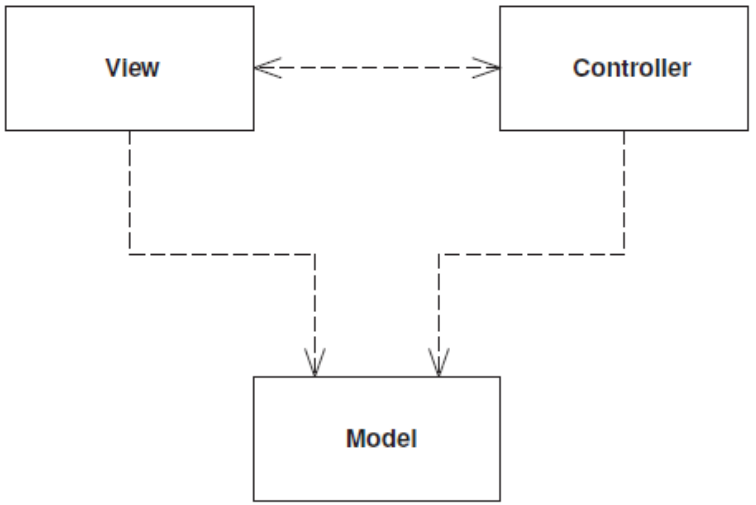
SOEN 387: Web-Based Enterprise Application Design

Chapter 14. Model View Controller Patterns

Model View Controller

Splits user interface interaction into three distinct roles.



MVC considers three roles. The model is an object that represents some information about the domain.

It is an object containing all the data and behavior other than that used for the UI. In its most pure OO form the model is an object within a *Domain Model*.

The view represents the display of the model in the UI. Thus, if our model is a customer object our view might be a frame full of UI widgets or an HTML page rendered with information from the model.

The view is only about display of information; any changes to the information are handled by the controller.

The controller takes user input, manipulates the model, and causes the view to update appropriately. In this way UI is a combination of the view and the controller.

Separation of presentation from model is one of the most fundamental heuristics of good software design. This separation is important for several reasons.

1. Fundamentally presentation and model are about different concerns. When you are developing a view you are thinking about the mechanisms of UI and how to lay out a good user interface. When you are working with a model you are thinking about business policies, perhaps database interactions. Certainly you will use different very different libraries when working with one or the other. Often people prefer one area to another and they specialize in one side of the line.

2. Depending on context, users want to see the same basic model information in different ways. Separating presentation and view allows you to develop multiple presentations—indeed, entirely different interfaces—and yet use the same model code. The same model could be provided with a Web browser, a remote API, and a command-line interface. Even within a single Web interface you might have different customer pages at different points in an application.

3. Separating presentation and model allows you to test all the domain logic easily without resorting to things like GUI scripting tools.

A key point in this separation is the direction of the dependencies: the presentation depends on the model but the model doesn't depend on the presentation.

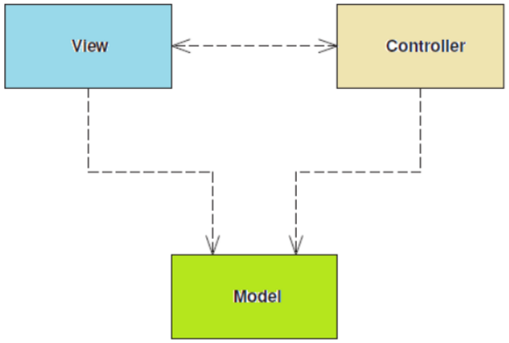
People programming in the model should be entirely unaware of what presentation is being used, which both simplifies their task and makes it easier to add new presentations later on. It also means that presentation changes can be made freely without altering the model.

This principle introduces a common issue. With a rich-client interface of multiple windows it's likely that there will be several presentations of a model on a screen at once. If a user makes a change to the model from one presentation, the others need to change as well.

To do this without creating a dependency you usually need an implementation of the Observer pattern, such as event propagation or a listener.

The presentation acts as the observer of the model: whenever the model changes it sends out an event and the presentations refresh the information.

