Pattern Language for

Data Driven Presentation Layer for Dynamic and Configurable Web Systems

Sharad Acharya
s.acharya@computer.org

July 26, 2004

ABSTRACT

Data driven presentation layer is one of the alternatives for prevailing presentation layer architectures that can be adopted to creating dynamic Web systems for attaching to modern middleware in loosely coupled manner. Rendering and navigation aspects of a data driven presentation layer are driven by three sets of data. The first set is presentation layer metadata that defines a page structure such as field applicability, type, style, and so on. Other than exceptional situations, domain specific business rules of a Web application are enforced in the business layer. Applying such business rules and based on metadata, business layer provides second set of data to the presentation layer which has dedicated component to interpret such data and generate client specific pages at rendering time. Processing user submitted data; business layer returns third set of data that is used to decide where the control should be forwarded. In this paper, data driven presentation layer architecture is proposed and discussed in the form of Pattern Language that can be adopted in creating Web systems that are flexible, maintainable yet dynamically modifiable.

PROLOGUE

Today, Internet is undeniably the most widely used media for information presentation and retrieval. Because of its global scope, enterprises of all domain and all sizes want to make their computing assets (business process, enterprise data, etc) available over the Internet in most efficient way. Although need of making enterprise computing assets available over the Internet is alluring; it is challenging too. Heterogeneity of modern middleware systems; domain specific discrete business rules and several categories of enterprise users are some of the major factors that may make presentation layer architecture of enterprise Web applications fall short to meet expectations of extensibility, maintainability, and manageability requirements. In addition to fulfill existing requirements, presentation layer architecture of modern web applications should be resilient enough to evolve in meeting changing requirements throughout the application lifecycle.

This paper is about Pattern Language that proposes an alternative architecture that can be adopted to create presentation layer, the pages of which show different contents based on several factors. The patterns discussed are

those that I was able to observe as part of teams that put reasonable amount of time and effort in multiple projects and in different platforms but with the same goal—attaching dynamic and configurable Web interfaces to modern middleware systems. Here are some of the recurring problems in broader context that these patterns try to address in general.

- In addition to managing domain specific workflow, how to leverage capability of modern middleware systems to drive presentation layer with 'in context' data?
- How to manage navigation aspect of a Web system in well-defined manner, which is independent of main application logic?
- Can a Web page whose contents change based on some rules be built without using server side scripting?
- Can a page structure be externalized so that modification of layout, style, applicability, and other structural attributes are independent of application logic?

PATTERN LANGUAGE CATALOG

Figure 1 is catalog of patterns discussed in this pattern language along with some related patterns that may be included in future (future candidates.)

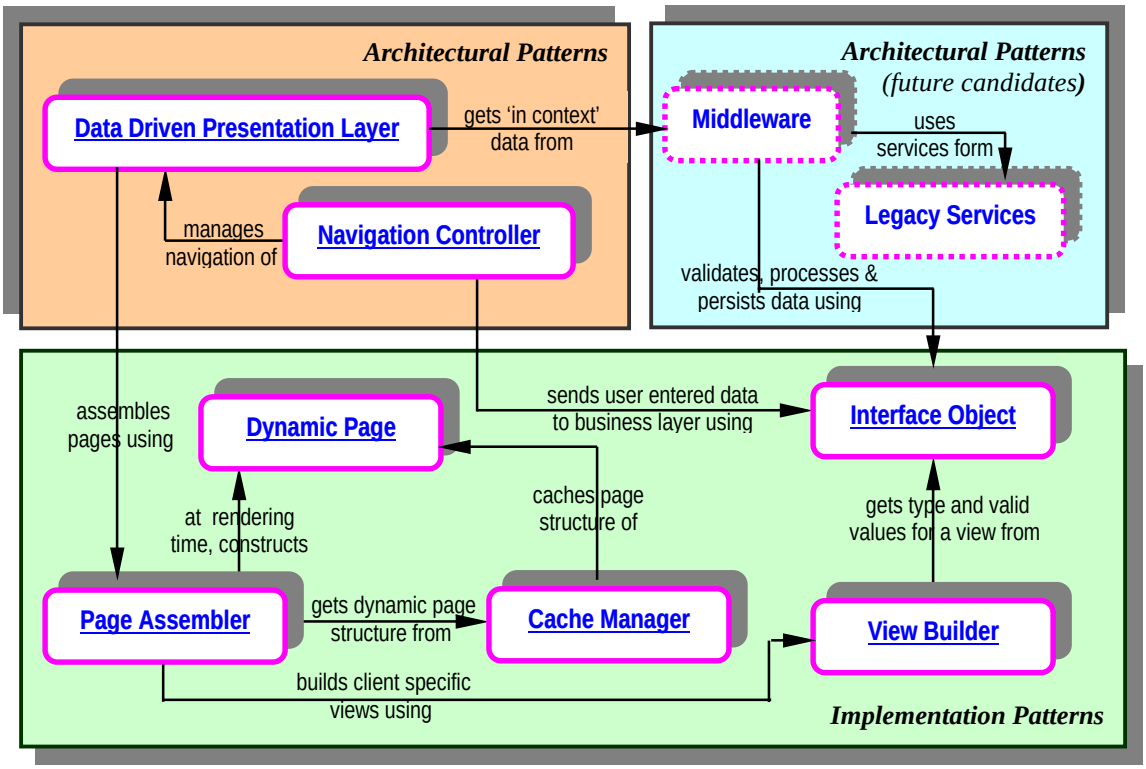


Figure 1: Patterns discussed in this pattern language and their relationships

To establish a relationship in figure 1, read the pattern name with originating arrow, text with the arrow and the pattern name with the arrowhead. Here is an example:

Data Driven Presentation Layer *assembles pages using* ***Page Assembler***.

The main objective of this pattern language is to make you able to select and implement presentation layer architecture for a Web application whose pages present dynamic contents. Hence, the patterns discussed in this paper are divided into two broad categories of Architectural and Implementation patterns.

Architectural Pattern

An Architectural Pattern provides solution that can be adopted for solving recurring problems in some broader context and how such solution fits in 'the bigger picture' of enterprise systems. Table 1 lists architectural patterns discussed in this pattern language. (Number in parenthesis next to pattern name indicates the page in which that particular pattern is discussed.)

Table 1: Architectural patterns, name and their purpose

Pattern Name	**Purpose**
Data Driven Presentation Layer(4)	Allows create a framework for generating pages dynamically based on data from some external source.
Navigation Controller(7)	Provides standard and decent way to manage navigation aspect of a dynamic web application.

Implementation Patterns

An Implementation Pattern provides solution to some specific problem that is part of page generation framework in a dynamic Web application. Table 2 lists implementation patterns discussed in this pattern language.

Table 2: Implementation patterns, name, and their purpose

Pattern Name	**Purpose**
Dynamic Page(14)	Provides a way to assemble client specific pages dynamically at rendering time
Interface Object(19)	Provides structure of a composite object used to send information between presentation and business layers.
Page Assembler(22)	Provides a way to assemble dynamic pages as an aggregation of client specific views.
View Builder(25)	Provides a way to create client specific views based on data available in some object.
Cache Manager(29)	Provides caching required by presentation layer so that data is available in the application level cache reducing database access, xml re-parsing or object reconstruction.

Note to PLoP' 04 Workshop: The patterns discussed follow order as they appear in this pattern language. Hence, Workshopping of this paper is expected in the sequential order.

DATA DRIVEN PRESENTATION LAYER

A Data Driven Presentation Layer pattern allows you to create a framework for generating pages dynamically based on data from some external source.

Context

You are creating presentation layer of a dynamic Web application that can be attached to the business layer in such a way that these two layers are loosely coupled, pages are completely unaware of domain specific business rules, and change management for the presentation layer is easy.

Problem

A dynamic page usually presents contents using server side scripting. Such a page displays different contents using one or more conditions that are usually part of business rules. This approach works for simpler pages in which the factors effecting dynamic behavior are none to very lesser in number. When such factors grow, a page may result in bulkier, more obscure, and lesser manageable scripts. How do you avoid using heavy scripting so that pages are manageable; presentation layer change management is easy; and it can take care of virtually unlimited number of current or future business rules?

Forces

A data driven presentation layer should balance some or all of the following forces.

- Domain specific business rules are usually applied in business layer. Presentation layer should minimize or avoid using intelligence of such rules to generate dynamic contents.
- Tight coupling between presentation and business layer components is undesirable.

Solution

Use presentation layer metadata to define page structure and data returned by business layer based on such metadata to construct a dynamic page.

To implement data driven architecture, presentation layer should be supplied with three sets of data--presentation layer metadata, data returned from business based on such metadata and status supplied by business layer based on user submitting a page. Presentation layer metadata defines views applicable to a page and fields applicable to a view including page layout, field style, field applicability, etc. Presentation layer metadata is usually defined in one or more database tables, read and cached usually at application startup time. Based on metadata, business layer should apply domain specific business rules and return appropriate data to the presentation layer, which should construct a page based on such data to present the user. When user submits a page, business layer processes user-supplied data in current page and returns result to presentation layer based on which the control is transferred to appropriate page.

Structure

Figure 2 shows some architecturally significant elements of data driven presentation layer architecture. Although shown, Legacy System is out of scope of this version of the paper.

