



**CS319 Object Oriented Software  
Engineering  
Project Design Report  
First Iteration**

**RISE OF EMPIRES**

**Group 2E**

Can Kılıç - 21703333  
Ege Çetin - 21802305  
Emre Erciyas - 21600991  
Furkan Demir - 21802818  
Muzaffer Köksal - 21803125

## TABLE OF CONTENTS

<b>TABLE OF CONTENTS</b>	<b>2</b>
<b>1.Introduction</b>	<b>5</b>
<b>1.1 Design Goals</b>	<b>5</b>
<b>1.1.1 Trade Offs</b>	<b>5</b>
<b>1.1.1.1 Development time vs. Performance</b>	<b>5</b>
<b>1.1.1.2 Functionality vs. Usability</b>	<b>5</b>
<b>1.1.1.3 Efficiency vs. Portability</b>	<b>5</b>
<b>1.1.1.4 Understandability vs. Functionality</b>	<b>6</b>
<b>1.1.2 Criteria</b>	<b>6</b>
<b>1.1.2.1 Usability</b>	<b>6</b>
<b>1.1.2.2 Performance</b>	<b>6</b>
<b>1.1.2.3 Extendibility</b>	<b>6</b>
<b>1.1.2.4 Portability</b>	<b>7</b>
<b>1.1.2.5 Response Time</b>	<b>7</b>
<b>2. System Architecture</b>	<b>7</b>
<b>2.1 Subsystem Decomposition</b>	<b>7</b>
<b>2.2. Hardware/Software Mapping</b>	<b>8</b>
<b>2.3 Data Management</b>	<b>9</b>
<b>2.4 Access Control and Security</b>	<b>9</b>
<b>2.5 Boundary Conditions</b>	<b>9</b>
<b>3. Subsystem Services</b>	<b>10</b>
<b>3.1 User Interface Subsystem</b>	<b>10</b>
<b>3.2 Game Manager Subsystem</b>	<b>11</b>
<b>3.3 Model Subsystem</b>	<b>12</b>
<b>4. Low-Level Design</b>	<b>13</b>
<b>4.1 Final Object Design</b>	<b>13</b>
<b>4.2 Packages</b>	<b>14</b>
<b>4.2.1 Game Related Packages</b>	<b>14</b>
<b>4.2.1.1 Menu Package</b>	<b>14</b>

4.2.1.2 Game Management Package	14
4.2.1.3 Map Management Package	14
4.2.1.4 Game Entities Package	14
4.2.1.5 File Management Package	14
4.2.2 JAVA Related Packages	14
4.2.2.1 java.util	14
4.2.2.2 javafx.scene.layout	14
4.2.2.3 javafx.scene.events	15
4.2.2.4 javafx.scene.input	15
4.2.2.5 javafx.scene.image	15
4.2.2.6 javafx.scene.paint	15
4.3 Class Interfaces	15
4.3.1 Event Interfaces	15
4.3.1.1 ActionListener	15
4.3.1.2 MouseListener	15
4.3.2 Game Classes	15
4.3.2.1 Troop Class	15
4.3.2.2 Land Class	16
4.3.2.3 Country Class	17
4.3.2.4 Dice Class	17
4.3.2.5 Leader Class	18
4.3.2.6 Player Class	18
4.3.2.7 BonusEffects Class	19
4.3.2.8 Research Class	19
4.3.2.9 Menu Class	20
4.3.2.10 GameController Class	21
4.3.2.11 PlayerController Class:	22
4.3.2.12 ExchangeCardsController Class:	22
4.3.2.13 Credits Class	23
4.3.2.14 Startmenu Class	23

<b>4.3.2.15 Card Class</b>	<b>23</b>
<b>4.3.2.16 Artillery Class</b>	<b>23</b>
<b>4.3.2.17 Tank Class</b>	<b>23</b>
<b>4.3.2.18 Infantry Class</b>	<b>24</b>
<b>4.3.2.19 Nerds Class</b>	<b>24</b>
<b>4.3.2.20 Save Class</b>	<b>24</b>
<b>4.3.2.21 SoundController Class</b>	<b>25</b>
<b>4.3.2.22 ResearchTree Class</b>	<b>25</b>
<b>4.3.2.23 BoardController Class</b>	<b>25</b>
<b>4.3.2.24 BoardView Class</b>	<b>25</b>
<b>4.3.2.25 InputController Class</b>	<b>26</b>
<b>4.3.2.26 Effects Interface</b>	<b>26</b>

## 1.Introduction

It was the coldest night of 1939. Europe is at the edge of another chaos. Dark times are upon us! Take command of your troops and save your homeland from total annihilation. Empower your troops and defend your land. When you are no longer satisfied with defending your land, conquer the enemy lands, and expand your empire.

Rise Of Empires is a turn-based combat strategy game with GUI and designed for Windows desktop machines. The main point of the game is to command your troops each turn and try to either defend or attack your opponent's lands in order to take control of the map. In order to conquer all the maps, the player needs both luck and strategy as the attack and defending the value of the units are decided by research (base) and dice (multiplicative).

### 1.1 Design Goals

We are aiming to improve our non-functional requirements we did in our analysis report. We will examine our non-functional requirements by their testability. Also, by making choices between trade offs which explained below, we are aiming to upgrade our game.