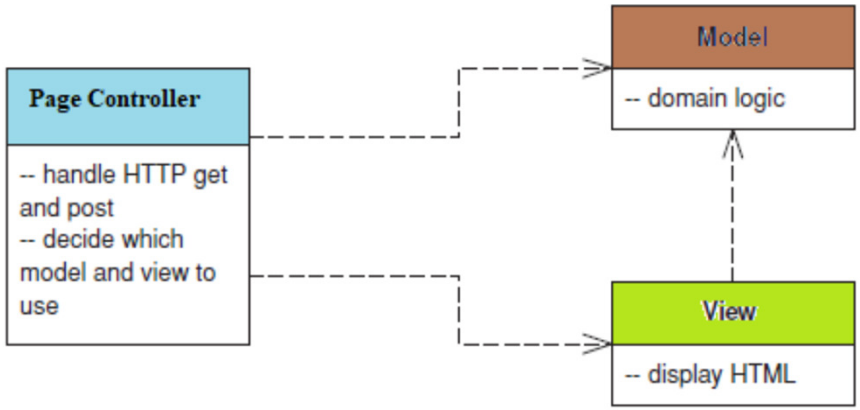
The second division, the **separation of view and controller**, is less important.



The fact that most GUI frameworks combine view and controller has led to many misquotations of MVC. The model and the view are obvious, but where is the controller? The common idea is that it sits between the model and the view,

Page Controller

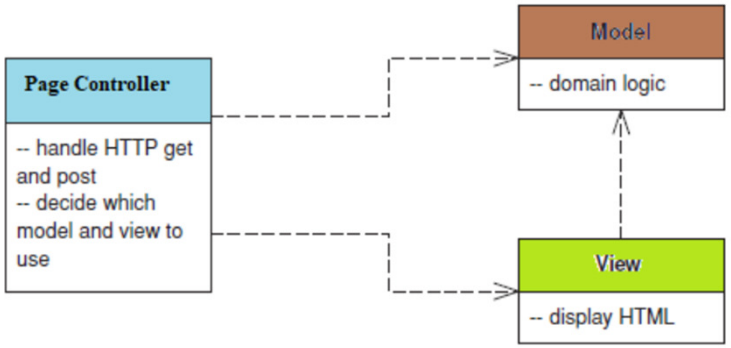
An object that handles a request for a specific page or action on a Web site.



Most people's basic Web experience is with static HTML pages. When you request static HTML you pass to the Web server the name and path for a HTML document stored on it. The key notion is that each page on the Web site is a separate document on the server.

With dynamic pages things can get much more interesting since there is a much more complex relationship between path names and the file that responds. However, the approach of one path leading to one file that handles the request is a simple model to understand.

The basic responsibilities of a *Page Controller* are:



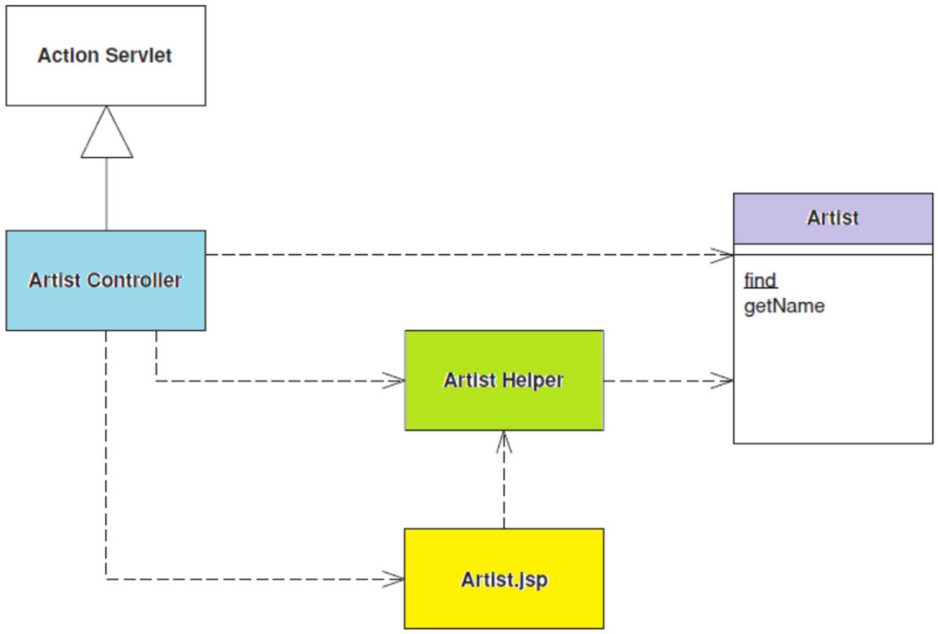
- Decode the URL and extract any form data to figure out all the data for the action.

