Systems for Design and Implementation

2015-2016

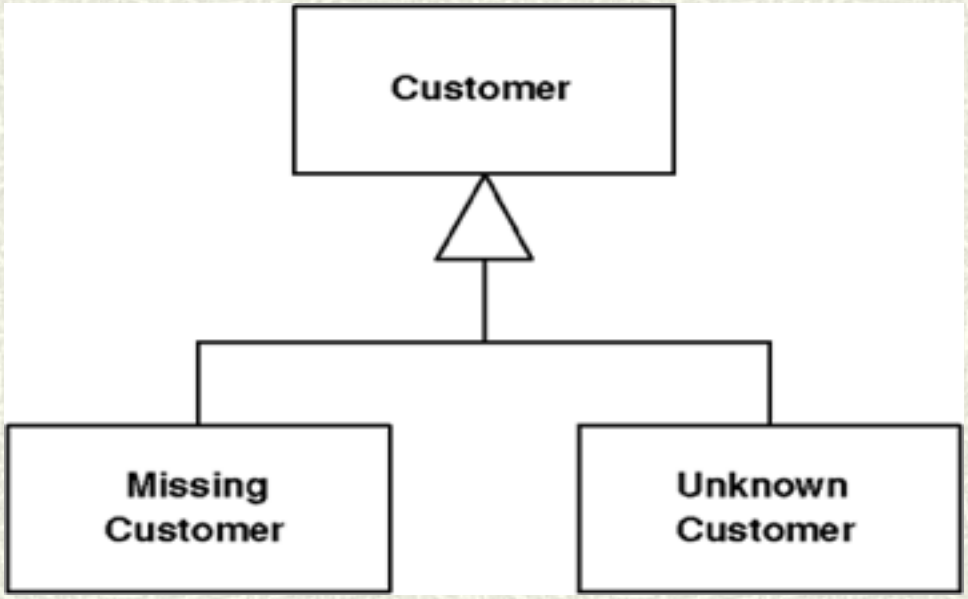
Course 11

Contents

- Special Case Pattern
- Distribution Patterns
- Web Presentation Patterns
- Session State Patterns

Special Case Pattern

- A subclass that provides special behavior for particular cases.
- Sometimes the result returned by a method is null (special case).
- Instead of returning null, return a *Special Case* that has the same interface as what the caller expects.



Special Case Pattern

- The basic idea is to create a subclass to handle the Special Case.
- E.g, if you have a customer object and you want to avoid null checks, you make a null customer object.
- All the methods for customer are overriden in the Special Case to provide some harmless behavior. Then, whenever the result should be null, an instance of the Special Case is returned.
- A null can mean different things:
  - a null customer may mean no customer
  - it may mean that there is a customer but we do not know who it is.
- In this case it is appropriate to have separate Special Cases for missing customer and unknown customer.

Special Case Pattern - Example

```
class Employee{
      public virtual String Name {
         get {return _name;}
         set {_name = value;}
      }
      private String _name;
      public virtual Contract Contract {
         get {return _contract;}
      }
      private Contract _contract;
   …
}
```

Special Case Pattern

```
class NullEmployee : Employee, INull {
      public override String Name {
         get {return "Null Employee";}
         set {}
      }
    public override Contract Contract {
         get {return Contract.NULL;}
      }
   …
}

class Contract{
   public static const Contract NULL=new NullContract(…);
   …
}
```

Distribution Patterns

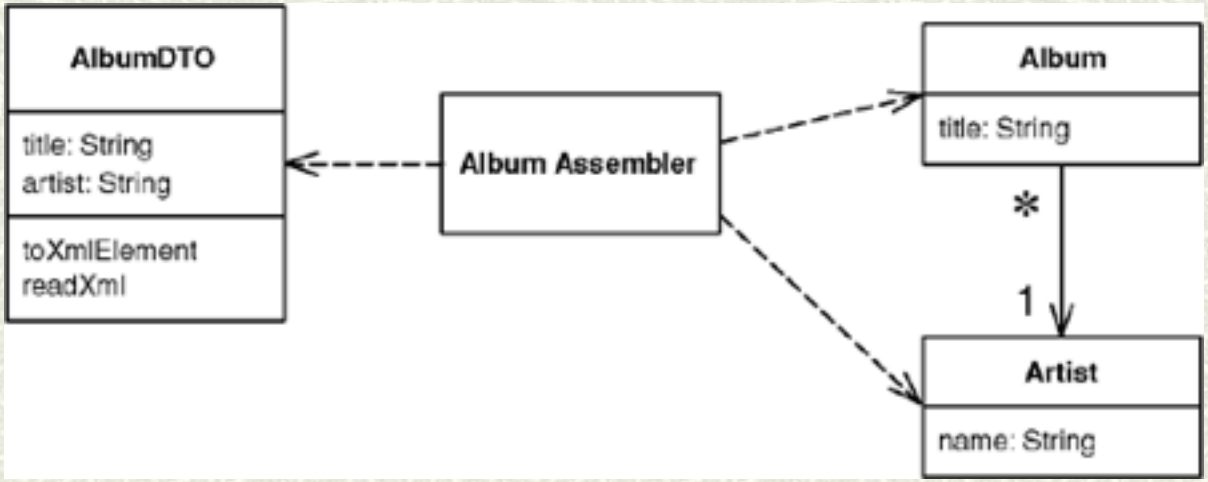
- A procedure call within a process is very, very fast.
- A procedure call between two separate processes is much more slower. If a process is also running on another machine then the communication is even slower, depending on the network bandwidth and load.
- The interface for an object to be used remotely must be different from that for an object used locally within the same process.
- Patterns:
  - Data Transfer Object
  - Remote Façade

Data Transfer Object

- An object that carries data between processes in order to reduce the number of method calls.
- When working with a remote service, each call to it is expensive. As a result the number of calls should be reduced, meaning that more data must be transferred with each call.
- One way to do this is to use more parameters:
  - often awkward to program
  - in some case impossible (i.e., in Java you can return only one value).
- The solution is to create a *Data Transfer Object* that can hold all the data for the call. It needs to be serializable (binary, XML, etc.) to be passed across the connection. Usually another object, called *assembler,* is used on the server side to transfer data between the DTO and any domain objects.

