# Architectural Design

We used MVC architectural design for developing the project. First we'll see a brief description of MVC.

MVC stands for Model, View and Controller. MVC separates application into three components - Model, View and Controller.

**Model**: Model represents shape of the data and business logic. It maintains the data of the application. Model objects retrieve and store model state in a database. It is responsible for managing the program's data (both private and client data). Model provides an internal interface (API) to enable other parts of the program to interact with it. It is responsible for maintaining the integrity of the program's data, because if that gets corrupted then it's game over for everyone

**View**: View is a user interface. View display data using model to the user and also enables them to modify the data.

**Controller**: Controller dandles the user request. Typically, user interact with View, user requests will be dandled by a controller. The controller renders the appropriate view with the model data as a response.

The View/Controller is responsible for providing the outside world with the means to interact with the program's client data. View/Controller provides an external interface (GUI/CLI/web form/did- level IPC/etc.) to enable everything outwits the program to communicate with it. The View/Controller is responsible for maintaining the integrity of the UI, making sure all text views are displaying up-to-date values, disabling menu items that don't apply to the current focus, etc. In our Project View/Controller are the same .

Now, here's the test of a true MVC design: the program should in essence be fully functional even without a View/Controller attached.

Why is this possible? The simple answer is that thanks to the low coupling between the Model and View/Controller layers. What's key to the whole MVC pattern is the *direction* in which those connection goes: ALL instructions flow *from* the View/Controller *to* the Model. The Model NEVEF tells the View/Controller what to do. Why? Because MVC is about creating a clear separation of concerns., while the View/Controller is permitted to know a little about the Model (specifically, the Model's API), but the Model is not allowed to know anything about the View/Controller.

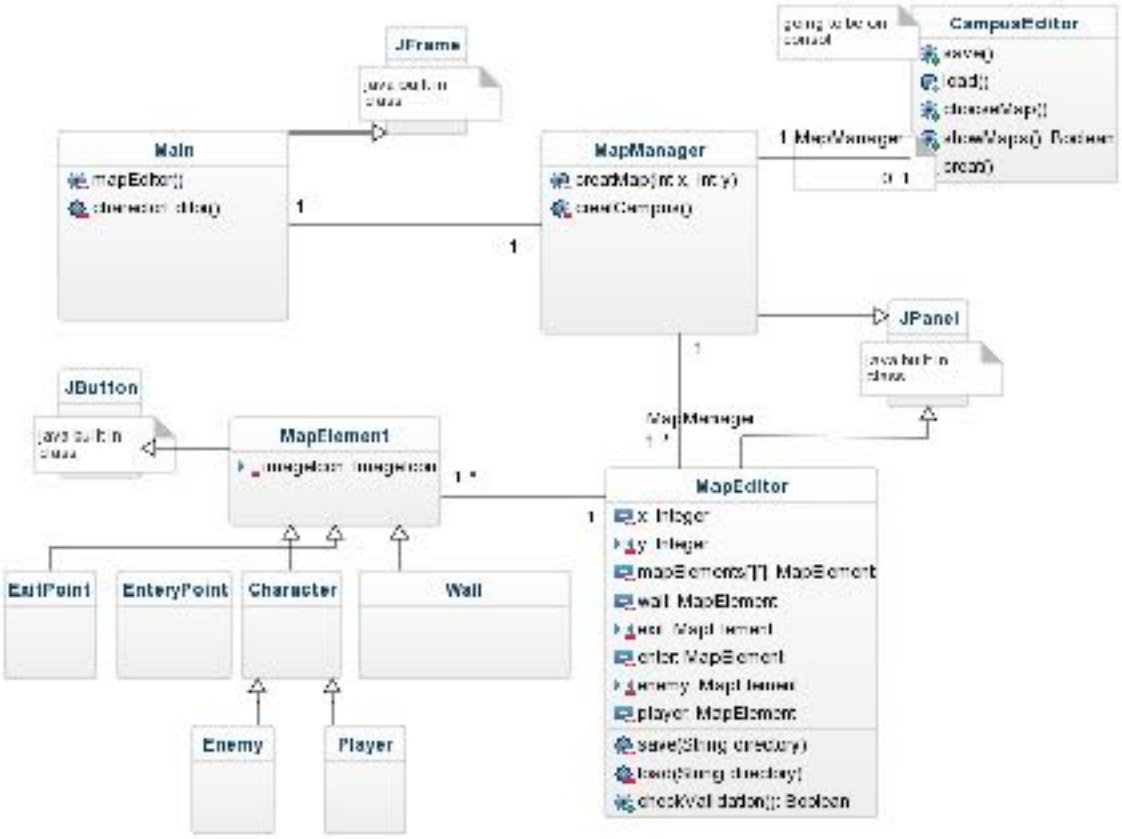
Why MVC is cares about separating concerns ? To help prevent program complexity getting out of control and burying the developer under it. The bigger the program, the greater the number of components in that program. And tde more connections exist between those components, the harder it is for developers to maintain/extend/replace individual components, or even just follow dow the whole system works.