- 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.

Example: Simple Display with a Servlet Controller and a JSP View (Java)

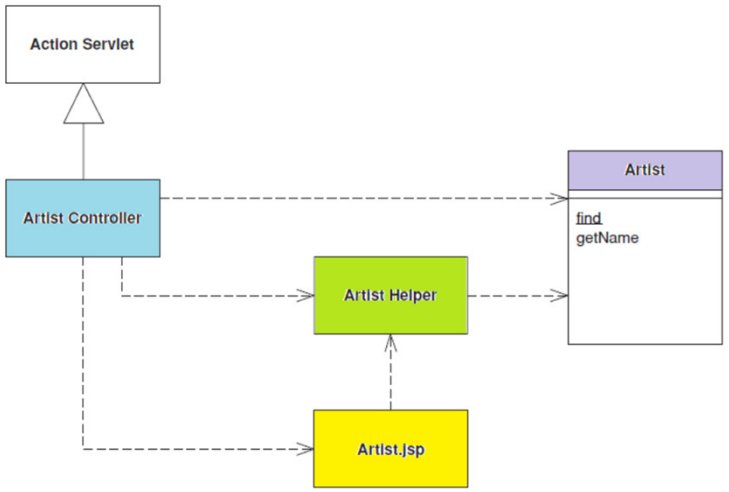
A simple example of a ***Page Controller***. Here we will show it displaying some information about a recording artist.
The URL runs along the lines of
http://www.thingy.com/recordingApp/artist?name=danielaMercury.



Classes involved in a simple display with a Page Controller *servlet and a JSP view.*

The artist controller needs to implement a method to handle the request.

```
class ArtistController..
public void doGet(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException {
Artist artist = Artist.findNamed(request.getParameter("name"));
if (artist == null)
          forward("/MissingArtistError.jsp", request, response);
else {
          request.setAttribute("helper", new ArtistHelper(artist));
          forward("/artist.jsp", request, response);
}
}
```

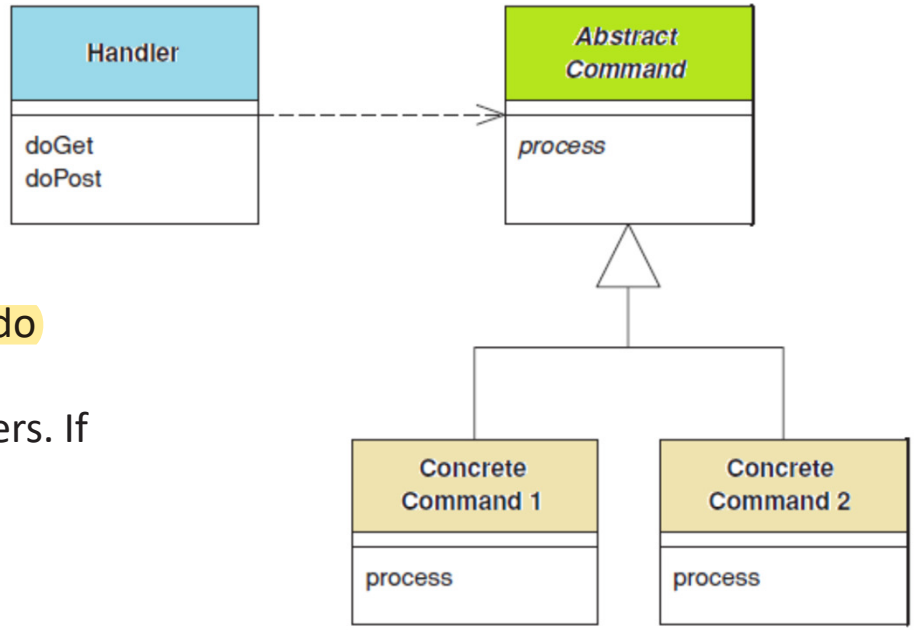


Classes involved in a simple display with a Page Controller *servlet and a JSP view.*

```
class ActionServlet...
protected void forward(String target, HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
          RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(target);
          dispatcher.forward(request, response);
}
```

Front Controller

A controller that handles all requests for a Web site.

