**Model View Controller Pattern**

The Model-View-Controller (MVC) pattern organizes and separates your software into three distinct roles:

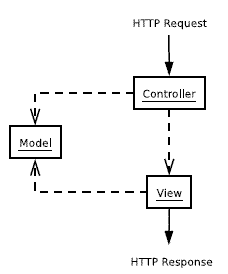
* The **Model** encapsulates application data, application flow, and business logic.
* The **View** extracts data from the Model and formats it for presentation.
* The **Controller** (aka Input controller) directs application flow and receives input and translates it for the Model and View.

**The Origins of MVC**

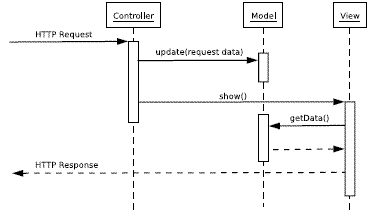
* The Model-View-Controller pattern was originally developed by **Trygve Reenskaug** at Xerox’s Palo Alto Research Center (PARC) in the late 1970s.
* The original reference implementation was coded in Smalltalk-80, and was originally designed to solve the GUI interaction problem in applications.

**Implementation**

* Unlike other design patterns, the MVCpattern does not map directly to a class structure that is coded and deployed.
* Instead, MVC is more of a conceptual guideline or paradigm.
* The conceptual MVC pattern is depicted as the relationship between three objects, the Model, the View, and the Controller.



* The ***Controller*** and the ***View*** both depend on the ***Model***, because both the View and the Controller may request data from the Model.
* Any inputs to your system enter through the Controller, which selects a View to emit results.
* In simple terms, the Controller handles each incoming HTTP request and the View generates the HTTP response.



* A request comes in to an input controller, which pulls information off the request. It then forwards the business logic to an appropriate model object.
* The model object talks to the data source and does everything indicated by the request as well as gather information for the response.
* When it's done it returns control to the input controller, which looks at the results and decides which view is needed to display the response.
* It then passes control, together with the response data, to the view.
* The input controller's handoff to the view often isn't always a straight call but often involves forwarding with the data placed in an agreed place on some form of HTTP session object that's shared between the input controller and the view.

**Advantage of MVC**

The first, and most important, reason for applying Model View Controlleris to ensure that the models are completely separated from the Web presentation. This makes it easier to modify the presentation as well as easier to add additional presentations later.

**Model**

* The Modelcontains all application logic and data and is likely the primary driver of value in an application.
* The Model has no presentation-related features and is also completely decoupled from the responsibility to process HTTP requests.
* Never include HTML tags or **$\_GET** superglobals in any PHP Model.

**Controller** (Input Controller) **Patterns**

The Controller is the one role of MVC that most MVC frameworks address.

This is reasonable considering that Models are specific to the application and nearly every developer already has their favorite template engine, a major component of the View.

That leaves interpreting the HTTP response, and controlling application flow (selecting the appropriate action to take or view to display), both approachable tasks for a generic framework.

There are two patterns used for designing input controller.

* Page Controller
* Front Controller

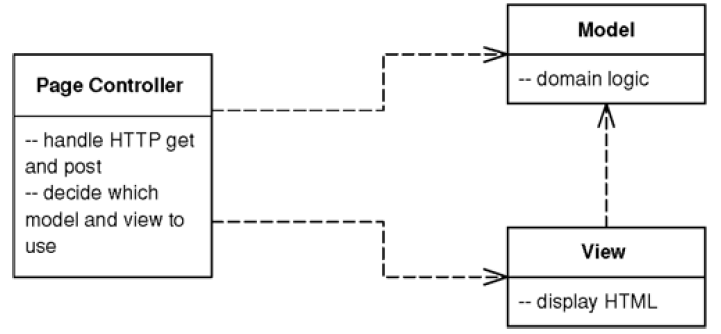
**Page Controller**

**Intent**

*An object that handles a request for a specific page or action on a Web site.*

Page Controller has one input controller for each logical page of the Web site. That controller may be the page itself, as it often is in server page environments, or it may be a separate object that corresponds to that page.

**Implementation**



**How it works**

The basic idea behind a *Page Controller* is to have one module on the Web server act as the controller for each page on the Web site.

The basic responsibilities of a Page Controllerare:

* Decode the URL and extract any form data to figure out all the data for the action.
* Create and invoke any model objects to process the data. All relevant data from the HTML request should be passed to the model so that the model objects don't need any connection to the HTML request.
* Determine which view should display the result page and forward the model information to it.

The Page Controllerneedn't be a single class but can invoke helper objects.

This is particularly useful if several handlers have to do similar tasks.

A helper class can then be a good spot to put any code that would otherwise be duplicated.