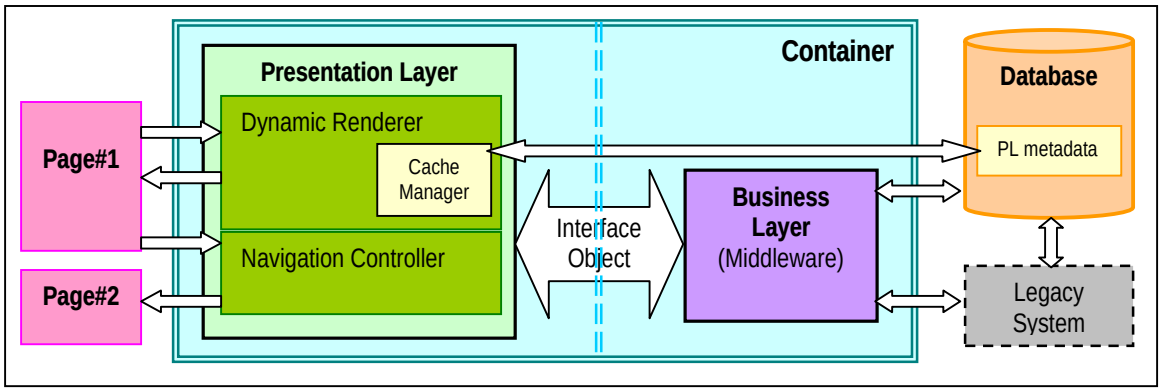


Figure 2: Architecturally significant elements of an enterprise Web system

Dynamics

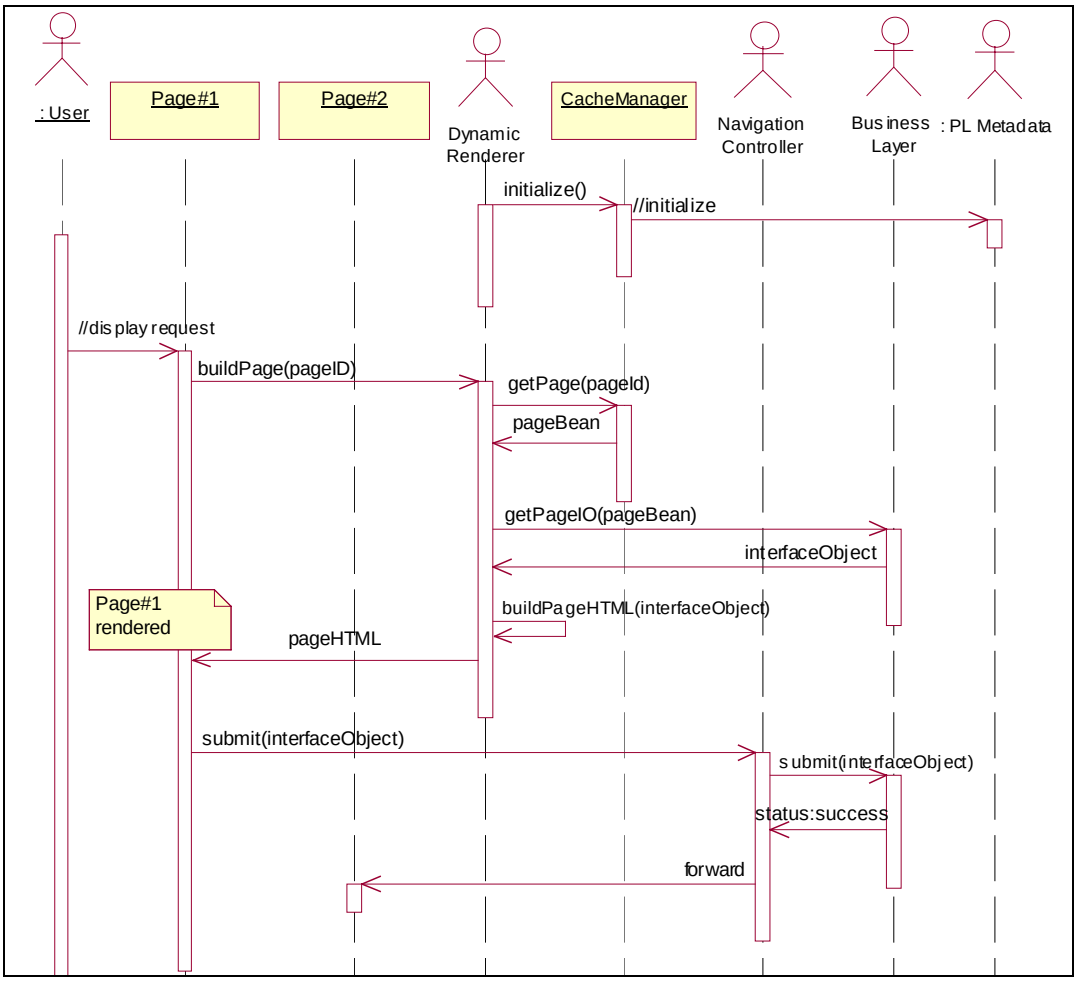


Figure 3: HTML page generation sequences for data driven presentation layer

Participants and Responsibilities

Page #1/ Page #2

A page is used by user to send request to the system and by system to present its response back to the user. A page for dynamic web applications is implemented using Java Server Page on J2EE applications, Active Server Page on .NET applications and so on. In data driven approach, a page is dynamically constructed using data returned by business layer.

Dynamic Renderer

This is the main component of data driven architecture. The main task of this component is to supply client specific dynamic content to the requesting page. To achieve this, this component retrieves page structure from cache and calls appropriate business layer method. The business layer call returns an instance of InterfaceObject (19) based on which this component constructs client specific dynamic contents. For a Web client, it generates HTML text representing the page and writes to client's browser window.

Cache Manager

Provides caching of frequently used information from appropriate sources such as database, XML files, and so on.

Business Layer

Business layer drives presentation layer by supplying data related to current page's context.

Navigation Controller (7)

This component manages navigation aspects of the application.

PL Metadata

Holds page, view, and field specific data and their relationships usually in a database table.

Consequences

This approach has following benefits.

- Change management is easy. Page structure can be altered changing presentation layer metadata, no need to change application code.
- Since pages are constructed based on data that results by applying domain specific business rules, no scripting needed for achieving dynamic behavior. It results in better maintainable pages.
- Loose coupling between business and presentation layers. Replacement of presentation layer is easy.

This approach has following liabilities.

- Relatively higher response time compared to scripting. Appropriate Cache Manager (29) should be provided for in order to cache repetitively used data to reduce response time.
- Complexity. Page Renderer component is added to the system.
- Difficult to decide the portion of a page to be generated dynamically or implement using scripting. The later can be better choice for some portion of a page.

NAVIGATION CONTROLLER

Navigation Controller pattern provides standard and descent way to manage navigation aspects of a dynamic Web application.

Example

Role based resources access needs to authenticate a user and authorize him according to his role. In a typical site navigation scenario, different page is presented to a user based on his role. Figure 4 shows an activity diagram for first few pages of such an authentication process. Initially, the user is presented with Login page allowing entering his user id and password before submitting the page. If system finds user id, then he is authenticated against the supplied password. In case of successful authentication, the user is presented with Catalog page if his role is customer, Catalog administration page if his role is site administrator. In case if user id does not exist, he is presented with Registration page that allows him to register before he can browse a Catalog page. For all other cases, the user is informed with appropriate message back in the Login page.

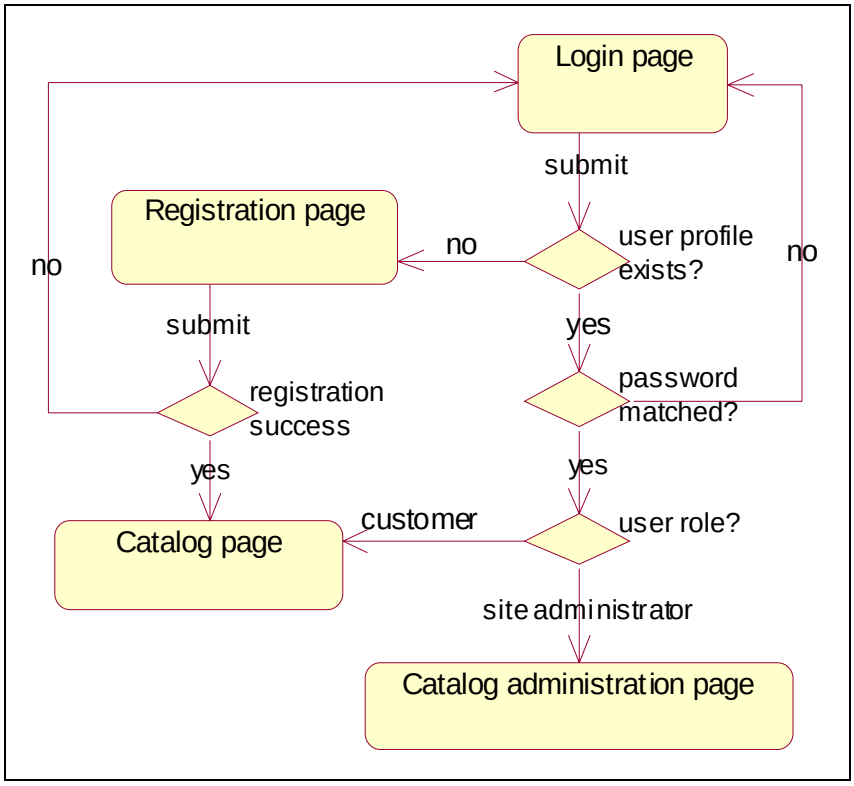


Figure 4:Page navigation activity diagram for user authentication

Context

Your Web application has complex and variable navigation aspects, which you want to manage in standard and decent way.