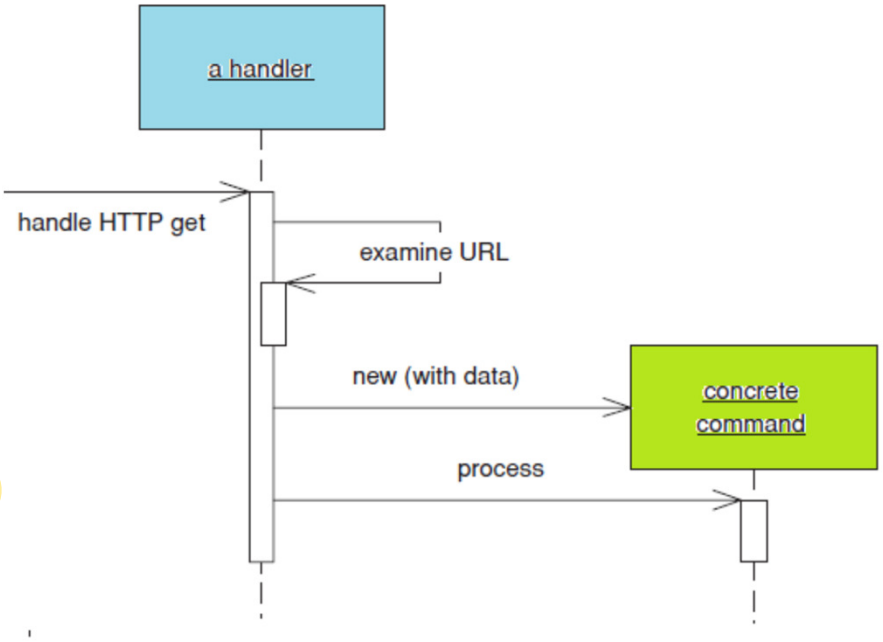


In a complex Web site there are many similar things you need to do when handling a request. These things include security, internationalization, and providing particular views for certain users. If the input controller behavior is scattered across multiple objects, much of this behavior can end up duplicated. Also, it's difficult to change behavior at runtime.

The *Front Controller* consolidates all request handling by channeling requests through a single handler object. The handler then dispatches to command objects for behavior particular to a request.

A *Front Controller* handles 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 get 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.



How the Front Controller *works.*

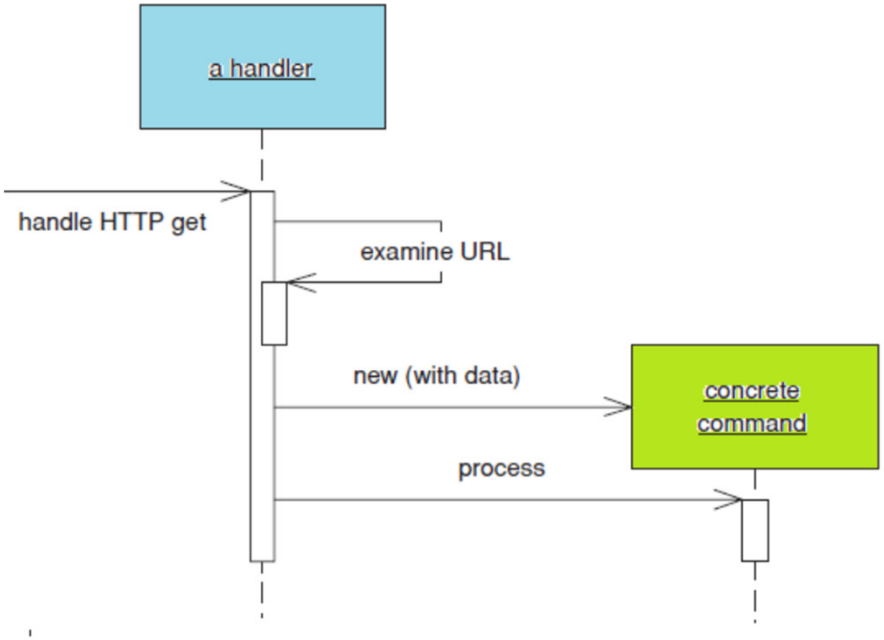
The Web handler can decide which command to run either statically or dynamically.

