Separating and Representing Navigation Concerns in Web Applications

Minmin Han
Lehigh University
Bethlehem, PA 18015
1-610-758-6972
mih9@lehigh.edu

Christine Hofmeister
Lehigh University
Bethlehem, PA 18015
1-610-758-4103
Hofmeister@cse.lehigh.edu

ABSTRACT
The navigation concerns of a web application are the parts of the code involved in displaying a web page in response to a user request. Often a request passes through multiple components on the server, each of which may do some processing for the request and/or modify or redirect the request on the way to creating a response. Current approaches to representing this navigation information are severely limited, in that they describe only the request and response, not the intermediate steps. Furthermore, the navigation concerns are typically spread throughout the code, making it difficult to trace the path a request takes through the code.

We present an approach that uses aspects to separate these navigation concerns without constraining the application structure to suit the navigation. We also present a new representation for navigation concerns: a navigation routing diagram, which presents a rich but concise description of the navigation. The diagram can be easily extracted from the aspect code, and conversely it is straightforward to create these aspects given a navigation routing diagram. With our approach it is easy to locate the navigation code and easy to understand the navigation concerns, thus greatly improving the maintainability of web applications.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques.

General Terms
Design, Documentation, Languages.

Keywords
Separation of concerns, aspect-oriented programming, navigation map, web application, object-oriented design.

1. INTRODUCTION
Web applications are significantly more complex than web sites: the latter support an interlinked set of web pages, some of which may contain chunks of code that execute on the client side (on the user's machine). Web applications also interact with the user via web pages, but 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.

In both cases the user "navigates" through the web pages, so a *navigation map* describes the possible paths through the pages. For web sites the navigation information is embedded in the web pages themselves, but for web applications the components on the server side can and typically do modify and redirect requests, making the resulting navigation very difficult to discern.

Understanding the navigation of a web application is crucial for doing most kinds of maintenance on these applications. It is used when external links embedded in pages change, when the content of a page changes, when the processing required for a page request changes, and when the navigation itself changes.

To illustrate the navigation-related code or *navigation concerns* we use Duke's Bank [26], an application developed by Sun to explain how to use their J2EE technology for developing web applications.[1] Figure 1 shows a small portion of the navigation map for Duke's Bank. It shows that the user navigates from page transferFunds_cpage to page transferAck_cpage in order to transfer money between accounts, and from page atm_cpage to page atmAck_cpage in order to get money from one account.

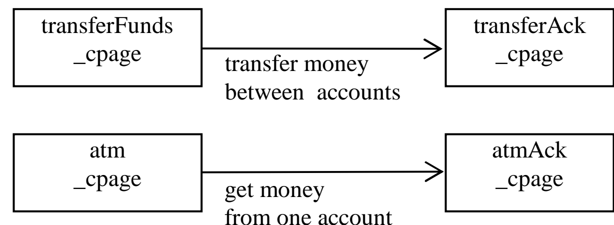

Figure 1 Navigation Map

In order to create such a navigation map we must start with each page request and trace its path through the source code until a response page is displayed. For this part of Duke's Bank we start with transferFunds_cpage, a composite page consisting of banner_page.jsp for displaying header information, link_page.jsp for managing the links that are available on every page, and transferFunds_page.jsp for the body part. (transferAck_cpage contains the same banner and link parts, with transferAck_page.jsp as the body part.)

The starting point for tracing a request is the link or form from which it originates. The first part of Figure 2 shows how the form in transferFunds_page.jsp is defined. Its name is "transfer_form" and its action or target URL is "transferAck_cpage". In the simplest case the target URL is simply the name of the response

[1] For this paper we appended classification information to the names used in Duke's Bank. Forms end with "_form"; pages end with "_page"; composite pages end with "_cpage".

page, but in Duke's Bank as in most web applications it identifies an intermediate target URL that may later be mapped to the final target URL.

```
<form name="transfer_form" method="post"
        action="transferAck_cpage" >
```

from transferFunds_page.jsp

```
<servlet-mapping>
  <servlet-name>Dispatcher</servlet-name>
  <url-pattern>/transferAck_cpage</url-pattern>
</servlet-mapping>
```

from web.xml

```
public void doPost (HttpServletRequest req,
HttpServletResponse res) {
    // Get intermediate target URL.
    String selectedScreen = req.getServletPath();

    if (selectedScreen.equals("transferAck_cpage")) {
    …// code related with the transfer action
    }else  if (selectedScreen.equals("atmAck_cpage")) {
    …// code related with the atm action
    }
    …
    try {
      // The following is to display a composite page.
      // The name of the composite page comes from
      // "selectedScreen".
        goToCompositePage(selectedScreen);
       }
    …
}
```

from Dispatcher.java

Figure 2 Selected Source Code from Duke's Bank

The next step for J2EE applications is controlled by a file named web.xml. If any of the servlet or filter classes named in this file map to the target URL, these requests are routed to the appropriate server or filter class (the first one in the file that matches the target URL). The middle of Figure 2 shows that the Dispatcher servlet receives requests whose target URL is "transferAck_cpage".

A servlet is a special type of class. It implements doGet() and doPost() methods in order to receive incoming requests. Its instantiation is handled by the web server, which typically creates one instance to handle all requests from all users. Thus the next step for tracing the navigation is to look at the doGet() and/or doPost() methods in Dispatcher.java.

The bottom of Figure 2 shows that the Dispatcher's doPost() method first retrieves the request's target URL, then if the value is "transferAck_cpage" it does some processing and forwards the request to "transferAck_cpage." (In this case the Dispatcher does not change the name of the target URL, but clearly it could.)

After tracing through three files and locating the navigation-related code, we finally understand the navigation of this part of Duke's Bank, which is a comparatively simple example. To trace the navigation in more complicated web applications, we may have to go through a number of filters and servlets and look at a number of utility classes.