#### 1.1.1 Trade Offs

##### 1.1.1.1 Development time vs. Performance

Java was chosen as the programming language to minimize development time, as it is easier to design graphical user interfaces with it. However, Java is a less powerful language than languages like C++ which means performance will take a hit.

##### 1.1.1.2 Functionality vs. Usability

For functionality a program should offer many functions and for usability the program must be easily usable so it should avoid too much complexity. In RISE of Empires we are just creating a simple game to increase player's enjoyment. Moreover, our game is not designed for too complex functions, the functions like drawing a card, making a research is tried to be as easy as possible in order to increase usability.

##### 1.1.1.3 Efficiency vs. Portability

Between the efficiency and portability, we will go with efficiency. Since our game will run on one computer and be played without the internet, also, as it is explained above, we want to escape complex functions. We will not make our game portable to cross platforms. It will be an efficient pc game, run with JAVA Virtual Machine.

We are using Java as our programming language which is a highly portable language that can support apps in android. However the API that we are using, JavaFX, cannot be directly used in mobile environments so we consider this tradeoff to be a middle ground between portability and efficiency.

#### 1.1.1.4 Understandability vs. Functionality

As we introduced above we want to escape too complex functions in our game. So, from this decision we believe that our game's understandability will be increased too. Together with usability and understandability, a player will be able to play the game with full understanding and he/she will have more fun.

### 1.1.2 Criteria

#### 1.1.2.1 Usability

While developing RISE, our first target is making RISE enjoyable and easy to understand. So, in order to succeed we are trying to make a user-friendly interface which the player will not lose him/herself in too many functions instead of focusing the game. Furthermore, RISE gives the player a chance to see tips and tutorials with how to play section.

#### 1.1.2.2 Performance

While developing RISE, our second target is we are trying to increase the performance of our game by using libraries related with GUI.

#### 1.1.2.3 Extendibility

While developing RISE, we also think about the future and make our game extendible depending on the feedback we will get.

#### 1.1.2.4 Portability

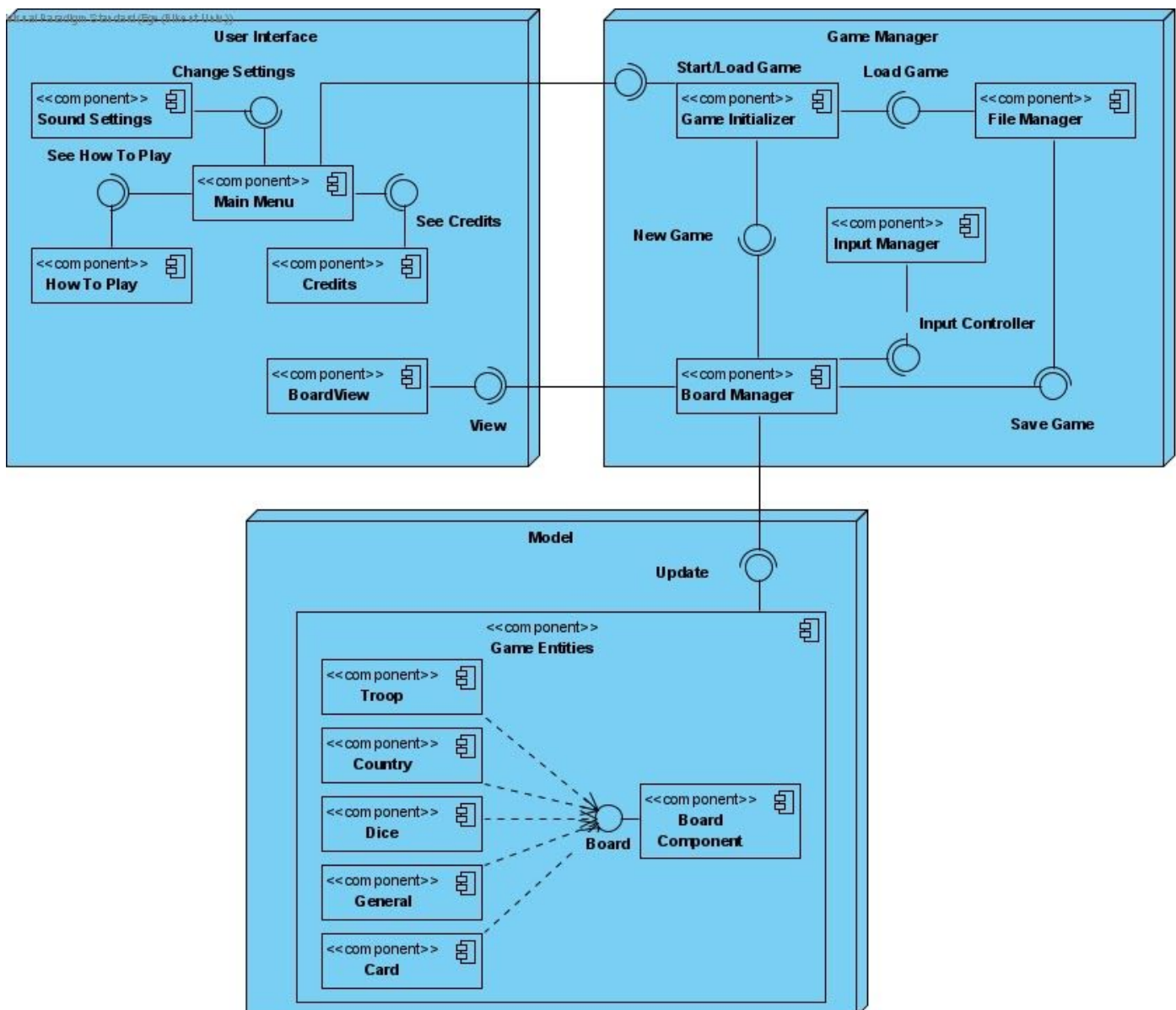
While developing RISE, we think about portability and thus we decided to use JAVA. Together with JAVA Virtual Machine our game will be portable between different pc platforms.

