

Modeling Navigation Routing in J2EE Web Applications

Minmin Han
Lehigh University
mih9@lehigh.edu

Christine Hofmeister
Lehigh University
Hofmeister@cse.lehigh.edu

Abstract

Navigation routing for web applications describes how requests from web pages are routed through components on the server. Many maintenance tasks require the developer to understand the navigation routing, but this is a labor-intensive task.

We describe an approach for improving the maintainability of J2EE web applications. We define categories of navigation needed for typical applications, implementation conventions for supporting navigation, and a model for describing navigation routing. While the implementation conventions are an improvement because they separate navigation-related code from processing code, the bigger improvement comes from the model, which allows the developer to quickly understand the routing and locate relevant code.

We use Z to define the navigation routing model, and provide tools to extract and analyze the model. We have applied this approach to a number of existing applications up to 34K LOC, showing improvement via the number of affected files and typical maintenance scenarios.

1. Introduction

Web applications are applications that interact with the user via web pages, but in contrast to web sites they typically have much more processing associated with each page request, this processing takes place on the server side, and often the processing updates a database or other form of persistent storage.

For these applications the user “navigates” through the web pages, so a *navigation map* describes the possible paths from page to page. However, the components on the server side can and typically do modify and redirect requests, making the resulting navigation very difficult to discern.

Common maintenance tasks for web applications are changing forms or links on a page, changing the processing of a request, adding or removing web pages, and displaying a different response page. All of

these types of changes require the programmer to understand how requests are routed through the components on the server; we call this the *navigation routing*. Some of these maintenance tasks change the routing itself, while for others the programmer uses the navigation routing in order to locate dependent code. For example, a change in a form means that the code processing the form may need to be changed.

Unfortunately, the code involved in supporting the navigation routing can be quite complex. In J2EE, the platform currently used in this research, there are many possible ways to encode the navigation routing. In addition, the navigation-related code is not readily encapsulated since each component in a navigation route contains parts of it. Finally, interpreting the navigation-related code requires a good understanding of the platform.

In J2EE a web application typically uses many types of components: JavaServerPages (JSPs), servlets, filters, JavaBeans, EntityBeans, and SessionBeans, but the navigation-related code is usually located in the first three types. A filter is a special kind of servlet, one that filters requests before passing them along to a servlet for processing. A JSP is intended to be used for displaying a page in the user’s web browser. It should contain mostly html code for formatting the page, although it can contain arbitrary chunks of Java code. JSPs are translated into servlets before execution. Although these conventions can be ignored, we use them in the following description of typical navigation routing.

Requests originate in a JSP, either from a link or a form in the displayed page. The JSP generates a “target URI” associated with the request. In the JSP this target URI could be: a hardcoded string, a Java expression that evaluates to a string, a JSP expression that evaluates to a string, or empty. If the target URI is the name of another JSP (a string ending in “.jsp”), this page is displayed in the client’s browser without invoking other components on the server side.

More typically a request must be processed by one or more components on the server, thus involving a file

named “web.xml” in the navigation routing. Filters and servlets are not explicitly invoked; instead the system uses web.xml to find the filter or servlet associated with the given target URI, then sends the request to this filter or servlet. The filter or servlet may do some processing of the request, and then can chain, forward, or redirect it using another (or the same) target URI. The system again uses web.xml to find the next component to execute, or if the target URI is a JSP, displays the result page in the user’s browser. Filters and servlets need not change the target URI; in this case, rather than repeatedly invoking the same filter or servlet, the system steps through the list of filters and servlets in web.xml.

Often a filter or servlet handles requests from more than one page. An extreme case is when the front controller design pattern is used, and one servlet receives all requests. In these cases the servlet may need to determine the origin of the request in order to do the proper processing. There are many ways of doing this, including using the target URI, using the value of a session variable, etc.

Therefore there are many variations on how navigation can be encoded, even if J2EE conventions are followed. The navigation-related code is dispersed among a number of JSPs, servlets, filters, and web.xml. The developer must also understand the different component types, the role of web.xml, and a number of J2EE navigation-related functions in order to trace the navigation routing. For platforms other than J2EE the details are different but they are similarly complex.

One way to improve the maintainability of navigation routing would be to change the platform. This is not practical, and improvements for navigation routing could well degrade the support for other aspects of web applications.

Our approach is first to define the kinds of navigation needed for typical applications. Then we define conventions for how these should be supported in the implementation. These conventions separate the navigation-related code from the rest of the application, making it easier to locate the code and easier to follow the navigation routing. Using conventions also reduces the variability in how navigation can be written.

However, it is still difficult to follow the navigation routing, because the navigation-related code makes heavy use of the J2EE infrastructure and the developer must do a lot of interpretation in order to extract the navigation routing. To handle this problem we extract a model of the navigation routing from the code. Developers can look at the model to determine navigation routing, then quickly locate the relevant

portions of the code. We also provide some automated analysis of the model, both for correctness of the navigation routing and to support maintenance tasks. More sophisticated analyses such as design pattern detection may be possible in the future.

The next section describes how this model improves two typical maintenance tasks. Section 3 describes the possible categories of navigation routing, and Section 4 describes the conventions we use to separate navigation concerns in the implementation. In Section 5 we give a formal basis for the model. Related work is described in Section 6. Tool support and an evaluation of our approach are covered in Sections 7 and 8.

2. Navigation Routing and Maintenance

To show how our approach improves maintainability we start with an application used by Sun as a tutorial [26]. We examine the petstore subsystem of the PetStore application. Figure 1 uses a *navigation routing diagram* to show the navigation routing for a portion of this subsystem. This type of diagram is a visualization of the underlying navigation routing model that we will describe in Section 5.

Figure 1 shows that the web page Cart.jsp contains a form named checkout. When a request originates from checkout, it has a target URI of enter_order.screen, and this request is first routed to the filter EncodingFilter. The filter invokes method setLanguage for this request, then sends it along with the same target URI, enter_order.screen. Next the request is routed to filter SignOnFilter, and function testSignOn is invoked. If the result of testSignOn is “yes”, it sends the request along with the same target URI (enter_order.screen). Since the diagram does not show any subsequent filters or servers for the request, at this point enter_order.screen is treated as a composite page to be displayed in the user’s browser.