Problem

Configuration management is an important non-functional requirement of large-scale Web systems. It should allow alteration of navigation sequence between pages that arise because of several factors such as different user categories, different type of business processes, or some future business rules that are yet to be identified. Moreover, change in post-deployment navigation sequences should not require code rework. How do you manage navigation aspect of Web application that can handle such requirements?

Forces

Navigation controller pattern should balance some or all of the following forces.

- Although it is possible to keep navigation logic within the same page, more complex navigation aspects result in bulky and hard to mange pages.
- If navigation aspect of a Web application can be externalized, it results in flexibility to add, remove, and alter navigation sequences of the application at any point of application lifecycle, if required.
- In general, every legitimate user generated event should result in change of contents at the same or different page. In data driven approach, since rendering of a page is driven by business layer data, it is necessary to make some methods call in business layer. There should be some standard place at presentation layer that will act as 'plug-point'.

Solution

Define an action for every form in a configuration element and attach input pages, forward pages, and other required elements to this action.

Before a request is processed, allow ActionController read the configuration element, instantiate all required classes, and cache required objects for managing navigation of the application. When an HTTP request is submitted because of some user-generated event in a page, let the controller intercept, inspect, interpret the request; do current action specific processing and forward control to appropriate page based on the return value.

Structure

Figure 5 shows some required components of navigation controller architecture of a typical web application.

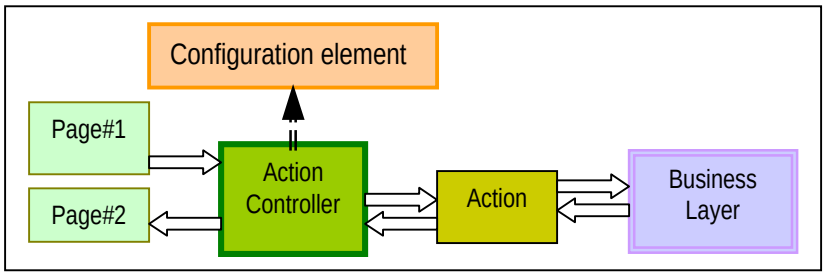
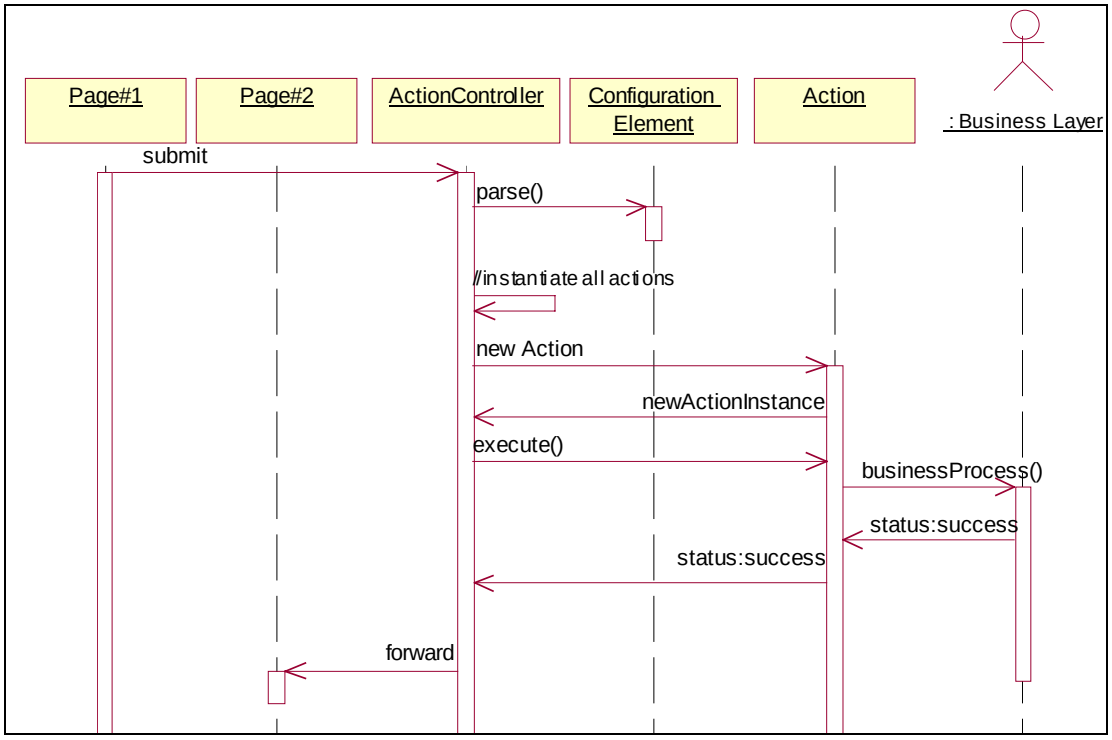


Figure 5: A typical navigation controller structure

Navigation Controller Dynamics



Page 6: Navigation controller dynamics

Participants and Responsibilities

The participants shown in navigation controller structure are described here.

Page #1/ Page #2

A page collects user input, submits HTTP request to the system, and presents system response to the user.

Action Controller

An Action Controller is the heart of this pattern. It should orchestrate request processing and response forwarding on behalf of the current request. Before serving any request, it parses the configuration element, instantiates action objects, and caches such objects for future reuse. It keeps reference to all user generated events that is meaningful to this application. In other words, for a given user generated event, it knows which method of which action class to execute and where to send a response based on return value of such call.

Action

An action is ‘unit of work’ consisting of several intermediate events to fulfill a request. It starts by user initiating some legitimate event such as pressing a button in a Web page. Presentation layer should process the request by populating required objects, communicating with appropriate business layer methods, and forwarding response based on return value of such business method call. For this reason, it can be considered as sub-layer within presentation layer that serves as ‘plug point’ to the business layer.

Configuration Element

A Configuration Element is place for defining action mappings. It consists of mapping for every page's user generated events that the application supposed to respond and process. It also specifies the forward pages where the control should be forwarded in case of success and some different page in case of error. Modern web applications usually define such configurations in XML file.

Business Layer

Business layer usually consists of application logic that implements domain specific business rules. Applying such rules, this layer gives presentation layer an indication in the form of return values where the response should be forwarded.

Consequences

Benefits

- Externalized configuration. This results in navigation aspects independent of application code. Altering navigation sequence and similar aspects at any point of application lifecycle can be done without touching the application code.
- Because it is a separate element, individuals who know domain specific business rules can write such configuration element, which has better accuracy.
- Separation of responsibilities among participating entities.
- Scalability.

Liabilities

- Complexity.

Implementing custom navigation controller as framework is difficult and time consuming. Instead of 're inventing the wheel' by writing your own framework, one of the already available frameworks can be adopted to manage navigation aspect of a Web application. One relatively newer of such frameworks is Jakarta Struts [7]. It is one of the most popular and hence widely used web development frameworks. In following sections this framework is discussed in brief and how it can be adopted to manage navigation aspect of a Web application.

Jakarta Struts framework

Jakarta Struts framework follows Model View Controller (MVC) pattern architecture. In Struts, ActionForm or its sub class represents model element, the instance of which is used by framework to populate HTML form data on page submission. There is usually one-to-one correspondence of an action and a HTML form. A Java Server Page represents view element that uses HTML form to post a HTTP request on submission of such form. An ActionServlet represents a controller component, which can directly be used or sub-classed to add application specific behavior. The ActionController shown in Figure 5 is supposed to provide responsibility of the controller. In Struts, it reads the configuration element, instantiates view forms and attaches such forms to action, and calls

specified method in the instance of action class every time the form is submitted. Following section outlines 4 main steps to implement navigation controller using Struts framework for the example discussed in Example section of this pattern (Figure 4, page 7).

Step By Step Implementation Example

Step 1: Define form bean and action mapping in configuration element. It is standard practice to use framework provided name of such configuration file, which is named as struts-config.xml. Code example 1 shows portion of such configuration file.

```
<form-beans>
  <form-bean name="loginForm"
          type="forms.LoginForm"/>
<!--… … other form-beans definition… … …-->
</form-beans>

<!--… … … other elements… … …-->

<action-mappings>
  <action path="/login"
          name="loginForm"
          type="actions.LoginAction"
          input="login.jsp">
      <forward name="success_admin" path="catalogadmin.jsp"/>
      <forward name="success_user" path="catalog.jsp"/>
      <forward name="no_profile" path="registration.jsp"/>
  </action>
<!--… … other action mappings … … …-->
</action-mappings>
```

Code example 1: Struts form bean and action definition

This action mapping shows two important sections in configuration file. The 'form-beans' section defines a form-bean name and fully qualified class that should be instantiated and assigned to that name. The 'action-mappings' section defines an action with a path, name, type, an input, and one or more forward names for this action. This instructs the action servlet attach loginForm to 'login' action that can be invoked by '/login.do' as URL. When invoked, the servlet calls execute() method in LoginAction instance and based on the return value of which the control will be forwarded to one of the pages specified by forward name. A return value of success_admin will forward to catlogadmin.jsp and vice versa. The framework forwards control to the page specified by input attribute in case of error.

