Separation of Navigation Routing Code in J2EE Web Applications

Minmin Han and Christine Hofmeister

Computer Science and Engineering Dept, Lehigh University
19 Memorial Dr. W., Bethlehem, PA 18015, USA
mih9@lehigh.edu, Hofmeister@cse.lehigh.edu

Abstract. The navigation routing code of a web application is the part of the code involved in routing a request from a web page through the appropriate components on the server, typically ending with the display of a response page. Common maintenance activities are to change the sequence of pages or the processing for a page, and for these activities the navigation routing code must be located, understood, and possibly modified. But in J2EE applications this code is spread among a number of components, making maintenance costly. We describe an approach for separating this navigation routing code, using either Aspect Oriented Programming (AOP) or conventional OO techniques. We demonstrate how this improves maintainability by converting three exemplar applications from Sun and Oracle, with a 4- to 11-fold reduction in the number of files containing navigation routing code and in the lines of code in these files.

1 Introduction

A navigation map, which describes the possible sequences of web pages displayed to a user, is typically part of the documentation of a web application. But to get from one page to another, a request from one page is usually routed through a series of components on the server, ending with the display of the response page. We define the *navigation path* of a request to be the sequence of components that handle this request. Then we define an application's *navigation routing* to be the union of all possible navigation paths for requests in that application. Navigation routing must be understood and potentially modified when the navigation map changes, when external links embedded in pages change, when the content of a page changes, or when processing required for a request changes.

Since changes of these types are common in web applications, the developer should be able to efficiently understand and modify the navigation routing. However, existing web application platforms such as J2EE make this difficult. One problem is the complexity of the navigation routing code: it is usually scattered through many components, interspersed with processing code, and uses a wide variety of techniques for routing a request. A second problem is that the platform invokes the server components implicitly, using information in a deployment descriptor file (web.xml in J2EE). This requires the developer to have a thorough understanding of the platform's navigation routing rules in order to follow the navigation routing.

In this paper we address the first problem, the complexity of the navigation routing code, by presenting an approach that

- separates the navigation routing from the rest of the application, and
- uses a small set of implementation conventions to reduce the variability in navigation routing code.

D. Lowe and M. Gaedke (Eds.): ICWE 2005, LNCS 3579, pp. 221–231, 2005.

Even in well-designed J2EE applications, navigation routing code is located in several types of components: JSPs (Java Server Pages), filter classes, servlet classes, and a deployment descriptor file (web.xml) [18].

The starting point of a navigation path is a request originating from a link or form in a JSP page. The "target URI" of the request determines where the request is routed. In the simplest case the target URI is another JSP page that is displayed by J2EE server. Otherwise the web server looks up the target URI in web.xml in order to determine which filter or servlet will next receive the request.

Upon receipt of a request, a filter or servlet often does some processing for the request, perhaps using the target URI in order to distinguish requests from different JSPs. For example, when the Front Controller design pattern is used [1], a single component receives all requests from the client then passes them to the appropriate component. To pass the request along to another component or display the response page, a component invokes one of several platform methods (chain, forward, sendRedirect), passing along a target URI. Each time the request is passed to another component, web.xml uses the target URI to determine which component should receive the request. Of course, the target URI does not have to be a hardcoded string that is passed immutably through the components. It could come from any expression that evaluates to a string, and could be changed by any component along the navigation route.

All of these variations are possible while still following the recommended design guidelines. But these guidelines are not obligatory, and in addition to having navigation routing code located in any number of JSPs, filters, servlets, and web.xml, it could be located in any number of other classes on the server side. The guidelines do nothing to prevent navigation routing code from being interspersed with code for processing the request, and they do not limit the large variety of ways the target URI can be set.

Our approach for addressing the complexity of navigation routing code separates it from the processing code, locates it in a small number of modules, and does not force a particular structure on the application. For example, some existing approaches put navigation routing in a separate table but require the use of a Front Controller servlet provided by a library. With our approach, requests can be routed through any number of components in any order with any desired target URI.

Separating out the navigation routing code already applies some implementation conventions, which helps reduce the variability of this code, but it still leaves too much variability in how a component can modify a target URI as it processes a request. After looking at existing applications for different ways that a component determines the outgoing target URI for a given incoming target URI, and considering other ways that might be used, we see that these are covered by three categories.

The first is *static routing*, where there is only one possible outgoing URI for a given incoming URI. This could be hardcoded in the component or looked up from a table. It could even be computed by the component, although this is unlikely since in static routing the outgoing URI is determined at development time, not runtime.

The second is *conditional routing*. In this case the set of possible outgoing URIs is determined at development time, but the choice among them is made at runtime. When processing a request the component can use a number of different ways to make the choice, such as using data located in the request, using a session variable,

looking up data in a table or database, using the current date and time, etc. Both static and conditional routing are supported by many current table-driven frameworks.