**When to Use It**

The main decision point is whether to use Page Controller or Front Controller. Of the two, Page Controller is the most familiar to work with and leads to a natural structuring mechanism where particular actions are handled by particular server pages or script classes.

The trade-off is thus the greater complexity of Front Controller against the various advantages of Front Controller, most of which make a difference in Web sites that have more navigational complexity.

Page Controller works particularly well in a site where most of the controller logic is pretty simple. In this case most URLs can be handled with a server page and the more complicated cases with helpers. When the controller logic is simple, Front Controller adds a lot of overhead.

It's not uncommon to have a site where some requests are dealt with by Page Controllers and others are dealt with by Front Controllers, particularly when a team is refactoring from one to another. Actually, the two patterns mix without too much trouble.

**Front Controller**

It’s often helpful to centralize the control of application flow at a single point.

Centralization can help to understand how a complex system operates and it also provides a single place where to insert global code such as an **Intercepting Filter**pattern.

A Front Controlleris perfect for centralization.

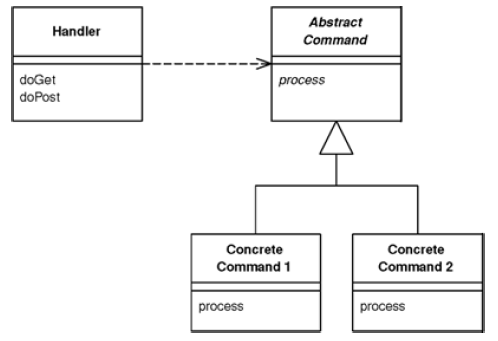
**Intent**

It is a controller that handles all requests for a Web site.

**Implementation**

The *Front Controller* consolidates all requests handling by channeling requests through a single handler object.

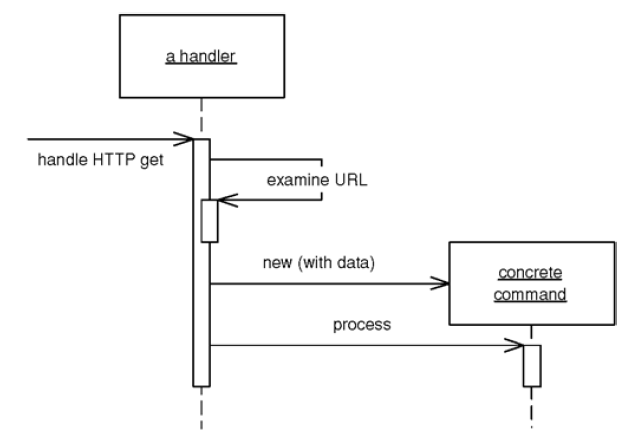
This object can carry out common behavior, which can be modified at runtime with **decorators**. The handler then dispatches to command objects for behavior particular to a request.

****

**How it works**

A Front Controllerhandles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy.

The Web handler is the object that actually receives post or gets requests from the Web server. It pulls just enough information from the URL and the request to decide what kind of action to initiate and then delegates to a command to carry out the action.



The Web handler can decide which command to run either statically or dynamically. The static version involves parsing the URL and using conditional logic; the dynamic version usually involves taking a standard piece of the URL and using dynamic instantiation to create a command class.

The static case has the advantage of explicit logic, compile time error checking on the dispatch, and lots of flexibility in the look of URLs. The dynamic case allows addition of new commands without changing the Web handler.

With dynamic invocation it is possible to put the name of the command class into the URL or use a properties file that binds URLs to command class names. The properties file is another file to edit, but it does make it easier to change class names without a lot of searching through Web pages.

A particularly useful pattern to use in conjunction with *Front Controller* is ***Intercepting Filter***.

This is essentially a decorator that wraps the handler of the front controller allowing you to build a **filter chain**(or pipeline of filters) to handle issues such as authentication, logging, and locale identification.

Using filters allows dynamic set up of filters that can be used at configuration time.

**View Patterns**

The View manages all aspects of presentation.

A Viewextracts data from a Modeland might format it as HTML for a web page, as XML for a web service, or as text for email.

One good way to identify if you’ve succeeded in separating your code into well-defined roles is to try substituting (at least conceptually) another Viewthat produces completely different output.

While the Viewhas access to the Model, it is bad form for a Viewto call methods of the Modelthat change state—updates should only be performed by the Controller. The Modelmethods called by the Viewshould be read-only data retrieval methods with no side effects.

There are two design patterns commonly used in Views:

* TemplateView
* TransformView

**Template View**

**Intent**

A template view renders information into HTML by embedding markers in an HTML page.

**Implementation**

This pattern uses a template file (usually HTML) that includes special markers that are replaced with data from the Modelwhen the TemplateViewis executed.