If instead the result of testSignOn is “no”, the diagram shows that composite page signon.screen is displayed. In this case SignOnFilter does one additional thing: it sets a session variable named original_URL with the value “enter_order.screen”. It does so because after the user successfully signs on, he or she should be returned to page enter_order.screen. This is a case where the navigation routing depends on the navigation history.

To continue with the navigation routing we use the composite page table (Table 1). This lists all composite pages and their constituent parts, typically as JSPs. Any constituent part from which a request can originate will appear at the left side of a navigation routing diagram, since it is the start of a navigation path. Thus signon.screen contains signon.jsp, which

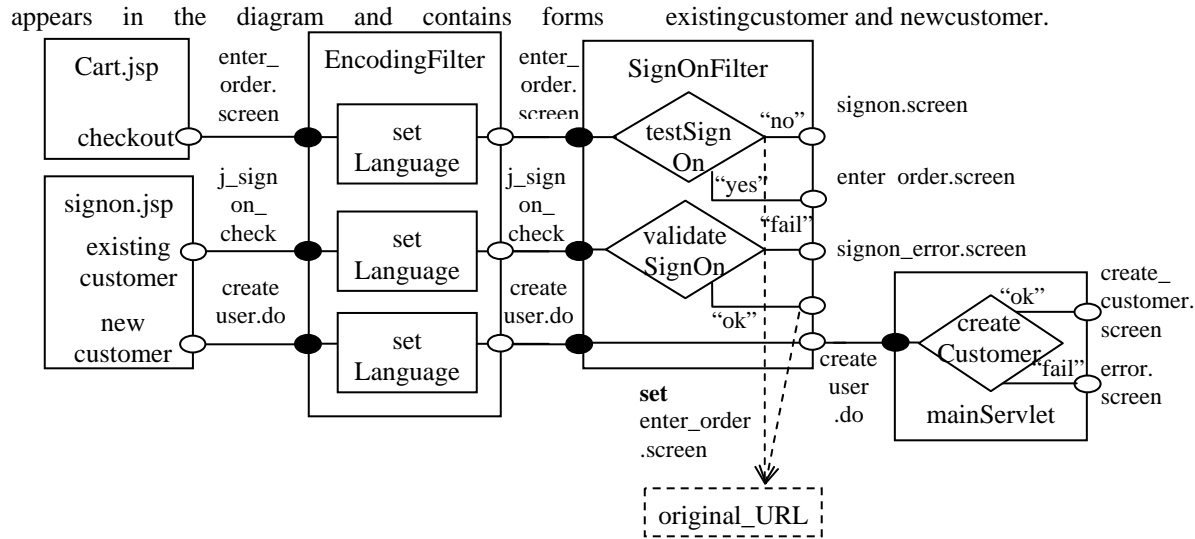


Figure 1 Navigation Routing Diagram for Pet Store

If the request originates from the existingcustomer form, it has a target URI of j_signon_check. From the diagram we see that this request is routed through method setLanguage in EncodingFilter, then through function validateSignOn in SignOnFilter. When the function returns “fail” the composite page signon_error.screen is displayed. When it returns “ok”, the value of original_URL is used as the composite page to be displayed. This should have been set by the page that was displayed prior to the signon.screen.

The last type of request shown in Figure 1 originates from the newcustomer form. This request is treated by EncodingFilter in the same way as the others, but SignOnFilter does not do any processing on it. SignOnFilter simply sends it along with the target URI of createuser.do. Then the request is routed to servlet MainServlet, where the composite page create_customer.screen or error.screen is displayed depending on the result of function createCustomer.

Table 1 Excerpt of PetStore Composite Page Table

Composite Page	Constituent parts
signon.screen	Banner.jsp, sidebar.jsp, signon.jsp, footer.jsp
enter_order.screen	Banner.jsp, sidebar.jsp, enter_order_information.jsp, footer.jsp
signon_error.screen	banner.jsp, sidebar.jsp, signon_failed.jsp, footer.jsp

2.1. First Maintenance Scenario

For the first maintenance scenario the customer has requested that a new input field, zip code, be added to the newcustomer form of the signon screen. With our

approach for supporting navigation, the developer uses the navigation routing model to quickly locate code that may be dependent on the change to the form. The developer has only the following tasks:

1. Update signon.jsp with the new field.
2. Check method setLanguage of EncodingFilter for code related to the data in the newcustomer form. No changes are needed.
3. Check function createCustomer of MainServlet for code related to the data in the newcustomer form.
4. Make a number of changes to MainServlet and classes it uses, in order to correctly process the new field.

In contrast, to accomplish the same change on the application as it was originally written, the developer must do the following:

1. Update signon.jsp with the new field.
2. Check signon.jsp for the target URI in the newcustomer form: createuser.do.
3. Read web.xml to find the first filter/servlet first for createuser.do. The answer is EncodingFilter.
4. Check EncodingFilter for code related to the data in the newcustomer form. No changes are needed.
5. Check EncodingFilter for navigation-related code. Note the target URI: createuser.do.
6. Read web.xml to find the next filter/servlet for createuser.do: SignOnFilter.
7. Check SignOnFilter for code related to the data in the newcustomer form. No changes are needed.
8. Check SignOnFilter for navigation-related code. Note the target URI: createuser.do.
9. Read web.xml to find the next filter/servlet for createuser.do: MainServlet.
10. Check MainServlet for code related to the data in the newcustomer form.
11. Make a number of changes to MainServlet and classes it uses.

12. Check `MainServlet` for navigation-related code. Note the target URIs: `create_customer.screen` and `error.screen`.
13. Read `web.xml` to find the next filter/servlet for `create_customer.screen` and `error.screen`: `TemplateServlet`.
14. Check `TemplateServlet`. It simply displays the page, so no changes are needed and the navigation path stops here.

For this scenario, the biggest benefit of our approach is that the developer does not need to figure out the routing. An additional benefit is that we put the processing a filter or servlet does for an incoming target URI into separate methods or functions. Thus the developer can go straight to the appropriate method or function rather than examining all the processing code in the filter or servlet.