#### 1.1.2.5 Response Time

While developing RISE, we are highly aware of the importance of the FPS and response time. We make our implementation in a way that RISE will not get lower than 20 frames per second and when the player makes an action game will not exceed 0.5 seconds to reply according to that action.

## 2. System Architecture

### 2.1 Subsystem Decomposition



*Figure 1 System Decomposition*

While we were designing our system, we decided to decompose our system into subsystems in order to reduce unnecessary dependencies between components. While doing so we also reduced the dependencies between the subsystem which made the maintenance of our system easier. Because we made our system as a group of subsystems, it is also easier to add new elements or change elements of the system.

We decided to use the MVC (Model-View-Controller) design pattern. As it can be seen in the above figure our system is divided into subsystems based on the MVC (see figure 1).

We found that MVC is a favorable design pattern because of the following reasons:

- We have 3 subsystem which can be viewed as categorized as:
  - Model: Model subsystem holds all of the game entities inside. It only interacts with the Game Manager and also is controlled by it.
  - View: User Interface subsystem includes UI components which the user can interact with.
  - Controller: Game Manager subsystem holds the controllers for the whole system. It manages the initialization of the game and the continuation of the game loop.
- Our subsystem's relations can be easily reviewed as parts of the MVC pattern.

Before giving a detailed information about our subsystems, let us give the functionality of our subsystems and the relations between them.

- As mentioned earlier, User Interface subsystem provides an user interface for the user. The only relation between this subsystem and the Game Manager subsystem is for the initialization of the game and the update of the board the game is played on. Users can only use this interface to start a new game or load the previous game. The other interaction between the User Interface and the Game Manager is to update the board according to user inputs.
- Game Manager subsystem acts as the controller of the system.
  - Input Manager is responsible for getting the user input.
  - File Manager is responsible for saving the game each turn or providing the saved file to the Game Initializer.
  - Game Initializer starts a new game with default values or with the given values from the save files gotten from the File Manager.
  - Board Manager is responsible for the main game loop. This is where the game logic is located. It also updates the BoardView and the Game Entities.
- Model Subsystem provides the game entities to the Game Manager and its only updated by Board Manager.

## 2.2. Hardware/Software Mapping

We will implement the Rise of Empires game in Java, JAVAFX libraries. Thus, RISE of Empires will require Java Runtime Environment to execute properly, and also RISE of



Empires will require at least the Java Development Kit 8 or a newer version because these versions include JAVAFX libraries.

Moreover, RISE of Empires will require a mouse as a hardware requirement. Mouse will be used to interact with the game components. Users will control the menu selections, unit actions, end turn and cast dice with mouse inputs. With the need of these two requirements, our game's system requirements will be minimum, a computer with essential software will be sufficient to compile the game and run it.

For storage of the game, the text files will be used in order to store players' troop counts, the last turn's map (because we will save at the end of each turn), research tree progress, general and leader selections.

RISE of Empire will be played on a single computer with multiple players taking their turns to play. Therefore, RISE of Empire will not require any internet connection or database to operate.


## 2.3 Data Management

The game instances of RISE of Empires like cards, troops, map, etc will be stored in the player's hard drive. Moreover, we will use text files to store game data. Thus, a complex data storage system or database is not needed in this project. On the one hand, some of these text files will be instantiated during the implementation like map designs. These files cannot be modified, thus fixed. On the other hand, other text files will be stored in text files that can be modified. For instance, settings(sound settings, preferences), the cards, troops, and the lands players conquered should be modified during the game. Therefore, these files will be stored in modifiable text files. Furthermore, the .wav format will be used to keep sounds and .gif will be used to keep images.

## 2.4 Access Control and Security

In RISE of Empires, any kind of internet/network connection or database will not be needed. As a result, the game will be accessible for everyone if the game is loaded. Therefore, no action will be taken in terms of access control and security to prevent malicious actions or data leaks.

## 2.5 Boundary Conditions

Our game doesn't need any kind of installation; it will not have an executable with .exe extension. Our game will have an executable jar file and the game will be executed by a .jar extension. Thus, our game will be portable. It can be transferred to another computer without any kind of installation and will be executable. The game can be shut down by clicking  button on the top right or clicking "Exit" in the main menu.

If the game collapses or freezes during game time because of a performance or design issue, the unsaved data will be lost.

