Architectural Design Document

**Introduction**

Our aim is to build a simple version of a Risk Strategy game. We have implemented a Model View Controller (MVC) architectural design model. We have used extreme programming approach for the smooth development of software by implementing features like pair-programming, simple design, etc.

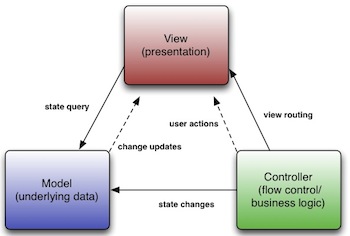
**Model View Controller**

Model View Controller design aims for separation of Concerns, meaning the components should not do more than one thing by dividing it into three parts a model, a view, a controller.

**Model** – It is the lowest level of the pattern which is responsible for maintaining data. In enterprise software, a model often serves as a software approximation of a real-world process.

**View** - It is responsible for displaying all or portion of the data to the user. If the model data changes, the view must update its presentation as needed. This can be achieved by using a push model, in which the view registers itself with the model for change notifications, or a pull model, in which the view is responsible for calling the model when it needs to retrieve the most current data.

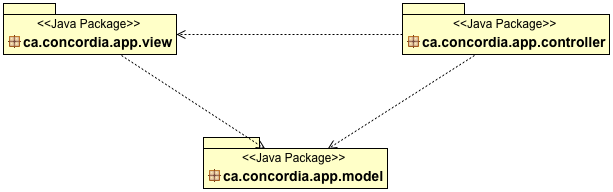
**Controller** – It is a software code that controls the interaction between the model and the view. The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone application, user interactions could be button clicks or mouse over events. A controller may also change the view as and when the action wants.



**Fig1. Basic MVC architecture**

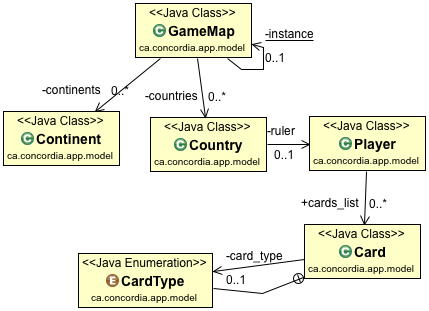
As the figure 2 goes, this is how MVC has been implemented in the project.

* Different Models (Card, Continent, Country, etc.) manages the data of the application domain. If the model gets a query for change state from views ( MapEditorView, NewGameView, etc.) they respond to the instruction via controllers (NewGamePhaseController, NewGameSelectorController).
* **Views** on the other hand renders the model into a form suitable for visualization or interaction, in a form of UI (user interface). If the model data changes, the view must update its presentation as needed.
* **Controllers** are designed to handle user input and initiate a response based on the event by making calls on appropriate model objects. Thus accept various input from the user and instruct the model to perform operations.
* The controller translates the user's interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view.
* Controller is also responsible for invoking new views upon conditions.