Once the navigation has been traced, the results are typically documented in a navigation map such as Figure 1. However, the limited information described in such a diagram is insufficient when changes must be made to the application.

If a developer wants to add an extra field to transfer_form, he/she must know which files besides transferFunds_page.jsp could be affected. Dispatcher may need to be changed because it accepts requests from transferFunds_page.jsp. If transferAck_cpage changes its name to safeTransferAck_cpage, all three files are potentially affected.

The traditional navigation map provides insufficient information for these kinds of maintenance activities. This forces the developer to repeat the laborious navigation tracing. One way to address this problem is to improve the representation of the navigation concerns. We propose a *navigation routing diagram* for this purpose. Figure 3 shows such a diagram for Duke's Bank.

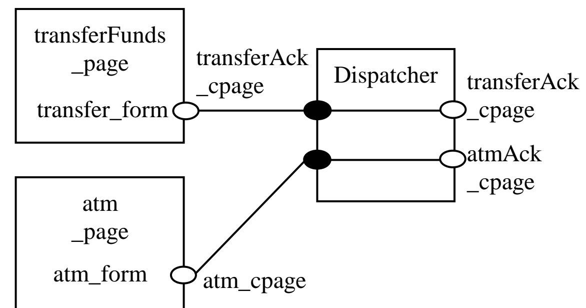

Figure 3 Original Duke's Bank Navigation Routing Diagram

The navigation map shows how a user can navigate through the web pages, so the elements in the map are typically the composite pages displayed to the user. The navigation routing diagram shows how requests are handled by the server-side software, and how the responses are generated. Since the requests typically originate from some part of a composite page, these parts are the elements at the left side of the diagram.

For a well-designed J2EE application, navigation concerns are usually located in the JSP pages, filter classes and servlet classes. [28] Filters are similar to servlets, but they implement a doChain() method rather than doGet() and doPost(), and they typically precede servlets in the request processing. JSP pages are automatically transformed into Java classes that produce web pages.

Thus the boxes in a navigation routing diagram represent JSP pages, filter classes, and servlet classes. A white oval is an exit point for the requests. It is attached to a box to indicate that a request may be generated/forwarded from that page/class and the target URL of the request is shown to the right of the oval. A black oval is an entry point for a request. It is attached to a box to indicate that requests with the specified target URL will be received by that class. So only white and black ovals with the same label will be connected. The exit points for JSP pages have

an additional label inside the box indicating the name of the form or link that generates the request.

Inside a filter or servlet class there may be a line connecting an entry point to an exit point, indicating that requests coming in with a particular target URL are forwarded to the target URL shown on the exit point. It is understood that some processing may occur inside the filter or servlet.

Completing the navigation information is a table specifying the parts of each composite page (Table 1). This is needed because in the navigation routing diagram the requests originate from a page part but the response pages (the unconnected white ovals) are often composite pages. We don't discuss the details of page composition in this paper; we put the page composition information in a table and build a function to forward the request to a composite page for display. Usually page composition information is in an XML table or a JSP page, and it is easy to locate and understand.

Table 1 Duke's Bank Composition Table

	Banner	**link**	**Body**
transfer _cpage	banner_page .jsp	link_page .jsp	transfer _page.jsp
transferAck _cpage	Banner_page .jsp	link_page .jsp	transferAck _page.jsp

So Figure 3 shows that requests coming from the transfer_form of the transferFunds_page are sent to the Dispatcher with a target URL of "transferAck_cpage". The Dispatcher receives this request and forwards it with the target URL "transferAck_cpage". No other servlet or filter class continues to process this request, so this is the final target URL. Thus the response is to display a composite page transferAck_cpage.

It is now clear to the developer that if a new field is added to transfer_form, all modules along the navigation routine could be affected. Here both the Dispatcher and transferAck_cpage must be updated. If transferAck_cpage changes its name to "safeTransferAck_cpage," the target URL changes. The developer can keep the target URL of transferFunds_page the same but change the target URL coming from Dispatcher, or can change them both. Of course the composition table must also be updated.

Thus the navigation map is insufficient for representing navigation concerns. In this paper we propose the navigation routing diagram instead. However, if it is derived and updated by a developer manually reading the code, it is prone to errors perhaps made when creating the diagram, but more commonly made when the code is updated but the diagram is not.

Rather than attempting to automatically extract the navigation concerns from web applications written in the traditional way, we will instead show how to separate out the navigation concerns, describing them in the source code in such a way as to establish traceability between the navigation concerns in the code and their representation in a navigation routing diagram. We use aspect-oriented programming (AOP) to achieve this separation of concerns, then show how the resulting aspect module is traced to the navigation routing diagram and vice versa.

The paper is structured as follows. Section 2 briefly explains how we write aspects in AspectJ. Section 3 discusses two approaches for separating navigation concerns using the Duke's Bank example. Section 4 discusses the correspondence between the navigation routing diagram and aspect modules using a more complex example, Petstore. Section 5 discusses related work, and Section 6 concludes the paper.

2. ASPECTJ

Aspect-oriented programming [10] is a technique for separating concerns at the source level when traditional procedural or object-oriented programming techniques do not suffice. The source code related to a particular concern is placed in aspect modules, and a tool called an aspect-weaver interjects them into the rest of the source code. The places where the aspects should be interjected are also specified in the aspect modules.

Because we are working with J2EE web applications, we use AspectJ [9] an aspect programming language for Java applications. In AspectJ the two parts of the aspect module are: advice, which contains the source code related to the concern; and pointcuts, which specify where the advice should be interjected.

The first part of the aspect module in Figure 4 shows an example pointcut. This pointcut, named "dispatcherPoint", specifies a set of methods: those in Dispatcher where the name starts with "do" and having two parameters of type HttpServletRequest and HttpServletResponse. The first * matches private, protected or public methods, and the second * is a wildcard for part of the method name. Finally, the parameters of dispatcherPoint and args(req,res) mean that the two arguments of a matching method may be used in the advice. There are other ways to define a pointcut, but in this paper we will define pointcuts like this one.