The third type of navigation routing is *cached routing*. As in conditional routing the set of possible outgoing URIs is known at development time and the choice is made at runtime, but in this case the choice is not made by the component. Instead the choice was made by another component when processing a prior request or possibly earlier in the processing of this request. Thus the outgoing URI is retrieved from a variable in which it was cached earlier. Note that cached routing could be treated as a case of conditional routing where the selection of outgoing URI is made by retrieving it from a variable. However, because this case occurs frequently, creating a third category allows us to preserve the particular semantics of this type of routing.

In our experience these three categories cover all of the variants on navigation routing that occur within web applications. The case where a user enters an arbitrary URI is not relevant because we are interested only in the navigation routing within an application, where the routing choices should be known at development time and only the selection is made at runtime. Having defined these three categories, we can create standard implementation conventions for each, thus greatly reducing the variability in navigation routing code.

In the next section we describe our approach, comparing it to current table-driven frameworks. Section 3 presents our evaluation of the approach. Section 4 describes related work, and Section 5 concludes the paper.

2 Separation of Navigation Routing Code

2.1 Our Approach

Our approach to separating navigation routing code has two variants. Our first version uses aspect-oriented programming (AOP), which supports the modularization of concerns that would otherwise be spread across many modules. This is done by creating aspect modules that contain *advice*, the code for that concern, and *pointcuts* that identify where the advice should be inserted in the application. Because we are working with J2EE web applications, we use AspectJ [5], an AOP language for Java applications.

We create three aspect modules, one each for the navigation routing code in web pages, filter classes and servlet classes. For JSP pages, we place all navigation routing code (the URIs) in a PageNavigation aspect. Since AspectJ cannot define pointcuts in JSP script language, we create a dummy function insertURI() and call this function to get the URI for each form or link in a web page. Then a PageNavigation aspect module intercepts all calls to insertURI() and computes the URI for all requests generated from JSP pages. It accepts two parameters, page name and form name, and returns a string containing the URI.

To create a ServletNavigation aspect, first any processing needed for a request is encapsulated in a method, so that each incoming request has a single processing method (or perhaps none at all). These methods have two parameters, for the request and response objects, and return either nothing or a string containing the processing result.

The doPost/doGet methods that normally process a request are left empty, and instead the navigation routing code is placed in a ServletNavigation aspect module. This module may contain several pairs of pointcuts and around advice, one pair for each servlet. The pointcut says that the corresponding advice should be executed instead of the servlet's doPost or doGet method. Inside the advice, the incoming URI is used to invoke the appropriate processing method. For static routing the outgoing URI is hardcoded then the request is passed along. For conditional routing the result of the processing method is used to determine the outgoing URI, and for cached routing the outgoing URI is retrieved from a session variable.

Fig.1 shows the pointcut and around advice for a servlet named Dispatcher. For incoming requests with URI "/transferAck.do", the "makeTransfer" method is called to process the request. Then the request is forwarded with URI "/transferAck.screen" if makeTransfer returns "success". Thus this is an example that uses conditional routing.

```
aspect ServletNavigation {
  pointcut DispatcherNav(HttpServletRequest req, HttpServletResponse res):
    execution(* Dispatcher. do* (HttpServletRequest, HttpServletResponse))
              && args(req,res);
  void around (Dispatcher dp, HttpServletRequest req, HttpServletResponse res) :
              DispatcherNav(req,res) && target(dp)  {
    String targetURI=req.getServletPath();  //Get target URI.
    if (targetURI.compareTo("/transferAck.do") == 0)   {
      String returnValue = dp.makeTransfer(req,res);
      try  {
        if (returnValue.compareTo("success") == 0) {
          // forward to transferAck web page
          req.getRequestDispatcher("/transferAck.screen").forward(req,res);
          } …
```

Fig. 1. ServletNavigation Aspect Module

Filters are treated almost exactly the same as servlets are, except that the pointcut specifies doChain, which is the normal processing method for filters, and the pointcuts and advice go into the FilterNavigation aspect.

Besides the three aspect modules, there is some navigation routing information in web.xml. Since the J2EE platform forwards the requests according to the deployment description in web.xml, we cannot modify it to look someplace else. Thus we leave web.xml as it is.

The number of the aspect modules can be altered since aspect modules could be combined or split without affecting the functionality. But according to our experience with small- and medium-sized applications, it works well to use the three aspect modules we describe above.

