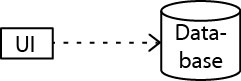
**Model-View-Controller**

Model-view-controller is an architectural pattern for the global organization of the code to develop a system. Originally it was used for developing desktop GUI systems, and still is. Now it is also used in web applications: Ruby, Php, ASP.NET, AngularJS, Ember, Backbone, and many others.

**Context**

1. The purpose of many computer systems is to allow users to view and manipulate (add, edit, delete) data stored in a database store and display it for the user. One approach to coding such a system is to put all the code in a user interface class.



1. A closer examination of this scenario reveals different types of logic embedded in the user interface:
2. Presentation Logic (PL) – code to display data
3. Business Logic (BL) – code to encode, validate, interpret data
4. System Logic (SL) – code to coordinate activities, retrieve and save data

In this “coupling model”, these three things are tied up together!

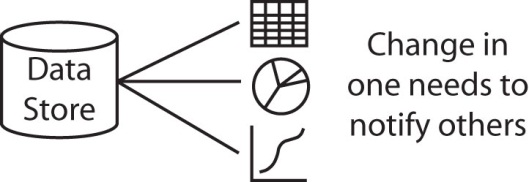
1. Problems:
2. PL tends to change more frequently than BL.
3. BL tends to be more substantial than PL.
4. BL & SL can’t be shared among other UI’s (in the same system or others).
5. When PL, BL, and SL are tightly coupled the hard to test.

**Problem**

How do you modularize the user interface functionality of an application?

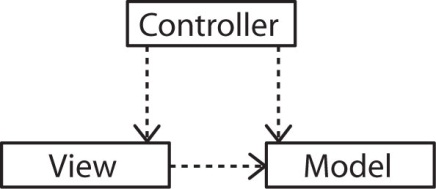
**Forces**

1. , but they both change! If BL is embedded in PL then:
2. More likely to introduce errors in BL as we change PL
3. Must test all BL even after minimal change in PL



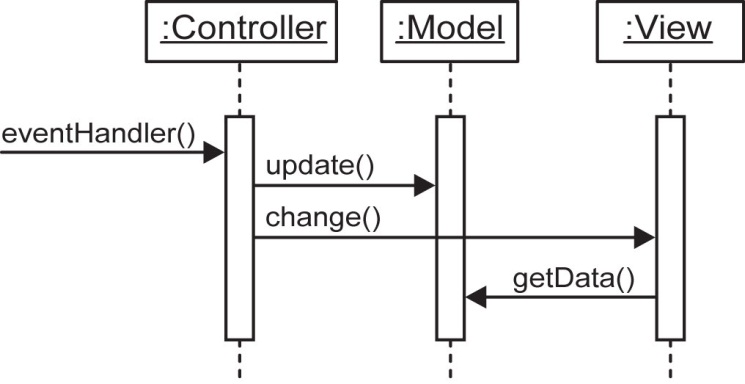
1. Usually different skills are involved to develop a visually appealing and efficient UI as compared to the BL. It is useful to separate the development effort.
2. PL code is more device-dependent than BL. If you want to migrate from an application to a web app, or mobile app, the PL changes, but not the BL.
3. Automated testing for the PL is usually more difficult and time consuming than for BL. Thus, reducing the amount of code directly tied to the PL makes testing the PL easier.

**Solution**

1. The Model-View-Controller (MVC) is an architectural pattern that separates the modeling of the domain, the presentation, and the user actions into separate components.
2. Model – The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the View), and responds to instructions to change state (usually from the Controller).
3. View – The view manages the display of information.
4. Controller - The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.
5. Dependencies
6. Controller depends on View and Model
7. View depends on Model
8. Model does not depend on either
9. Benefits of this separation:
10. Model can be built and tested independently of the other components
11. Model can be reused
12. Alternate Views (or Controllers) can be integrated
13. Note, the separation of View and Controller is often blurred with common implementations. For instance, the use of anonymous inner classes for event handlers in Java.

