattern for both in the general structure of the game and in the screens package. We used MVC design pattern because it separates the logic from the view and makes our implementation of both parts easier for us. It also provides us with extendibility so if we want we can implement our game to different platforms more easily.

4.3 Screen Classes

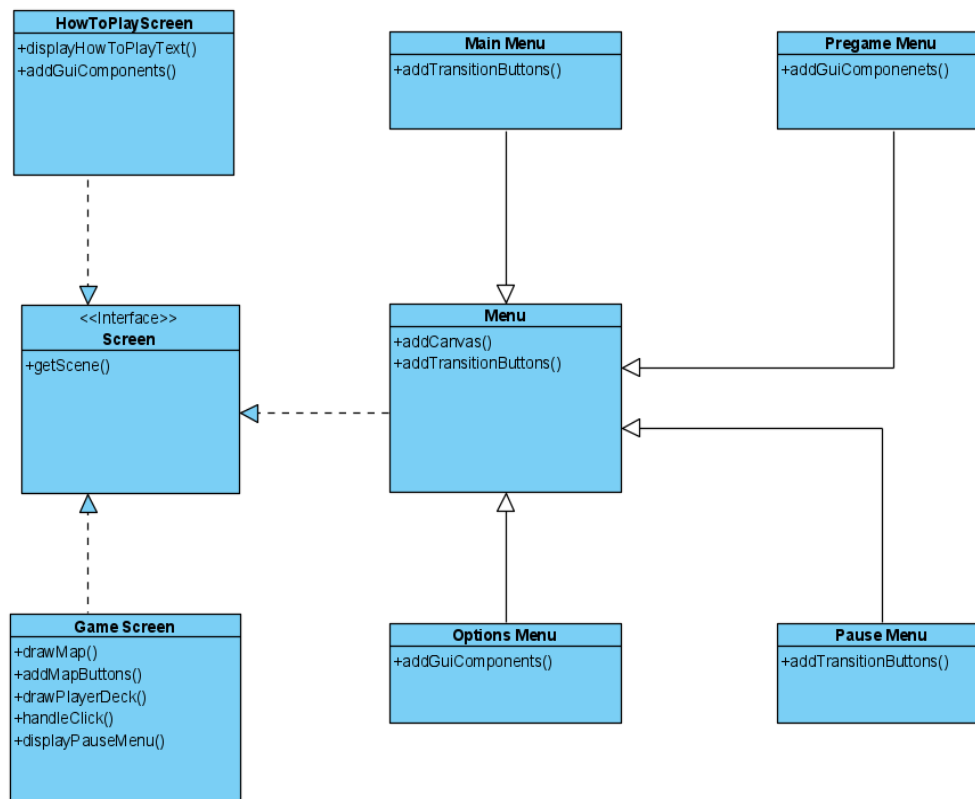


Figure 3: Screen Classes

The screen classes constitute the Screens package in our application. They undertake the role of both being a view and being a controller by handling the user input and transmitting the necessary data to our game logic and updating the view accordingly.

Screen

Screen is an interface that all of our important screen classes implement. So it sets the ground rules for our screens in our application.

HowToPlayScreen

HowToPlayScreen is a class that displays our tutorial of the game so that the user can understand the game mechanics.

GameScreen

GameScreen will be the class that displays our game screen, which consists of our map, player deck, and handles user input to the map and player deck and updates them accordingly.

Menu

Menu class will be the parent class of our menu system, all other menu classes will inherit it. It will create a basic menu structure and help us to implement other menus more easily.

Main Menu

Main Menu class will represent our opening screen, it will have navigation buttons to the other screens, it will handle the user input and ensure the transition to other screens.

Options Menu

Options Menu class will represent our options menu, it will be connected to our game logic since it can make some differences on it. It is also responsible for handling user input and displaying the options screen.

Pregame Menu

Pregame Menu class will represent our pregame menu, it will mainly display the pregame menu and handle user input. Also since it is an initialization screen it will send data to the Game class to initialize the game.

Pause Menu

Pause Menu class will represent our pause game menu, the button that opens pause game menu will position on the player deck section of our game screen. It will display the options that are available to the player on the pause screen and it will handle user input.

4.4 Manager Classes

The purpose of the *manager* package is to rarefy the complicated algorithms and application-dependent functions from GameModel and GameView classes. In such a way, complex challenges can handle single method calls by separating the managers.