Data Transfer Object

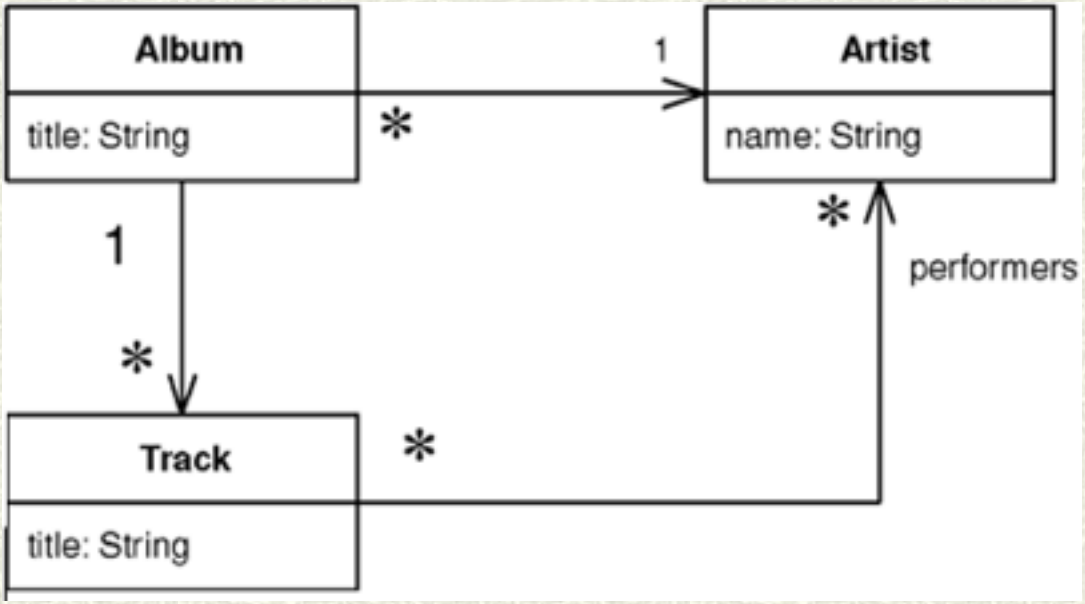
- A DTO usually only contains a lot of fields and the getters and setters for them.
- Whenever a remote object needs some data, it asks for a suitable DTO. The DTO will usually carry much more data than what the remote object requested, but it should carry all the data the remote object will need for a while.
- A single DTO usually contains more than just a single object. It aggregates data from all the objects that the remote object is likely to want data from.



Data Transfer Object

- A Data Transfer Object should be used whenever you need to transfer multiple items of data between two processes in a single method call.
- Alternatives:
  - To use a setting method with many arguments or a getting method with several pass-by reference arguments. The problem is that many languages, such as Java, allow only one object as a return value. Although this can be used for updates, it cannot be used for retrieving information.
  - To use some form of string representation directly, without an object acting as the interface to it. Here the problem is that everything else is coupled to the string representation.

Data Transfer Object - Example



Data Transfer Object - Example

```
class AlbumAssembler{
   public AlbumDTO writeDTO(Album subject) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(subject.getTitle());
      result.setArtist(subject.getArtist().getName());
      writeTracks(result, subject);
      return result;
   }
   private void writeTracks(AlbumDTO result, Album subject) {
      List<TrackDTO> newTracks = new ArrayList<TrackDTO>();
      Iterator<Track> it = subject.getTracks().iterator();
      while (it.hasNext()) {
         TrackDTO newDTO = new TrackDTO();
         Track thisTrack = it.next();
         newDTO.setTitle(thisTrack.getTitle());
         writePerformers(newDTO, thisTrack);
         newTracks.add(newDTO);
      }
result.setTracks(newTracks.toArray(new TrackDTO[newTracks.size()]));
   }
```

Data Transfer Object - Example

```
private void writePerformers(TrackDTO dto, Track subject) {
      List<String> result = new ArrayList<String>();
      Iterator<Artist> it = subject.getPerformers().iterator();
      while (it.hasNext()) {
         Artist each = it.next();
         result.add(each.getName());
      }
 dto.setPerformers(result.toArray(new String[result.size()]));
   }
```

Remote Facade

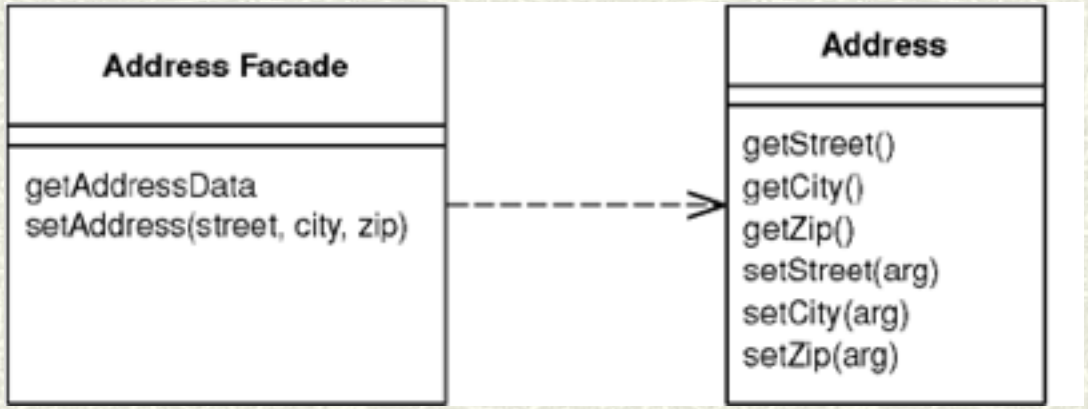
- In an object-oriented model, small objects that have small methods are better. One of the consequences of such fine-grained behavior is that there is usually a great deal of interaction between objects, and that interaction usually requires lots of method invocations.

- Within a single address space fine-grained interaction works well, but this is not true for calls between processes. Remote calls are much more expensive (data may have to be marshaled, security may need to be checked, packets may need to be routed through switches). Any inter-process call is much more expensive than an in-process call—even if both processes are on the same machine.

- Any object that is intended to be used as a remote objects needs a coarse-grained interface that minimizes the number of calls needed to get something done. This affects the method calls, and also the objects.

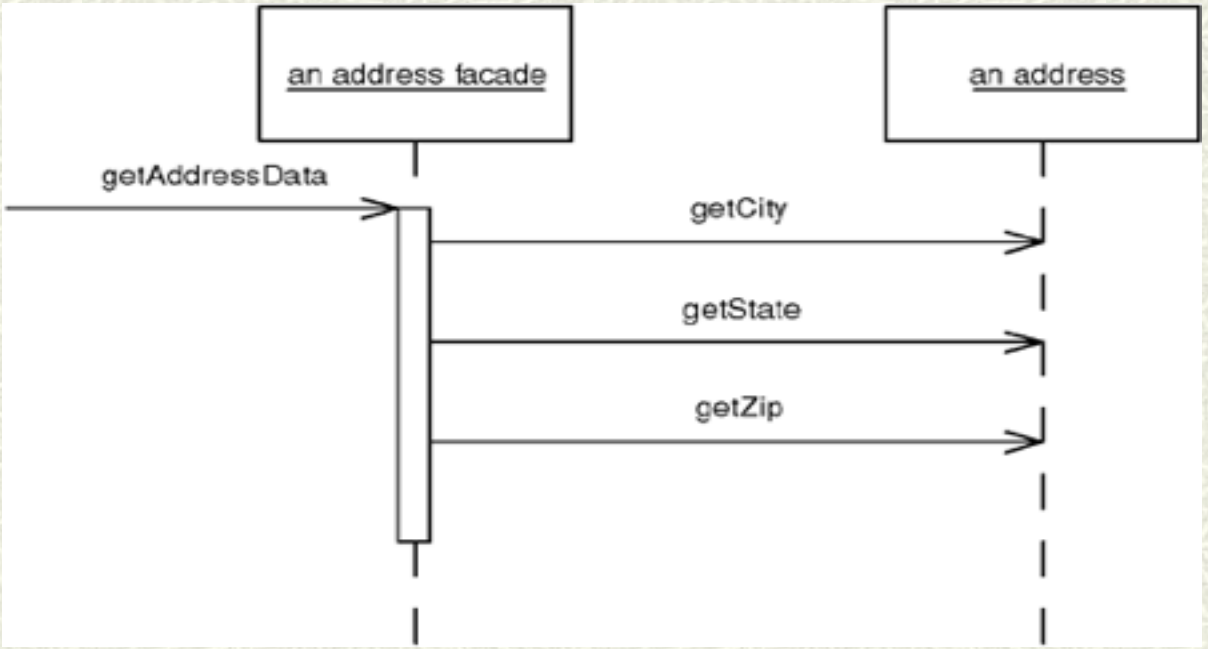
Remote Facade

- A Remote Facade is a coarse-grained facade over a web of fine-grained objects. None of the fine-grained objects have a remote interface, and the Remote Facade contains no domain logic. All the Remote Facade does is translate coarse-grained methods onto the underlying fine-grained objects.
- A Remote Façade provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.