2.2. Second Maintenance Scenario

In the second maintenance scenario the client has requested a new feature: the ability for a customer to look at their past orders. A new order history page is needed to support this feature, and its navigation should be similar to `Cart.jsp`. To replicate this navigation routing for the new page, the developer uses the navigation routing model to understand the routing for `Cart.jsp` and `signon.jsp`, then modifies the code based on this information.

Using our approach the developer does the following:

1. Create a new `OrderHistory` page containing a link with target URI `order_history_info.screen`. This will be part of a new composite page named `order_history_info.screen`.
2. Check method `setLanguage` to see whether it contains processing specific to `enter_order.screen`. There is none.
3. Check function `testSignOn` to see whether it contains processing specific to `enter_order.screen`. There is none.
4. Check the navigation code for filters, located in the Filter aspect module. For `EncodingFilter` there is no special processing for `enter_order.screen`, so no changes are needed. For `SignOnFilter` there is. Based on the results of invoking `testSignOn`, it forwards the request either to `enter_order.screen` or `signon.screen`. In the latter case it sets `original_URL` to `"enter_order.screen"`.
5. Modify the Filter aspect module, copying the processing for `enter_order.screen` but replacing the target URI with `order_history_info.screen`.
6. Modify `TemplateServlet` to display the constituent parts of `order_history_info.screen`.
7. Update the navigation routing model.

To make the change on the application as it was originally written, the developer must follow the navigation paths originating at checkout, existing-

customer, and newcustomer. Locating the relevant code involves approximately three times the number of steps as in the first scenario, since three navigation paths must be followed. To make the changes the developer does the same thing as in our approach; the only difference is that the new navigation code is located in `SignOnFilter` rather than in the Filter aspect module.

As in the first scenario, separating the filter and servlet code into processing code and navigation code in one improvement of our approach. The bigger improvement comes from having an accurate model of the navigation routing so that the developer can quickly understand the routing and locate the relevant parts of the implementation.

3. Categories of Navigation Routing

Before describing the conventions we use to support navigation routing at the implementation level, we first describe three categories of navigation routing that cover the needs of typical applications. These describe how a filter or servlet routes an incoming URI to an outgoing URI.

Static routing is when the outgoing URI depends only on the incoming URI, and their values are fixed by the developer or deployer, e.g. hardcoded in the application or in a table. Since nothing needs to be decided at the run time, static routing is the simplest form of navigation routing. In the maintenance example, all navigation routing in the `EncodingFilter` is static routing. JSPs use static routing, except they have a form or link instead of an incoming URI.

Conditional routing is when the outgoing URI is selected based on the incoming URI and the result of processing, and their values are fixed by the developer or deployer. Conditional routing is used frequently in web applications. In our maintenance example, `SignOnFilter` uses conditional routing for incoming URIs `enter_order.screen` and `j_signon_check`, and uses static routing for `createuser.do`.

Routing depending on navigation history is when the outgoing URI depends on the incoming URI, and having followed another navigation path prior to this one. Again the values of the incoming URI and associated outgoing URIs are fixed by the developer or deployer. In the maintenance example, this type of routing is used in `SignOnFilter` for incoming URI `j_signon_check` when `validateSignOn` returns "ok".

There are other ways of implementing navigation routing, such as when the user enters a target URI that is used for navigation. However, we believe the above three categories cover most navigation routing in web applications. Although within these three categories

there are still many variations on how navigation could be implemented, next we define implementation conventions that are sufficient for supporting the three categories. In the future we may expand our support to cover more categories, if we find others that are commonly used.

4. Support at the Implementation Level

The basic approach for our implementation conventions is to separate the navigation routing from the rest of the request processing. The approach should: support the three categories of navigation routing; minimize the performance impact; not dictate or overly constrain the application structure.

We currently use aspect-oriented programming (AOP) techniques to separate the navigation concerns. AOP is considered a new approach to separating the concerns in applications, since techniques in object-oriented programming and procedural programming do not suffice. The basic idea is to define and implement the crosscutting concerns in aspect modules and weave these aspect modules with other OO modules before executing.[9] Since our platform is J2EE, we use a Java-based aspect language: AspectJ. [10] The details of our use of aspects is described elsewhere[7]. Here we summarize the approach.

As the example in Section 2 showed, navigation routing code is located in JSPs, filters, servlets, and web.xml. Because web.xml does not contain processing code for requests, we do not apply any special conventions to it.

Inside a JSP the target URI must be specified by invoking the function:

```
string getTargetURI (string  
pageName, string formOrLinkName);
```

This function returns the appropriate target URI for all possible pairs of pageName and formOrLinkName.

With AOP we use a dummy function getTargetURI in every JSP. Then the Page aspect module catches the execution of getTargetURI and returns the target URI.

For filters and servlets there are two conventions. The first is that any processing needed for an incoming target URI must be wrapped in a function or method. The second is that the doGet/ doPost/ doChain methods do nothing more than route incoming target URIs to outgoing URIs, invoking no more than one function or method for each incoming URI.

With AOP this is done by leaving the doGet/ doPost/ doChain methods empty. The routing that belongs in the filters is put into a Filter aspect module, and the routing that belongs in servlets is put into a Servlet aspect module. The Filter aspect module catches the execution of doChain methods of the filters

and inserts the routing code. The Servlet aspect module does a similar thing with the doGet and doPost methods.

The last step is to weave the Page aspect module, Servlet aspect module, and Filter aspect module with the JSPs, servlet classes, and filter classes. The only catch is that JSPs are not written in Java, so they do not work with AspectJ. As a workaround the developer has to weave the generated Java classes for the JSPs with the aspect module.

It should be clear that it is possible to separate the navigation code without using AOP. AOP has the advantage that the code in the aspect modules can be written as if it were physically located inside the corresponding JSP, filter, or servlet. Thus for example the Servlet aspect module can contain references to private attributes of a servlet. With AOP we can not only separate the navigation code, we can gather it together, confining it to the three aspect modules and web.xml.

It is also possible to use a table-driven approach for separating the navigation code. One table-driven approach is to separate the navigation concern of page, servlet and filters into XML files and read the XML tables at runtime. However, this adds complexity because the programmer has to write many extra lines of code to retrieve data from the XML tables in web pages, servlet classes and filter classes.