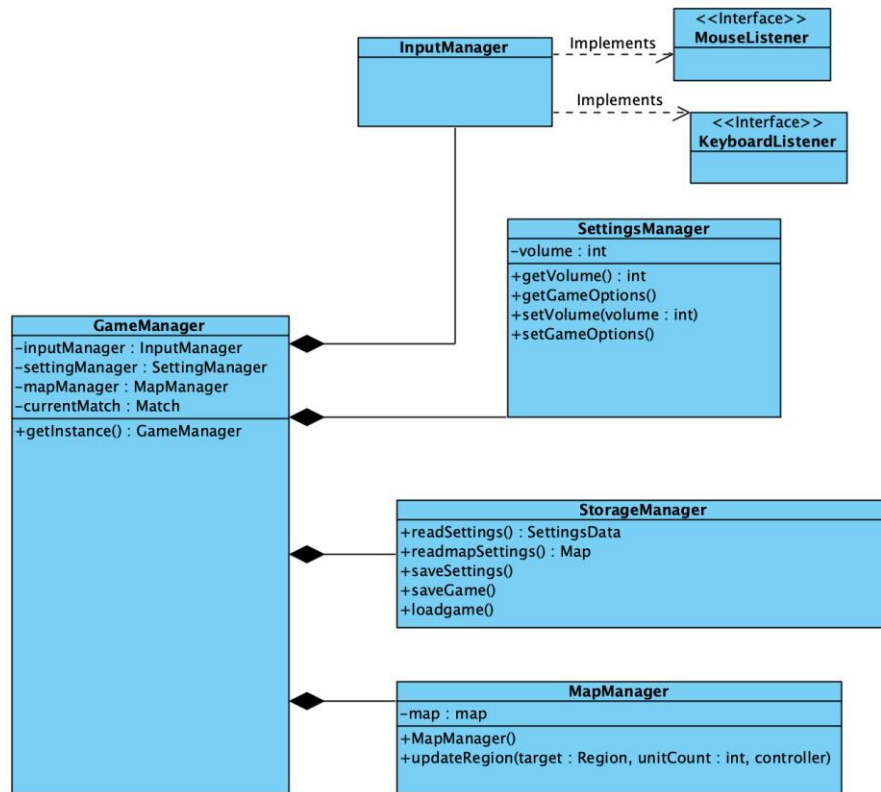


Figure 4: Manager Classes

Game Manager

The **GameManager** class allows to control other sub-control classes and holds other control classes as attributes. This class is the core of the game mechanics since it updates the model according to the *Input Manager* and *Battle Manager*.

Storage Manager

The **StorageManager** class allows to store the settings data as files in the game and saving of the user process as well as the preferred game settings.

Battle Manager

The **BattleManager** class is designed to simplify the battle case of the game. Using this class, the result of the battle will be clarified and the appropriate model and view classes will be updated.

Input Manager

InputManager class allows user interactions. The class handles the inputs from the user and updates the game by using other control classes.

Settings Manager

SettingsManager class allows changing in game settings such as volume and game options according to the interactions received from *OptionsScreen*.

Map Manager

This class is for updating the status of the target region according to the unit count, annually. Also this class updates the map synchronously with the BattleManager class.