Remote Facade

- In a simple case, a Remote Facade replaces all the getting and setting methods of the regular object(s) with one getter and one setter, often referred to as *bulk accessors*. When a client calls a bulk setter, the facade reads the data from the setting method and calls the individual accessors on the real object and does nothing more.



Remote Facade

- Remote Façade should be used whenever you need remote access to a fine-grained object model. You gain the advantages of a coarse-grained interface while still keeping the advantage of fine-grained objects.

- The most common use of this pattern is between a presentation and a Domain Model, where the two may run on different processes(i.e.between a swing UI and server domain model, or between a servlet and a server object model if the application and Web servers are different processes).

Web Applications

- A web application is an application that is accessed via a web browser over a network such as the Internet or an intranet.
- It is also a computer software application that is coded in a browser-supported language (such as HTML, JavaScript, Java, etc.) and that relies on a common web browser to render the application executable.
- Web applications commonly use a combination of server-side script (ASP, PHP, etc) and client-side script (HTML, Javascript, etc.) to develop the application. The client-side script deals with the presentation of the information while the server-side script deals with operations like storing and retrieving the information.
- Web Presentation Patterns:
  - Model View Controller
  - Page Controller, Front Controller
  - Template View, Transform View, Two Step View
  - Application Controller

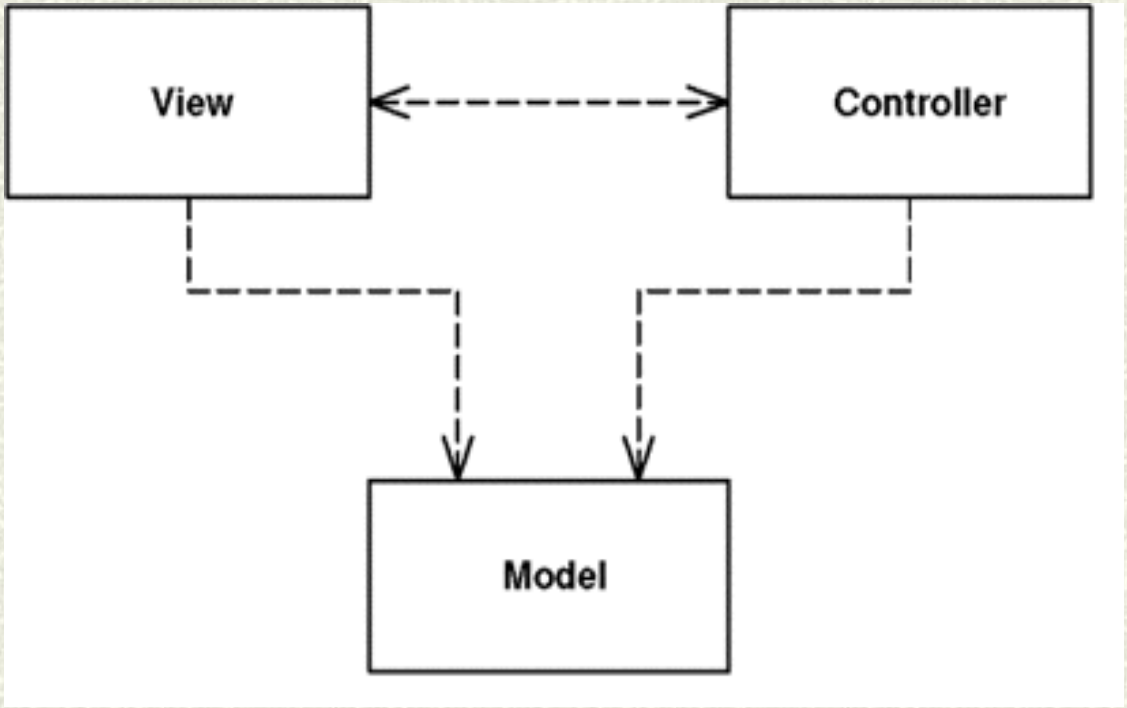
Model-View-Controller

- 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 a nonvisual object containing all the data and behavior other than that used for the UI.
  - The view represents the display of the model in the UI. 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.
- Two principal separations: separating the presentation from the model and separating the controller from the view.

Model-View-Controller

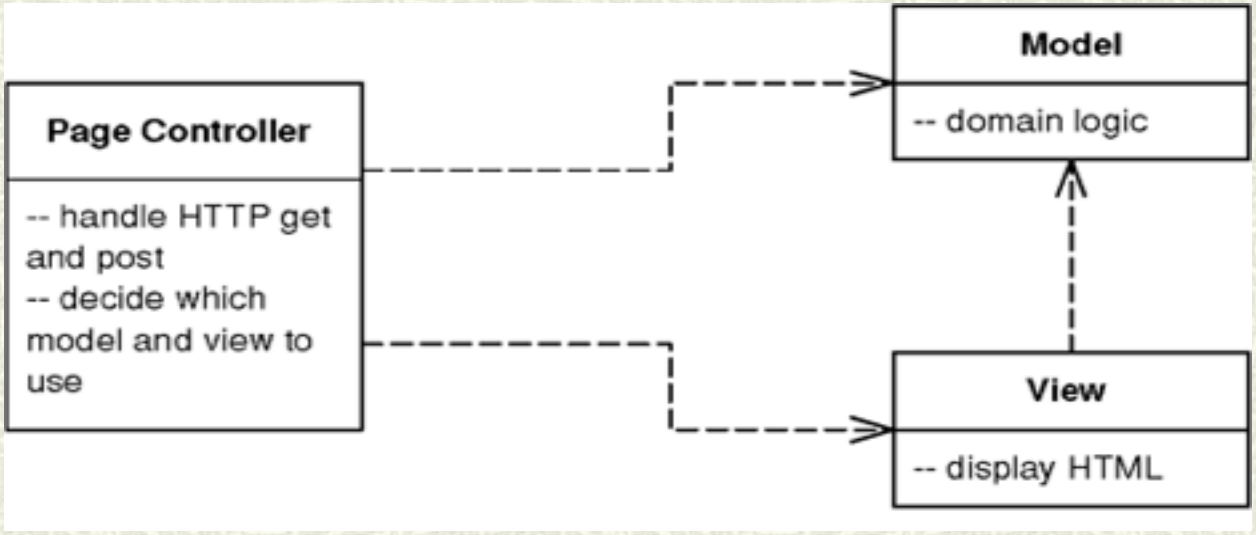
- Separation of the presentation from the model is fundamental:
  - Presentation and view are about different concerns. When you are developing a view you are thinking about the mechanisms of UI and how to lay out a good user interface. When you are working with a model you are thinking about business policies, database interactions, etc.
  - Separating presentation and model allows the development of multiple presentations, entirely different interfaces that use the same model code.
  - Non-visual objects are usually easier to test than visual ones. Separating presentation and model allows you to test all the domain logic easily.
  - The direction of the dependency is: the presentation depends on the model but the model does not depend on the presentation. People developing the model should be entirely unaware of what presentation is being used.
- Separation of view and controller, is less important.
  - It is more common in Web applications where the controller is separated out.