The use of the ActionServlet class from the Struts library can alleviate this burden: by using this class the developer need only fill in the file struts-config.xml. The disadvantages are that this forces a front-controller structure on the application, and that only static and conditional routing are possible.

5. The Navigation Routing Model

For a formal representation of our navigation routing model we use the Z notation[23]. As the maintenance example shows, the model is used by the developer in order to understand the navigation routing and to locate code that implements the routing or processes a request. Depending on tool support, the developer could make navigation routing changes in the model and have the implementation automatically updated. Tool support is discussed in Section 6. In addition, the developer can apply analysis on the model in order to produce complexity metrics, estimate the impact of a proposed maintenance task, check the correctness of the navigation routing, etc.

The following tables describe the schemas we defined in Z to represent the navigation routing. In Table 2, all but the last two are set explicitly, e.g. by

extracting them from the implementation. Navigation and Connection are computed from the others.

There are four kinds of nodes, to represent the components involved in the navigation. PageNodes are the JSPs from which a request can originate, thus they are at the beginning of a navigation path. CompositePageNodes are at the end of a navigation path; they represent the web page that is displayed as a response to the request. FilterNodes and ServletNodes are internal nodes in a navigation path.

For each target URI used by a node we create a unique port. Entry ports are for incoming target URIs, and exit ports are for outgoing URIs. We use

InNodeConnections to record which exit port(s) are associated with each entry port in a node. A method or function can also be associated with one or more of the InNodeConnections, when the node does some processing of the request.

The schema Navigation records connections between nodes; these are determined by connecting exit and entry ports with matching URIs according to the sequencing of nodes. The sequencing of nodes dictates that PageNodes precede FilterNodes; FilterNodes are in a strict sequence; FilterNodes precede ServletNodes; and ServletNodes precede CompositePageNodes.

Table 2 State Schemas

Schema Name and Description	Variables used for storing state	Sample contents from maintenance example
PageNodes: JSPs from which a request can originate	PNodes	{ cart.jsp, signon.jsp }
FilterNodes: Sequence of Filter classes	FNodes	< EncodingFilter, SignOnFilter >
ServletNodes: Set of Servlet classes	SNodes	{ MainServlet }
CompositePageNodes: Names and constituent parts of composite pages	CPageNodes	{ signon.screen, enter_order.screen... }
	CPageNodesContent	{ (signon.screen, {banner.jsp, sidebar.jsp, signon.jsp, footer.jsp}), ... }
URIVarsandValues: Target URIs, including values of URI variables	URIVariables	{ original_URL }
	URIVarsValue	{ (original_URL, {enter_order.screen}) }
	URIValues	{ enter_order.screen, j_signon_check... }
AllPorts: Entry and exit ports, associated nodes, associated URIs	Ports	{ port1, port2, port3, port4, port5... }
	AttachedNode	{ (port1, cartjsp), (port2, signonjsp) ... }
	PortType	{ (port1, EXIT), (port2, EXIT) ... }
	PortURIVar	{ (port16, original_URL) }
	PortURIValue	{ (port1, enter_order.screen)... }
	PortFormOrLink	{ (port1, checkout)... }
InNodeConnection: Entry ports and their associated exit ports within a node	InNodePort Connections	{ (port4, port7), (port5, port8), ... (port10, port13)... }
	GlobalVarSet	{ ((port10, port13), (original_URL, enter_order.screen)) }
MethodsAndFunctions: Methods or functions associated with an InNodeConnection	Methods	{ setLanguage }
	Functions	{ testSignOn, validateSignOn... }
	MethodPorts	{ ((port4, port7), setLanguage) ... }
	FunctionPorts	{ ((port10, port13), (testSignOn, "no"))... }
Navigation: Connections between the exit port of one node and the entry port of another	Links	{ (port1, port4), (port2, port5), ... , (port7, port10), ... }
Connection: Directly and indirectly connected ports	ConnectedPorts	{ (port1, port13), (port1, port4), ... }
	ConnectedNodes	{ (cart.jsp, signon.screen), (cart.jsp, EncodingFilter), ... }

The schemas in Table 3 are computed on demand and have no state of their own. These are used to

update the other schemas that do have state, and to provide some additional analysis of the navigation routing. For example, UnusedNodes and UnusedPorts

can help the developer determine the correctness of the navigation routing. The number of paths in `EndToEndConnection` gives an indication of the complexity of the navigation routing of an application, as does the length of these paths. Another indication of navigation routing complexity is the size of `PortURIVar` and `GlobalVarSet`, which indicates how many places the navigation routing depends on the navigation history. `EndToEndConnection`, `Depending`, and `Depended` could be useful in estimating the impact of a proposed maintenance task. As a last example, the developer can surmise the use of the front controller design pattern when a servlet or filter node has as its `Depending` nodes all the `PageNodes` and no other nodes.

Table 3 Operation Schemas

Schema and Description
<code>UpdateNavigation</code> : Calculate and update Navigation
<code>GenerateNavigationMap</code> : <code>PageNodes</code> and the <code>CompositePageNodes</code> that appear in Connection
<code>Depending</code> : For each node, the set of nodes that precede it in some Connection
<code>Depended</code> : For each node, the set of nodes that follow it in some Connection
<code>UnusedNodes</code> : The set of nodes not involved in any Connection
<code>UnusedPorts</code> : The set of ports not involved in any Connection
<code>EndToEndConnection</code> : The set of paths from a form or link to a composite page, stored as the sequence of ports on the path.

The navigation routing model supports the three categories of navigation from Section 3 as follows:

- **Static routing**: A pair of ports that appears in `InNodePortConnections`, does not appear in `FunctionPorts`, and where the exit port appears in `PortURIValue`.
- **Conditional routing**: A pair of ports that appears in `InNodePortConnections`, appears in `FunctionPorts`, and where the exit port appears in `PortURIValue`.
- **Routing depending on navigation history**: A pair of ports that appears in `InNodePortConnections`, appears in `GlobalVarSet`, and where the exit port appears in `PortURIVar`.

