Risk Game

Build 1 – Design Document

|  |  |
| --- | --- |
| **Student Name** | **Student ID** |
| Kourosh Aziz-Nejad | 40014215 |
| Sandeep Swainch | 40036994 |
| Ranjitha Shetty | 40038087 |
| Mitra Azari-Sissi | 40016446 |

Table of Contents

[1. Introduction 3](#_Toc496224977)

[2. Architecture Design 4](#_Toc496224978)

[3. Use Case Diagram 7](#_Toc496224979)

[4. Software Versioning Repository 8](#_Toc496224980)

[5. API Documentation 9](#_Toc496224981)

[6. Unit test 9](#_Toc496224982)

[7. Coding Standards 12](#_Toc496224983)

# 1. Introduction

The Risk Game game follows the mvc and singleton pattern in parts. The code is organized in five packages.

1. com.risk.controller :

This package contains Listeners

1. com.risk.lib

This package contains Jar files

1. com.risk.model

This package contains all Object (models) of game.

1. com.risk.test

This Package contains all Unit tests and TestSuite

1. com.risk.utility

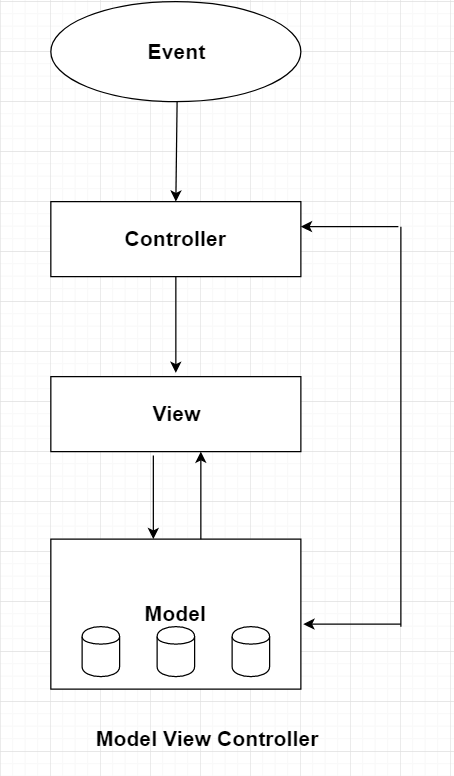
This package contains all the utilities used in the game

1. com.risk.view

This package contrains UI Components (Buttons, Panels etc)

# 2. Architecture Design

To separate the logic of our game in terms of persistence, view and the logical layer, for performing actions such as user clicking a button to start a game, deciding on number of players, allocating armies per countries and listening to various user inputs, changing the state of the game– MVC architecture is the best fit. Model plays the crucial role of maintaining the state of the application and fetching the latest changes via controller to the model. In addition to it, user is not aware of the complexity of the game and the view abstracts the core logic of game. View updates the model via the controller.



MVC is popular as it isolates the application logic from the user interface layer and supports separation of concerns. Here the Controller receives all requests for the application and then works with the Model to prepare any data needed by the View. The View then uses the data prepared by the Controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.