The second part of the aspect module in Figure 4 shows an example of "around" advice. The parameters of the advice allow its body to access the pointcut parameters and the target object (the Dispatcher). In the woven code (Figure 5), the body of the advice is used "around" (instead of) code block B or C. The proceed() statement in the advice means the contents of the original method (block B or C) is executed after block A ends. Of course it's possible to not "proceed()", so that the original code is not executed at all.

3. USING ASPECTS FOR NAVIGATION CONCERNS

This section describes the two approaches we used for separating navigation concerns using aspects.

3.1 First Approach: Encapsulate

The first approach gathers all navigation-related code together into a single aspect module.

The first step is to remove the navigation related code from the JSP files. As we discussed before, the navigation information in a form is its action value (target URL). Since every form and link must have a target URL, we cannot simply omit this information. Instead we use the same generic target URL for every form and link in the web application, e.g. "FrontController". Then we add

```
pointcut dispatcherPoint (HttpServletRequest req, HttpServletResponse res): execution(* Dispatcher.do*(HttpServletRequest,HttpServletResponse))&& args(req,res);
```

```
void around (Dispatcher dp, HttpServletRequest req, HttpServletResponse res) : dispatcherPoint(req,res) && target(dp){
…// code block A
proceed( );
}
```

Aspect Module

```
public void doPost (HttpServletRequest req, HttpServletResponse res) {
    … //code block B
}

public void doGet (HttpServletRequest req, HttpServletResponse res) {
    … //code block C
}
```

from Dispatcher class

Figure 4 An Example of Pre-woven Code

```
public void doPost (HttpServletRequest req, HttpServletResponse res) {
    … // code block A
    ….// code block B
}

public void doGet (HttpServletRequest req, HttpServletResponse res) {
    … //code block A
    …// code block C
}
```

Figure 5 The Results after Weaving

```
<form name="transfer_form" method="post"
        action="FrontController">
   <input type="hidden" name="pagename"
          value="transferFunds_page">
   <input type="hidden" name="formname"
          value="transfer_form">
```

Figure 6 First Approach: Changes to transferFunds_page.jsp

two hidden fields in the form, giving the page name and the form name. Now the request contains no information about navigation but contains information about where it comes from. Later the aspect module can use this information to differentiate requests. Figure 6 shows how transferFunds.jsp is modified.

The second step is to remove the navigation concerns from web.xml. We do this by modifying web.xml to guide every request to a front controller servlet class (following the Façade design pattern [5]).The front controller servlet accepts every request before any other server module works on it. In Duke's Bank, the Dispatcher acts as the front controller, so all requests are first received by the Dispatcher. (See Figure 7)

```
<servlet-mapping>
  <servlet-name>Dispatcher</servlet-name>
  <url-pattern>/FrontController</url-pattern>
</servlet-mapping>
```

Figure 7 First Approach: Changes to web.xml

The third step is to modify the servlet and filter classes. There is only one servlet class in Duke's Bank: the Dispatcher. We turn to the method in the servlet or filter that receives the requests, does optional processing on them, and forwards them. (This is doGet or doPost in a servlet, and doChain in a filter class.) We break up the processing in this method as needed, so that the processing related to each target URL is in a separate method. The remaining navigation-related code is put into the navigation aspect module, leaving the original method empty. We discuss later how the advice in the navigation module calls the new methods in the servlet and filter classes.

```
public void doPost (HttpServletRequest req, HttpServletResponse res) { }

public void doGet (HttpServletRequest req, HttpServletResponse res) { }

public void makeTransfer (HttpServletRequest req, HttpServletResponse res) {
    …// code related with the transfer action
    }

public void makeATM (HttpServletRequest req, HttpServletResponse res) {
    …// code related with the atm action
}
```

Figure 8 First Approach: Dispatcher.java

For Duke's Bank, we created new methods makeTransfer() for requests coming from transfer_form and transferFunds_page, and makeATM() for requests coming from atm_form and atm_page (Figure 8).

The last step is to build the navigation aspect module. It intercepts calls to doPost() and doGet() of the front controller class. It checks where the request comes from, calls the appropriate processing method, then forwards the request to the appropriate target URL.

So before using the aspect-weaver, the navigation concerns are all located in the navigation aspect and it is easy to trace through the navigation. After the aspect is woven into the rest of the source, the navigation-related code is inserted back to into the front controller class.

Figure 9 shows the navigation aspect module for Duke's Bank. A request from the transferFunds_page and transfer_form is

processed by makeTransfer(), then forwarded to a composite page "transferAck_cpage".

```
pointcut dispatcherPoint (HttpServletRequest req,
HttpServletResponse res): execution(*
Dispatcher.do*(HttpServletRequest,HttpServletResponse))
&& args(req,res);

void around (Dispatcher dp, HttpServletRequest req,
HttpServletResponse res) : dispatcherPoint(req,res) &&
target(dp){
 …// get page_name and form_name from the request.
 if ((page_name.equals("transferFunds_page"))
 && (form_name.equals("transfer_form"))) {
    dp.makeTransfer(req,res);
    try {
          goToCompositePage(req, "transferAck_cpage");
          …
     }
 …
}
// A function to display a composite page
void goToCompositePage(…)
```

Figure 9 First Approach: NavigationAspect.java

Advantages and Disadvantages

One advantage of this approach is it is easy to understand. Developers can read one aspect module and understand the navigation of the whole application.

