
**Advanced Programming Practices**
**SOEN 6441 --- Fall 2018**

## Project: Risk – The World Conquest Game

Bitbucket Repository : https://RiskGameTeam@bitbucket.org/RiskGameTeam/soen6441_riskgame.git

## Architectural Design Description (Project Build 1)

**Team 4:**


| Team Member | Student ID |
|---|---|
| Karan Bhalla | 40047162 |
| Kunal Sharma | 40053616 |
| Yogesh Choudhary | 40071287 |
| Shivam Aggarwal | 40080017 |


**Submitted to**: Dr. Joey Paquet

## Table of Contents

## Introduction

This document provides description of Architectural design for this build, which demands

*Map editor*

- User-driven creation of map elements, such as country, continent, and connectivity between countries.
- Saving a map to a text file exactly as edited (using the "conquest" game map format).
- Loading a map from an existing "conquest" map file, then editing the map, or create a new map from scratch.
- Verification of map correctness upon loading and before saving (at least 3 types of incorrect maps).

*Game Play*

Implementation of a game driver implementing the game phases according to the Risk rules.

1. *Startup phase*

- The game starts by user selection of a user-saved map file.
- The map is loaded as a connected graph, which is rendered effectively to the user to enable efficient play.
- The user chooses the number of players, then all countries are randomly assigned to players.
- Players are allocated a number of initial armies, depending on the number of players. In round-robin fashion, the players place their given armies one by one on their own countries.

2. *Reinforcement phase*

- Calculation of the correct number of reinforcement armies according to the Risk rules.
- Player place all reinforcement armies on the map.

3. *Fortification phase*

- Implementation of a valid fortification moves according to the Risk rules.

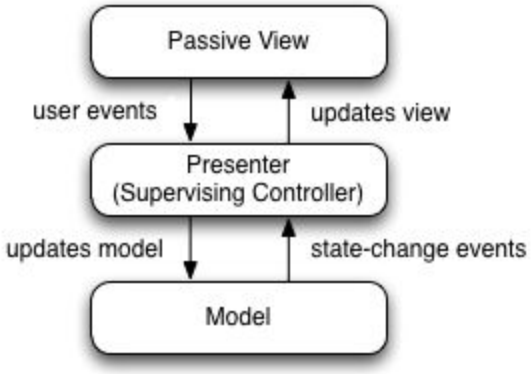
## Architecture design

Our application architecture is a variation of MVC architecture and sometimes also known as MVP which stands for Model-View-Presenter (Supervising Controller).

In traditional MVC

- The model manages the data and business logic of the application.
- The view is responsible for representing information to the user.
- The controller is responsible to capture user's input or actions and asks the model to do processing based on that user action.

And then Model and View interact with each other to update the view for the user.