However, this approach has some shortcomings. A developer need not be an expert in AOP because our implementation conventions describe exactly how to write the aspects. But without some understanding of AOP, he or she may have difficulty understanding how advice is woven into the application at the locations specified in the pointcuts. Another problem is the complexity of compilation and deployment of the aspect modules, especially for JSP web pages. Because AspectJ does not understand the JSP scripting language, JSPs must be converted into Java files before weaving in the aspects.

But it turns out that AOP is not essential to our approach. The reason for this is that the weaving we require is simple: we ask the weaver to replace a dummy method with the advice given in the aspect rather than asking it to weave advice into various places in the body of a method. Thus our approach can also be implemented using only standard OO techniques.

For the OO variant of our approach, we use regular classes instead of aspect modules. For the navigation routing code in JSPs, we use a PageNavigationManager class. It provides a function that is called whenever the navigation routing information is needed. This function takes the web page name and form/link name as parameters and returns the URI.

The servlet classes need to be structured in almost the same way as for the AOP variant: divide processing into functions for different URIs, but in this case remove the doGet/doPost methods.

For the navigation routing code in the servlet, we use a class ServletFramework instead of the servlet aspect module. In this class, both doGet and doPost call a `proceed` function. The `proceed` function looks much like the servlet advice: identify the incoming request's URI, call corresponding functions in original servlet class, set session variables if necessary and forward the request.

All servlet classes inherit from this ServletFramework class. Since we do not define doPost/doGet in the servlet classes, every time a request reaches a servlet class, the `proceed` function in ServletFramework is invoked by its doGet/doPost. Function `proceed` uses introspection to find out which servlet class was invoked, then invokes the appropriate method.

As before, filters are treated in a similar way to servlets.

Fig. 2 shows part of the proceed function in the ServletFramework class corresponding to Fig.1. First it checks if the instance is the Dispatcher class; then identifies the incoming request's URI; invokes the method "makeTransfer" in the Dispatcher, and forwards the request with URI "/trasferAck.screen" if the return value is "success".

```
private void proceed(HttpServletRequest request, HttpServletResponse response {
  Class theServletClass = getClass();
  String incomingURI = request.getServletPath();
  …
  if (theServletClass.getName().compareTo("com.sun.ebank.web.Dispatcher")==0) {
       if (incomingURI.compareTo("/transferAck.do")==0) {
            theMethod = theServletClass.getMethod("makeTransfer", paramTypeList);
            returnValue = theMethod.invoke(this, paramList);
            if (((String)returnValue).compareTo("success")==0)
                      forwardingURI = "/transferAck.screen";
  ...}}
  request.getRequestDispatcher("/"+forwardingURI).forward(request, response);
  …
```

Fig. 2. Proceed Function in ServletFramework Class

2.2 Comparison with Current Table-Driven Frameworks

We also studied the existing frameworks (Struts[14] and others) for developing web applications to find out how the current techniques handle navigation routing. Most

frameworks are similar with regard to navigation routing, so we will only discuss Struts in this section. The discussion of other web application frameworks is in section 5.

J2EE web applications using the Struts use the ActionServlet servlet class of the Struts library as a Front Controller. For all requests received by the ActionServlet class, part of the navigation routing is defined in a separate XML file, which includes incoming requests' URI, the Action class that will process the request, the possible return values, and the corresponding forwarding URI. Thus this table-driven framework supports both static and conditional routing.

```
<action-mappings>
    <action path="/transferAck"
            type="com.sun.ebank.web.transferFundsAction"  …>
            <forward name="success" path="/transferAck.screen" />
    </action>    …
```

Fig. 3. Struts-config.xml

Fig. 3 shows part of the struts-config.xml for the same Dispatcher servlet shown in Fig.1 and Fig. 2. The incoming request with URI "/transferAck" will invoke the Action class "transferFundsAction", which works as the makeTransfer function in our approach. Then the transferFundsAction class forwards the request to the URI ("path" in Fig. 3) associated with "success".

We compare our approach and Struts in Table 1 according to seven criteria. There are two main concerns for a developer choosing an approach for supporting navigation routing: coverage and effort. For coverage, we see if the approach covers all three navigation routing categories described in Section 1, which types of components it covers, and what kind of constraints it applies to the application structure. For effort, we see how easy it is to understand, compile and deploy, debug, and implement the approach.

A checkmark in the table means the approach satisfies this criterion well. Most of the table content is self-explanatory so we do not repeat the details here. Our AOP and OO variants separate the navigation routing code in the same basic way, but each has some shortcomings. The AOP variant is more difficult to compile and deploy. The OO variant requires the servlet and filter classes to inherit from our navigation routing class framework.

Struts does not separate the whole navigation routing but only those parts used by the front controller. Strut uses XML tables to store navigation routing information. It cannot support cached routing because its table structure does not allow that. We can also use tables for our approach, rather than hardcoding the same information in the source code. Then we provide developers the navigation routing classes as a library. Since our tables cover navigation routing code for all component types, the table must have different sections for different component types. They must also support cached routing.

