Name

Model-View-Controller

Problem

Applications often need to present data to a client—a human user or another program—so that the client can inspect, manipulate, and comprehend it with ease. How do you design such a system, which fits a sensible mental model of the data, provides intuitive usability and responsiveness, and is modular in implementation?

Context

In applications which have a client, the architecture should allow the client to interact with the data and change it, updating the client interface with the new data. For the sake of modularity, it is desirable to divide an application with this kind of interaction into an architecture that separates out the data, its manipulation, and its presentation to the client; this allows the application to be easily extended by modifying the correct subsystem.

Ideally, the application is centered on the data itself and its manipulation. There may be a number of simultaneous clients interacting with the same data, which brings the problems similar to those in repository-style architecture, such as those associated with merging conflicting changes to the data. More interestingly, a single client may want to have multiple views, with some coordination between them.

Forces

The following forces create tensions which determine the architecture:

- **Responsiveness and programmability/maintainability**: When a client initiates an action, the system should respond in a reasonable time, where the meaning of "reasonable" may be application-dependent. Meanwhile, the code behind the system as a whole must still be easily amenable to changes and fixes, and thus must avoid premature optimization which may result in unnecessary obfuscation.

- **Fault tolerance/correctness and scalability**: Data may be lost and communication links can fail, especially in network and web-based applications, and further data storage can fail. Multiple clients may attempt modifying the system simultaneously resulting in conflicts. But the system must also be easily scaled to potentially accommodate very large datasets, high throughput, and a large number of clients.

- **Presentation and Portability**: The data must be presented to the client in an easily comprehended and visually appealing manner. There may need to be more than one distinct way of displaying the data, and the client may wish to have some freedom in how to view it. However, the client may wish to interact with the data through a variety of devices, which may each require different interfaces.
Solution

Divide the application into three interacting subsystems: the model, the view, and the controller.

Model

The model is the data store and its related components. Any logic relating pieces of raw data, as well as the data store itself, are encapsulated in this module. Any relations between data items are specified in and maintained by the model. At minimum, the model exposes an interface to read, write, and modify the data. This alone suffices for constructing a passive-model MVC (Model-View-Controller) system, in which the model is unaware of existence of controllers and views. If the model changes because of sources outside the MVC triad, then there needs to be an update mechanism that notifies views/controllers of the change. Still, the model is typically only loosely aware of the views or controllers as the listeners of the state-change event. In a parallel environment, the model enforces atomicity, ensuring any interaction, even in parallel, with the model is safe.

View

The view, also referred to as a viewport, presents the model to the client. While both the view and the controller handle requests from the client, the view only handles requests from the client which change the way the model is displayed to it. The controller, discussed
below, is reserved to handle client requests to alter the model. Both the view and controller make use of the interface exposed by the model in order to accomplish these goals.

Each view in the Model-View-Controller pattern has a distinct use; one can think of providing a different view for each class of client, with each view providing interactions tuned to the client's particular class. Alternatively, a single client may wish to view data contained in the model according to a variety of available schemes. For instance, a human client may wish to view a document as HTML, as a PDF, or as XML. The view is responsible for providing this variety of renderings of the underlying model.

**Controller**

The controller interacts with both the view and the model, allowing the client to change the model, if the client has permission via the model. It is not an omnipresent piece of the pattern; some interactions (such as changing the ordering of data presentation in a view) do not require consulting the controller. However, all write and modify operations should pass through the controller, which is responsible for properly updating the data and presenting updated views to the client. The controller also passes along rule changes, such as permissions modifications or the addition of novel datatypes, to the model.

This separation of the three pieces improves maintainability and programmability by allowing each piece to be updated without changing all three--often referred to as "separation of concerns." For example, adding an "administrative" interface to an application does not need involve changing the model. In addition, it preserves fault tolerance and correctness by making the model, in its roll as a data controller, ensure that all updates to the data are correct. This not only improves the maintainability and ease with which the code can be changed, but also enables more parallel code development and more developer specialization.

Lastly, scalability is improved because views, which do not need write access to the data, can be independently run; the controller and model are responsible for synchronization. Thus read-only access is much lighter weight than full write access, and full write access has the arbitration in the model.

**Invariant**

1. The controller is the only piece that writes directly to the model and preserves correctness.
2. The client only interacts with the view.
3. The controller interacts with the view and indirectly with the client to update the model.
4. The view does not directly cause modifications in the model, but all modification requests go through the controller.

**Examples**

A company wants to create an online store. It begins by creating a model of the data: each item in the store is represented as a row in a database, with data like how many of each item are available. An example row in the database looks like:

```
item_id item_name item_description | number_available
```
Given this data representation, they then build a simple view which outputs all the item names and descriptions. The view handles display change requests by the client—for instance, to sort the displayed names alphabetically, or according to the number available. A controller is added that interfaces with the view to pass along requests to the model to change the number in stock, or alter the description of an item. At this stage, the store is ready for business: a model contains all the data, while the view allow customers to buy items by passing on purchase requests through the controller.

Later, the company decides it needs a separate interface for an internal purchasing department, one that would alert employees when the inventory of an item gets low or when an item is described as poor in function. For this, another another view/controller combination is built for employees only; much of the code from the customer view and controller is reused, but this new view includes functionality specific to purchasing department use. Thus, the company is able to easily add functionality without having to rewrite the entire application—only the portion of the problem specific to the novel situation (in this case, the purchasing department) is added. The separation of concerns between the data and the presentation/logic layers allows this to happen easily.

**Known Uses**

Model-View-Controller is implemented in: web frameworks such as Ruby on Rails, Struts, and Django, a number of GUI implementations such as Core Data in Mac OS X and Java Swing, and very many business logic applications—in the case of UC Berkeley, Bear Facts, for example.

**Related Patterns**

MVC is related/connected to the following patterns:

1. Event-based: the synchronization between the model, the view and the controller is usually event-based.
2. Agent-repository: if one model supports multiple controller-views simultaneously, we have a repository-style architecture at the model end.

**References**

1. Trygve Reenskaug: [http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html](http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html)

**Authors**

Rhishikesh Limaye, Hovig Bayandorian, Shoaib Kamil