Model-View-Controller



Page Controller

- 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 is in server page environments, or it may be a separate object that corresponds to that page.
- 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.

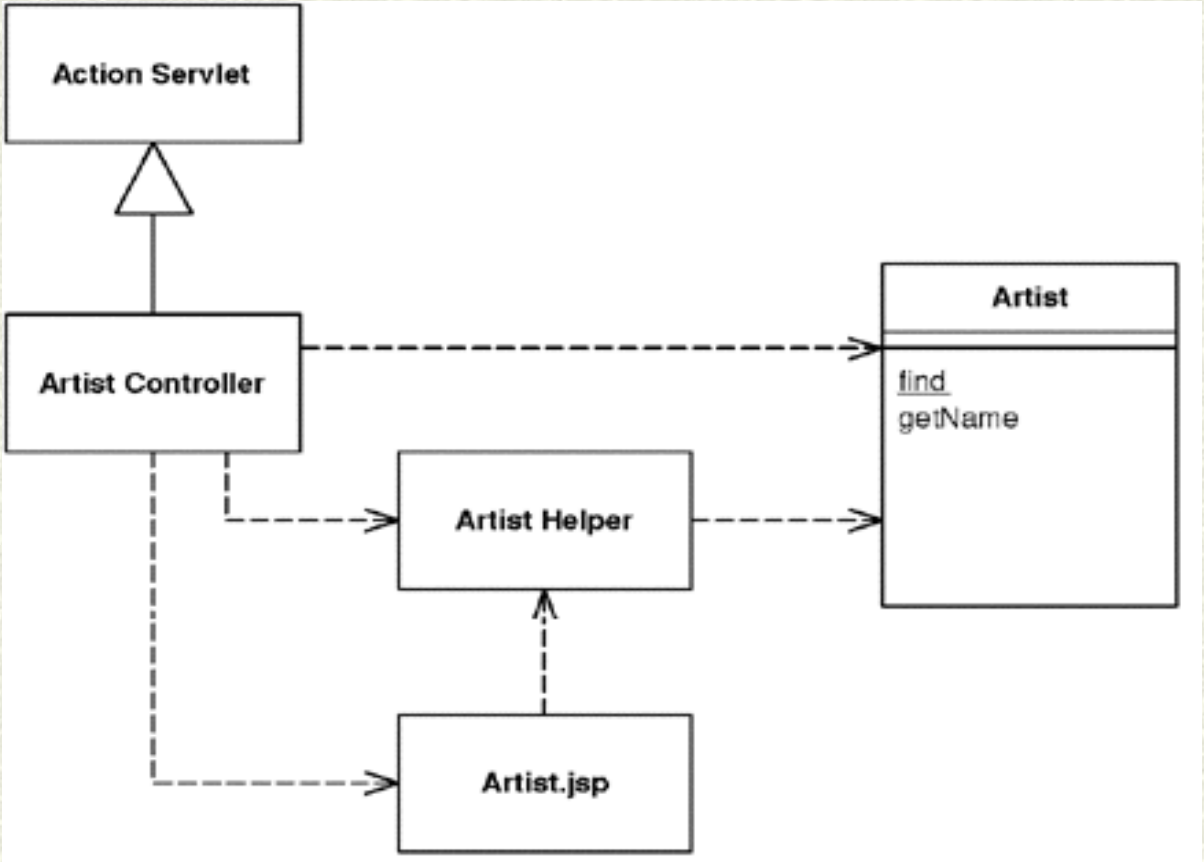


Page Controller

- The Page Controller can be structured either as a script (CGI script, servlet, etc.) or as a server page (ASP, PHP, JSP, etc.).
- Basic responsibilities:
  - 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 do not need any connection to the HTML request.
  - Determine which view should display the result page and forward the model information to it.
- The Page Controller can invoke helper objects.
- 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.

Page Controller - Example

http://www.xyz.com/recordingApp/artist?name=IRIS



Page Controller - Example

The Web server needs to be configured to recognize `/artist` as a call to `ArtistController`.

Tomcat: add code in the `web.xml` file:

```
<servlet>
   <servlet-name>artist</servlet-name>
   <servlet-class>actionController.ArtistController</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>artist</servlet-name>
   <url-pattern>/artist</url-pattern>
</servlet-mapping>
```

Page Controller - Example

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);
      }
   }
```

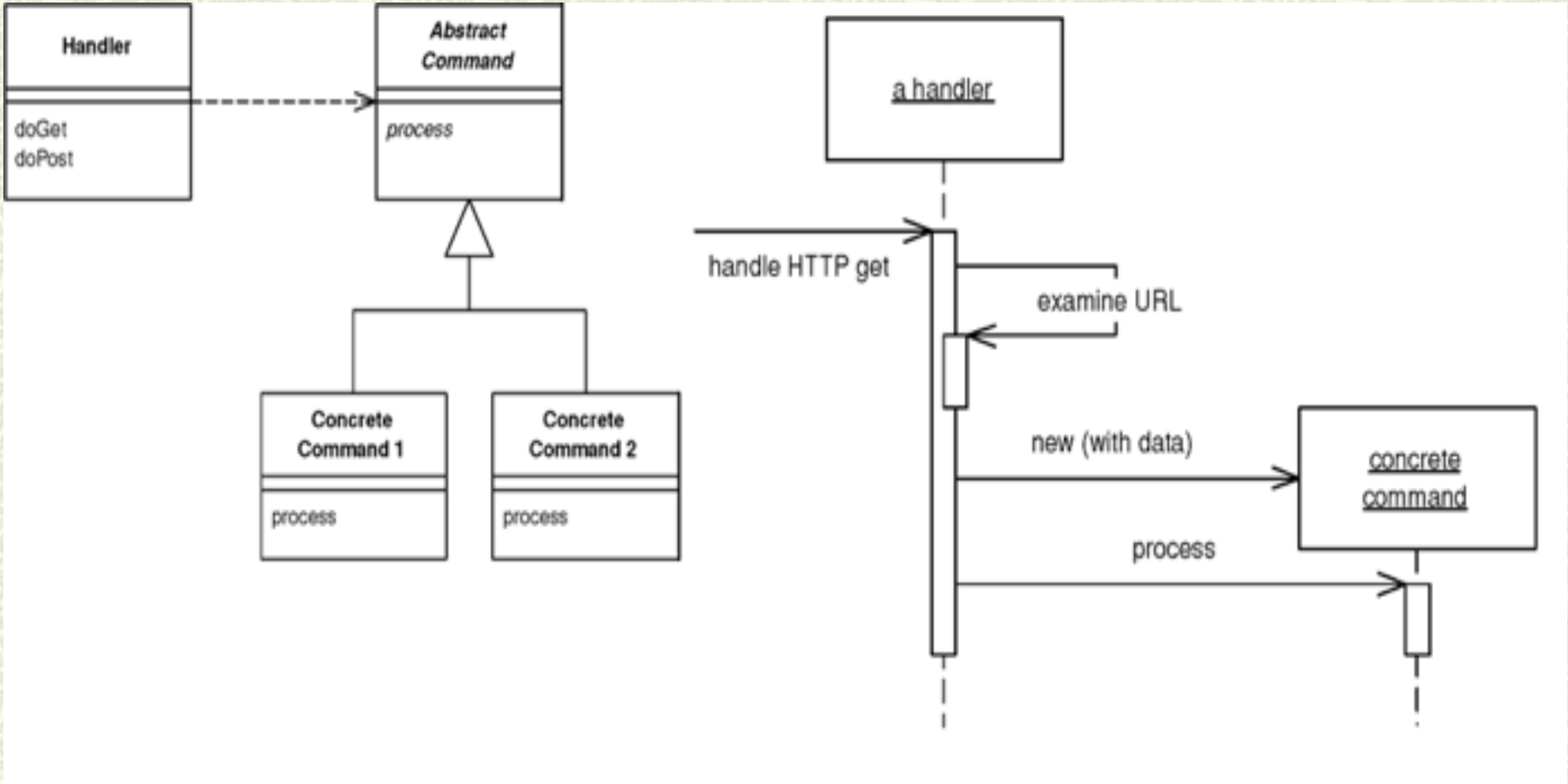
Page Controller - Example