Step 2: Define Action Form class.

A specific action form is subclass of ActionForm provided by the framework. In most general case, it defines properties current form is supposed to handle. For framework to be able to populate from request parameters, the form should have matching properties that are exposed by getter and setter methods. When the HTML form attached with this action form is submitted, matching form attributes with the HTML input parameter name are automatically populated.

```
/**
* all import statements go here
*/

public class LoginForm extends ActionForm {
   private String password;
   private String userid;
   public String getPassword() {return password;}
   public String getUserid() {return userid; }
   public void setPassword(String password) {this.password = password;}
   public void setUserid(String userid) { this.userid = userid;}
}
```

Code example 2: Login action form

Step 3: Define a Java Server Page that uses the form and action.

Code example 3 shows a Java Server Page that shows two text fields for user to enter his userid and password. This uses struts provided html tag that allows define Struts form and display html elements. It is very interesting to see how much work the framework does behind the scene for attaching a form to this JSP. The `<html:form action="/login.do">` is usual syntax to define form. When such tag is processed, framework makes sure that the form is instantiated and attached to the action specified. This JSP also displays two text boxes fields-- userid and password. Since it uses 'login.do' as current action, which is attached with loginForm, any initialized attribute in action form is available in JSP. Any changes in JSP fields are also populated when the form is submitted.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
  <HEAD>
    <%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
    <TITLE>login.jsp</TITLE>
  </HEAD>
  <BODY>
     <html:form action="/login.do">
          <TABLE border="1">
               <TR>
                 <TD>User Id</TD>
                 <TD><html:text property="userid"/> </TD>
              </TR>
             <TR>
                 <TD>Password</TD>
                 <TD><html:password property="password" /> </TD>
             </TR>
        </TABLE>
    </html:form>
  </BODY>
</HTML>
```

Code example 3: Login JSP using login action.

Step 4: Define LoginAction class that is invoked by the framework.

LoginAction class is shown in code example 4. The execute (…) method of this class is called by the framework (by controller to be specific) when 'login.do' action is invoked. This is the most appropriate place to make business layer calls because the page to forward is decided by controller based on the return value of this method. This also is the last point where application related data may be manipulated before the framework takes control. In the example discussed, the return value of business layer call, authenticateUser (userid, password) is used by the controller, to forward one of the pages defined in forward section of configuration file. In this scenario,

business layer will return appropriate forward string based on user role. Navigation control is not usually manipulated in action classes. In case if there is new forward name to be added because of some business rules, then it should be added in struts-config.xml file. There is no need to change presentation layer code because business layer provides the forward string.

```
/**
* all import statements go here
*/

public class LoginAction extends Action {
    public ActionForward execute( ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception
    {
       LoginForm loginForm = (LoginForm) form;
       String userID = loginForm.getUserid();
       String password = loginForm.getPassword();
       //get business layer interface instance by remote
       //lookup to business layer interface (lookup code not shown here)
       String loginStatus = blInterface.authenticateUser(userID, password);
       return mapping.findForward(loginStatus);
     }
}
```

Code example 4: Login action

See Also

MVC [6,9]

MVC is short for Model, View, and Controller. A Model component encapsulates application state, notifies views about changes, responds to state query and exposes application functionality. In general, it is the application data and business logic operating in those data. A View component renders model, sends user gesture to controller and allows controller to select views. A Controller component defines application behavior, maps user actions to model updates and selects view to response. Navigation Controller pattern is based on the MVC. Any page in this pattern corresponds to view component; ActionController corresponds to controller component and action form (LoginForm is one such form) correspond to model component.

Command [1], Command Processor [9]

Command pattern allows encapsulate service request into objects. Command Processor pattern builds upon the Command pattern and provides more details in handling command objects. Navigation controller builds upon this idea by encapsulating Action classes as command objects. An action class is a command whose execute() method is called by the controller based on mapping defined in configuration element. In previous example, ActionController invokes LoginAction.execute() method as part of login action making LoginAction object as command object.

DYNAMIC PAGE

Dynamic page pattern provides a way to assemble client specific pages dynamically at rendering time.

Example

There are several scenarios, in which the same page should be able to present different contents or same contents differently. Let us consider how same contents in a page should be presented differently for different user roles.

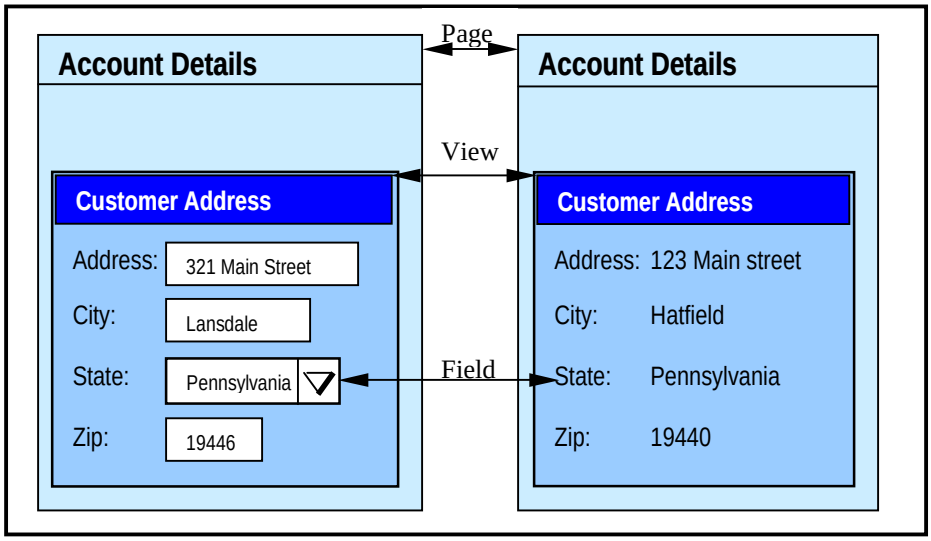


Figure 7a:'Type A' user's view of a page *Figure 7b: 'Type B' user's view of a page*

Let us assume that 'Type A' and 'Type B' are two user roles are. Also assume there is a hypothetical business rule-- user 'Type A' should be able to add or modify customer address whereas user 'Type B' can only view the existing data. This leads to the fact that for 'Type A' user role, the fields should be shown editable (Figure 7a) whereas for user 'Type B' the same fields should be shown read only (Figure 7b).

Context

You are designing presentation layer of a Web application in which same page may present different contents based on domain specific business rules.

Problem

Layout, style, applicability, and other structural information of Web pages may change because of domain specific business rules. Server side scripting with or without client side scripting can be used to present dynamic contents in such pages. In this approach, however, as the complexity grow, the amount of scripting may grow to such extend that may result in less maintainable pages. How to minimize or even avoid scripting so that pages that present dynamic contents become better maintainable?

Forces

A dynamic page should balance some or all of the following forces.

- Server side scripting is one of the most widely used methods to achieve dynamic behavior of a page. As factors affecting dynamic behavior increase, scripting may soon result in code that is difficult to read and manage.
- Layout, style, applicability, and other structural information of a Web page may change because of domain specific business rules. Externalization of page structure makes it easy for change management.
- Client side scripts can also be used to make a page behave like dynamic by showing only those fields that are applicable to current context and hiding all other fields. This approach may result in pages that may produce different results based on client's environment settings. An application, for example, may use JavaScript to open a popup as child to a main window to show some additional information. If a user has installed 'popup blocker' software in his browser to block unwanted pop-ups. It turns out that the application would not behave as expected since the popup cannot be opened.

Solution

Assemble pages dynamically at rendering time that contain only required fields with appropriate style and valid values applicable to current context.

In dynamic rendering approach, a page may not rely on scripting for dynamic behavior that may be affected by multiple factors. When a rendering request is received from a page, presentation layer should send current page structure to the business layer using a common object (a PageBean instance). Business layer applies business rules and populates the page structure with data pertaining to the current page. The presentation layer receives an instance of Interface Object (19), which is used by Page Assembler (22) to generate client specific page.

Further, in a Web page, fields tend to fall in groups. For scope of this pattern, we will use a term called 'view' to represent logical unit that presents fields related to each other to form some meaningful entity. Example in Figure 7 shows a 'Customer Address' view in 'Account Details' page. A dynamic page should be constructed as aggregation of views instead of aggregation of fields. This promotes the reusability in coarser level of granularity.

Structure

Figure 8 shows UML representation of classes for a typical dynamic page structure.

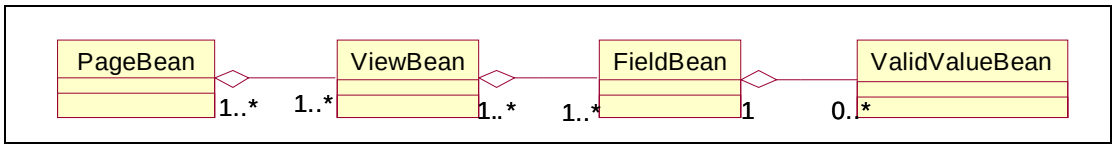


Figure 8: Participating classes for a dynamic page structure

Participants and Responsibilities

PageBean

A PageBean is a class an instance of which should be able to completely describe structure of a dynamically generated area of a page. It holds a list of views applicable to the current page for current context. In Figure 7, the main content area corresponds to an instance of a PageBean. Although this example shows only one view, the actual number of views applicable to a page is defined in presentation layer metadata.

ViewBean

An instance of ViewBean should hold necessary and sufficient information to generate a view within the containing page. It holds a list of field beans applicable to the current view. A view structure is also defined at presentation layer metadata.

FieleBean

An instance of FieldBean holds information about a field that is to be displayed within a view. In general, this represents one row in a view as drawn in page. A FieldBean instance should hold complete information required to construct appropriate field type, style, valid values, and so on. An instance of 'state' field bean in figure 7a, holds three values--data for field label which is 'State'; default selection which is 'Pennsylvania'; and valid values to display in drop down which is list of applicable states. In figure 7b, however, field bean for state should contain two label values-- 'State' label and actual state value 'Pennsylvania'. Business layer populates these values based on the business rule (user role in this case), presentation layer need not consider the user role to display the field properly.

ValidValueBean

An instance of this class represents a valid value in a field that shows it.

Code Example

A typical semi pseudo code for Java Server Page element that represents page in figure 7 may look like this.