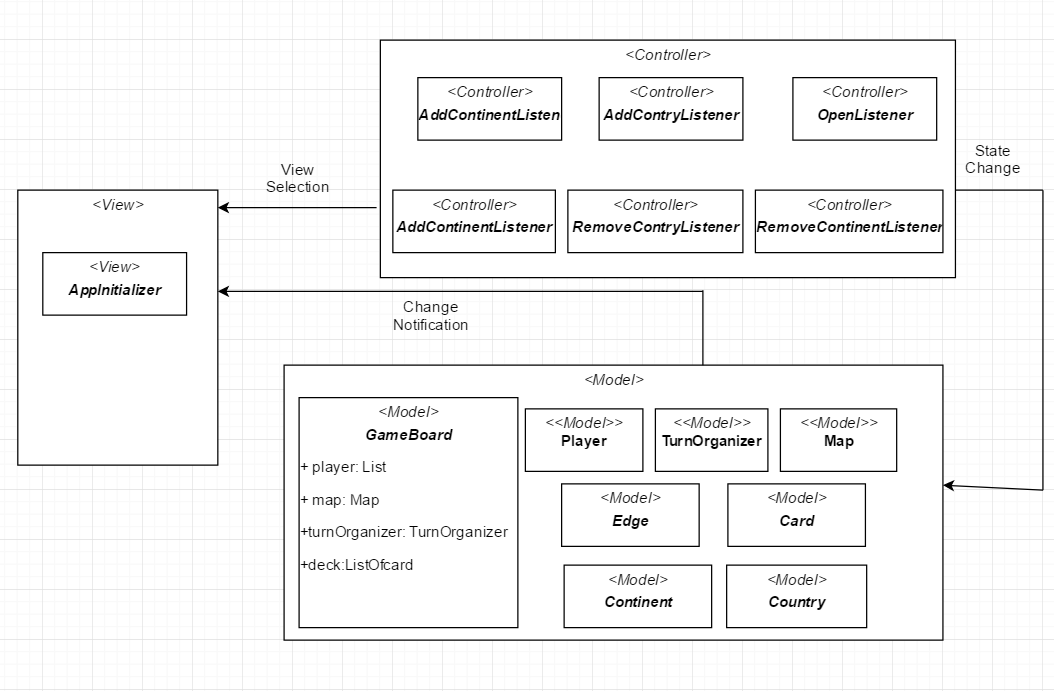


Fig: Model View Controller Architecture for Risk Game

**The View:**

View is responsible for displaying all or a portion of the data to the user. In the Risk game, user loads a map interacting with a with the application From View layer. **Console** Class loads the **applicationwindow** which renders GUI of the application. The result is prepared by the view and returns the result to the user.

**The controller:**

The controller is responsible for responding to user input and perform interactions on the data model objects. The controller receives the input, it validates the input and then performs the business operation that modifies the state of the data model.In Risk Game **Listener** changes the state of the application when user perform some action with the help of a controller.

**The Model:**

The model is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

**Gameboard Class** keeps map, players deck and TurnOrganizer. It distributes request from Controller and view. It performs various actions such as it Load and save Map, Starting the game, setup the Players, Decides Player turn in Round Robin Fission, Assigns Country Randomly, Reinforcement action, fortification actions.

**Player Class** Represents Player’s attributes.

**Map Class** Adds and Remove Continent, Country and Connectivity, Create Map.

**FactoryLand Class** constructs objects of continents and Countries.

**TurnOrganiser Class** Organize the State of the game such as Started Game flag, Current Player Id, Current phase of the game.

**Edge**: Contains connectivity between countries.

**Land**: Parent class for continent and country classes

**ILand**: Interface for Land

**Utilities:**

Map Parser Class: Parses the File (.map file) to the application and write map to a file. It validates the file.

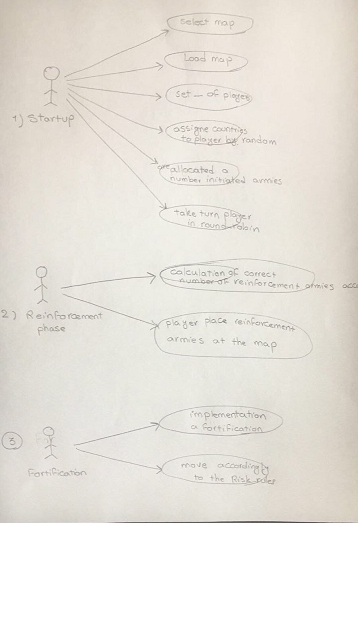
**Ecards Class** is a Enum Type which contains Type of cards.

