**The Model-View-Controller Pattern**

**The Problem**

Presentation logic usually refers to the user interface of an application. Business logic refers to the procedures and data of application domain. For example, in the banking domain we would expect to find data representing accounts, transactions, and account holders as well as implementations of procedures for transferring funds between accounts.

Several applications might use these same data and procedures: on-line banking for customers, apps for tellers, apps for financial advisors, and ATM machines. All of these applications would have different user interfaces. Also, as a rule-of-thumb business logic is more stable than presentation logic. User interfaces are always changing, but the logic of banking hasn't changed much in the last 100 years!

If business logic depends on presentation logic, if, for example, the declaration of an account class included the way it was displayed in an on-line banking window, then it would be difficult to reuse this class in a teller application. Also, any changes to the way the way the account should be displayed would require changes to the account class itself. In other words, the account class isn't reusable in other applications or with different user interfaces.

Here's a simple example:

double square(double x) {  
  double result = x \* x;  
  println("the square of " + x + " = " + result);  
  return result;  
}

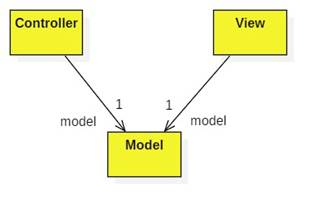
This function contains a toxic mixture of business logic—computing the square of x—and presentation logic—displaying the result in a console window. Clearly we couldn't use this function with a calculator application that displays answers in a special window, nor could we use it in an engineering application that calls square as a low-level function.

**A Solution**

The **Model-View-Controller Architecture** pattern partitions applications into three pieces: the model contains the business logic, views display the business logic to the user, and controllers execute user commands by updating the model. Together, the views and controllers (there can be more than one) make up the presentation logic or user interface. They present to the user the list of available commands (through menu items, buttons, consoles, and other controls) and display to the user what's going on in the model (through monitors, charts, meters and other output components).

The important feature of the Model-View-Controller Architecture is that the model does not depend, reference, or use the views or controllers. We can replace views and controllers without needing to make any changes to the model.

Here is a UML sketch of the architecture:



Notice that the controller and view contain one-way references to the model, but not vice-versa.