```
<table>
    <tr><td span=2><include top.jsp></td></tr>
    <tr>
        <td><include teftnav.jsp></td>
        <td><include main.jsp></td>
    </tr>
    <tr><td span=2><include bottom.jsp></td></tr>
</table>
```

Code example 5: Server side script for a page

All server side scripting elements may be implemented as dynamic page but this may not always be required. The server pages representing top, bottom, and left in Code example 5 (not shown in example), will not have substantial amount of dynamic behavior because, the structure are usually same, except a few things. Hence, such elements are better candidates for implementing using scripting. The server page representing main content area is the one whose contents are expected to change substantially. Code snippet in example 6 is a scripting element (Java Server Page) that is used to render such a page dynamically.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<!—taglib declarations -- >
<!—page level imports -- >
<TITLE>AccountDetails.jsp</TITLE>
</HEAD>
  <BODY>
  <!  … … … Start table and row tags … … -- >
  <!-- build this screen dynamically  -->
        <%
        PageAssembler.getInstance().buildPageHTML(
           request,
           pageContext,
           GlobalConstants.PAGE_ID_ACCOUNT_DETAILS);
        %>
  <!  … … … End row/ end table tags … … -- >
  </BODY>
</HTML>
```

Code example 6: Main.jsp building page contents dynamically

One of the most important features of this JSP is that it neither has to declare any scripting element nor multiple scriplet to control display logic. The rendering is completely controlled by single call to Page Assembler (22) component. The return value of this method is the actual text representing current dynamic page.