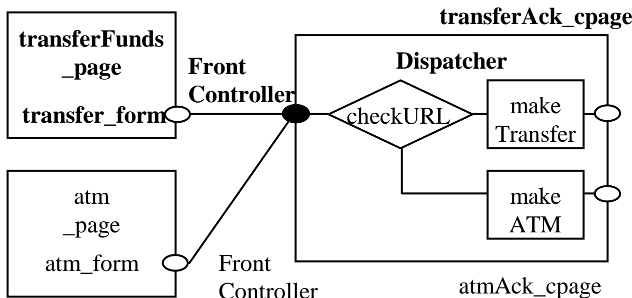

Figure 10 First Approach: Navigation Routing Diagram

There are, however, some problems. First of all, if we compare this navigation routing diagram (Figure 10) with the original (Figure 3), there are a number of differences. All intermediate target URLs are now "FrontController". After weaving, the front controller class decides every part of the navigation. If there are other servlet and/or filter classes besides the front controller class, the front controller may call methods in them to do processing, but they no longer receive any requests. Separating the navigation concerns in this way can simplify future maintenance, but changing the original design to this extent is not so desirable.

Second, if there was no front controller servlet originally, we must create one before we build an aspect module for it. It is not good design to have a front controller class containing only empty methods, simply for the aspect to override these methods. In addition, every request must go through this front controller class, creating an execution bottleneck where there had been none.

Finally, this solution is based on encapsulation. If encapsulation can solve the problem, we should use a component instead of aspect oriented programming [10]. Hannemann and Kiczales have also shown that using AspectJ for the Façade design pattern (the front controller) brings no benefit [6].But even if a component were used instead of an aspect module, the first problem remains.

3.2 Second Approach: Preserve Application Structure

The second approach does not encapsulate the navigation concerns in a single aspect module but instead creates a number of aspects that are woven back into the application in their original locations.

First step is to remove from the JSP pages the navigation-related code (the target URLs) and place these in an aspect. Since AspectJ can't define pointcuts in markup language, we add a function insertFormAction() and call this function wherever we wish to insert aspect code. It accepts two parameters, page name and form name, and returns a string containing the target URL (Figure 11).

```
<form name="transfer_form" method="post"
        action="insertFormAction("transferFunds_page",
                  "transfer_form")">
```

```
public String insertFormAction(String pageName,
String formName){
      // The following line will be replaced after weaving.
      return ("nothing");  }
```

from transferFunds_page.jsp

```
aspect PageNavigation {
   pointcut JSPPageNavigation(String pageName, String
formName): execution(* *.insertFormAction(String,
String))&& args (pageName, formName);

String around (String pageName, String formName):
JSPPageNavigation (pageName, formName) {
  if (pageName.equals("transferFunds_page")) {
         if (formName.equals("transfer_form"))
            return("transferAck_cpage");
       }
   …
}
```

from PageNavigation.java

Figure 11 Second Approach: Page Navigation

Then we define a PageNavigation aspect module that computes the intermediate target URL for all JSP pages. This module catches all calls to insertFormAction() and returns the intermediate target URL. Figure 11 shows that it returns "transferAck_cpage" for the transfer_form in the transferFunds_page. Before weaving, the JSP navigation information is in the aspect module, and after weaving, it replaces the contents of insertFormAction(). Thus after weaving each form or link in a JSP page executes a big function containing all JSP navigation. The aspect could potentially be split into multiple aspects, but this may be more difficult for the developer to follow and the performance impact of the current approach is not large.

The second step is to extract the navigation concerns from the servlet classes. This is done in nearly the same way as in the first approach, except that each servlet has its own aspect, and the aspect identifies a request by its intermediate target URL (as set in the JSP page) rather than by the form and page from which it originated (Figure 12).

```
aspect ServletNavigation {
   pointcut DispathcerNavigation(HttpServletRequest req,
HttpServletResponse res): execution(* Dispatcher. do*
(HttpServletRequest, HttpServletResponse))&&
args(req,res);

   void around (Dispatcher dp, HttpServletRequest req,
HttpServletResponse res) : DispatcherNavigation(req,res)
&& target(dp){
  //Get intermediate target URL.
  String selectedScreen=req.getServletPath();
  if (selectedScreen.equals("transferAck_cpage"))   {
    dp.makeTransfer(req,res);
    try{
       // forward the request and display a composite page
       dp. goToCompositePage(req, "transferAck_cpage");
    }
  }
  …
}}
```

Figure 12 Second Approach: ServletNavigation.java

The third step is to extract the navigation concerns from the filter classes. This is nearly the same process as for servlet classes, except that the pointcut matches doChain() rather than doGet() and doPost() methods.

There is one technical problem in creating aspects for JSP pages: since they contain markup language, AspectJ can't work with them directly. However, all JSP pages are automatically transformed into regular Java classes by the web server and only the resulting servlets are executed. So our current work-around is to modify the generated servlet files rather than the original JSP pages. Figure 13 shows what the generated servlet contains when the JSP contains the script shown in Figure 11.

```
out.write("<form name=\"transfer_form\" method=\"post\"
action=\"" + insertFormAction("transferFunds_page",
"transfer_form")+"\" >\n");
```

Figure 13 transferFunds_page$jsp._jspService(...)

Advantages and Disadvantages

In comparison to the first approach, there is no longer a front controller class responsible for all navigation. In the woven code, the navigation concerns are located in the same places as the original design. But in the pre-woven source files, the aspect modules contain all navigation-related code originally in JSPs, servlets, and filters.

Thus we have preserved the original structure of the application. This can also be seen by looking at the navigation routing diagram (Figure 14). It is identical to the original (Figure 3), except that now we see the use of the new methods makeTransfer() and makeATM() in the Dispatcher.