### 3. Subsystem Services

#### 3.1 User Interface Subsystem

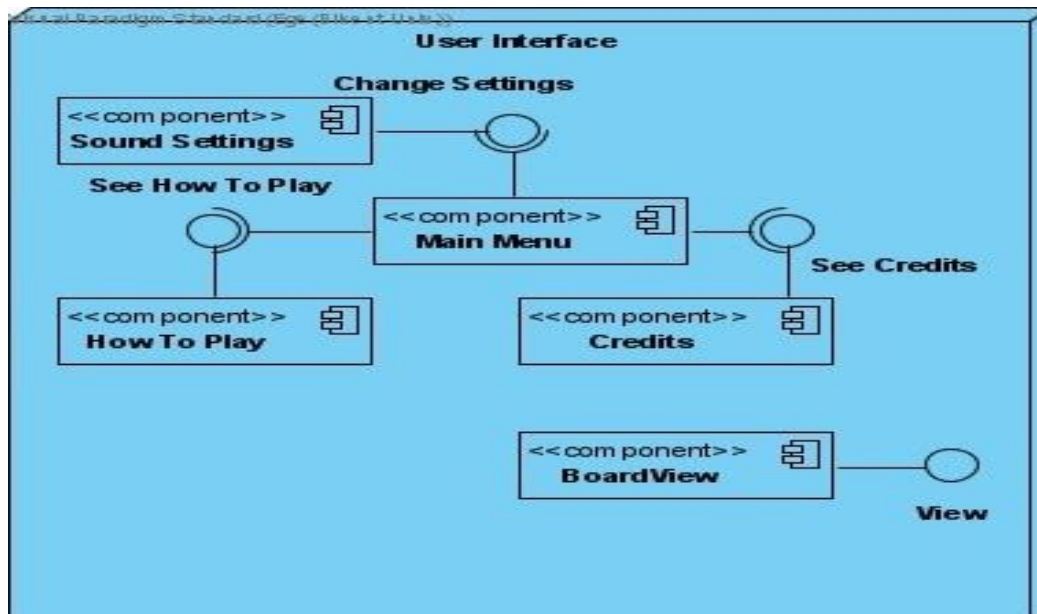


Figure 2.1 User Interface Subsystem Decomposition

This subsystem consists of 5 major components:

1. Main Menu
2. Sound Settings
3. How To Play
4. Credits
5. BoardView

When the user launches the program, they are met with the Menu. This menu is provided by the Main Menu component of the UI subsystem. Using the menu, users can access different parts of the game such as credits, how to play, settings, new game or load game.

BoardView provides users with an understandable model view. This component displays all of the views such as the boardView, playerView, troopView and such. This component is updated by Board Manager when a change occurs.

Users can access how to play, credits or the sound settings only on the main menu.

When the user wants to start a game, they can only do so from the main menu. This way, it becomes easier to handle the initialization of a game.

### 3.2 Game Manager Subsystem

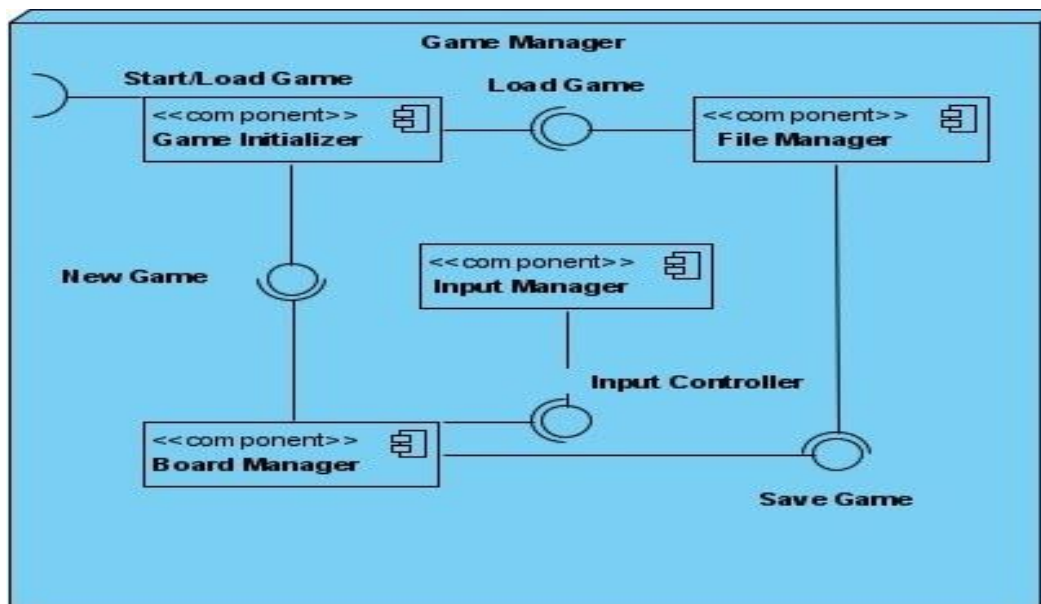


Figure 2.2 Game Manager Subsystem Decomposition

This subsystem consists of 4 major components:

1. Game Initializer
2. File Manager
3. Input Manager
4. Board Manager

As can be deduced from the name, Game Initializer takes a new game or load game input from the user and initializes the game. After the initialization, it passes the game info to the Board Manager to handle later user inputs.