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 your URLs.

The dynamic case allows you to add new commands without changing the Web handler.

With dynamic invocation you can put the name of the command class into the URL or you can use a properties file that binds URLs to command class names.



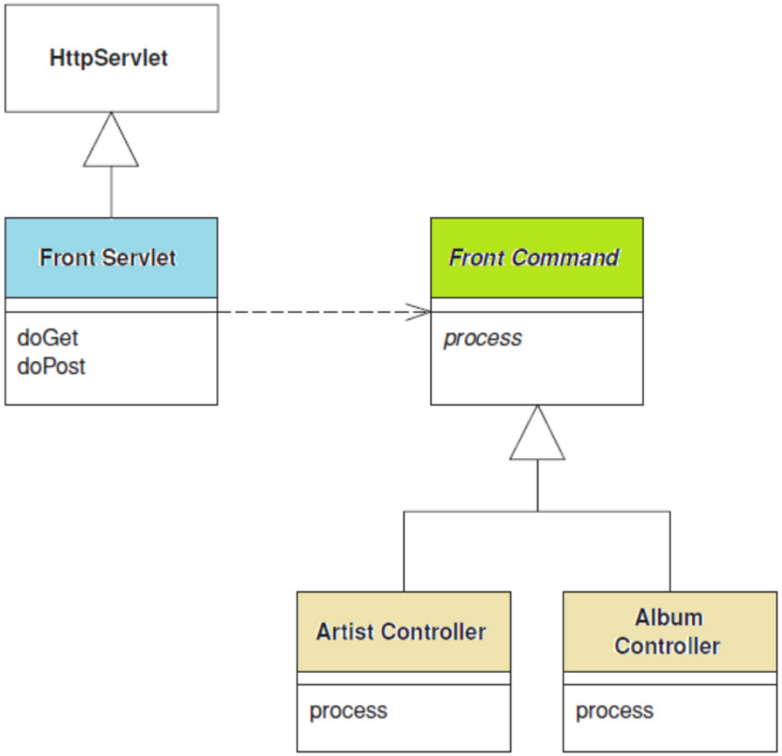
How the Front Controller *works.*

Example: Simple Display (Java)

Here is a simple case of using *Front Controller* for displaying information about a recording artist. We will use dynamic commands with a URL of the form: http://localhost:8080/isa/music?name=barelyWorks&command=Artist. The command parameter tells the Web handler which command to use.

```
class FrontServlet...
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
            FrontCommand command = getCommand(request);
            . . . . . .
            command.process();
}
private FrontCommand getCommand(HttpServletRequest request)
{
try
{
    return (FrontCommand) getCommandClass(request).newInstance();
}
catch (Exception e)
{           throw new ApplicationException(e); }
}

private Class getCommandClass(HttpServletRequest request) {
Class result;
final String commandClassName = "frontController." + (String)
request.getParameter("command") + "Command";
try { result = Class.forName(commandClassName); }
catch (ClassNotFoundException e) {result = UnknownCommand.class;}
return result;
}
```



The classes that implement Front Controller.

The handler tries to instantiate a class named by concatenating the command name and “Command.” Once it has the new command it initializes it with the necessary information from the HTTP server.

The *Front Controller* is a more complicated design than its obvious counterpart, *Page Controller* . It therefore needs a few advantages to be worth the effort.