We use navigation routing diagram (NRD) as a visualization of the model. A node is shown as a box with ports attached on the sides. One exception is that in the NRD composite web pages are shown as the page name at the end of a navigation path. The white ovals represent entry ports and the black ovals

represent exit ports. The URI value associated with a port is noted next to the port, but port names do not appear in the diagram. `InNodeConnections` and `Navigation` links are shown as lines between ports. If there is a method or function associated with one in-node connection, it is shown as a box or a diamond shape in the middle of the connection. The global variables used for exit port URIs are shown in a dashed-line box, with dashed arrows from an in-node connection for setting the value or from a port for getting the value.

6. Related Work

Our research belongs to the research area of web engineering. Deshpande et al. discuss the idea of web engineering as “the application of systematic, disciplined and quantifiable approaches to development, operation, and maintenance of web-based applications”. [3][4]

It is clear that researchers believe navigation concerns are an important part of web applications. Rossi [21] identifies a number of common navigation patterns used in web applications. Reina [19] states that it is necessary to separate the navigation concerns into aspect modules but does not present a specific solution.

There is no one standard notation for a navigation model. Some of the discussion occurs in the context of web application modeling. These approaches focus on navigation models that contain conceptual objects and show how the conceptual objects connect. Examples are the “navigation space model” and “navigation structure model” used by Koch [12], the navigation schema in Autoweb system [6], and the navigation diagram in RMM [8]. In our work we want the navigation model or representation to correspond much more closely to the source code.

More typically a navigation model is quite similar to the “Architecturally Significant Navigation Map” in Pearl Circle [18], which contains request page, response page, an arrow to connect them, and a label on the arrow to explain why the user wants to go to the response page. Similar ones include the “site design diagram” in Microsoft InterDev [15], the ADM-d model for “dataintensive” web sites in the Araneus approach [14], the navigation diagram in WebML [2] and W3DT [1], “Navigational Contexts Schema” in OOHDM [22], and “WebApp Top-Level Navigational Charts” by Enguix [5].

Another type navigation model is presented by Leung et al. [13]. They use statecharts to describe navigation. Each page is a state and a link between pages is represented by a transition between states.

These navigation models and diagrams contain only the request and the response pages but not the intermediate processing modules and target URL, as our model does. Our model provides enough information that developers do not need to trace the navigation each time there is an update requirement in the web application.

Others have applied AOP to web applications but not for separating navigation concerns. Reina et.al. [20] follow the proposals of Kilesev [11] to use AOP to address security, design by contract, exception handling, logging, tracing, profiling, pooling and caching concerns in web application development, but they only present a solution for authentication.

7. Tool Support

There are a number of tools that are useful for support the navigation routing model. The first is essential, because it implements the Z specification for the model. Thus it supports the input of data for a navigation model and the execution of operations on this data. A general-purpose Z animator could serve this purpose, but we have found no suitable animator. Thus we have written NavR, a program that implements the specification described in Section 4. NavR accepts the sample data input in ASCII format, and it executes the part of the executable schemas that describe the operations. Developers can use NavR to enter, examine, and analyze a navigation routing model. Currently we have implemented all schemas described in section 4, and will continue to update NavR as new schemas for additional analysis operations are specified.

The next most important tools are those that support traceability between a navigation routing model and the corresponding implementation. One of these, NavRExtract, performs reverse engineering, extracting the navigation model from the source code. The input for NavRExtract includes the three aspect files and web.xml file, and the output is state data for the NavR tool. The navigation information related to web pages (page nodes, ports, and URIs) is extracted from the page aspect module. The navigation information for filters (filter nodes, ports, URI values and variables, in-node connections, methods, and functions) is extracted from the filter aspect module and web.xml. The navigation for servlets (servlet nodes, ports, URI values and variables, in-node connections, methods and functions) is extracted from the servlet aspect module and web.xml.

The current version of NavRExtract uses flex and bison to parse Page, Filter, and Servlet aspect modules. It does not yet extract information for the composite

page nodes; this would come from the composite page table. We require the aspect modules to be written in a certain style so the tool can parse the files. We support the use of wild cards (*) in web.xml, but not currently in the aspect modules. The developer is thus responsible for expanding the wild card to explicit target URIs. In the next version of NavRExtract we plan to support wild cards also in the aspect modules. Then the tool will perform the expansion of target URIs. So far we have used NavRExtract on PetStore, the most complex of the examples we describe in the next section.

To accomplish the forward engineering we plan to use a NavRGen tool, which will generate source code from a NavR specification. This functionality may be combined with NavR instead of being a standalone tool, since much of the code it needs already exists in NavR.

Finally, there are two additional tools that would help the developer interact with model. A visualizer would be very useful. We currently use navigation routing diagrams such as that shown in Figure 1 to visualize a navigation routing model, but we create these by hand. A diagram editor that can be used to directly create and update a model would also be very useful, but would require a large investment.

8. Evaluation and Discussion

We have applied this approach to a number of existing applications. Some of them are small applications provided with the J2EE server 1.4, including four Hello applications and two Duke's Bookstore applications. Other applications are comprehensive ones: Duke's Bank[24] and PetStore [25] from Sun's tutorial and Virtual Shopping Mall [17] from Oracle's tutorial. Duke's Bank is a comparatively small on-line banking application which uses the front controller pattern, static routing, and some conditional routing. Virtual Shopping Mall is a medium-size on-line retail store application which also has a front controller, but it is the ActionServlet from the Struts library. It uses an XML table for navigation and mostly conditional routing. Pet Store is another medium-size on-line store application, and it uses all three types of navigation routing. Table 4 shows some statistics for these three applications as they were originally written.

Our approach worked well on all of these examples. For Virtual Shopping Mall, since the generic Struts ActionServlet is used, much of the navigation-related code in the servlet is already separated into XML files. The ActionServlet also contains some navigation-related code, since it handles the routing based on data

it reads from the XML file. Thus for this application we chose not to apply the AOP approach to the servlet, and we used it only for the web pages. With the help of the NavRExtract tool we created the navigation model for these three applications partially automatically. Table 5 shows some statistics for these applications after our approach was applied.