```
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.
- It 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.
- The Web handler is almost always implemented as a class rather than as a server page, as it does not produce any response. The commands are also classes rather than server pages and do not need any knowledge of the Web environment, although they are often passed the HTTP information.
- The Web handler itself is usually a fairly simple program that does nothing other than decide which command to run.

Front Controller



Front Controller

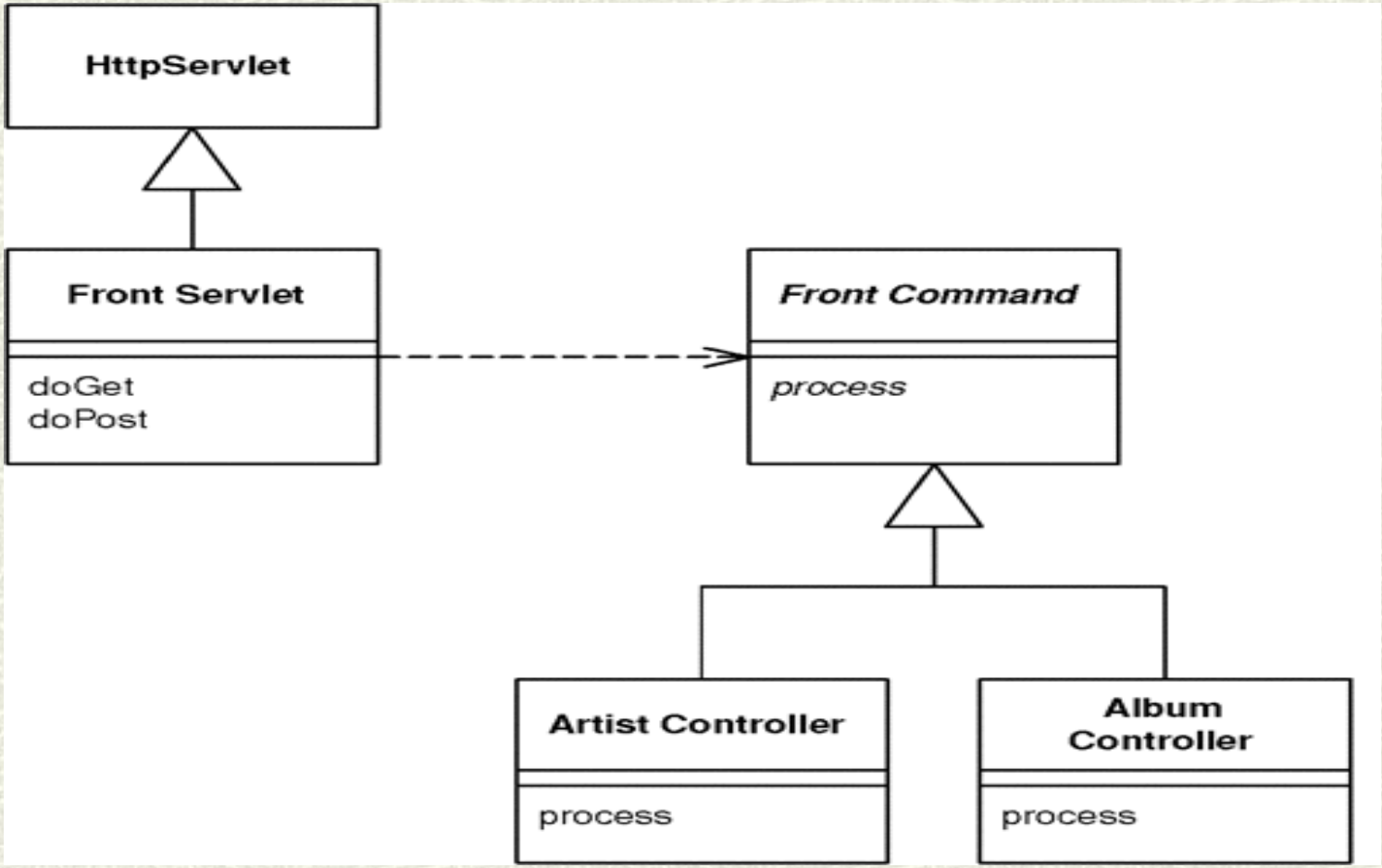
- The Web handler can decide which command to run either statically or dynamically. The static version involves parsing the URL and using conditional logic; the dynamic version usually involves taking a standard piece of the URL and using dynamic instantiation to create a command class.
- The static case has the advantage of explicit logic, compile time error checking on the dispatch, and lots of flexibility in the look of the 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. The properties file is another file to edit, but it does make it easier to change your class names without a lot of searching through your Web pages.

Front Controller

- Benefits:
  - Only one Front Controller has to be configured into the Web server; the Web handler does the rest of the dispatching. This simplifies the configuration of the Web server.
  - With dynamic commands you can add new commands without changing anything. They also ease porting since you only have to register the handler in a Web-server-specific way.
  - New command objects are created with each request, so you do not have to worry about making the command classes thread-safe.
  - The controller can easily be enhanced with new behavior at runtime using decorators. You can have decorators for authentication, character encoding, internationalization, and so forth, and add them using a configuration file or even while the server is running.