File Manager is responsible for the save files. After each turn, it gets the board and player values from the Board Manager and saves it. It also handles pictures and sound files.

Input Manager handles user inputs and passes these inputs to the Board Manager for further processing.

Board Manager is the most important component of this subsystem. It handles the game loop for continuous game turns. This component also updates the game entities and the BoardView for an updated view.

### 3.3 Model Subsystem

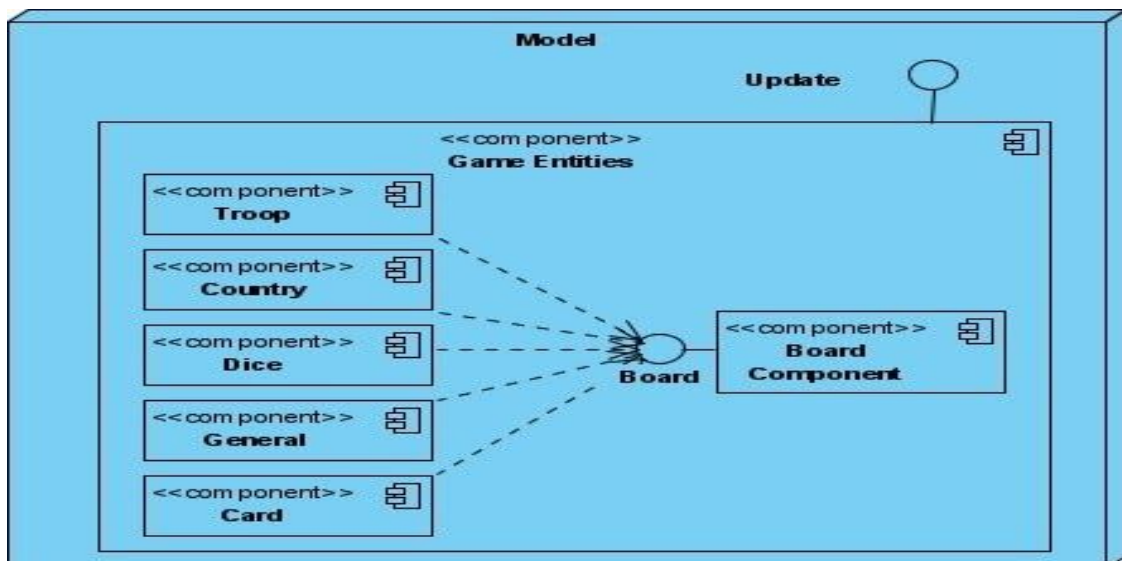


Figure 2.3 Model Subsystem Decomposition

This subsystem consists of 1 main component which consists of smaller components. This component is controlled and updated by the Board Manager.

Board Component represents the board game is played on.

Troop Component represents the troops that a player can have

Country Component represents the country the player has chosen and its attributes.

Dice Component represents the dice cast for each attack or defence option.

General Component represents the general the player has chosen and its attributes.

Card Component represents the cards users collected through the gameplay.