Table 4 Statistics for the original applications

	Duke's Bank	Mall	Pet Store
Total LOC	7133	19248	34048
# web pages	10	35	16
# filters	0	0	2
# servlets	1	7	2
# composite pages	10	35	19
# navigation paths	14	113	48
Avg. length nav. path	3	3	4
# nav. paths related to nav. history	0	0	6
# modules with nav.	12	44	21
LOC modules with nav.	1015	6154	9059

Table 5 Statistics for the modified applications

	Duke's Bank	Mall	Pet Store
# modules with nav.	3	4	4
LOC modules with nav.	200	518	370
# func. and methods	3	27	11
LOC func. and methods	120	3175	2016

Since the aspect modules are woven into the other source code before executing, the performance is not affected. Also applying our approach does not restrict how the application is structured with respect to other concerns. Some table-driven approaches force the application to use front controller pattern, such as when the ActionServlet from Struts is used. With our AOP approach the only constraints on application structure are that the navigation concerns are separated into up to three aspect modules.

As evidence that our approach improves maintainability, we first look at the number and size of files that contain navigation concerns before and after applying our approach. These are shown in the two tables above. For example, in PetStore originally there were 21 files containing navigation concerns: 16 web pages, 2 filters, 2 servlets, and web.xml. After applying our approach there are only the three aspect

files (Page, Filter, and Servlet aspect modules) and web.xml. Since the developer can focus on few, smaller files, the maintenance work related to navigation should be easier to perform. Duke's Bank and Virtual Shopping Mall have similar improvements: from 12 to 3 (there are no filters, so no Filter aspect) and from 44 to 4 (including the original XML table for the ActionServlet).

However, maintainability is improved primarily because the model can be used to understand and analyze the navigation routing. With the two maintenance scenarios in Section 2 we described the reduced effort required by the developer when the model is used to follow navigation paths. We have also begun providing operations that perform analysis on the model, and more can easily be added. An alternate way of handling the first maintenance scenario in Section 2 is for the developer to use the Depended operation. This finds all filters, servlets, and composite pages that follow signon.jsp, and then the developer can go directly to those files to examine all functions and methods for code related to the newcustomer form.

We can also use the navigation model as an indication of the complexity of an application's navigation routing, without even looking at the source code. Looking at the number of navigation paths (EndToEndConnections) and the average length of these paths, we see in Table 4 that Duke's Bank has 14 navigation paths, which is many fewer than the 113 in Virtual Shopping Mall. Since the average length of these paths is the same for both applications and neither has navigation paths that depend on the navigation history, we conclude that the navigation routing in Duke's Bank is simpler than in Virtual Shopping Mall because there are so many fewer paths. PetStore has an intermediate number of navigation paths, but with a longer average length, and a number of these depend on the navigation history. Thus its navigation routing is potentially the most complex.

Currently our implementation conventions are based on AOP, but other kinds of conventions are possible. Although our conventions improve the maintainability of the implementation, another important purpose is to reduce the variability for how navigation routing can be encoded, so that a tool can readily extract the navigation routing model from the code. If other conventions are used, they should be easy to understand for typical developers, and of course they must cover all three categories of navigation routing. Automatic generation of source code from a model helps the developer properly follow the conventions. New forward and reverse engineering tools would be needed to support another set of implementation conventions, but the tool that supports

the model (NavR) and any visualization and editing tools would need no changes.

9. References

- [1] M. Bichler, and S. Nusser, "Developing structured WWW-sites with W3DT". In *Proceedings of the WebNet - World Conference of The Web Society, USA*, October 16 -19, 1996.
- [2] S. Ceri, P. Fraternali, and A. Bongio, "Web Modeling Language (WebML): a modeling language for designing Web sites". In *Proceedings of the 9th International WWW Conference on Computer Network*, North-Holland Publishing Co. Amsterdam, The Netherlands, Amsterdam, The Netherlands, 2000, pp.137-157.
- [3] Y. Deshpande and Y. Hansen. "Web Engineering: Creating a Discipline among Disciplines". *IEEE Multimedia*, 8, 1-2, 2001.
- [4] Y. Deshpande, S. Murugesan, A. Ginige, S. Hansen, D. Schwabe, M. Gaedke, and B. White. "Web Engineering". *Journal of Web Engineering*, 1, 1, 2002, pp.3-17.
- [5] C.F. Enguix, and J.G. Davis, "Filling the Gap: New Models for Systematic Page-based Web Application Development & Maintenance". In *Proceedings of the Workshop on Web Engineering 8th International World Wide Web Conference*, Toronto, Canada, May 11, 1999.
- [6] P. Fraternali, and P. Paolini, "Model-Driven Development of Web Applications: The Autoweb System". *ACM Transactions on Information Systems*, 28, 4, Oct, 2000, pp.323-382.
- [7] M. Han and C. Hofmeister. "Separating and Representing Navigation Concerns in Web Applications". Lehigh University Technical Report. LU-CSE-04-004
- [8] T. Isakowitz, E.A. Stohr, and P. Balasubramanian, "RMM: A Methodology for Structured Hypermedia Design". *Communications of the ACM*, 38, 8, Aug. 1995.
- [9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Pal, and W.G. Griswold, "An Overview of AspectJ. J". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, Springer-Verlag LNCS 2072, Budapest, Hungary, June 18-22, 2001, pp. 327-353.
- [10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, Springer-Verlag LNCS 1241, Finland, June 1997, pp.220-242.
- [11] I. Kilesev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, 2002.
- [12] N. Koch and A. Kraus. "The Expressive Power of UML-based Web Engineering". In *Proceeding of the Second Int. Workshop on Web-Oriented Software Technology (IWWOST '02)*, Malaga, Spain, June, 2002, pp.105-120.
- [13] K.R.P.H. Leung, L.C.K. Hui, S.M. Yiu, and R.W.M. Tang. "Modelling Web Navigation by Statechart". In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC '00)*, October 2000.
- [14] P. Merialdo and P. Atzeni. "Design and Development of Data-intensive Web Sites: The Araneus Approach". *ACM Transactions on Internet Technology*, 3, 1, Feb 2003, pp.49-92.
- [15] Microsoft Visual InterDev Reference. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vidref98/html/dvconsignedesign.asp>
- [16] ORA Canada. Z/EVES. <http://www.ora.on.ca/z-eves/welcome.html>
- [17] Oracle J2EE sample code. Virtual Shopping Mall. http://www.oracle.com/technology/sample_code/tech/java/j2ee/vsm13/index.html
- [18] Rational Software. Pearl Circle Online Auction Reference Application Reference Application Software Architecture Document, Issue 0.2, Rational Software 2001.
- [19] A.M. Reina, and J. Torres. "Analysing the Navigational Aspect". In *Proceedings of Second Aspect-Oriented Software Development Workshop (AOSD '02)*, Germany, February 2002.
- [20] A.M. Reina, J. Torres and M. Toro. "Aspect-Oriented Web Development vs. Non Aspect-Oriented Web Development". In *Workshop of Analysis of Aspect-Oriented Software (AAOS '03)*, Darmstadt, Germany, July 2003.
- [21] G. Rossi, D. Schwabe, and F. Lyardet. "Improving Web Information Systems with Navigational Patterns". In *Proceedings of the 8th International Conference on WWW*, Elsevier North-Holland, Inc., New York, NY, Toronto, Canada, 1999, pp.1667-1678.
- [22] D. Schwabe, and G. Rossi, "An Object Oriented Approach to Web-Based Application Design". *Theory and Practice of Object Systems*, 4, 4, 1998. Wiley and Sons, New York.
- [23] J.M. Spivey. The Z Notation: a reference manual. <http://spivey.oriel.ox.ac.uk/~mike/zrm/>
- [24] Sun Java Center. The Duke's Bank Application. <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Ebank.html>
- [25] Sun Java Center. Java Petstore 1.1.2. http://java.sun.com/developer/releases/petstore/petstore_1_1_2.html
- [26] Sun Java Center. Java Servlet Technology. <http://java.sun.com/products/servlet/>