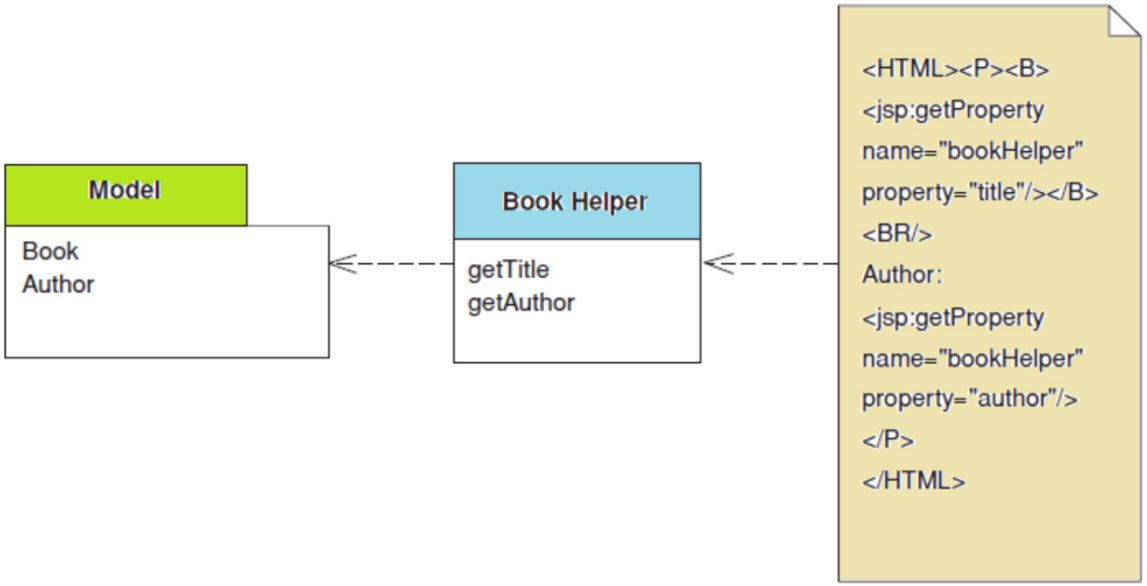
Page Controller works particularly well in a site where most of the controller logic is pretty simple. In this case most URLs can be handled with a server page and the more complicated cases with helpers. When your controller logic is simple, *Front Controller* adds a lot of overhead.

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.

With dynamic commands you can add new commands without changing anything.

Template View

Renders information into HTML by embedding markers in an HTML page.

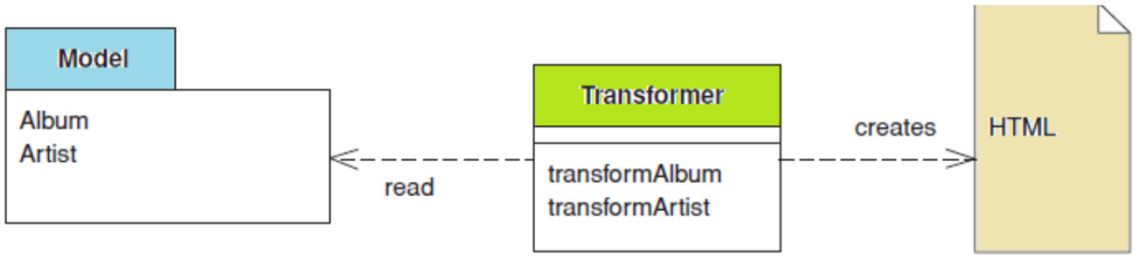


One of the most popular forms of *Template View* is a **server page** such as ASP, JSP, or PHP. These actually go a step further than the basic form of a *Template View* in that they allow you to embed arbitrary programming logic, referred to as **scriptlets**, into the page.

The most obvious disadvantage of putting a lot of scriptlets into a page is that it eliminates the possibility of nonprogrammers editing the page

Transform View

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



Using *Transform View* means thinking of this as a transformation where you have the model's data as input and its HTML as output.