Front Controller - example

- http://localhost:8080/isa/music?name=IRIS&command=Artist



Front Controller - example

```
class FrontServlet... {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
      FrontCommand command = getCommand(request);
      command.init(getServletContext(), request, response);
      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;
   }
```

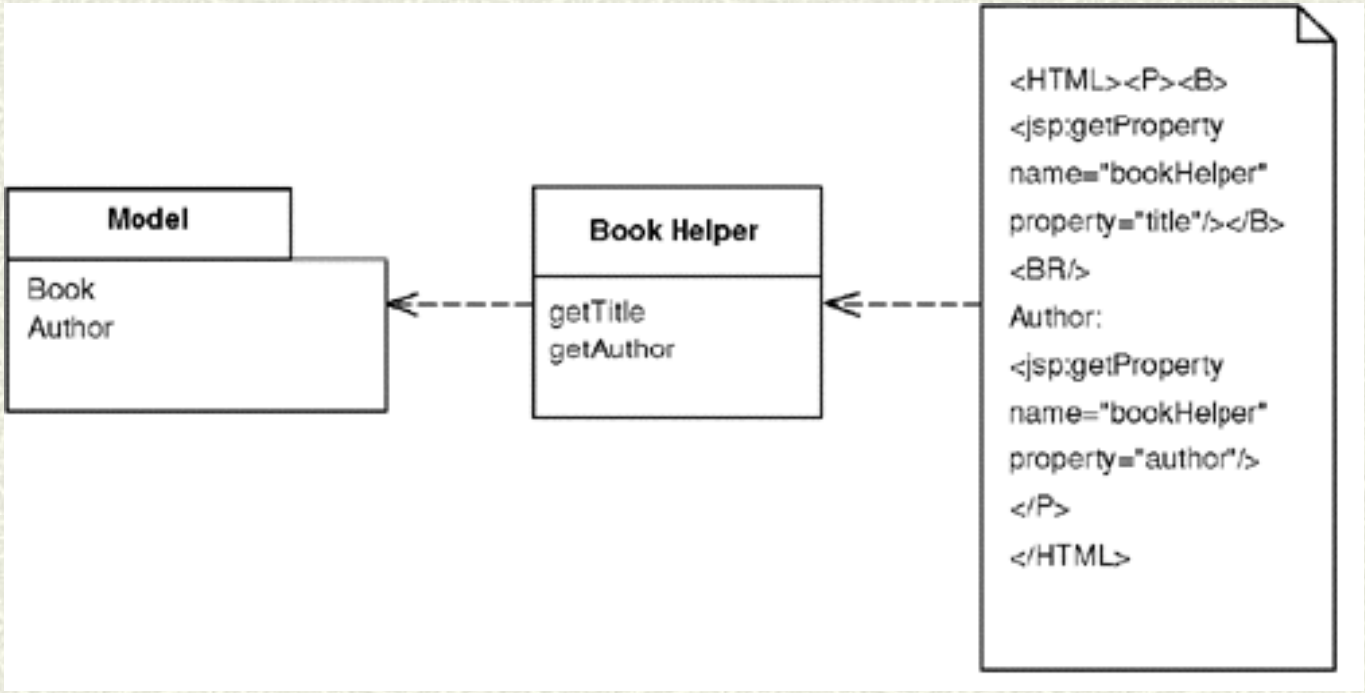
Front Controller - example

```
abstract class FrontCommand... {
   abstract public void process()throws ServletException, IOException ;
   protected void forward(String target) throws ServletException, IOException
   {
      RequestDispatcher dispatcher = context.getRequestDispatcher(target);
      dispatcher.forward(request, response);
   }}
class ArtistCommand... {
   public void process() throws ServletException, IOException {
      Artist artist = Artist.findNamed(request.getParameter("name"));
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp");
   }}
class UnknownCommand... {
   public void process() throws ServletException, IOException {
      forward("/unknown.jsp");
   }}
```

Template View

- Renders information into HTML by embedding markers in an HTML page.
- The basic idea is to embed markers (scriplets, tags) into a static HTML page when it is written.
- When the page is used to service a request, the markers are replaced by the results of some computation (i.e.,database query). In this way the page can be laid out (displayed) 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.
- The primary function of this pattern is to play the view in Model View Controller

Template View



Template View

Template View has two significant weaknesses:

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

Template View

```
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);
      }
   }
```

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>
```

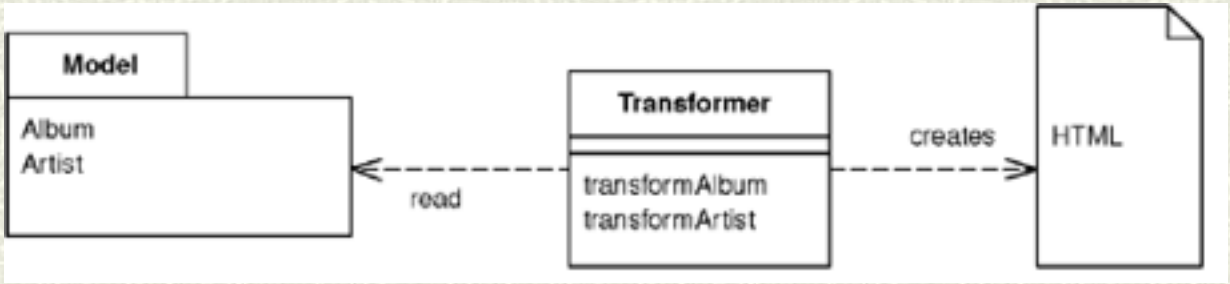
Template View

```
class ArtistHelper…{

   private Artist artist;
   public ArtistHelper(Artist artist) {
      this.artist = artist;
   }
   public String getName() {
      return artist.getName();
   }
}
//artist.jsp file

<B> <%=helper.getName()%></B>
or
<B><jsp:getProperty name="helper" property="name"/></B>
```

Transform View

- A view that processes domain data element by element and transforms it into HTML.
- The basic notion of Transform View is 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 View and Template View is the way in which the view is organized. A Template View is organized around the output. A Transform View is 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.
- You can write a Transform View in any language, however, the dominant choice is XSLT.

Transform View



Transform View - Example

```
class AlbumCommand... {
   public void process() {
      try {
       Album album = Album.findNamed(request.getParameter("name"));
       PrintWriter out = response.getWriter();
 XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
      out.print(processor.getTransformation(album.toXmlDocument()));
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
```

Transform View - Example

```
<album>
   <title>ABC</title>
   <artist>A A </artist>
   <trackList>
      <track><title>B B</title><time>1:37</time></track>
      <track><title>C C</title><time>2:24</time></track>
      <track><title>D D</title><time>1:23</time></track>
      <track><title>F F</title><time>3:01</time></track>
   </trackList>
</album>
```

Transform View - Example