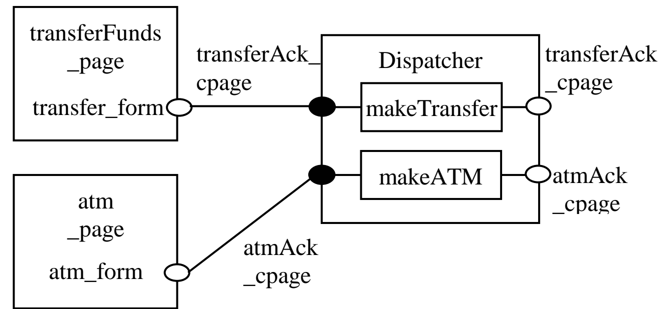

Figure 14 Second Approach: Navigation Routing Diagram

What we do not see in the diagram is that now all the navigation decisions are located in aspect modules or in web.xml. The entry points, exit points, their associated target URLs, and the control flow inside the servlets and filters are described in aspect modules. Connections from the exit point of one box to the entry point of another are described in web.xml.

One possible disadvantage is a developer must examine the web.xml file and a number of aspect modules to understand the navigation, whereas with the encapsulation approach the navigation concerns were consolidated in one aspect. We believe the advantages of this approach far outweigh this disadvantage.

4. TRACING ASPECTS TO DIAGRAMS

Next we show how to build the aspects using a navigation routing diagram and vice versa. We have chosen a significantly more complex example: the Petstore application from Sun's J2EE tutorial [27], focusing on a small portion of this application.

From the shopping cart page, customers can click the "check out" button to start the check out process. If the customer has signed in, the response page is enter_order.screen. If the customer has not signed in, the response page is the sign-on page (signon.screen). Current customers sign in then are taken to enter_order.screen. New customers are taken to create_customer.screen.

4.1 Navigation Routing Diagram

We start by showing the navigation routing diagram. For a new application this diagram would likely be created during design, then the aspects would be written based on the diagram. For converting an existing application (as we did with the Petstore), the developer may first extract the navigation concerns into aspects then derive the navigation routing diagram.

Because the Petstore is more complex than Duke's Bank, we introduce some new elements in the navigation routing diagram (Figure 15). Sometimes the servlet or filter chooses a different target URL for the exit point depending on the results of its processing. For example, a checkout request (enter_order.screen) is forwarded to enter_order.screen if the customer has already signed in, and is forwarded to signon.screen if not. We show this by putting a diamond shape inside the servlet or filter. Inside the diamond is the name of the function whose return value will determine the target URL.

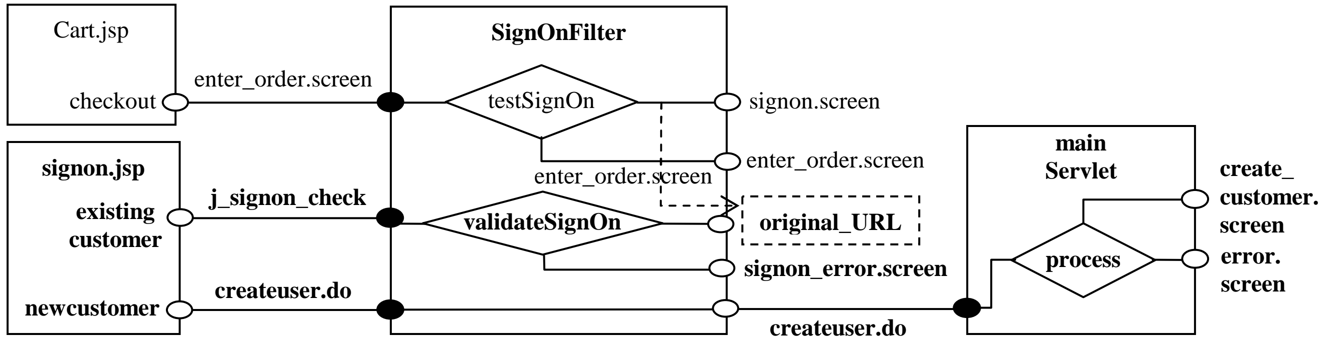

Figure 15 Petstore: Naviagtion Routing Diagram

The target URL of a successful sign-on depends on which page the customer was using before the sign-on. We show this in the diagram with a dashed box, meaning the target URL is a value in a variable. Usually there are dashed arrows pointing to the box showing the setting of this variable. Figure x shows that when an enter_order.screen request goes to signon.screen, part of the processing is to place the value "enter_order.screen" into the variable "original_URL".

As before, we use a composition table to describe composite pages and their parts (Table 2).

Table 2 Petstore Composite Table

composite page name	Banner	Sidebar	body	footer
signon.screen	banner. jsp	sidebar. jsp	signon. jsp	footer. jsp
enter_order. screen	banner. jsp	sidebar. jsp	enter_ order.jsp	footer. jsp
signon_error. screen	banner. jsp	sidebar. jsp	signon_ failed.jsp	footer. jsp
create_ customer. screen	banner. jsp	sidebar. jsp	create_ customer .jsp	footer. jsp
error.screen	banner. jsp	sidebar. jsp	general_ error.jsp	footer. jsp

4.2 From Diagram to Aspects

Now the aspects can be written, using the navigation routing diagram as a guide. The two JSP pages shown in the navigation routing diagram are cart.jsp and signon.jsp. These are built using tag libraries, so they are very different from the JSPs in Duke's Bank.

However, since all we are interested in is the page name, form name, and target URL, the JSP page aspect module is still very similar to the one in Duke's Bank. Figure 16 shows part of this aspect module, the part related to page signon.jsp. In the "around" advice each JSP page shown in the navigation routing diagram appears as an "if" statement checking the page name, then returning the appropriate target URL for each form or link name.

```
aspect JSPPageNavigation{
pointcut PageNavigation(String pageName,String
formName): execution(* *.insertFormAction String,
Sring))&& args(pageName, formName);

String around (String pageName,String formName) :
PageNavigation(pageName,formName) {
    if (pageName.equals("signon"))
    {
      if (formName.equals("existingcustomer"))
        return("j_signon_check");
      if (formName.equals("newcustomer"))
        return ("createuser.do");
    }
    if (pageName.equals("cart"))
    …
  }
}
```

Figure 16 Petstore: JSPPageNavigation.java

To follow the navigation routing diagram, Petstore's web.xml must route all target URLs to a filter class SignOnFilter.