Now the question is dow can the Model inform the View/Controller of changes in the Model's user data when the Model isn't even allowed to connect to the View/Controller?

How do you keep the UI's display synchronized with the Model's state in an MVC-based system?

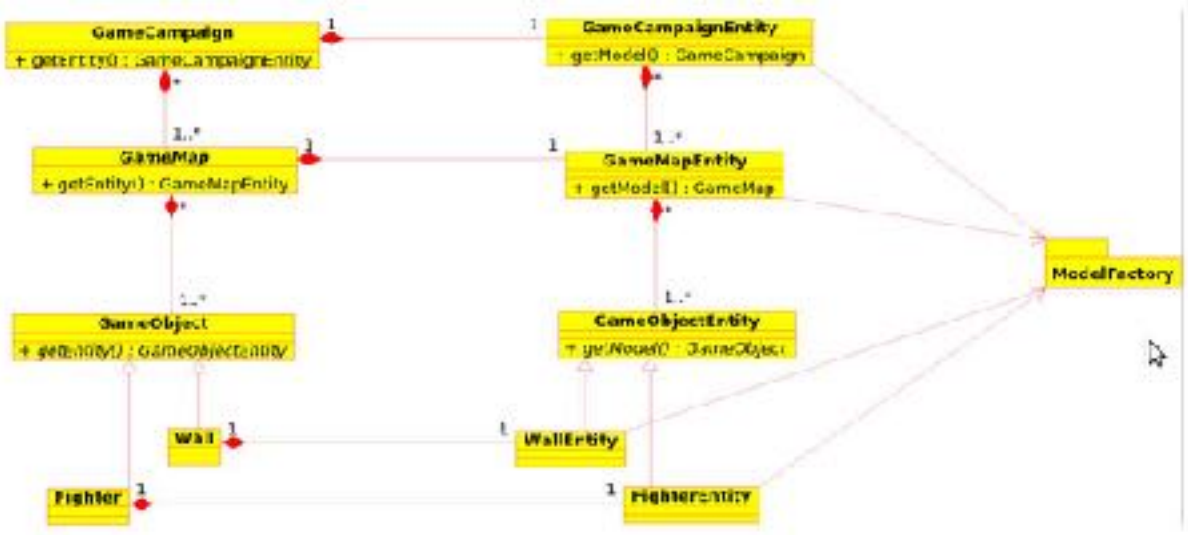
The answer is to set up a notifications system, providing the Model layer with a place it can announce to no-one in particular that it das just done something interesting. Other layers can then post listeners with that notification system to listen for those announcements that they're actually interested in. The Model layer doesn't need to know anything about who's listening (or even if anyone is listening at all!); it just posts an announcement and then forgets about it. And if anyone dears that announcement and feels like doing something afterwards - like asking the Model for some new data so it can update its on-screen display, it is done. The Model just lists what notifications it sends as part of its API definition; and what anyone else does with that knowledge is up to them.

Our class diagram design at the start of project is simply shown in below.




in this design the bottom left part is our model and other classes are our view/controller classes . of corse the above design had some problems but later on we add other models like (character, item, ability, etc..) and other view/controller(itemBuilder, characterEditor ,etc..) to our first design and fix some problems about inheriting and use couple of interfaces to complete the design with getting it to complicated.we also reorganized our packages and rename them at the end to make it more understandable.

and here is the class diagram of how an approach that we used to saved our classes into file .




illustrating , we are creating modelEntity for each of our model . every modelEntity contains only the data of our model and the business of every model handled in their main model class. basically, we separate data from functionality in our model class to be able to save them more efficiently.