```
<xsl:template match="album">
   <HTML><BODY bgcolor="white"> <xsl:apply-templates/>  </BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
   <h1><xsl:apply-templates/></h1> </xsl:template>
<xsl:template match="artist">
      <P><B>Artist: </B><xsl:apply-templates/></P> </xsl:template>
<xsl:template match="trackList">
   <table><xsl:apply-templates/></table> </xsl:template>
<xsl:template match="track"> <xsl:variable name="bgcolor">
      <xsl:choose>
         <xsl:when test="(position() mod 2) = 1">green</xsl:when>
         <xsl:otherwise>white</xsl:otherwise>
      </xsl:choose>    </xsl:variable>
   <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="track/title">
   <td><xsl:apply-templates/></td> </xsl:template>
<xsl:template match="track/time">
   <td><xsl:apply-templates/></td> </xsl:template>
```

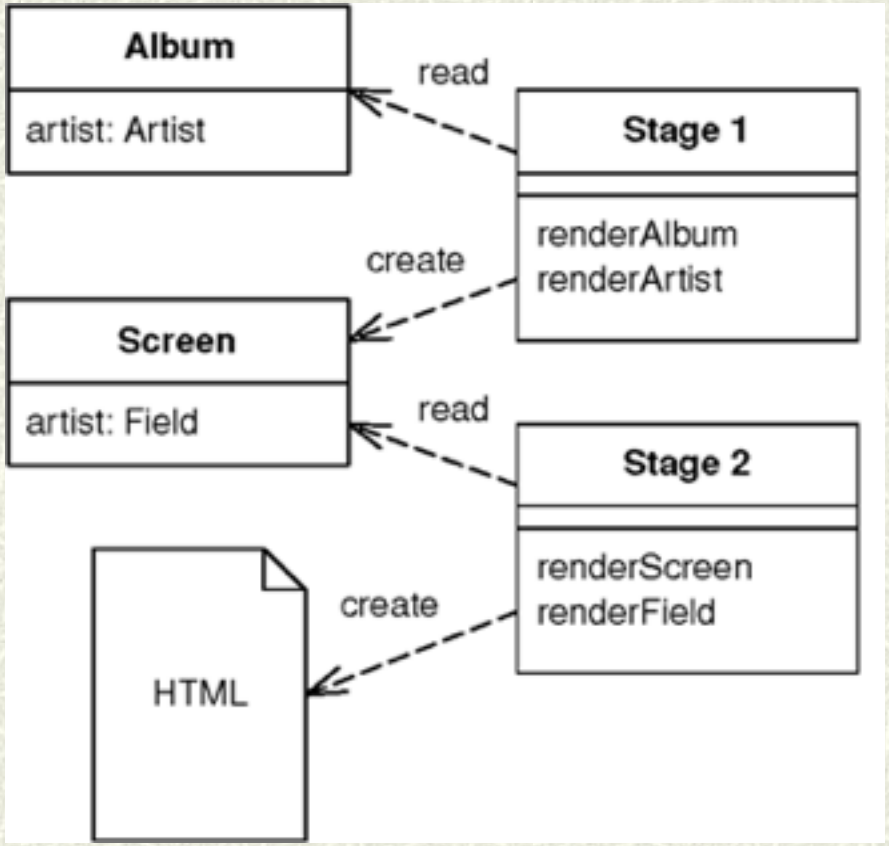
Two Step View

- Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.
- The key to this pattern is in making the transformation to HTML a two-stage process. The first stage assembles the information in a logical screen structure that is suggestive of the display elements yet contains no HTML. The second stage takes that presentation-oriented structure and renders it into HTML.
- This intermediate form is a kind of logical screen. Its elements might include things like fields, headers, footers, tables, choices, and the like. It is presentation-oriented and imposes the screens to follow a specific style.

Two Step View

- The first stage's responsibility is to access a domain-oriented model, either a database, an actual domain model, to extract the relevant information for that screen, and then to put that information into the presentation-oriented structure.
- The second stage turns the presentation-oriented structure into HTML. It knows about each element in the presentation-oriented structure and how to show it as HTML.

Two Step View

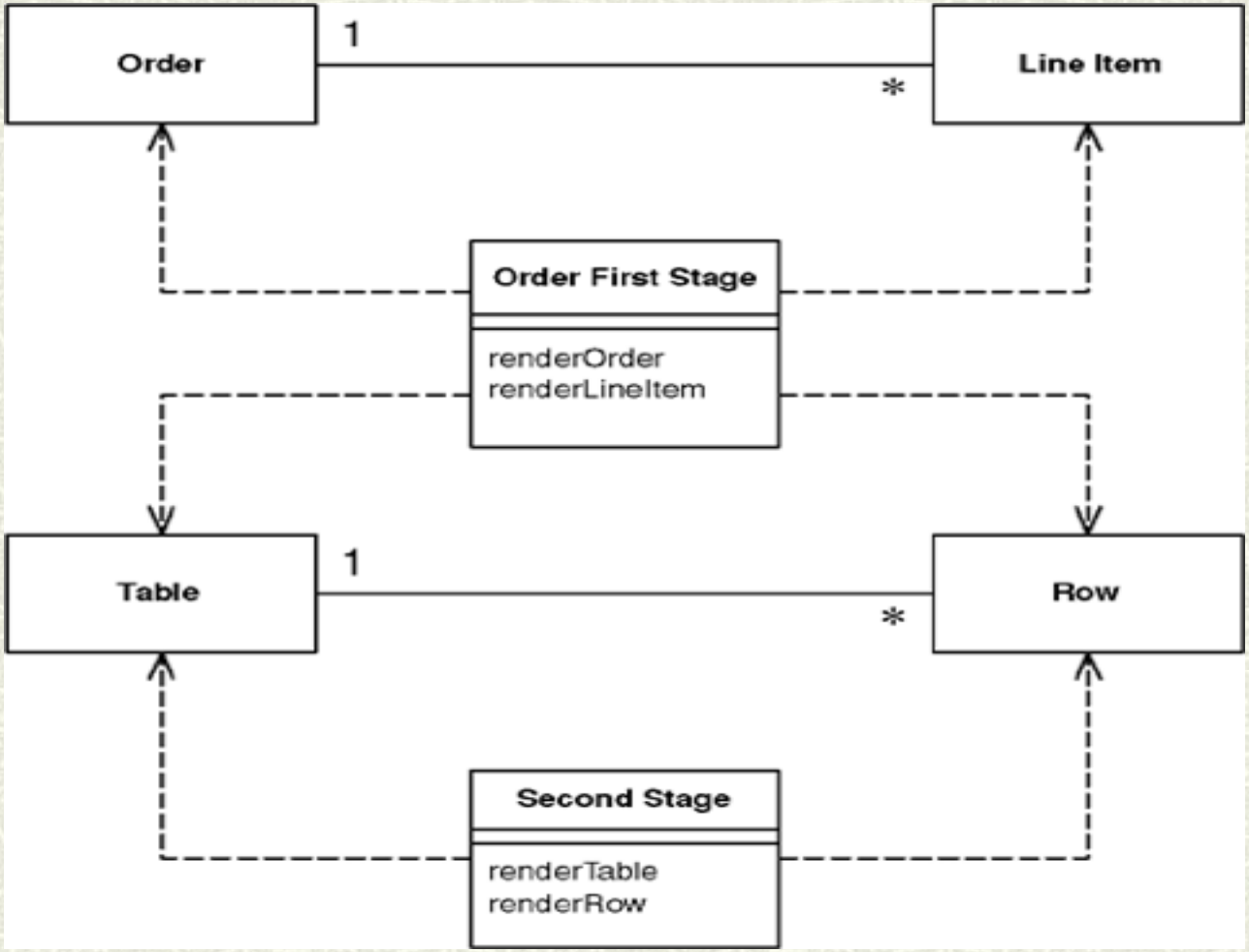


Two Step View

There are several ways to build a Two Step View:

- The easiest is with two-step XSLT. There are two XSLT style sheets. The first-stage style sheet transforms the domain-oriented XML into presentation-oriented XML, the second-stage style sheet renders that XML into HTML.
- Another way is to use classes. The developer defines the presentation-oriented structure as a set of classes: with a table class, a row class, etc. The first stage takes domain information and instantiates these classes into a structure that models a logical screen. The second stage renders the classes into HTML, either by getting each presentation-oriented class to generate HTML for itself or by having a separate HTML renderer class to do the job.

Two Step View



Two Step View

```
<album>
   <title>ABC</title>
   <artist>A A</artist>
   <trackList>
      <track><title>B B</title><time>1:39</time></track>
      <track><title>C C</title><time>2:30</time></track>
      <track><title>D D</title><time>3:00</time></track>
      <track><title>E E</title><time>3:05</time></track>
      <track><title>F F</title><time>2:50</time></track>
</trackList>
</album>
```

Two Step View