## 4. Low-Level Design

### 4.1 Final Object Design

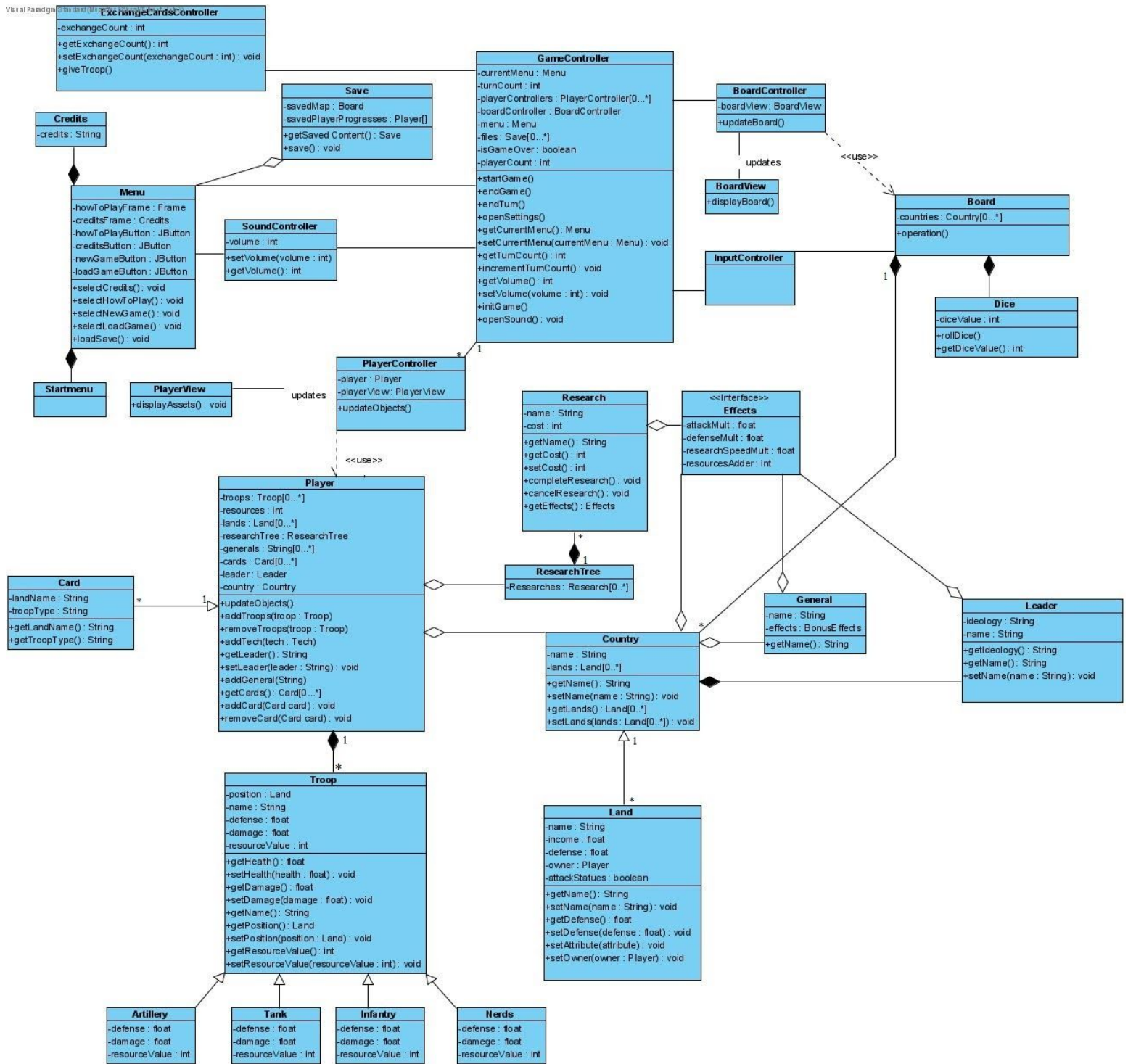


Figure 3 Final Object Design

## 4.2 Packages

While implementing RISE of Empire two kinds of packages: Game Related and JAVA Related. Game Related packages are simply the ones that we, developers, added for including the classes. The second one JAVA related packages are basically for external libraries.

### 4.2.1 Game Related Packages

#### 4.2.1.1 Menu Package

This package is basically responsible for the game menu classes and/or interfaces.

#### 4.2.1.2 Game Management Package

This package is responsible for RISE's heart. Most of the actions in the game are handled by this package's classes and interfaces.

#### 4.2.1.3 Map Management Package

This package is responsible for providing classes and/or interfaces for RISE's map.

#### 4.2.1.4 Game Entities Package

This package is responsible for providing classes and/or interfaces for the game objects.

#### 4.2.1.5 File Management Package

This package is responsible for providing file management in RISE.

### 4.2.2 JAVA Related Packages

#### 4.2.2.1 java.util

This package contains ArrayList which will be used for controlling multiple variables like players' lands, leader/general buffs and troop counts etc.

#### 4.2.2.2 javafx.scene.layout

This package will simply be used for creating a layout for the player interface with GUI.

#### 4.2.2.3 javafx.scene.events

This package will be used for the events and their control. This package gives users to deliver different inputs between different hardware components like a mouse.

#### 4.2.2.4 javafx.scene.input

This package is for taking inputs from keyboard and mouse via the handlers this package provides.

#### 4.2.2.5 javafx.scene.image

In some parts of the game like how to play section we intended to add images. In order to fulfill this desire this package will be used.

#### 4.2.2.6 javafx.scene.paint

For handling the visual output of a game this package will be used.

### 4.3 Class Interfaces

#### 4.3.1 Event Interfaces

##### 4.3.1.1 ActionListener

This interface will be used for receiving the events when an action occurs.

##### 4.3.1.2 MouseListener

This interface will be used for tracking mouse actions invoked by the player.

#### 4.3.2 Game Classes

##### 4.3.2.1 Troop Class

###### **Attributes:**

- private Land position: Stores which land on the map the troop is positioned.
- private String name: Stores the name of the troop.

- private float defense: Stores the defense value of the troop.
- private float damage: Stores the damage value of the troop.
- private int resourceValue: Stores the exchange value of the troop. For example, infantry has resourceValue = 1, so it takes 3 infantry to get 1 artillery of resourceValue = 3.

#### **Constructors:**

- public Troop(): Empty constructor that will be overloaded by Troop's subclasses.

#### **Methods:**

- public float getDamage(): Returns the damage value of the troop.
- public float getDefense(): Returns the defense value of the troop.
- public void setDamage( float damage): Changes the damage value of the troop to the value int the damage parameter.
- public String getName(): Returns the name of the troop.
- public Land getPosition(): Returns the land on which the troop is standing.
- public void setPosition(): Changes the position of the troop.
- public int getResourceValue(): Returns the resourceValue of the troop.
- public void setResourceValue( int value): Sets the resourceValue of the troop to the value parameter.

#### 4.3.2.2 Land Class

##### **Attributes:**

- private String name: Stores the name of the land.
- private float defense: Stores the base defense value of the land.
- private Player owner: Stores which player owns the land.

##### **Constructor:**



- `public Land(String name, float defense, Player player):` Initializes the land object.