4.5 Application Classes

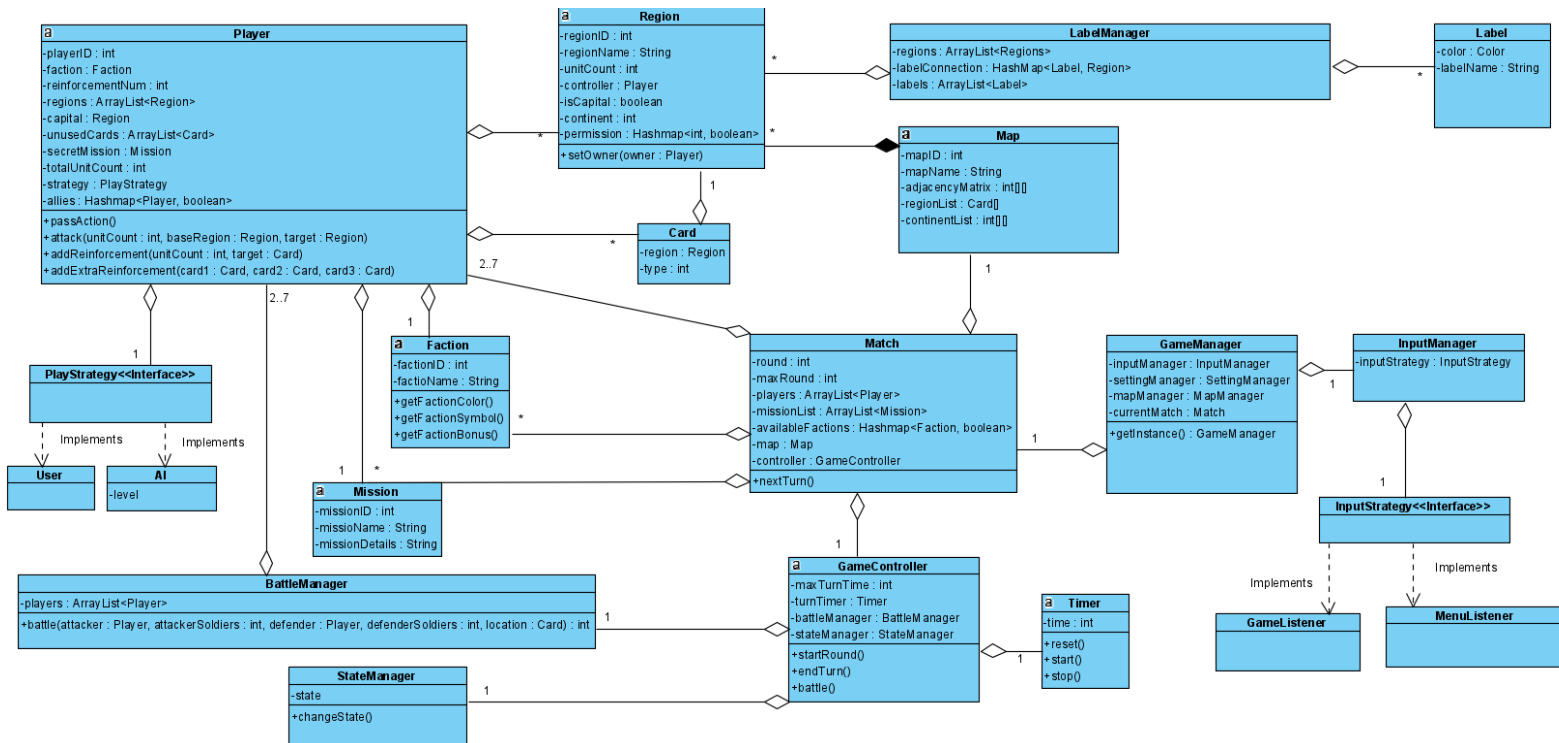


Figure 5: Application classes

Player Class

Player	
-playerID : int	
-faction : Faction	
-reinforcementNum : int	
-regions : ArrayList<Region>	
-capital : Region	
-unusedCards : ArrayList<Card>	
-secretMission : Mission	
-totalUnitCount : int	
-strategy : PlayStrategy	
-allies : Hashmap<Player, boolean>	
+passAction()	
+attack(unitCount : int, baseRegion : Region, target : Region)	
+addReinforcement(unitCount : int, target : Card)	
+addExtraReinforcement(card1 : Card, card2 : Card, card3 : Card)	

Attributes:


- int playerID: This is the identifier of the player and is used to refer to the instance in different classes and functions.
- Faction faction: This the players faction. Different factions provide bonuses different bonuses
- int reinforcementNum: This is the current reinforcement number according to the number of Region cards the player possesses.
- ArrayList<Regions> regions: All the regions the player possesses.
- Region capital: The capital region of the player. This attribute is used if the game mode includes capitals.
- ArrayList<Cards> unusedCards: The list of region cards the player possesses that has not been used for extra reinforcements.
- Mission secretMission: The mission given to the player; it is the player's own victory condition.
- int totalUnitCount: The total number of units the player possesses in all of the regions.
- HashMap<Players,boolean> allies: The allies this player possesses.

Methods:

- passAction(): Passes the current action phase (attack stage, maneuver stage) for the player.
- attack(int unitCount, Card baseRegion, Card target): Method for player's attack onto enemy region. Specifies the unit count, origin of attack's location and the target location.