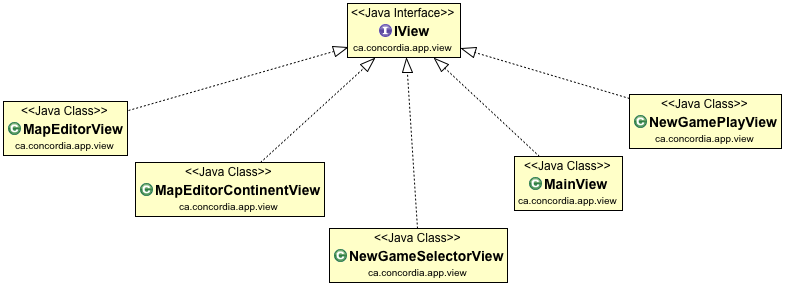


**Fig. 2 Class Diagram for overall MVC structure**

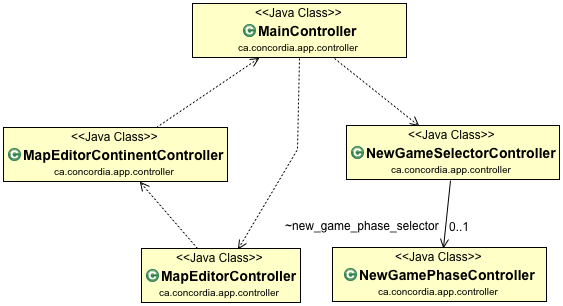
Following the layout of figure 2 the figure3, 4 and 5 describe the actual class diagrams inside each and every packages. Viz. Model Package in figure 3, View Package in figure 4, Controller Package in figure 5.



**Fig.3 Model Class Diagram**

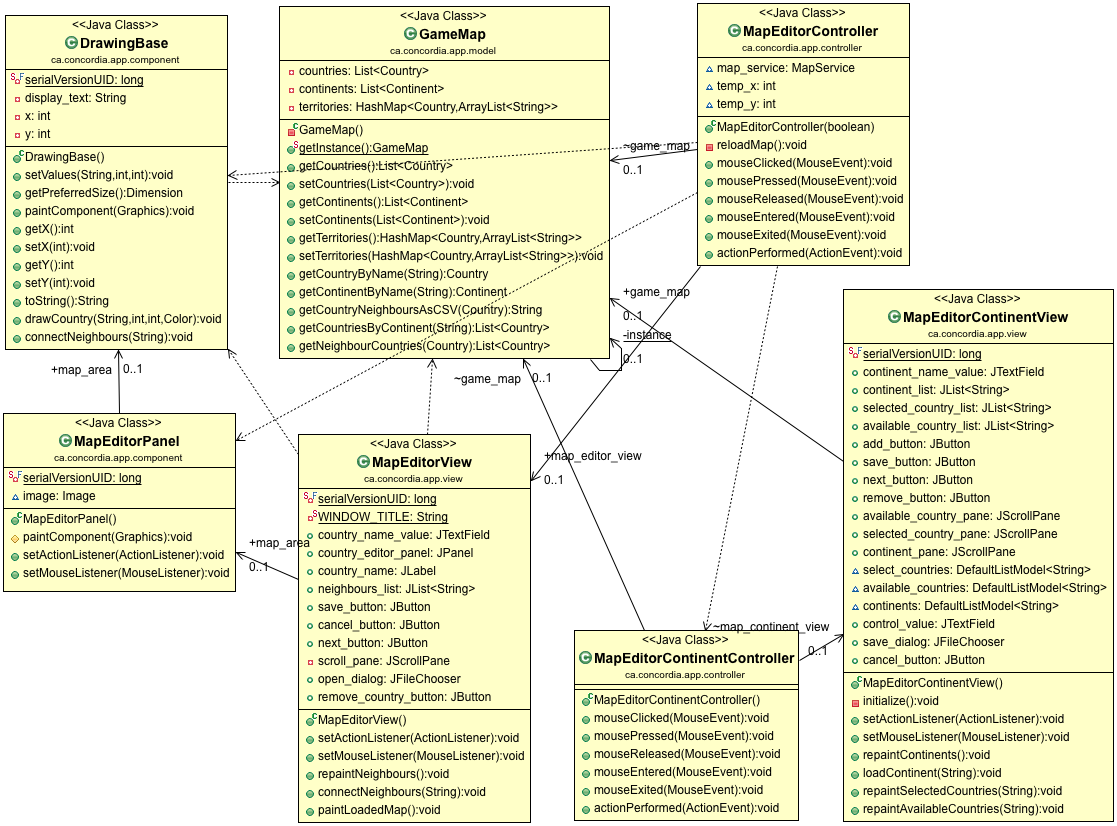


**Fig.4 View Class Diagram**



**Fig.5 Controller Class Diagram**

Figure 6 shows one particular class diagram (of many such possible class linkage possible) where in it is shown how MVC structure is implemented in the actual sense. With each and every package of model controller and view linking with each other in event-driven system.