Consequences

Benefits

- Reduced number of presentation elements. If chosen to do so, only one JSP is sufficient to present the contents for whole application.
- No scripting to generating dynamic contents. Pages not use any business rules for generating dynamic content.
- Maintainable pages. Writing maintainable dynamic pages has always been a daunting task for page authors because they tend mix markup language tags, server side scripting elements and scriplets. This leads to the fact that serve side elements becomes less maintainable. Code example 6 shows (JSP) which makes only a single call to Page Assembler component that creates necessary HTML text to display this page appropriately. This scripting element (JSP) implementation is ‘thin’, clean, and completely unaware of what contents it will receive to present because this is driven by business layer data. Instead, Code example 7 shows how scriplet and HTML tags can mix up that results in JSP code difficult to understand and manage.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML> <HEAD>
<!—taglib declarations -- >
<!—page level imports -- >
<TITLE> AccountDetails.jsp</TITLE>
</HEAD>
  <BODY>
  <!  … … … Start table and row tags … … -- >
  <!-- build this screen using scriplet -->
       <% if (account.getUser().getUserRole.equals(“TypeA”)){%>
            < ! — display state as list -- >
            <TD>State : </TD><TD><SELECT name=”states”>
            <% //for loop for applicable states list go here %>
           <OPTION value=”” >
         </SELECT></TD>
      <%} else{%>
            < ! — display state as label-- >
     <%}%>
  <!  … … … more field display logic … … -- >
  <!  … … … end row/ end table tags … … -- >
  </BODY>
</HTML>
```

Code example 7: Building state list using scriplet to take account of user role

When we compare code example 6 with 7, example 6 is, of course better in terms of maintainability.

Liabilities

- Response time**.** Since string manipulation is considered relatively heavy operation, there can be reasonable amount of performance penalty unless suitable caching mechanism is implemented. Although this approach involves heavy string manipulation, some scenarios urge to refrain from adopting server side scripting and adopt this approach. If your application will be used by range of users to perform several business processes, dynamic rendering approach is definitely a better choice.

See Also

Two Step View [9]

Two Step View [8] is one of such patterns that can be used to generate pages dynamically. This pattern proposes to generate HTML pages in two steps. At the first step, domain data is transferred into some kind of logical page, usually a XML document. At the next step, logical page from the first step is transferred to HTML by applying XSL transformation. Although Two Step View and Dynamic Page approaches have similar goal—present dynamic content based on some data, there are some fundamental differences. Two Step View mainly depends on transformation, creating intermediate views and transforming such views to target final view such as HTML or PDF. A Dynamic Page approach however depends only on the object returned from business layer to generate a client specific view. No intermediate transformation is necessary.

INTERFACE OBJECT

Instance of an interface object provides a common object structure to send information between presentation and business layers.

Example

Communication between presentation and business layers usually happens via remote method calls over the network. In data driven architecture, presentation layer assembles client specific pages at rendering time based on data provided by business layer. Presentation layer makes calls to business layer, which usually exposes the 'services' via interface methods. Because of extend of data transfer between two layers, the network traffic should be minimized by providing a coarse grained object structure that should be able to hold necessary and sufficient information for presentation layer to construct a page. When user submits a page, the object should hold the newly entered data by the user for business layer for further processing. Figure 9 shows a schematic representation of communication between presentation and business layers.

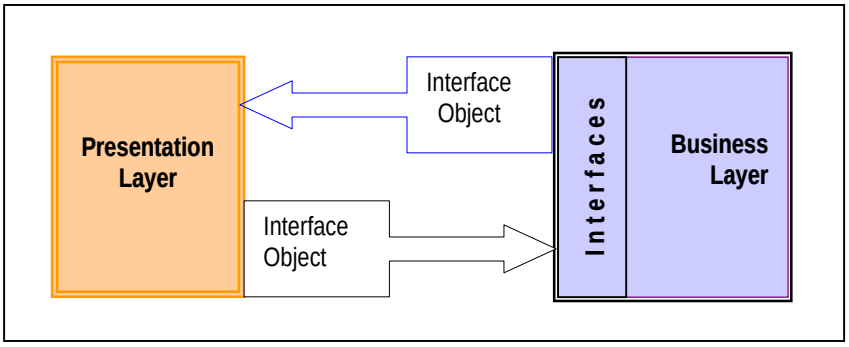


Figure 9: Interface Object usage scenario

Context

You have decided to adopt data driven architecture for your presentation layer to generate pages dynamically. You need to have a common object that can be interpreted by presentation layer to completely generate a page and by business layer for further processing.

Problem

It is necessary for presentation layer to completely understand data returned from business layer and vice versa. This leads to the fact that a generic object is required to share information by both presentation and business layers. Presentation layer uses this object to generate pages dynamically. Business layer uses this object to obtain data submitted from user for further processing. What kind of object is suitable to serve the both purpose?

Forces

An interface object should balance some or all of the following forces.

- Modern web applications are heterogeneous, composed of several components running across the network. In data driven approach, presentation layer depends on business layer for virtually every piece of information. Presentation and business layer components are usually part of network instead of same physical machine. A 'coarser grained' object is required to reduce network traffic that presentation and business layers can share so that sending and retrieving information can be completed in single call.
- Business layer can operate on HTML form values available, as part of HTTP request. This approach is not recommended because it is easy to compromise. If name of a field is changed in presentation layer, the business layer needs to be changed as well or it does not get appropriate value. Business layer should not depend on HTTP request object for further processing.

Solution

Define a common object and let business and presentation layers communicate using this object to send required information back and forth between these layers.

An Interface Object is a coarse grained class composed of several other classes. The object of this class holds required data structure and value objects based on which pages are constructed dynamically. This object should also be updated to reflect data changed by a user so that business layer can take appropriate action.

Structure

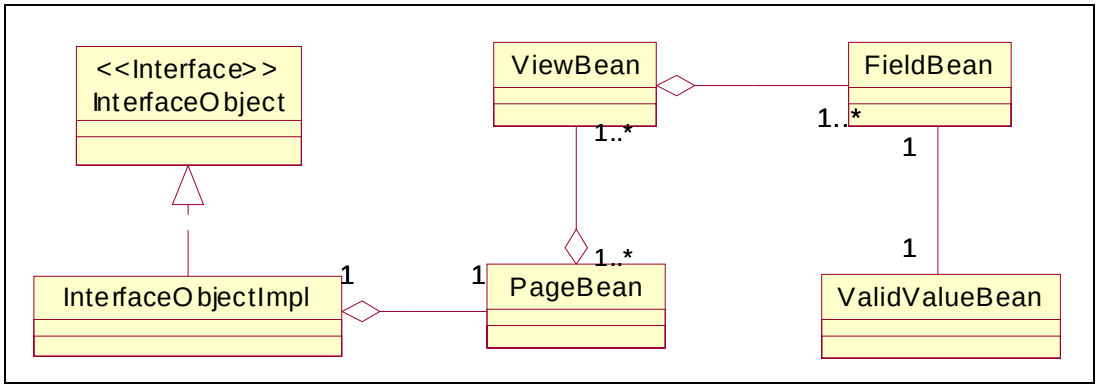


Figure 10: Interface Object classes

Participants and Responsibilities

InterfaceObject

This interface defines InterfaceObject object type and exposes common operations in such object instances.

InterfaceObjectImpl

This class is actual implementation of the interface object. This object holds, in addition to others, an updated PageBean instance at any point of time.

PageBean

A PageBean holds list of ViewBeans applicable for this page for any given context.

ViewBean

A ViewBean holds information to completely render a view in a page that displays it. It holds one or more FieleBeans applicable to this view.

FieldBean

A FieldBean represents a particular field to be displayed within a view. It should define all required properties to render it properly, which includes type of (this) field, size, style, as well as list of valid values, if applicable.

ValidValueBean

A ValidValueBean represents a valid value that is applicable to the associated field. For example, a valid value bean for a state field of a US address may have a state's display name and actual name that is used to submit as part of HTTP request parameter.

Consequences

Benefits

- Single, sharable object that can be used by both presentation and business layers
- With coarser grained object structure, information can be retrieve in single method invocation, hence reducing network traffic between presentation and business layers.

Liabilities

- Complex structure of Interface Object makes relatively difficult to manage its lifecycle.

PAGE ASSEMBLER

Page Assembler pattern allows generating a dynamic page by assembling client specific views.

Example