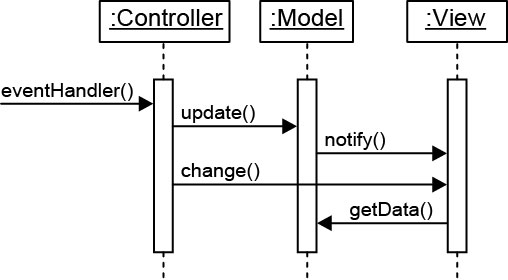
**Variations**

1. The Passive model



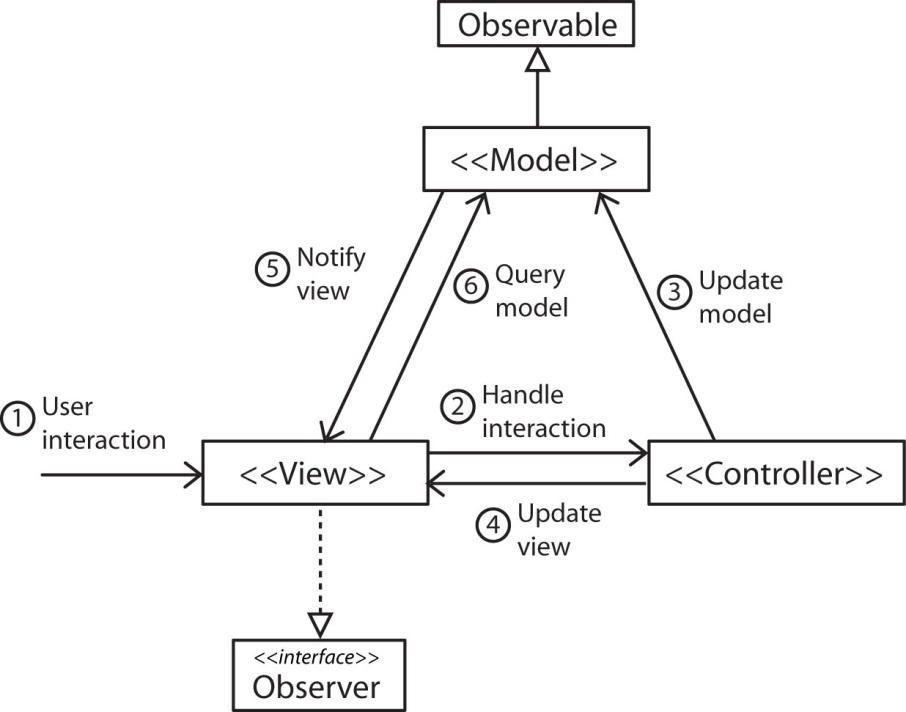
The Model cannot detect and report changes. It is completely independent of the Controller and View.

1. The Active model



In some situations, for instance if the model is changed by multiple sources, it may necessary for the Model to notify the View (or Controller) that it has changed. However, this reintroduces dependency.

1. One way to implement the Active model is to use the Observer pattern where we make the Model observable and the View (and/or Controller) an observer.



**Testing**

1. If the model, view, and controller behaviors are tightly coupled, *e.g.* one classthen it is hard to setup tests even for just basic operations. Also, it is hard to find where the errors are.

1. Using MVC and separation of concerns leads to simpler and more efficient testing. Why:
2. The Model is tested separately.
3. Then, the View can be tested against the Model which results in fewer tests because the Model is already tested.
4. Finally, test the Controller with the View and Model with the same benefits.

**Benefits**

1. Testing is easier and more reliable.
2. Supports multiple Views.
3. Accommodates change in Views and Model.

**Liabilities**

1. Complexity – The MVC pattern introduces more levels of indirection and thus increases the complexity of the solution somewhat.
2. Cost of frequent updates – If the Model undergoes frequent change it can flood the View with update requests. The View can fall behind. One solution is to batch update notifications.