**Methods:**

- `public String getName():` Returns the name of the Land object.
- `public void setName(String name):` Sets the name of the Land object.
- `public float getDefense():` Returns the base defense value of the Land.
- `public void setDefense(float defense):` Sets the base defense value of the Land objects.
- `public void setOwner(Player owner):` Sets the owner of the Land object.

#### 4.3.2.3 Country Class

**Attributes:**

- `private String name:` Stores the name of the country.
- `private ArrayList<Land> lands:` Stores the lands the country has.

**Constructor:**

- `public Country():` Initializes the country object.

**Methods:**

- `public String getName():` Returns the name of the country object.
- `public void setName(String name):` Sets the name of the country.
- `public ArrayList<Land> getLands():` Returns the lands the country has.
- `public void setLands(ArrayList<Land> lands):` Sets the lands of the country to the lands given in the parameter.

#### 4.3.2.4 Dice Class

**Attributes:**

- `private int diceValue:` The current value of the dice. Between 1 and 6.

**Constructor:**

- `public Dice(int value)`: Initializes the dice object.

**Methods:**

- `public int rollDice()`: Randomly determines a new value for the dice.
- `public int getDiceValue()`: Returns the value of the dice.

#### 4.3.2.5 Leader Class

**Attributes:**

- `public String ideology`: Stores the explanation of the leader's ideology and the areas of the game he affects.
- `public String name`: Stores the name of the leader.

**Constructor:**

- `public Leader()`: Initializes the leader objects.

**Methods:**

- `public String getIdeology()`: Returns the ideology explanation of the leader.
- `public String getName()`: Returns the name of the leader object.
- `public void setName(String name)`: Sets the name of the leader object.

#### 4.3.2.6 Player Class

**Attributes:**

- `private ArrayList troops<Troop>`: Stores a linked list of troops that belong to the player.
- `private int resources`: Stores the new troops that will be given to the player at each turn.
- `private ArrayList lands<Land>`: Stores a linked list of the lands that belong to the player.
- `private ArrayList techs<Tech>`: Stores a linked list of the techs that the player has invested in so far.
- `private ArrayList generals<String>`: Stores a linked list of the generals that the player has.

- private ArrayList cards<Card>: Stores a linked list of the board cards that the player has.
- private Leader leader: Stores the leader of the player's country.

**Constructor:**

- public Player(): Default constructor for the player class.

**Methods:**

- public void addTroops(Troop troop): Adds a troop to the player's hand.
- public void removeTroops(Troop troop): Removes a troop from the player's hand.
- public void addTech(Research tech): Adds a research bonus from the tech tree.
- public String getLeader(): Returns the leader of the player's country.
- public void setLeader(String leader): Sets the leader of the player's country.
- public void addGeneral(String general): Adds a general to the player's hand.
- public ArrayList<Card> getCards(): Returns the cards in the player's hand.
- public void addCard(Card card): Adds a card to the player's hand.
- public void remove Card(Card card): Removes a card from the player's hand.

#### 4.3.2.7 BonusEffects Class

**Attributes:**

- public float attackMult: Stores the attack multiplier coming from the research tree and/or leader/general.
- public float defenseMult: Stores the defence multiplier coming from the research tree and/or leader/general.
- public float researchSpeedMult: Stores the change value of the research speed.

#### 4.3.2.8 Research Class

**Attributes:**

- public String name: Stores the name of the research option.
- public int cost: Stores the cost associated with that research.

**Constructor:**

- public Research(): Initializes the research object

**Methods:**

- public String getName(): Returns the name of the research option.
- public int getCost(): Returns the cost of the research option.
- public int setCost(): Sets the cost of the research option.
- public void completeResearch(): When a research completed, bonus effects enabled and research tree updated
- public void cancelResearch(): If the player cancels the research or gets in a situation where research has to be cancelled, this cancels the research and resets the research tree
- public BonusEffects getEffects(): Returns the effect of research.

#### 4.3.2.9 Menu Class

**Attributes:**

- public Frame howToPlayFrame: The frame that will show the tutorial of the game with images.
- public Credits creditsFrame: The frame that will show the names and pictures of the developers of the game.
- public JButton howToPlayButton: The button that will open how to play
- public JButton creditsButton: The button that will open credits
- public JButton newGameButton: The button that will initiate a new game.
- public JButton loadGameButton: Opens a prompt that will allow the user to load a saved game.

**Constructor:**

- public Menu(): initializes the menu

**Methods:**

- public void selectCredits(): After clicking the button this method will be called
- public void selectHowToPlay(): After clicking the button this method will be called
- public void selectNewGame(): After clicking the button this method will be called
- public void selectLoadGame(): After clicking the button this method will be called
- public void loadSave(): In order to call the save file this method will be called

#### 4.3.2.10 GameController Class

**Attributes:**

- private Menu currentMenu: Stores the current menu that is shown on the screen.
- private int turnCount: Stores how many turns have passed since the start
- private PlayerController playerControllers: Stores the player controllers for each player.
- private BoardController boardController: The board controller object.
- private Menu menu:
- private ArrayList<Save> files: Stores an array list of save files for each player.
- private boolean isGameOver: Stores whether or not the game has ended.
- private int playerCount: Stores the number of players currently playing the game.