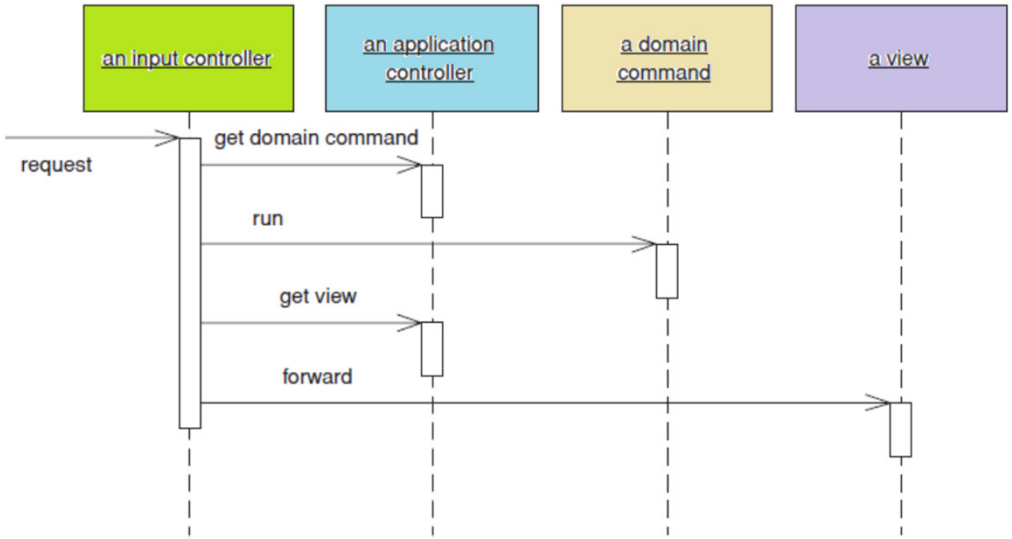
The basic notion of *Transform View* is writing a program that looks at domain data and converts it to HTML. The program walks the structure of the domain data and, as it recognizes each form of domain data, it writes out the particular piece of HTML for it.

Application Controller

A centralized point for handling screen navigation and the flow of an application.

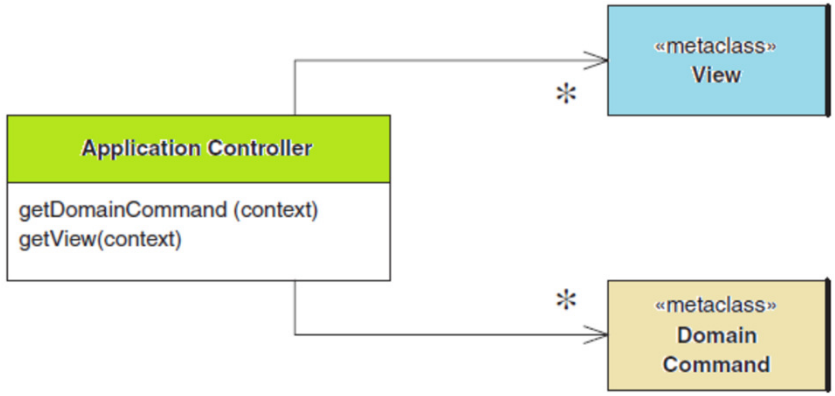
Some applications contain a significant amount of logic about the screens to use at different points, which may involve invoking certain screens at certain times in an application. This is the wizard style of interaction, where the user is led through a series of screens in a certain order.
In other cases we may see screens that are only brought in under certain conditions, or choices between different screens that depend on earlier input.

To some degree the various *Model View Controller* input controllers can make some of these decisions, but as an application gets more complex this can lead to duplicated code as several controllers for different screens need to know what to do in a certain situation.
You can remove this duplication by placing all the flow logic in an *Application Controller*. Input controllers then ask the *Application Controller* for the appropriate commands for execution against a model and the correct view to use depending on the application context.



An *Application Controller* has two main responsibilities: deciding which domain logic to run and deciding the view with which to display the response. To do this it typically holds two structured collections of class references, one for domain commands to execute against in the domain layer and one of views .

The *Application Controller* as an intermediate layer between the presentation and the domain. An application can have multiple *Application Controllers* to handle each of its different parts. This allows you to split up complex logic into several classes.



An application controller has two collections of references to classes, one for domain logic and one for view.

If the flow and navigation of your application are simple enough so that anyone can visit any screen in pretty much any order, there's little value in a *Application Controller*. The strength of an *Application Controller* comes from definite rules about the order in which pages should be visited and different views depending on the state of objects.