In the original application, for incoming target URL j_signon_check, SignOnFilter validates the sign-on then depending on the result, forwards the request to signon_error.screen or the value stored in variable "original_url."

Now the SignOnFilter class must provide an empty doChain() method, and provide methods for all processing shown inside the SignOnFilter box in the navigation routing diagram (e.g. testSignOn() and validateSignOn()).

The routing is now placed in the aspect module: the pointcut specifies the doChain() method of SignOnFilter, and for each entry point to SignOnFilter, the "around" advice contains an "if" statement that invokes the appropriate method and forwards the request to the page specified on the exit point.

So for j_signon_check, the advice executes validateSignOn(), then depending on the result either forwards the request to signon_error.screen or to the page name stored in original_url (Figure 17).

```
aspect FilterNavigation {
pointcut SignOnFilterNavigation(ServletRequest req, ServletResponse res, FilterChain chain): execution(* SignOnFilter.do*(ServletRequest, ServletResponse, FilterChain)) && args(req,res,chain);

void around (SignOnFilterNew signOnF, ServletRequest req, ServletResponse res, FilterChain chain) : SignOnFilterNavigation (req,res,chain) && target (signOnF){
        …//get targetURL from the request
        try{
          if (targetURL.equals("j_signon_check")) {
                returnValue=signOnF.validateSignOn
                                (req,res,chain);
                if (returnValue==1) {
                  targetURL = (String)hreq.getSession().
                      getAttribute ("original_url");
                  hres.sendRedirect(targetURL);
                }
                else
                  hres.sendRedirect("signon_error.screen");
          }
        …
        }
}}
```

Figure 17 Petstore: FilterNavigation.java

```
aspect ServletNavigation {
pointcut MainServletNavigation(HttpServletRequest req, HttpServletResponse res): execution(* MainServlet.do* (HttpServletRequest, HttpServletResponse))&& args(req, res);

void around (MainServlet mainS, HttpServletRequest req, HttpServletResponse res) : MainServletNavigation(req,res) && target(mainS){
        // get targetURL from the request
        if (targetURL.equals("createuser.do")) {
          try{
            returnValue=mainS.process(req,res);
            if (returnValue==1)
              req.getRequestDispatcher("create_customer
                  .screen").forward(req,res);
            else
              req.getRequestDispatcher("error.screen").
                  forward(req,res);
          }
        …
        }
        …
} }
```

Figure 18 Petstore: ServletNavigation.java

The last aspect module needed is the servlet navigation module. According to the navigation routing diagram, requests with the target URL “createuser.do” will be forwarded to MainServlet. Petstore’s web.xml forwards all those ending with “.do” to MainServlet.

As for all servlets shown in the navigation routing diagram, the MainServlet class must provide empty doGet() and doPost() methods, and provide methods for all processing shown inside the servlet box in the diagram.

The routing is located in a servlet navigation aspect module (Figure 18). In this case the pointcut specifies the doGet() and doPost() methods of MainServlet, and for createuser.do it invokes process(), then forwards the request to either create_customer.screen or error.screen.

These final target URL do not end with “.jsp” so they are not actual page names. Petstore does not use a “goToCompositePage” function but instead a servlet class that accepts a target URL as the composite page name and displays the composite page. Again, page composition is not included in the navigation map but in the composition table.

4.3 From Aspects to Diagram

To go the other direction, creating a navigation routing diagram from the navigation aspects and web.xml file, the following four steps are used.

The first step is to build the page elements. Pages, forms, related exit points and target URLs can be found in the page navigation aspect. From PageNavigation.java in Petstore (Figure 16), we build a page element “signon.jsp”, two forms “existingcustomer” and “newcustomer”, two exit points for these two forms and two related target URLs “j_signon_check” and “createuser.do”.

The second step is to build filter elements. Filters, related entry points, exit points, target URLs and the processing methods used can be found in the filter navigation aspects. From FilterNavigation.java in Petstore (Figure 17), we build a filter element “SignOnFilter”, an entry point for incoming requests with target URL “j_signon_check”, a diamond for method “validateSignOn” then two exits points with target URLs “original_URL” (a variable) and “signon_error.screen”.

The third step is to build Servlet elements. This step is very much like the second step except now we look in the servlet navigation aspects. So for Petstore, we check ServletNavigation.java (Figure 18). For MainServlet, we build an entry point with target URL “createuser.do”, a diamond for the “process” method, and two exit points with target URLs “create_customer.screen” and “error.screen”.

The last step is to connect the exit points of pages and/or filters to entry points of filters and/or servlets. This information comes from web.xml. Only the exit points and entry points with same target URL can be connected.

4.4 From Diagram to Navigation Map

It is also easy to extract a navigation map from a navigation routing diagram. To do so, we trace the requests from forms or links in JSPs through servers and filters to the target URL. The navigation map shows only these starting and ending points, and shows them as pages presented to the user. In a navigation routing diagram the target URL is the page presented to the user, but the JSPs at the left of the diagram may be only part of the page presented to the user. We use the composition table to find the

enclosing composite page. The arrow leaving the composite page can be annotated with the form or link name (Figure 19).

Notice that in Figure 19 there are no symbolic target URLs such as “original_URL” shown in the dashed box in Figure 15. In this situation the navigation map must show all possible target URLs this variable could hold. (There are many in Petstore, but in the portion we have shown only enter_order.screen is used.)