```
<screen>
   <title>ABC</title>
   <field label="Artist">A A</field>
   <table>
      <row><cell>B B</cell><cell>1:39</cell></row>
      <row><cell>C C</cell><cell>2:30</cell></row>
      <row><cell>D D</cell><cell>3:00</cell></row>
      <row><cell>E E</cell><cell>3:05</cell></row>
      <row><cell>F F</cell><cell>2:50</cell></row>

   </table>
</screen>
```

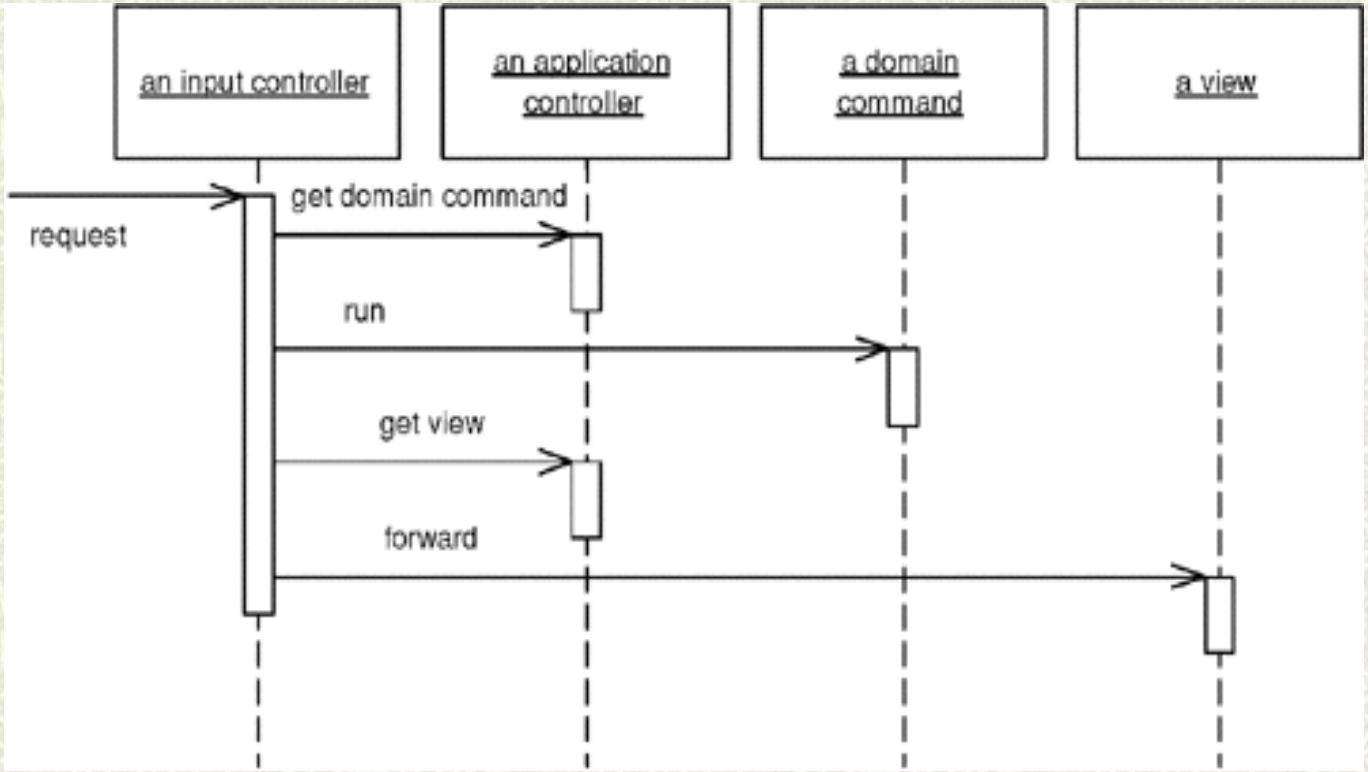
Two Step View

```
<xsl:template match="screen">
   <HTML><BODY bgcolor="white"> <xsl:apply-templates/>  </BODY></HTML>
</xsl:template>
<xsl:template match="title"> <h1><xsl:apply-templates/></h1>
</xsl:template><xsl:template match="field">
   <P><B><xsl:value-of select = "@label"/>: </B><xsl:apply-templates/></P>
</xsl:template>  <xsl:template match="table">
   <table><xsl:apply-templates/></table> </xsl:template>
<xsl:template match="table/row"> <xsl:variable name="bgcolor">
      <xsl:choose>
         <xsl:when test="(position() mod 2) = 1">green</xsl:when>
         <xsl:otherwise>white</xsl:otherwise>
      </xsl:choose>    </xsl:variable>
   <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr> </xsl:template>
<xsl:template match="table/row/cell">   <td><xsl:apply-templates/></td>
</xsl:template>
```

Application Controller

- A centralized point for handling screen navigation and the flow of an application.
- 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.
- If the flow and navigation of the application are simple enough so that anyone can visit any screen in pretty much any order Application Controller should not be used.
- The Application Controller should be used when there are definite rules about the order in which pages should be visited and different views depending on the state of objects.

Application Controller



Session State Patterns

- Client Session State
- Server Session State
- Database Session State

Client Session State

- Stores session state on the client.
- There are three common ways to do client session state: URL parameters, hidden fields, and cookies.
  - URL parameters are the easiest to work with for a small amount of data. Essentially all URLs on any response page take the session state as a parameter. The clear limit to doing this is that the size of an URL is limited, but if you only have a couple of data items it works well.
  - A hidden field is a field sent to the browser that is not displayed on the Web page. You get it with a tag of the form <INPUT type = "hidden">. To make a hidden field work you serialize your session state into it when you make a response and read it back in on each request. However, a hidden field is only hidden from the displayed page; anyone can look at the data by looking at the page source.
  - Cookies are sent back and forth automatically. Just like a hidden field you can use a cookie by serializing the session state into it.

Client Session State

- With just a few fields Client Session State works nicely. With large amounts of data the issues of where to store the data and the time cost of transferring everything with every request become prohibitive.

- Security issue. Any data sent to the client is vulnerable to being looked at and altered. Encryption is the only way to stop this, but encrypting and decrypting with each request are a performance burden. Without encryption you have to be sure you are not sending anything you would rather hide.

Server Session State

- Keeps the session state on a server system in a serialized form.
- In the simplest form of this pattern a session object is held in memory on an application server. You can have some kind of map in memory that holds these session objects keyed by a session ID; all the client needs to do is to give the session ID and the session object can be retrieved from the map to process the request.

- The great advantage of Server Session State is its simplicity. In a number of cases you do not have to do any programming at all to make this work. Whether you can get away with that depends on if you can get away with the in-memory implementation and, if not, how much help your application server platform gives you.

Database Session State

- Stores session data as committed data in the database.
- When a call goes out from the client to the server, the server object first pulls the data required for the request from the database. Then it does the work it needs to do and saves back to the database all the data required.
- In order to pull information from the database, the server object needs some information about the session, which requires at least a session ID number to be stored on the client.
- Database Session State is one alternative to handling session state.
- An important aspect to consider with this pattern is performance. In order to pull the data in and out of the database with each request time is needed. You can reduce this cost by caching the server object so you do not have to read the data out of the database whenever the cache is hit, but you still have over time for write operations.