10. Appendix: Formal Description of Navigation

[*NODE*]

PageNodes

PNodes: $\mathbb{P} \text{ NODE}$

CompositePageNodes

CPageNodes: $\mathbb{P} \text{ NODE}$
CPageNodesContent: $\text{NODE} \rightarrow \mathbb{P} \text{ NODE}$

 $\text{dom } CPageNodesContent = CPageNodes$

FilterNodes

FNodes: $\text{iseq } \text{NODE}$

ServletNodes

SNodes: $\mathbb{P} \text{ NODE}$

AllNodes

PageNodes
CompositePageNodes
FilterNodes
ServletNodes

 $PNodes \cap CPageNodes = \emptyset$
 $SNodes \cap CPageNodes = \emptyset$
 $PNodes \cap SNodes = \emptyset$
 $SNodes \cap \text{ran } FNodes = \emptyset$
 $\text{ran } FNodes \cap PNodes = \emptyset$
 $\text{ran } FNodes \cap CPageNodes = \emptyset$

[*PORT*]

PORTTYPE ::= *ENTRY* | *EXIT*

[*URIVAR*]

[*URIVALUE*]

URIVarsandValues

URIVariables: \mathbb{P} *URIVAR*
URIVarsValue: *URIVAR* \rightarrow \mathbb{P} *URIVALUE*
URIVValues: \mathbb{P} *URIVALUE*

$\text{dom } URIVarsValue = URIVariables$

[*FORMLINK*]

AllPorts

Ports: \mathbb{P} *PORT*
AllNodes
URIVarsandValues
AttachedNode: *PORT* \rightarrow *NODE*
PortType: *PORT* \rightarrow *PORTTYPE*
PortURIValue: *PORT* \rightarrow *URIVALUE*
PortURIVar: *PORT* \rightarrow *URIVAR*
PortFormOrLink: *PORT* \rightarrow *FORMLINK*

$Ports = \text{dom } AttachedNode$
 $Ports = \text{dom } PortType$
 $Ports = \text{dom } PortURIVar$
 $Ports = \text{dom } PortURIValue$
 $Ports = \text{dom } PortFormOrLink$
 $\text{ran } PortURIVar = URIVariables$
 $\text{ran } AttachedNode \subseteq PNodes \cup CPageNodes \cup \text{ran } FNodes \cup SNodes$
 $\forall a: Ports \mid PortType a = ENTRY \cdot AttachedNode a \notin PNodes$
 $\forall a: Ports \mid PortType a = EXIT \cdot AttachedNode a \notin CPageNodes$
 $\forall a: Ports \mid \exists b: URIVariables \cdot PortURIVar a = b$
 $\cdot \neg (\exists c: URIVValues \cdot PortURIValue a = c)$
 $\forall a: Ports \mid \exists b: URIVValues \cdot PortURIValue a = b$
 $\cdot \neg (\exists c: URIVariables \cdot PortURIVar a = c)$
 $\forall a: Ports \mid \neg (\exists b: URIVariables \cdot PortURIVar a = b)$
 $\cdot \exists c: URIVValues \cdot PortURIValue a = c$
 $\forall a: Ports \mid \neg (\exists b: URIVValues \cdot PortURIValue a = b)$
 $\cdot \exists c: URIVariables \cdot PortURIVar a = c$
 $\forall a: Ports \mid PortType a = ENTRY \cdot \exists b: URIVValues \cdot PortURIValue a = b$

InNodeConnections

AllPorts

InNodePortConnections: $PORT \leftrightarrow PORT$

GlobalVarSet: $PORT \times PORT \rightarrow URIVAR \times URIVALUE$

$\forall a$: *InNodePortConnections*

- *first a* $\in Ports$
 \wedge *second a* $\in Ports$
 \wedge *AttachedNode* (*first a*) = *AttachedNode* (*second a*)
 \wedge *PortType* (*first a*) = *ENTRY*
 \wedge *PortType* (*second a*) = *EXIT*

[*METHOD*]

[*FUNCTION*]

[*FUNCTIONRESULT*]

MethodsAndFunctions

InNodeConnections

Methods: $\mathbb{P} METHOD$

Functions: $\mathbb{P} FUNCTION$

MethodPorts: $PORT \times PORT \rightarrow METHOD$

FunctionPorts: $PORT \times PORT \rightarrow FUNCTION \times FUNCTIONRESULT$

Methods = ran *MethodPorts*

Navigation

AllPorts

Links: $PORT \leftrightarrow PORT$

$\forall a$: *Links*

- *first a* $\in Ports$
 \wedge *PortType* (*first a*) = *EXIT*
 \wedge *second a* $\in Ports$
 \wedge *PortType* (*second a*) = *ENTRY*
 \wedge *PortURIValue* (*first a*) = *PortURIValue* (*second a*)
 \wedge *AttachedNode* (*first a*) \neq *AttachedNode* (*second a*)