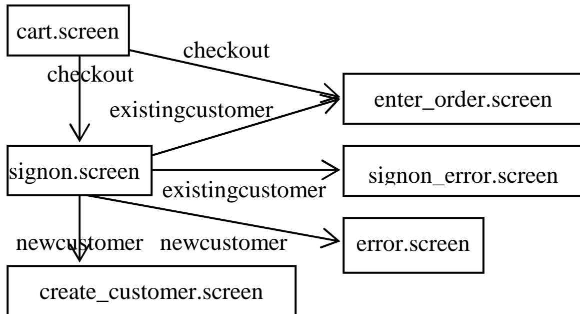

Figure 19 Navigation Map for Petstore

5. RELATED WORK

Others have applied AOP to web applications but not for separating navigation concerns. Reina et.al. [22] follow the proposals of Kilesev [12] 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. Kerston et.al. [8] present their experience using AspjectJ on the Atlas system, a web-based educational system. They successfully separate network context concerns in a distributed server broker framework.

It is clear that researchers believe navigation concerns are an important part of web applications. Rossi [23] identifies a number of common navigation patterns used in web applications. Reina [21] states that it is necessary to separate the navigation concerns into aspect modules but does not present a specific solution.

Another approach is to partially or fully separate the navigation concerns into XML files. Kojarski et.al. [14] build code at the abstract level for web applications. They put lower level design data, form fields, database structure, and navigation in XML files. The original Petstore [27] also puts some navigation information in XML files. The idea is that when navigation changes, developers no longer need to change the source code but instead just the XML files.

However, we believe these solutions are harder to understand than ours. Sometimes it is difficult because part of the navigation is in source code while the rest is in XML files, such as in the Petstore. Even when all navigation information is in XML files, there is usually no one specific XML file for navigation information. Developers must read through all source files and XML tables to understand the navigation. In addition, it would be very difficult to turn an existing non-XML based web application into one that is XML-based.

There is no one standard notation for a navigation map. 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[13], the navigation schema in Autoweb system[4], and the navigation diagram in RMM[7]. In our work we want the navigation model or representation to correspond much more closely to the source code.

More typically a navigation map is quite similar to the “Architecturally Significant Navigation Map” in Pearl Circle[20], 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[18], the ADM-d model for “data-intensive” web sites in the Araneus approach [17], the navigation diagram in WebML[2] and W3DT[1], “Navigational Contexts Schema” in OOHDM[24], and “WebApp Top-Level Navigational Charts” by Enguix[3].

Another type navigation map is presented by Leung et.al. [15]. 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 Navigation Routing Diagram does. Our diagrams provide enough information that developers do not need to trace the navigation each time there is an update requirement in the web application.

To the best of our knowledge, there is no other similar use of AspectJ for separating navigation concerns. AspectJ has been applied to many different kinds of applications to separate other kinds of concerns, such as concurrency and failures [11], distribution and persistence [25], and exception detection and handling [16]. Most of the results are considered a success and better than a pure Java implementation.

Early on Kiczales et.al. [10] defined two rules for using AOP. The first is: when concerns can be clearly encapsulated into one module, a component should be used instead of an aspect module. The second is: when the first rule does not apply, an aspect module should be used. Our results from the two approaches we used confirms these rules.

Also we believe an aspect-oriented solution should not dictate the structure of the application. AOP is for separating a concern into an aspect module so that developers can easily locate and understand the concern. When converting an existing application as we did, the code after weaving should be very similar to the original application. This is also desirable because the less we change the original design, the less likely we are to introduce errors.

Walker et.al. [29] describe two exploratory experiments where two groups of participants used AOP and OO approaches. Their two key insights were: 1) the scope of aspect should be well-defined and 2) using AOP may alter programming and debugging strategies. In our research, the scope of aspect is clear: navigation. The navigation concern is not tightly coupled with other concerns.

Kienzle [11] concludes that AOP can be good for code factorization but only for experienced programmers. Also since concurrency and failures are simulated by objects (not part of the

object semantics), they are very difficult to aspectize. Our results are that navigation is not simulated by objects and is not difficult to aspectize if our second approach is followed. We do not expect developers to be experienced with AOP. Developers can easily create and understand navigation routing diagrams, and there is a systematic technique for creating the aspect modules from these diagrams.

Murphy et.al. [19] present a study on applying separation of concern mechanisms to two common scenarios: separating concerns tangled within a method and between classes. They conclude that a concern may be easier to separate if it is encapsulated in methods and classes or if its code is in contiguous chunks. We use a similar idea in our second approach. For doPost/doGet/doChain, we first gather the processing for each intermediate target URL and put it into a separate method, such as "makeTransfer". After that we put the remaining navigation-related code into aspect modules.

6. CONCLUSION

In this paper we presented two approaches to using aspects for separating navigation concerns in J2EE web applications. The first involves encapsulating all navigation-related code in one module. This solution is weak because aspects are not needed when concerns are already encapsulated, but more importantly placing all navigation concerns in a separate module causes the application (or parts of it) to be structured to suit the navigation concerns. The benefit of using aspects is that concerns can be separated without their dictating the structure of the application.

The second, better approach uses aspects and a web.xml file to specify the navigation concerns but does not constrain the application structure. We have applied this approach to a number of existing applications, converting them without altering their original structure. The applications are examples provided with the J2EE server 1.4, including:

- four Hello applications,
- two Duke's Bookstore applications,
- and the Duke's Bank and Petstore applications presented in this paper.

These applications use different approach for navigation: some with a Front Controller and some without, some with multiple filters and servlets, some with JSPs only, some with servlets only, and some using XML for part of the navigation. Our second approach works for all of them, and in all cases the original application structure was preseved.

Our second approach is also easy to apply. The JSP pages need little change: to add an empty function and replace the target URLs in forms or links with a function call. There are some changes in the servlet and filter classes, to split the processing in the doGet/doPost/doChain method into separate methods for processing each type of request. However we believe this kind of work is not only good for the purpose of separating the navigation concerns but also good for future maintenance of the web application.

Writing the aspects themselves is straightforward because they follow a simple pattern and can be derived directly from a navigation routing diagram.