**Constructor:**

- public GameController(int playerCount): Initializes the game controller object.

**Methods:**

- public void startGame(): Starts the game.
- public void endGame(): Ends the game.
- public void endTurn(): Ends the turn.
- public void openSettings(): Shows the settings frame on the screen. \* CANCELLED \*
- public void getCurrentMenu(): Returns the current menu object.
- public void setCurrentMenu(Menu currentMenu): Sets the current menu object.

- `public int getTurnCount():` Returns the turn count of the game.
- `public void incrementTurnCount():` Increases the current turn count of the game.
- `public int getVolume():` Returns the current volume value.
- `public void setVolume(int volume):` Sets the volume value for the music.
- `public boolean initGame(ArrayList<Save> files):` Initializes the game from a save file.
- `public void openSound():` Opens the sound settings options.

#### 4.3.2.11 PlayerController Class:

##### **Attributes**

- `private Player player:` The Player model.
- `private PlayerView playerView:` The view object of the player model.

##### **Constructor**

- `public PlayerController():` Initializes the player controller object.

##### **Methods**

- `public void updateObjects():` Updates the playerView.

#### 4.3.2.12 ExchangeCardsController Class:

##### **Attributes:**

- `public int exchangeCount:` The number of card exchanges made through the game.

##### **Constructors:**

- `public ExchangeCardsController():` Initializes the Exchange cards controller object

##### **Methods:**

- `public int getExchangeCount():` Gets the number of card exchanges made through the game.
- `public void setExchangeCount(int exchangeCount):` Sets the number of card exchanges made through the game.
- `public void giveTroop():` Gives the player a set amount of troops for the cards exchanged.

#### 4.3.2.13 Credits Class

**Attributes:**

- public String credits: Stores the name of the developer's name.

**Constructors:**

- public Credits(): Initializes the credits object.

#### 4.3.2.14 Startmenu Class

**Constructors:**

- public Startmenu(): Initializes the startmenu object.

#### 4.3.2.15 Card Class

**Attributes:**

- private String landName: Stores the card name.
- private String troopType: Stores the troop type that the card affects.

**Constructor:**

- public Card(): Initializes the card object.

**Methods:**

- public String getLandName(): Returns the land name of the card.
- public String getTroopType(): Returns troop name of the card.

#### 4.3.2.16 Artillery Class

**Attributes:**

- public float defence: Stores the defence value of the artillery class.
- public float damage: Stores the attack value of the artillery class.
- public int resourceValue: Stores the cost of the artillery class.

**Constructors:**

- public Artillery(): Initializes the artillery object.

#### 4.3.2.17 Tank Class

**Attributes:**

- public float defence: Stores the defence value of the tank class.

- public float damage: Stores the attack value of the tank class.
- public int resourceValue: Stores the cost of the tank class.

**Constructors:**

- public Tank(): Initializes the tank object.

#### 4.3.2.18 Infantry Class

**Attributes:**

- public float defence: Stores the defence value of the infantry class.
- public float damage: Stores the attack value of the infantry class.
- public int resourceValue: Stores the cost of the infantry class.

**Constructors:**

- public Infantry(): Initializes the infantry object.

#### 4.3.2.19 Nerds Class

**Attributes:**

- public float defence: Stores the defence value of the nerds class.
- public float damage: Stores the attack value of the nerds class.
- public int resourceValue: Stores the cost of the nerds class.

**Constructors:**

- public Nerds(): Initializes the nerds object.

#### 4.3.2.20 Save Class

**Attributes:**

- public Board savedMap: Stores the map object that will be saved into the file.
- public Player[] savedPlayerProgress: Stores the players that will be saved into the file.

**Constructors:**

- public Save(): Initializes the save object.

**Methods:**

- public Save getSavedContent(): Returns the Save object



- public void save(): Writes the save file onto a document.

#### 4.3.2.21 SoundController Class

##### **Attributes:**

- public int volume: Stores the volume value.

##### **Constructors:**

- public SoundController(): Initializes the sound controller object.

##### **Methods:**

- public void setVolume(int volume): sets the volume to the given parameter
- public void getVolume(): Returns the int volume value.

#### 4.3.2.22 ResearchTree Class

##### **Attributes:**

- public Research[] researches: Stores all of the researches for a player.

##### **Constructors:**

- public ResearchTree(): Initializes the research tree object.

#### 4.3.2.23 BoardController Class

##### **Attributes:**

- public BoardView boardView: Have the game board object.

##### **Constructors:**

- public BoardController(): Initializes the board controller object.

##### **Methods:**

- public void updateBoard(): After each change, updates the boardView object.

#### 4.3.2.24 BoardView Class

##### **Constructors:**

- public BoardView(): Initializes the boardView object.

**Methods:**

- public displayBoard(): Displays the board as a frame.

**4.3.2.25 InputController Class****Constructors:**

- public InputController(): initializes the inputController object

**4.3.2.26 Effects Interface****Attributes:**

- public float attackMult: Stores the attack multiplier value
- public float defenseMult: Stores the defense multiplier value
- public float researchSpeedMult: Stores the research speed multiplier value
- public int researchAdder: Stores the research add value