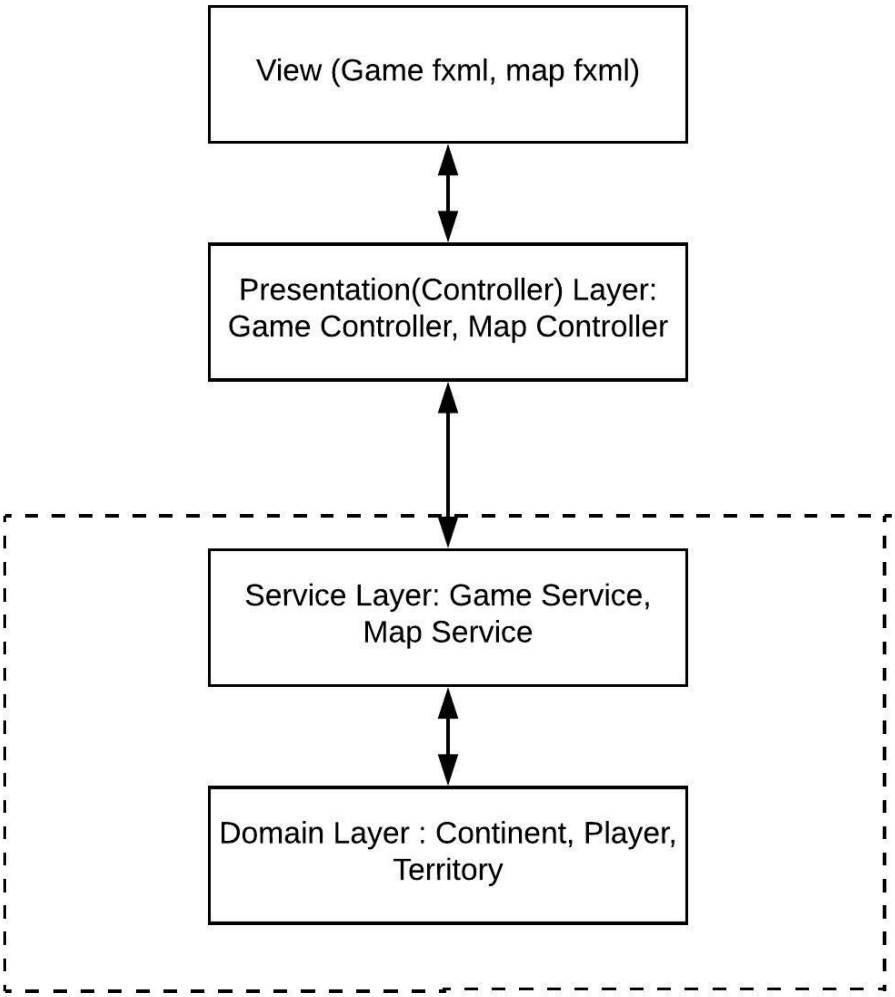



But in MVP we have Presenter which acts as a supervising controller between Model and View. The major difference between the two architecture is that in MVP there is no communication channel between Model and View, rather Presenter (Supervising Controller) is responsible for that. The Presenter users command in form of events generated by the user through View and update the Model and then fetch the data, format it and update the View accordingly.

There are few benefits of using MVP over MVC which are:

- It improves the separation of concern in various layers due to which code is easy to maintain and test.
- As View and Model are independent of each other so they can be developed independently.
- Business logic can be moved to either controller or to another layer to increase the modularity of application.

**Game Flow**




*Game Flow*

## Module description

- **Domain:** This module has Model classes which are plain old java object (POJO) classes which represents domain entities in our game.

    - *Continent*: It contains all the information related to a Continent in the game like the name of the continent, army value of the continent, list of territories it contains. All this information of the continent can be changed for each continent object and can be accessed anywhere in the project.

    - *Territory*: It contains all the information related to a Territory in the game like the name of the territory and the continent it belongs to, list of neighbouring territories of this territory and the name of the owner of this territory during gameplay. All this information of the territory can be changed for each territory object and can be accessed anywhere in the project.

    - *Player*: It contains all the information related to a Player in the game like the name of the player, list of the territories player owns, and the number of armies' player has. All this information changes according to the game flow and can be accessed anywhere in the project.

- **Services:** This is an additional module which has classes to implement various game rules and validation. This package manages all the service calls from Controller package(See below).

    - *MapService:* This class is used to handle all the service calls from MapController class(Package- 'Controller'). This class implements the required business logic. It manages the beginning interactions of the user with the game and contains business logic required for set up of game. It contains methods which deal with, saving of a generated map file, validations for the generated map and providing flexibility to the user to modify the map. The user can also choose a map file, here, the method in the class reads the file and parses it and provides it as an object for further use, this business logic and validation on the chosen file is done in this class.

    - *GameService:* This class handles all the service calls from GameController class(Package- 'Controller'). This class implements the required business logic for initial distribution of armies among players, placing of armies on territories which the player owns and also the required validations, calculation of player who will get the first turn (Using dice roll technique) and the required round-robin turn style, distributes armies for reinforcement phase, and required function for fortification phase. This class also manages to apply validation for each function that class performs so it ensures handling of user inputs accordingly.

- **User Interface:** This module has UI or view component which are used in JavaFX application to create a user interface such as fxml and css files.

    - *Risk:* This class is the main entry point of the application. It also contains a method for providing a user interface. Here, it enables the beginning of application with loading Map.fxml file. A user interacts with this window for generation and saving of map. It performs the set up required and allows a player to start the game.

    - *Map.fxml:* It is the fxml file which is shown on the screen and user to do set up required for the game and allows the user to start the game.

    - *Game.fxml:* It is the main view of risk game application, where a user plays the game.

- **Controller:** This module has Presenter or supervising controller classes which are responsible for capturing user-generated events, performing View related validation, calling required methods to do processing over models and updating View.

    - *MapController:* This class have methods which handle different actions performed by a user on User Interface for creating and updating the map for the game.

    - *GameController:* This class handles different actions performed by a user on Game.fxml file and implements the game driver. It contains methods which ensure proper functioning of game flow and ensuring right order for players. And it includes validation for various wrong inputs given by user making the application more robust.

- **Tests:** This contains some very useful unit test cases that ensured proper analyzing of code and made application robust. Junit 4 is used as the unit testing framework.