When the page is used to service a request, the markers are replaced by the results of some computation, such as a database query.

This way the page can be laid out in the usual manner, often with WYSIWYG editors, often by people who are not programmers.

The markers then communicate with real programs to put in the results.

Possible Implementations:

* Using PHP - PHP itself is an example of a specific type of Template View called a server page.

Disadvantages :

1. Putting a lot of scriptlets into a page is that it eliminates the possibility of nonprogrammers editing the page. This is particularly important when there are graphic/web designers creating the page design.
2. Putting a lot of scriptlets into the page makes it too easy to mingle the different layers of an enterprise application. When domain logic starts turning up on server pages it becomes far too difficult to structure it well and far too easy to duplicate it across different server pages.

* Using a View Helper Object

The key to avoiding scriptlets is to provide a regular object as a **helper** to each page. This helper has all the real programming logic.

The page only has calls into it, which simplifies the page and makes it a more pure TemplateView*.*

The resulting simplicity allows nonprogrammers to edit the page and programmers to concentrate on the helper.

* Template engines : Smarty or Savant3

**When to use IT**

For implementing the view in Model View Controllerthe main choice is between Template Viewand Transform View*.*

The strength of Template Viewis that it allows to compose the content of the page by looking at the page structure.

This seems to be easier for most people to do and to learn. In particular it nicely supports the idea of a graphic designer laying out a page with a programmer working on the helper.

Template Viewhas two significant weaknesses:

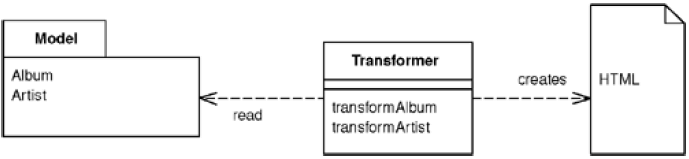
* First, the common implementations make it too easy to put complicated logic in the page, thus making it hard to maintain, particularly by nonprogrammers. There need to have good discipline to keep the page simple and display oriented, putting logic in the helper.
* The second weakness is that Template Viewis harder to test than Transform View. Most implementations of TemplateVieware designed to work within a Web server and are very difficult or impossible to test otherwise.

**Transform View**

**Intent**

A view that processes domain data element by element and transforms it into HTML.

**Implementation**



The Transform View extracts data from your model and transforms the data into the desired output format.

It essentially amounts to using a language to step through the elements of your data one by one, assembling the output along the way.

**How It Works**

The basic notion of Transform Viewis writing a program that looks at domain-oriented data and converts it to HTML.

The program walks the structure of the domain data and, as it recognizes each form of domain data, it writes out the particular piece of HTML for it.

The key difference between Transform Viewand Template Viewis the way in which the view is organized.

A Template Viewis organized around the output.

A Transform Viewis organized around separate transforms for each kind of input element.

The transform is controlled by something like a simple loop that looks at each input element, finds the appropriate transform for that element, and then calls the transform on it. (A parser)

Transform view can be written in any language.

XSLT (Extensible Stylesheet Language Transformations) is the preferred choice.

XSLT is mainly used for transforming XML documents into other XML documents, or other objects such as HTML for web pages, plain text or into XSL Formatting Objects which can then be converted to PDF, PostScript and PNG.

Typically, input documents are XML files, but anything from which the processor can build an XQuery and XPath Data Model can be used, for example relational database tables, or geographical information systems.

The logic for the transform is captured in an XSLT style sheet, which is also passed to the transformer.

The transformer then applies the stylesheet to the input XML to yield the output HTML, which can be written directly to the HTTP response.

**When to use IT**

The choice between a Transform Viewand a Template Viewmostly comes down to which environment the team working on the view software prefers.

The difference between the Template Viewand the Transform Viewis the direction of data flow.

* In the Template Viewstarts with a skeleton of the output and insert domain data into it.
* In the Transform Viewstarts with the data and build the output from it.

One of the strengths of XSLT is its portability to almost any Web platform. The same XSLT can be used to transform XML created from PHP, Java or .NET, which can help in putting a common HTML view on data from different sources.

XSLT is also often easier in case of building a view on an XML document. Other environments usually require transforming such a document into an object or to indulge in walking the XML DOM, which can be complicated. XSLT fits naturally in an XML world.

Transform Viewavoids two of the biggest problems with Template View:

* It's easier to keep the transform focused only on rendering HTML, thus avoiding having too much other logic in the view.
* It's also easy to run the Transform Viewand capture the output for testing. This makes it easier to test the view.

Transform Viewtransforms directly from domain-oriented XML into HTML.

If there is a need to change the overall appearance of a Web site, this can require to changing multiple transform programs. Using common transforms, such as with XSLT includes, helps reduce this problem.