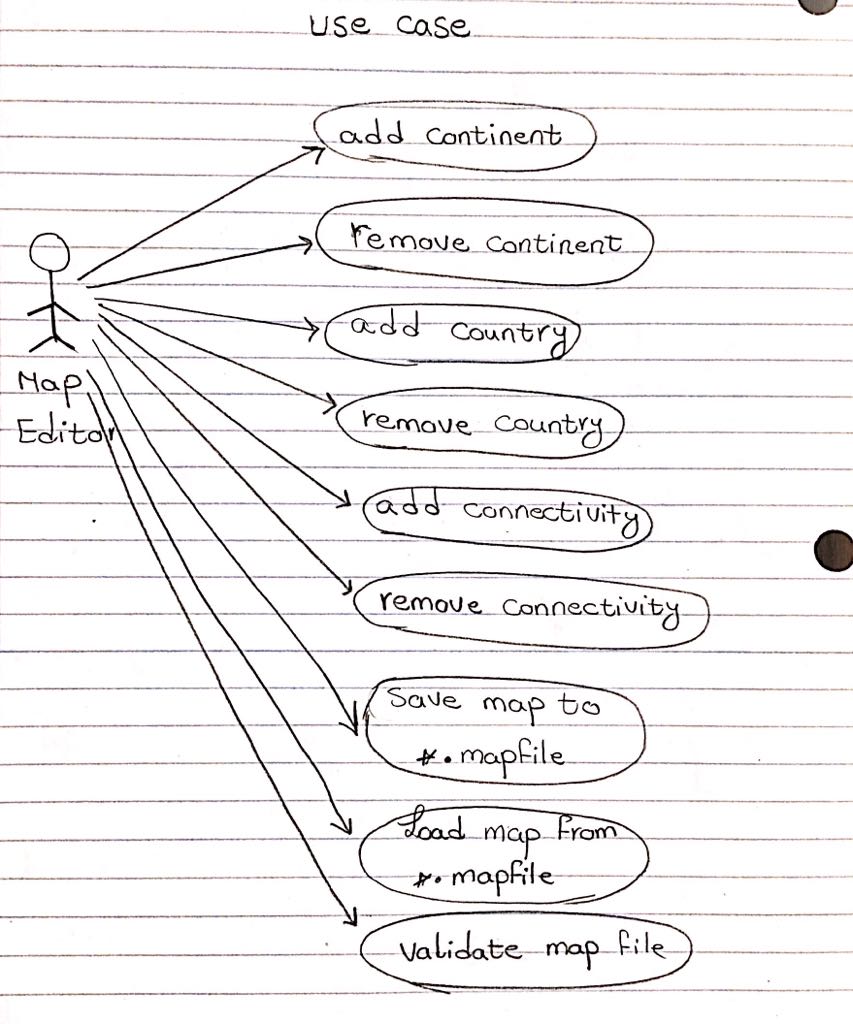
**staticApplicationVariable** maintains global variables