Connection

Navigation

InNodeConnections

ConnectingPorts: $PORT \leftrightarrow PORT$

ConnectingNodes: $NODE \leftrightarrow NODE$

$(\forall a, b: Ports \mid (a, b) \in Links \vee (a, b) \in InNodePortConnections$

- $(a, b) \in ConnectingPorts$)

$\vee (\forall a, b: Ports$

$\mid \exists c, d: Ports$

- $(a, c) \in ConnectingPorts$

$\wedge (d, b) \in ConnectingPorts$

$\wedge (c, d) \in InNodePortConnections \cdot (a, b) \in ConnectingPorts$)

$\forall a: ConnectingPorts$

- $AttachedNode (first\ a) \neq AttachedNode (second\ a)$

$\Leftrightarrow (AttachedNode (first\ a), AttachedNode (second\ a)) \in ConnectingNodes$

Init

AllPorts

$PNodes = \emptyset$

$CPageNodes = \emptyset$

$SNodes = \emptyset$

$FNodes = \diamond$

$Ports = \emptyset$

EmptyNavigation

$\Delta Navigation$

$Links' = \emptyset$

$Ports' = Ports$

$PNodes' = PNodes$

$FNodes' = FNodes$

$CPageNodes' = CPageNodes$

$SNodes' = SNodes$

$AttachedNode' = AttachedNode$

$PortType' = PortType$

$PortURIValue' = PortURIValue$

$PortURIVar' = PortURIVar$

SequenceFun

before: $PORT \leftrightarrow PORT$

AllPorts

$\forall a$: *before* • *first* $a \in Ports \wedge$ *second* $a \in Ports$

$\forall a, b$: *Ports* | *AttachedNode* $a \in PNodes \wedge$ *AttachedNode* $b \notin PNodes$

• $(a, b) \in$ *before*

$\forall a, b$: *Ports* | *AttachedNode* $a \notin CPageNodes \wedge$ *AttachedNode* $b \in CPageNodes$

• $(a, b) \in$ *before*

$\forall a, b$: *Ports* | *AttachedNode* $a \in SNodes \wedge$ *AttachedNode* $b \in SNodes$

• $(a, b) \in$ *before*

$\forall a, b$: *Ports* | *AttachedNode* $a \in \text{ran } FNodes \wedge$ *AttachedNode* $b \in SNodes$

• $(a, b) \in$ *before*

$\forall a, b$: *Ports*

| *AttachedNode* $a \in \text{ran } FNodes$

\wedge *AttachedNode* $b \in \text{ran } FNodes$

$\wedge (\exists i, j: \mathbb{N} \mid FNodes\ i = \text{AttachedNode } a \wedge FNodes\ j = \text{AttachedNode } b$

• $i < j$) • $(a, b) \in$ *before*

UpdateNavigation

Δ *Navigation*

SequenceFun

$\forall a, b$: *Ports*

| $(a, b) \in$ *before*

\wedge *PortURIValue* $a = \text{PortURIValue } b$

\wedge *PortType* $a = EXIT$

\wedge *PortType* $b = ENTRY$

$\wedge (\forall c$: *Ports*

• $((a, c) \in$ *before*

$\wedge (c, b) \in$ *before*

\wedge *PortType* $c = ENTRY$

\wedge *PortURIValue* $c \neq \text{PortURIValue } a)$

$\wedge (a, b) \notin Links \cdot Links' = Links \cup \{(a, b)\}$

NavigationMapGeneration

\exists *Connection*

NavigationMap!: $NODE \leftrightarrow NODE$

$\forall a$: *PNodes*; b : *CPageNodes* • $(a, b) \in$ *ConnectingNodes* $\Leftrightarrow (a, b) \in$ *NavigationMap!*

NavigationMap \cong *EmptyNavigation* § *UpdateNavigation* § *NavigationMapGeneration*

CalculateDependingNodes

Connection

thisNode?: NODE

dependingNodes!: P NODE

$\forall a: \text{dependingNodes!} \cdot (a, \text{thisNode?}) \in \text{ConnectingNodes}$

$\forall a: \text{ConnectingNodes} \mid \text{second } a = \text{thisNode?} \cdot \text{first } a \in \text{dependingNodes!}$

Depending \cong *EmptyNavigation* \S *UpdateNavigation* \S *CalculateDependingNodes*

CalculateDependedNodes

Connection

thisNode?: NODE

dependedNodes!: P NODE

$\forall a: \text{dependedNodes!} \cdot (\text{thisNode?}, a) \in \text{ConnectingNodes}$

$\forall a: \text{ConnectingNodes} \mid \text{first } a = \text{thisNode?} \cdot \text{second } a \in \text{dependedNodes!}$

Depended \cong *EmptyNavigation* \S *UpdateNavigation* \S *CalculateDependedNodes*

FindUnusedNodes

Connection

unusedNodes!: P NODE

$\forall a: \text{PNodes} \cup \text{ran } \text{FNodes} \cup \text{SNodes} \cup \text{CPageNodes}$

$\cdot (\forall b: \text{ConnectingNodes} \cdot a \neq \text{first } b \wedge a \neq \text{second } b) \wedge a \in \text{unusedNodes!}$

$\forall a: \text{unusedNodes!}; b: \text{PNodes} \cup \text{ran } \text{FNodes} \cup \text{SNodes} \cup \text{CPageNodes}$

$\cdot (a, b) \notin \text{ConnectingNodes} \wedge (b, a) \notin \text{ConnectingNodes}$

UnusedNodes \cong *EmptyNavigation* \S *UpdateNavigation* \S *FindUnusedNodes*

FindUnusedPorts

Connection

unusedPorts!: P PORT

$\forall a: \text{Ports}$

$\cdot (\forall b: \text{Ports} \cdot (a, b) \notin \text{ConnectingPorts} \wedge (b, a) \notin \text{ConnectingPorts})$

$\Leftrightarrow a \in \text{unusedPorts!}$

UnusedPorts \cong *EmptyNavigation* \S *UpdateNavigation* \S *FindUnusedPorts*