The navigation routing diagram is a new approach we proposed for representing navigation concerns. Traditional navigation maps are useful for presenting the user's view of the application, but do not contain the kind of information a developer needs in order to perform maintenance.

The navigation routing diagram is easy to understand and easy to create once the navigation is understood. We showed that it is easily derived from the navigation aspects and the web.xml file. We also showed how a navigation routing diagram can be used to guide the creation of aspects and the web.xml file.

Because it is so straightforward to map one to the other, we believe it would be possible to automatically generate a navigation routing diagram given the aspects and web.xml. Conversely, it looks feasible to generate most of the aspects and web.xml from the diagram. This automatic generation is planned for future work.

7. REFERENCES

[1] Bichler, M. and Nusser, S. Developing structured WWW-sites with W3DT. In *Proceedings of the WebNet - World Conference of The Web Society* (USA, October 16 -19, 1996)

[2] Ceri, S., Fraternali, P. and Bongio, A. Web Modeling Language (WebML): a modeling language for designing Web sites. In *Proceedings of the 9th International WWW Conference on Computer Network.* (Amsterdam, The Netherlands, 2000) North-Holland Publishing Co. Amsterdam, The Netherlands, 2000, 137-157.

[3] Enguix, C.F. and Davis, J.G. 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) 1999.

[4] Fraternali, P. and Paolini, P. Model-Driven Development of Web Applications: The Autoweb System. ACM Transactions on Information Systems, 28, 4, (Oct 2000), 323-382.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[6] Hannemann, J. and Kiczales, G. Design pattern implementation in Java and AspectJ. In *Proceedings of 17th Annual ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)* (Seattle, Washington, November 4-8, 2002) ACM Press, New York, NY, 2002, 161-173.

[7] Isakowitz, T., Stohr, E.A. and Balasubramanian P. RMM: A Methodology for Structured Hypermedia Design. *Communications of the ACM*, 38, 8, (Aug. 1995).

[8] Kersten, M. and Murphy G.C. Atlas: a Case Study in Building a Web-based Learning Environment Using Aspect-Oriented Programming. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 99)* (Denver, Colorado, November 1-5, 1999). ACM Press, New York, NY, 2003, 340-352.

[9] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Pal J., and Griswold, W.G. An Overview of AspectJ. J. In

Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01) (Budapest, Hungary, June 18-22, 2001) Springer-Verlag LNCS 2072, 2001, 327-353.

[10] Kiczales, G., Lamping J., Mendhekar A., Maeda C., Lopes, C. V., Loingtier, J. and Irwin, J. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)* (Finland, June 1997). Springer-Verlag LNCS 1241, 1997, 220-242.

[11] Kienzle, J. and Guerraoui, R. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '02)* (Malaga, Spain, June 10-14, 2002) Springer-Verlag LNCS 2374, 2002, 37-61.

[12] Kilesev, I. *Aspect-Oriented Programming with AspectJ.* Sams Publishing, 2002.

[13] Koch, N. and Kraus, A. 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) 105-120.

[14] Kojarski, S. and Lorenz, D.H. Domain Driven Web Development with WebJinn. In *Special Track on Domain Driven Development, International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA '03)* (Anaheim, California, October 26-30, 2003). ACM Press, New York, NY, 2003, 53-65.

[15] Leung, K.R.P.H., Hui, L.C.K., Yiu, S.M., and Tang, R.W.M. Modelling Web Navigation by Statechart. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC '00)* (October 2000).

[16] Lippert, M. and Lopes, C.V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)* (Limmerick, Ireland, June, 2000) ACM Press, New York, NY, 2000.

[17] Merialdo P. and Atzeni P. Design and Development of Data-intensive Web Sites: The Araneus Approach. ACM Transactions on Internet Technology, 3, 1, (Feb 2003), 49-92.

[18] Microsoft Visual InterDev Reference. http://msdn.microsoft.com/libarary.default.asp?url=/library/en-us/vidref98/html/dvconsitedesign.asp

[19] Murphy, G.C., Lai, A., Walker, R.J., and Robillard, M.P. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)* (Toronto, Canada, May 12-19, 2001) ACM Press, New York, NY, 2001, 275-284.

[20] Rational Software. Pearl Circle Online Auction Reference Application Reference Application Software Architecture Document, Issue 0.2, Rational Software 2001.

[21] Reina, A.M. and Torres, J. Analysing the Navigational Aspect. *In Proceedings of Second Aspect-Oriented Software Development Workshop (AOSD '02)* (Germany, February 2002).

[22] Reina, A.M., Torres, J. and Toro, M. Aspect-Oriented Web Development vs. Non Aspect-Oriented Web Development. In *Workshop of Analysis of Aspect-Oriented Software (AAOS '03)* (Darmstadt, Germany, July 2003).

[23] Rossi, G., Schwabe, D. and Lyardet, F. Improving Web Information Systems with Navigational Patterns. In *Proceedings of the 8th International Conference on WWW* (Toronto, Canada, 1999) Elsevier North-Hooland, Inc., New York, NY, 1999, 1667-1678.

[24] Schwabe, D. and Rossi, G. An Object Oriented Approach to Web-Based Application Design. Theory and Practice of Object Systems 4, 4, 1998. Wiley and Sons, New York.

[25] Soares, S., Laureano, E. and Borba, P. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of 17th Annual ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)* (Seattle, Washington, November 4-8, 2002) ACM Press, New York, NY, 2002, 174-190.

[26] Sun Java Center. The Duke's Bank Application. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Ebank.html

[27] Sun Java Center. Java Petstore 1.1.2. http://java.sun.com/developer/releases/petstore/petstore1_1_2.html

[28] Sun Java Center. Java Servlet Technology. http://java.sun.com/products/servlet/

[29] Walker, R.J., Baniassad E.L.A. and Murphy, G.C. An Initial Assessment of Aspect-oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 21)* (Los Angeles, California, May, 1999) ACM Press, New York, NY, 1999, 120-130.