**TurnPhases** is Enum Type which keeps the phases.

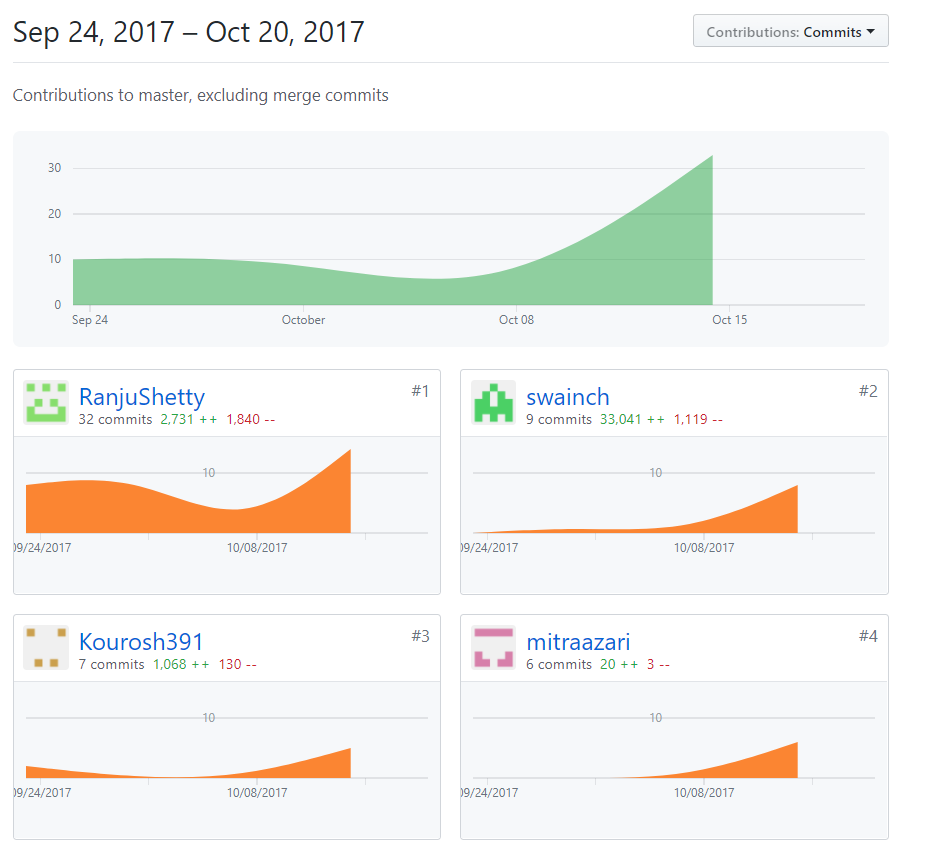
**output.txt :**

# 3. Use Case Diagram





# 4. Software Versioning Repository



# 5. API Documentation

API documentation is created in Eclipse -> Project -> generate JavaDoc

We Create javadoc for files, Classes and Methods

# 6. Unit test

In Risk Game we are following standard Unit testing framework(Junit4) to test the important part of the program.

@before annotation is used to run the method before each test method

@After annotation is used to run the method after each test method

@Test method attached to it run as test case

# 7. Coding Standards

Coding Conventions are a set of rules that use how to code are written. They are important to

programmers because code conventions improve the readability, understandability new code

quickly. Having the code conventions make develop and maintain the software. It improve

internal quality of the software and maximize the productivity.

They are including:

**File Organization**: All classes are declared as part of a package

For example:

Controllers are organized under the package:

com.risk.controller

**Comments**:

Single line comment:

/\*….. \*/

MultiLineComments:

/\*

\* Here is a block comment

\*/

Documentation Comment

/*\*\*  
 \* This class represents the map  
 \* it maintains the countries and continents   
 \* as a collection of lands  
 \* it has different methods to change the state  
 \*/*

**Class Declaration:**

Java Classes are named as \*.java. Java class contains single public class or interface.

**Variable Declaration:**

**Class variable:** All class variables are declared as below:

Ex: public class GameBoard {  
}

**Local variable:** All local variables are declared as below:

Ex: GameBoard gameboard;

**Parameters:** All parameters are declared as below:

**Ex: public** Continent(String prm\_name, **int** prm\_control)

**Constants:**

Constants are written in Capital Letters

public static int SCREEN\_WIDTH = 0;

**Naming Convention:**

They can also give information about the function of the identifier:

|  |  |
| --- | --- |
| **Identifier Type** | **Rules for Naming** |
| Filename | First letter lower case character and second letter starts with upper case |
| Packages | Written in lower case charecter |
| Classes | Start each word with upper case character |
| Methods | Start with upper case letters to separate words following Camel Script |
| Attributes | Use a lower-case letter and use underscore between word |
| Variables | Local variables are written entirely lower case without underscore |
| Constants | Use upper case letter with underscore between words |
| Parameters | Prm and name |