- `addReinforcement(int unitCount, Card target)`: Method for reinforcing a specified region with specified amount of forces during the reinforcement stage of player's turn.
- `addExtraReinforcement(Card card1, Card card2, Card card3)`: Used to add extra reinforcements using player's unused region cards.

Faction Class

 Faction
-factionID : int
-factionName : String
+getFactionColor()
+getFactionSymbol()
+getFactionBonus()


Attributes:

- `int factionID`: Identifier of the faction.
- `String factionName`: Name of the faction

Methods:

- `getFactionColor()`: returns the faction color for units and regions.
- `getFactionSymbol()`: returns the faction symbol image.
- `getFactionBonus()`: specifies the faction's starting bonus.

Map Class

 Map
-mapID : int
-mapName : String
-adjacencyMatrix : int[][]
-regionList : Card[]
-continentList : int[][]
+updateRegion(target : Card, unitCount : int, controller : Player)

Attributes:

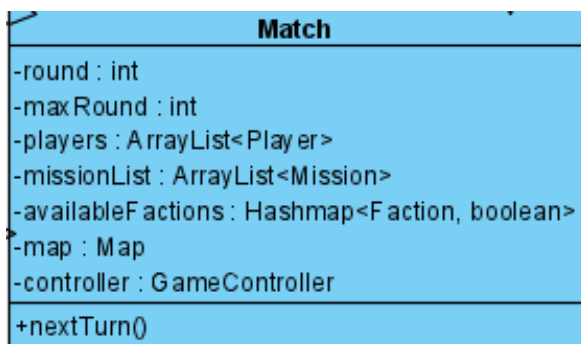
- `int mapID`: Identifying attribute of an instance.

- mapName: Name of the map.
- int[][] adjacencyMatrix: Since Map has a graph of regions inside, an adjacency matrix is used. The int values are the regions id's.
- Card[] regionList: list of all regions.
- int[][] continentList: List of all continents, by id, and their associated regions, by id.

Methods:

- updateRegion(Card target, int unitCount, Player controller): Updates a region's properties.

Match Class



Attributes:

- int round: Current round of the game
- int maxRound: Maximum round, if specified.
- ArrayList<Player> players: All the players in the current match
- ArrayList<Mission> missionList: All the missions of the game.
- HashMap<Faction, boolean> availableFactions: The list of all functions and whether they are taken or not.
- Map map: The current map.
- GameController controller: The GameController of this current game that controls the events between core models.

Methods:

- nextTurn(): Skips turn to next player in line.

Region Class

a	Region
-regionID : int	
-regionName : String	
-unitCount : int	
-controller : Player	
-isCapital : boolean	
-continent : int	
-permission : HashMap<int, boolean>	
+setOwner(owner : Player)	

Attributes:

- int regionID: Identifying attribute of the instance.
- regionName: Name of the region.
- controller: Current owner of the instance.
- boolean isCapital: Determines whether this region is a capital or not.
- HashMap<int,boolean> permission: This attribute is used for unit maneuvering. Allied players are allowed to use each other's regions when maneuvering; a player does not necessarily need to own a region in order to trespass it.

Methods:

- setOwner(Player owner): Changes the owner of the region.

a	Timer
-time : int	
+reset()	
+start()	
+stop()	

Timer Class

Attributes:

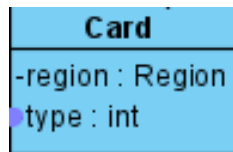
- int time: Stores the current time

Methods:

- reset(): Resets the current to 0.

- start(): Continues time if it was stopped.
- stop(): Stops the time

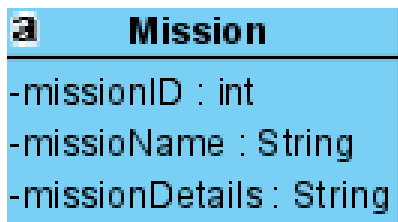
Card Class



Attributes:

- Region region: Specifies the region of the card
- int type: Stores the type of the card (Infantry, cavalry, artillery)

Mission Class



Attributes:

- int missionID: Identifying value of the instance
- String missionName: Mission's name
- String missionDetail: Details, objective and the win condition of the mission