# System Architecture

## Abstract

**Model–view–controller** (**MVC**) is a pattern that allows the GUI to be separated from the core application and divides the application into three interconnected parts. This pattern ensures decoupling of the major components resulting in code reuse and concurrent development of the different project components.

**Model-** The model is the central component of the pattern. It expresses the application's behavior in terms of the problem domain, independent of the user interface. It directly manages the data, logic and rules of the application.

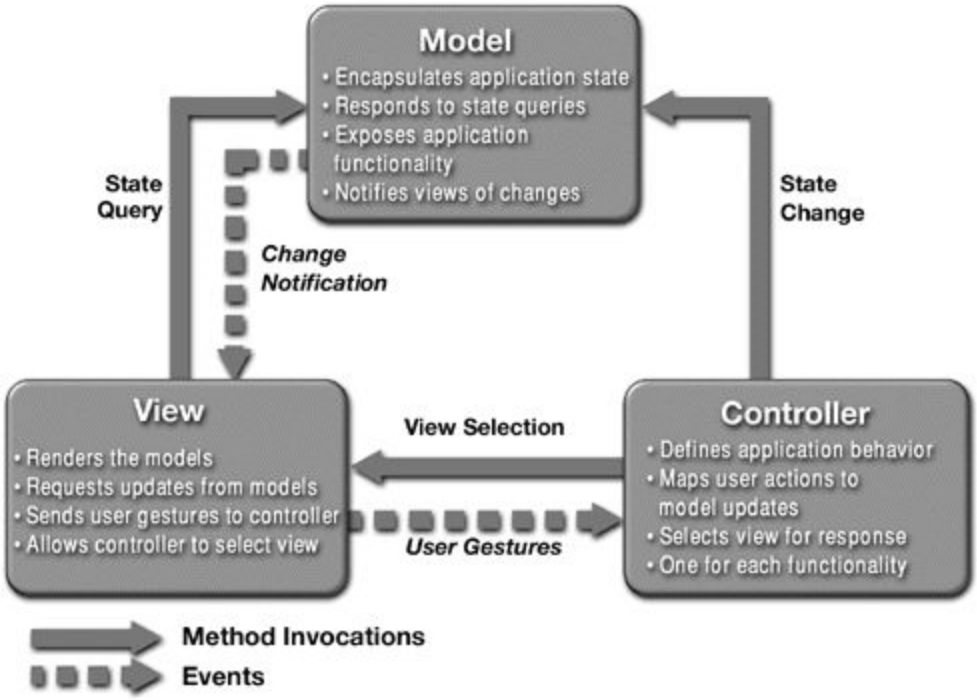
**View-** A view can be any output representation of information

**Controller-** The controller, accepts input and converts it to commands for the model or view.

## System Overview:

Our project “Risk” follows this MVC pattern. In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.

- A model stores data that is retrieved according to commands from the controller and displayed in the view.
- A view generates new output to the user based on changes in the model.
- A controller can send commands to the model to update the model's state . It can also send commands to its associated view to change the view's presentation of the model.



## Model

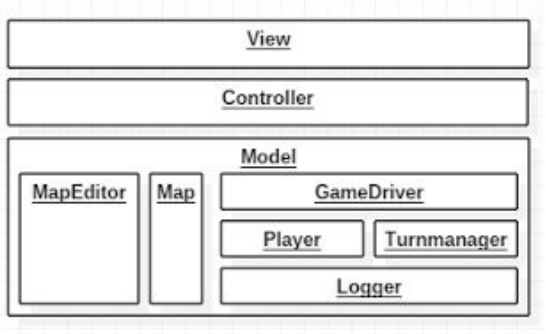
The model implements Map, Player, Cards , Countries and Continents using Map, Player, CountryNode and MapNode files. These objects are created when the call from the controller goes to the model. Model classes update the view object state.

### View

The View implements the graphical user interface through which the user interacts with the game and selects maps, creates player, edits map, creates map, place armies and plays reinforcement and fortification moves. The actions to these interactions implemented in Controller of the game. Since we are using push notification, model updates the view, whenever state of model is changed.

### Controller[1]

The controller is responsible for carrying out the actionlisteners to the view elements like selecting to playmove or edit maps on startup of the game, placing armies and creating necessary changes in the model. Controller implements the action listeners, so whenever there is any action performed by the user, controller notify responsible methods for the action. In some cases controller also take responsibility of notifying the view about any change in the model.




Architecture of Risk

The Observer pattern is implemented in the game through MapView. The Map model acts as the Observable and MapView is the observer. Whenever there is the change in the model the view gets updated accordingly.
Risk game is divided into different modules.
**GameDriver:** Controls the main flow in game. Game Model uses player, turnmanager and map class.
**Player Module:** It represents the players. Player module interacts with logger, Map and turnmanager module.
**TurnManager:** It handles the turn related functions or when to change a turn phase.
**Logger:** Logger module logs every step in game. Other modules use this module during game to record moves and game states at different intsances.

**MapEditor:** It represents the map editor and use map module.
**Map:** map module contains classes that read write or store map information.
**View:** View contains all the GUI classes.
**Controller:** Controller act as intermediate between model and view as described in [1].