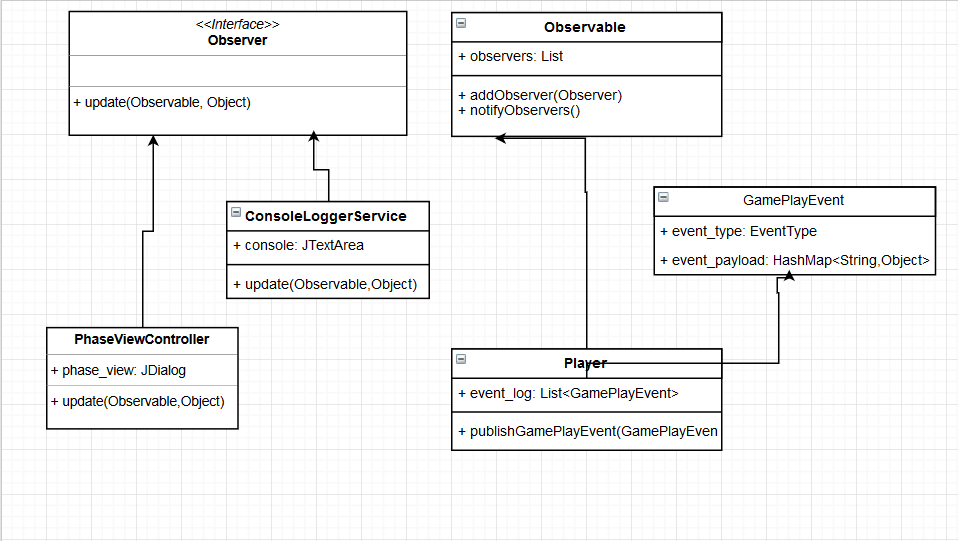


**Fig.6 Map architecture class diagram**

**OBSERVER PATTERN:**

Observer pattern is used when there is a one-to-many relationship between objects i.e. If one object is modified, its dependent objects are notified automatically. This pattern falls under Behavioral Pattern. It is mainly used to implement distributed event handling systems, in event-driven software. This pattern is a Key part in the model-view-controller(MVC) architecture.

We have two important parts in the observer pattern, The Observers and the Observables. The Observables are the ones that change and their job is to notify all the registered observers. The Observer are the ones that get notified and update themselves.

****

**STRATEGY PATTERN**

**Motivation**:

Sometimes we may want to change the behaviour of an object depending on some conditions that are only to be determined at runtime, or to easily add new definitions of certain behaviour without altering the class that is using it.

**Intent**:

Define a group of algorithms that applies to a family of classes. Encapsulate each algorithm separately. Make the algorithms interchangeable within that family

**Consequence**:

The Strategy Pattern lets the specific algorithm implemented by a method vary without affecting the clients that use it.

**Elements of the strategy pattern:**

Context Class: class that uses a certain behaviour that is to be changed during execution. It contains a Strategy object and provides a setStrategy() method to change its own strategy. The strategy is to be called through a executeStrategy() method that will delegate to a concrete strategy method.

Strategy Abstract Class: Superclass of all strategies containing the abstract executeStrategy() method to be implemented by all its subclasses.

Concrete Strategies: Subclasses of Strategy that provide a different implementation for the executeStrategy() method. In our project concrete strategy classes are AggressiveStrategy, BenevolentStrategy, CheaterStrategy, RandomStrategy, HumanStrategy.

**References**

* <http://www.oracle.com/technetwork/articles/javase/index-142890.html>
* <http://www.java-forums.org/attachments/ocmjea/3449d1333636384t-tutorial-review-web-tier-application-architecture-java-architect-exam-c5-conceptualmvc.jpg>
* <https://users.encs.concordia.ca/~paquet/wiki/index.php?title=SOEN6441_-_fall_2017>