In data driven approach, a page is constructed at rendering time. There will only be place holder construct, the contents for such page are supplied at the rendering time. A dedicated component is needed for delivering dynamic contents to a requesting page based on current context. Figure 11 shows a schematic representation of dynamic page assembling activity.

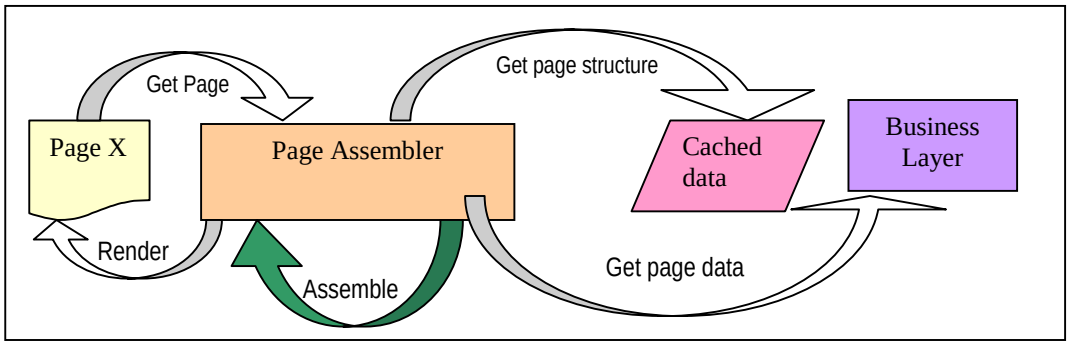


Figure 11: Page assembling activity example

Context

You have decided to use Dynamic Page (14) approach to construct a page. You need a component that has dedicated job of assembling a client specific page based on Interface Object (14) returned by business layer.

Problem

In applications that use dynamic rendering approach, a page is constructed using Interface Object (19). Presentation layer of such application need a dedicated component that can generate client specific page contents. What kind of component should it be to serve purpose of assembling page efficiently?

Forces

A Page Assembler should balance some or all of the following force(s).

- Modularity is one of the important aspects for application manageability. Because of its role in dynamic page generation framework, it should be implemented as separate component.
- Presentation layer architecture should be extensible. Assembler as component makes it easy to replace the client type by extending the base page assembler for specific type of client.

Solution

Implement a Page Assembler component and let this component assemble client specific pages. The page assembler component should provide appropriate services for building a page. The component should take a request from a page to build a client specific content. As assembling process, it finds a page structure from appropriate cache manager and based on this structure, this component makes business layer method calls and retrieves instance of Interface Object (19). It then operates on this object to construct current page.

Structure

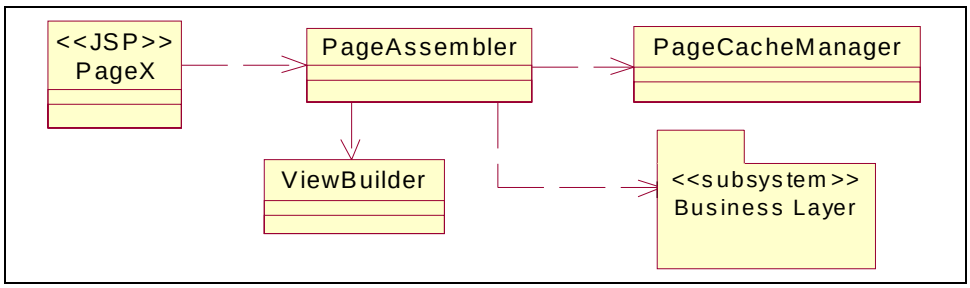


Figure 12: Classes and participating components for assembling dynamic page

Dynamic Page Assembling Dynamics

Figure 13 shows the sequence diagram of dynamic page assembling activity. The responsibilities of individual class or components are discussed in the following section.

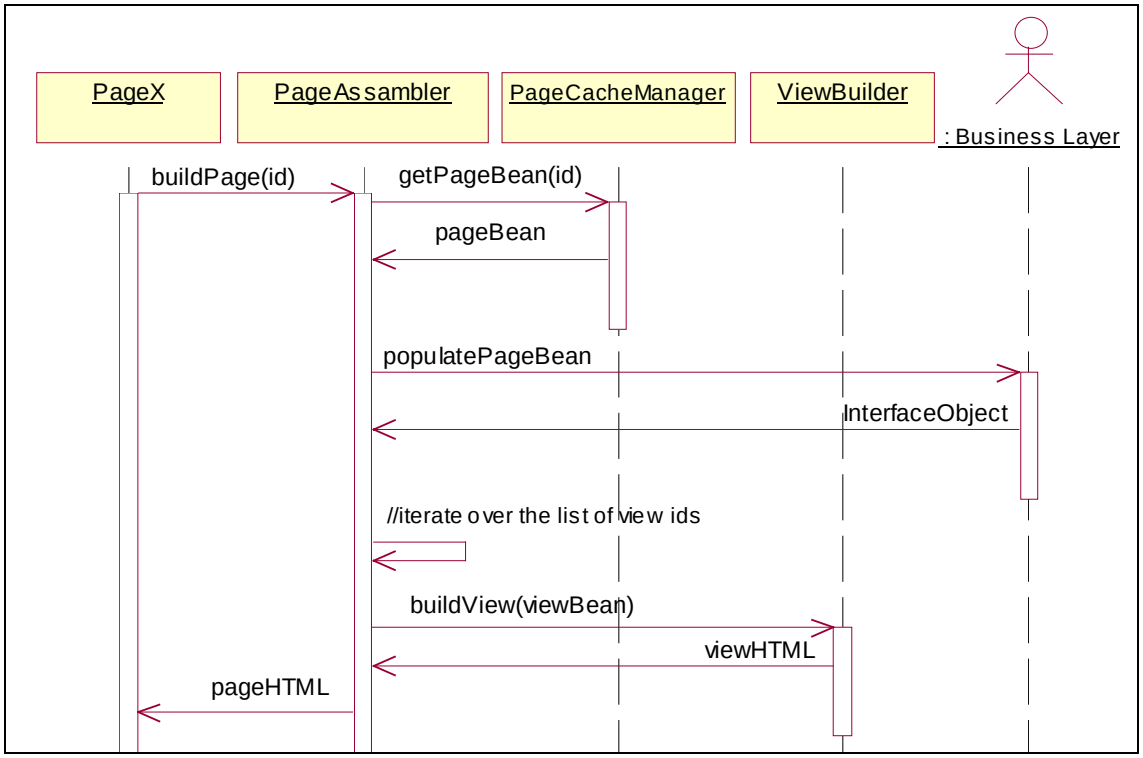


Figure 13: dynamic page assembling sequence diagram

Participants and Responsibilities

PageX

A page is used to send user request to the system and present system response to user.

Page Assembler

A Page Assembler is the heart of page assembling activity. Based on page id, it gets pageBean (structure) from appropriate cache manager, and requests business layer to provide page specific data. Business layer returns Interface Object (19) for this page. Page assembler uses custom View Builder (25) to assemble views. For HTML client, it creates HTML representing the dynamic page in the form of pageHTML, which is based the current instance of interface object.

Page Cache Manager

Page Cache Manager is a class that caches page level information for a given page, it holds information about all the views that can be shown in this page for all type of users and business processes. It reads and initializes itself using data in external data source so that database access frequency is minimized.

View Builder (25)

It constructs client specific view based on a ViewBean. For Web client, it constructs HTML text that represents current view.

Business Layer

Business Layer can be local or a reference layer. It provides Interface Object (19) instance that holds data for current page.

Consequences

Benefits

- Page assembling framework that can be extended to generate pages for different type of clients. Discussion was done generating HTML client. The same framework can be extended to different type of client, a WML for wireless client.
- Because it is implemented as separate component, page assembler promotes modularity, which in turn promotes reusability and maintainability.

Liabilities

- Intensive string manipulation may result in performance degradation unless appropriate caching mechanism is in place.

VIEW BUILDER

A View Builder allows creating client specific views based on data available in some pre-defined object.

Example

In dynamic page assembling approach, a page can be constructed in 'single shot' or as aggregate of views. Considering efficiency of page construction, the later approach is better than the earlier one because of the reason of reusability. Such presentation layer should be extensible enough so that any type of client can be served by providing contents for that type of client. In example of figure 14, two different sub types of view builder components are serving HTML and WML clients.

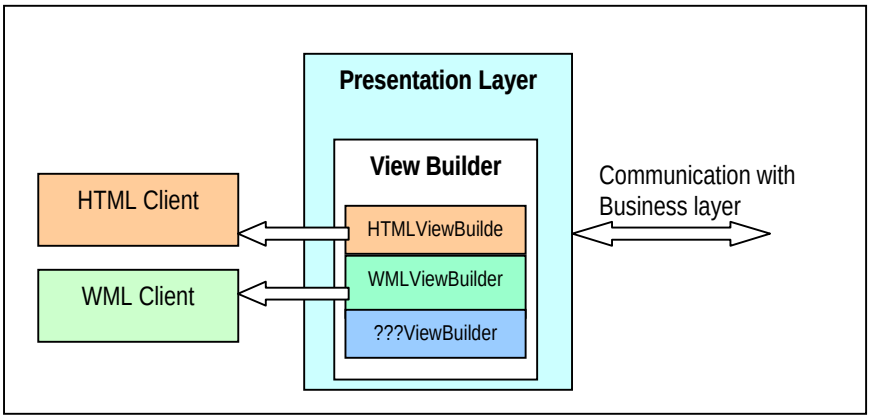


Figure 14: View Builder example

Context

You have selected to generate pages of your Web application dynamically and looking for an appropriate component that can build a client specific view.

Problem

How to efficiently generate client specific views of a Web application using dynamic page generation approach?

Forces

A view builder should balance some or all of the following forces.

- Construction of pages based on views promotes reuse since they can be used across the application**.**
- Presentation layer of a dynamic Web application should be extensible enough to generate contents for different types of clients.

Solution

Define a view builder component and let this component build client specific views on demand.

One of the most important factors of dynamic page construction architecture is the way they are constructed. In this approach, every field in a page is constructed at the time of rendering. Hence, there is reasonable amount of time spent while constructing such fields. Further, fields in a page tend to fall in some coarser grained entity called view. Because it is candidate of reuse, a client specific view once constructed, can be reused to improve system performance.

A typical view builder should have intelligence of constructing views based on a ViewBean instance (Figure 10). A ViewBean in domain model will represent a 'painted' view in a page. Hence, a ViewBean should hold necessary and sufficient information to construct a view for applicable page.

Structure

Figure 14 shows a view of participating classes for a view builder component. ViewBuilderFactory is factory class that will instantiate and return appropriate view builder instance on request-- HTMLViewBuilder instance for Web client and vice versa. HTMLViewBuilder uses HTMLFieldBuilder to construct view specific fields. It uses HTMLCacheManager to get field specific string to construct the current field. Figure 15 shows class diagram for a view builder component.

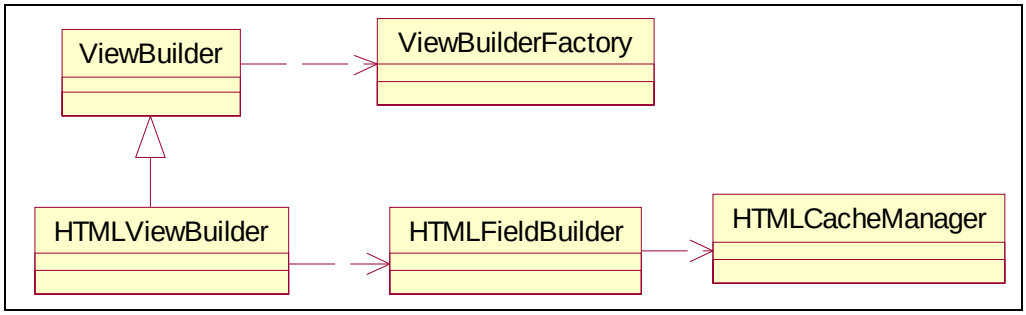


Figure 15: View Builder class diagram

Sequence diagram for a HTML view building process is shown in figure 16.

Dynamics of View Building

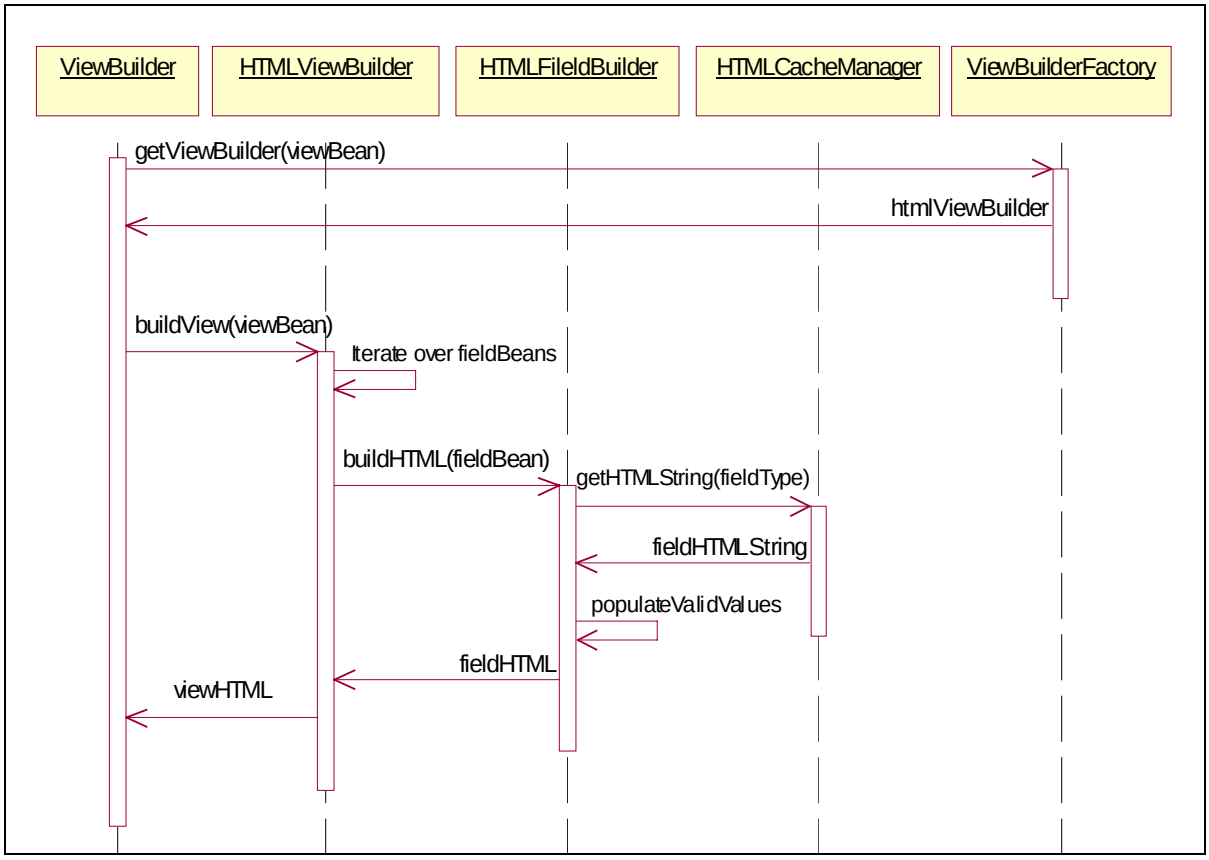


Figure 16: Sequence diagram for generating view HTML

Participants and Responsibilities

ViewBuilder

This is generic view builder class and it defines super type of all view builders.

HTMLViewBuilder

The instance of this class has an intelligence of building a view for HTML type clients. It iterates over the FieldsBeans and delegates calls to HTMLFieleBuilder to build individual fields.

HTMLFieldBuilder

The instance of this class knows how to build a field for HTML client. It builds equivalent HTML text that can be interpreted by Web browser to generate the type of field.

HTMLCacheManager

This cache implementation holds minimum text to paint HTML fields in client's browser window. It holds, in appropriate data structure, minimum text required for creating HTML fields such as text, checkbox, drop-down etc.

ViewBuilderFactory

A factory class that returns appropriate builder instance for given type of view. For a HTML client, it returns HTMLViewBuilder and vice versa.

Consequences

Benefits

- A component that can interpret and generate client specific.
- Dynamic generation of views.

Liabilities

- Extensive string manipulation.

Code Example

Here is code snippet that is used to generate a client specific field dynamically for which the data is available in interface object. The textHTML for a field can be retrieved from HTML cache manager by:

```
CacheManager htmlCacheManager = CacheManager.getCacheManager(CacheManager.HTML);
String textHTML = htmlCacheManager.getHTML(CacheManager.FIELD_TYPE_TEXT);
```

The first line returns an appropriate instance of cache manager, which is htmlCacheManager and knows how to return string representation for HTML client. The second call to cache manager returns a string representation for a text field, which may look like this:

```
<input type="text" name=name$ value=value$>
```

This text represents 'un-populated' version of the text field. In order to populate, the view builder can call following method to get the actual string representation.

```
String nameField = populateValidValue(textHTML, fieldBean);
```

In this example, the fieldBean instance holds the name-value pair to populate.

```
username=John Doe
```

The call will result in the full html text that is shown below.

```
<input type="text" name="username" value="John Doe" >
```

This is a valid HTML text that a browser can read and produce HTML text field that is shown in figure 16.

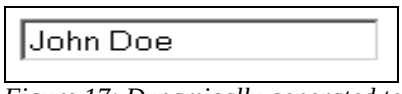


Figure 17: Dynamically generated text box field

CACHE MANAGER

Cache Manager provides caching required for presentation layer so that data is available in the application level cache reducing database access, xml parsing or object construction at rendering time.

Example

Lets consider a page that is being dynamically constructed. In this approach, individual view and hence field is constructed for given client type. In example figure 18 a page is shown that contains view 1 and view 2, view 2 has a 'Name' field.

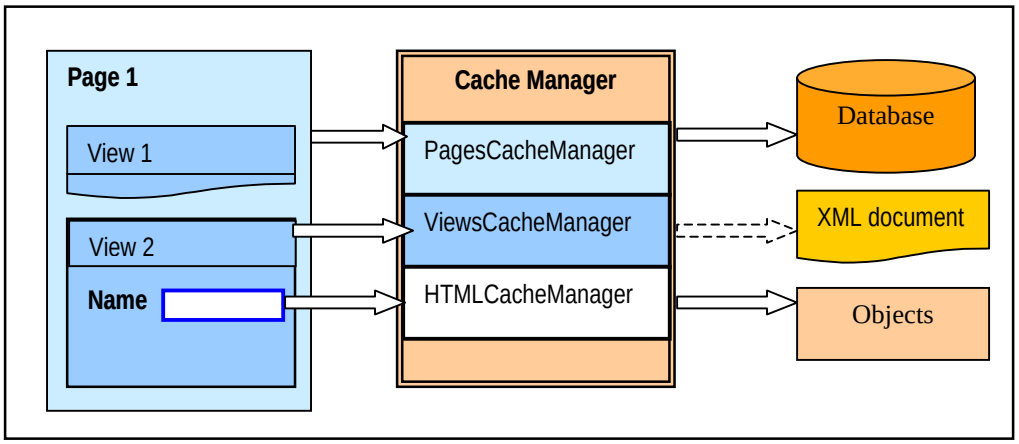


Figure 18: Example of cache manager to cache data from several sources

The information needed to construct such page may be available in database, XML document or an instance of simple objects. Accessing such information every time when needed is be time consuming. For this reason, an application level cache can provide better way of accessing such information by caching with appropriate cache manager objects. A typical cache manager has an intelligence of updating contents based on the need, for example, a PagesCacheManager will read and initialize at application startup time.

Context

Instead of using server or client side scripting for generating dynamic content, your presentation layer uses pages constructed at rendering time. The information required to successfully construct a page may come from multiple sources. You are looking to have a unified place for accessing data from such multiple sources.

Problem

Dynamic page rendering uses information from several sources. Accessing information from every possible source at page assembling time is not desirable because of performance and other reasons. How do you provide a

mechanism for caching application data from multiple sources so that accessing such data can be achieved from single place?

Forces

A cache manager should balance some or all of the following forces.

- View Builder (25) component should be able to construct individual views in time efficient way.
- It is desirable that frequently used information should be available in single application component.

Solution

Implement a Cache Manager, which caches required data for successful construction of pages at rendering time. Let such cache manager also have intelligence of resetting itself in case of data update or object attribute changes so that it is always in-sync with application state.

Several sources of data drive rendering of dynamic presentation layer. In order to improve the application performance; there is no question that application level data has to be cached. Because of the nature of data, a specific cache manager may be used to cache data from a specified source.

Structure

A CacheManager view of participating class is proposed in figure 16.

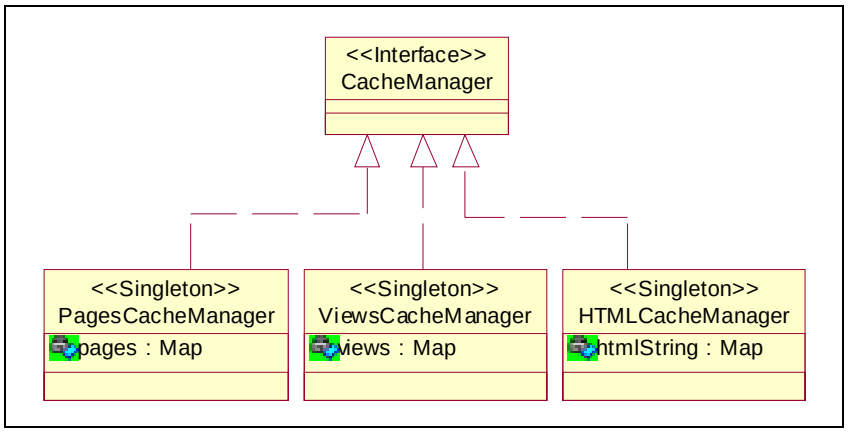


Figure 16: Cache Manager class hierarchy

Participants and Responsibilities

CacheManager

This interface defines object type and exposes common operations.

PagesCacheManager

A class that holds, in addition to others, page related information.

ViewsCacheManager

This class caches all views used across the application. Since views are reused, it makes necessary to keep repository of views instead of part of PageBean in PagesCacheManager.

HTMLCacheManager

This class holds appropriate data structure to cache possible 'minimum' string representation required to generate HTML elements.

Consequences

Benefits

- Repetitively used application data is cached that helps to increases application performance.
- Extensibility. Adding new type of cache manager is easy.
- Single point of data access.

Liabilities

- Complexity.

GLOSSARY

Business Layer: AKA business tier, it is a logical layer of programming constructs grouped together for the purpose of handling domain specific workflow and legacy communication. Presentation layer gets context related data from this layer.

Data Driven Architecture: In the context of Web application, a system, whose rendering and navigation aspects are driven by external data.

Field: A field is an actual presentation element that is shown to the user. A name text box for an HTML page is an example of a field.

Legacy Systems: For the context of this pattern language, a legacy system refers to a system that is not flexible enough to adapt new computing paradigms as they emerge, hence need to make them Web enabled.

Middleware: Collection of components of a large-scale enterprise web applications responsible to deliver contents to the application client that usually implement domain specific workflow and communicates to the back end systems. Middleware consists of business objects and server side scripts related to presentation layer as well as controller component that orchestrate program flow.

Page: Application element that is used to present the user with system response and let user enter and send request for further system processing. In the context of dynamic rendering architecture, a page consists aggregation of one or more views that are applicable to this page.

Pattern: Solution to a common and reoccurring problem in a particular context.

Pattern Language: Collection of related patterns grouped to solve a problem that occurs in some broader context.

Presentation Layer: AKA, presentation tier, is logical layer that hosts system elements for interacting with middleware and presenting contents to the client.

Quality of Services: The requirement of a system that it is supposed to provide when it is in service. There are several attributes to measure quality of services; some are more important that others for specific applications.

View: The presentation layer element that is used as unit of a page. A view is collection of fields that are grouped together to form a logical entity within a page. An example of a view is the Address view in a page, which includes some common fields such as line1, line2, city, state, zip for US postal address.

Web Enabling: Process that makes contents of a system accessible over the Internet.

REFERENCES

1. Gamma, Erich; Vlissides, John; Johnson, Ralph; Helm, Richard *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley; 15 January, 1995.
2. Britton, Chris, *IT Architecture and Middleware*, Strategies for Building Large, Integrated Systems, Addison Wesley, 2000.
3. Conallen, Jim; *Building Web Applications with UML*, Addison Wesley,2000
4. IBM Redbook on *Legacy Modernization with WebSphere Studio Enterprise Developer*
5. Resource available in IBM web site for *Web development Pattern* http://www.ibm.com/developerworks/patterns/
6. Core J2EE Patterns: Patterns index page: http://java.sun.com/blueprints/corej2eepatterns/Patterns/
7. Apache Struts resources page available at http://jakarta.apache.org/struts/
8. Fowler, M, *Patterns of Enterprise Application Architecture*, Addison Wesley, Oct 2002
9. Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerland Peter; Stal, Michael, *Pattern Oriented Software Architecture: A System of Patterns*

ACKNOWLEDGMENTS

Writing patterns is hard. Making them useful is even harder. I have given due attention to make these patterns as much useful and adaptable as I can, but as always, there may still be some room for improvement. The concepts expressed in this paper were observed as part of development teams in several projects that include mortgage, bank, and insurance domains. I want express my sincere thank to all who provided direct or indirect inputs towards this paper as part of team or otherwise. My special thanks to Peter Sommerland, who was shepherd for this paper for PLoP ’04 conference, whose insightful guidance and valuable feedbacks helped to improve this paper a great deal that it stands in today’s shape. I will also greatly appreciate your valuable comments during PLoP 04 and beyond in any patterns towards making them more useful and adaptable. Please direct your comments to s.acharya@computer.org.