The most obvious advantage of a table-driven approach is that to change navigation routing inside a table is easier than to change the source code in an aspect module or class. No recompilation and redeployment is necessary for modifications in a table.

Table 1. Struts vs Our Approach

		Struts	Our Approach	
			AOP Variant	OO Variant
Coverage	Navigation Routing Categories	Static and conditional routing.	√ Static, conditional, and cached routing.	
	Component Types	Only the front controller servlet.	√ JSPs, servlet classes, and filter classes.	
	Application Structure	Must use front controller pattern.	√ No pattern is enforced. Also it can be applied on top of Struts.	
Effort	Understandability	√ OK if the developer is familiar with Struts.	√ Ok if the developer understands AOP and weaving.	√ OK.
	Compilation and Deployment	√ Relatively easy.	Difficult. JSP files need to be converted to Java, and aspects need to be woven into other modules.	√ Relatively easy.
	Debugging	The library code is not open source, so it cannot be traced at runtime.	Aspect modules make the control flow hard to follow and debug.	√ Relatively easy.
	Implementation Support	√ Library code is provided.	√ Aspect module framework is provided.	√ Navigation routing class framework is provided.

Putting the navigation routing in a table may make it easier for non-specialists to understand and modify the navigation routing, since they do not need to understand a programming language like Java. However, when the table contents get complicated, it may be easier for a developer to understand code rather than a table. Supporting all component types and all three navigation routing categories significantly complicates the table structure.

3 Evaluation

To evaluate our solution for improving the maintainability of web applications by separating navigation routing code, we describe our experiences refactoring existing applications.

We[1] have applied our solution to a number of existing applications. Some are small sample applications provided with the J2EE server 1.4, but the three we discuss here are more comprehensive: Duke's Bank [16], PetStore [17] and Virtual Shopping Mall [10]. The first two of these exemplar applications are provided by Sun, and the third is provided by Oracle.

[1] Here "we" stands for one author and another graduate student who was not involved in the research but converted the applications after receiving an explanation of the approach. The numbers we give for effort (time) are the average time spent among us

Table 2. Metrics for the original applications

	Dukes Bank	Mall	Pet Store
Total LOC	7133	19248	34048
# web pages	10	35	16
# filters	0	0	2
# servlets	1	7	2
# modules with nav. routing	12	44	21
LOC modules with nav. routing	1015	6154	9059

Table 2 shows some metrics for the original applications and Table 3 shows how these changed after our AOP solution, OO solution, or the Struts technique was applied. Duke's Bank 1 uses the AOP solution, Duke's Bank 2 uses the OO solution, Duke's Bank 3 uses the Struts technique, and Mall and Petstore use the AOP solution.

Duke's Bank is a comparatively small on-line banking application. We spent three hours applying the AOP variant of our solution on this web application (Duke's Bank 1 in Table 3): one hour moving code from the JSPs to the Page Navigation aspect module and one hour moving code from the servlet to the Servlet Navigation aspect module, including separating out three functions for three types of request processing. The remaining hour was needed to compile, test, and deploy the aspect modules and modified java files.

When we used the OO variant of our approach (Duke's Bank 2), the time we spent was very similar: one hour moving code from JSPs to the Page Navigation class and one hour moving code from servlets to the ServletFramework class and separating out three functions. We spent less time compiling, testing, and deploying than with the aspect modules.

We also used Struts on Duke's Bank (Duke's Bank 3 in Table 3). We spent less time with Struts since not all navigation routing code is separated. JSPs remain untouched because Struts cannot separate the navigation routing code in JSP pages. Less than one hour was spent moving code from the original servlet to three new action classes. It is very much like separating out functions in our approach. One hour was spent compiling, testing, and deploying. The longer time here was because it is not easy to debug when Struts is used. The library source code is not available, so when there are errors the debugger cannot trace the servlet code.

Virtual Shopping Mall is a medium-sized on-line retail store application. Since we found that six of the web pages simply forward the request to other web pages, we treated them as servlets. This application uses the Struts framework, so part of the navigation routing code is already separated. We applied our approach on top of the Struts framework to separate the leftover navigation routing code (mostly in web pages), which took three hours. An additional two hours was spent compiling, testing, and deploying.

PetStore is another medium-sized on-line store application. Petstore uses WAF (a framework very similar to Struts for navigation routing). But there is still navigation routing code in filter classes and web pages. We spent one and a half hours converting the JSPs and creating the Page Navigation aspect module. One hour was spent modifying the servlet classes and creating the Servlet Navigation aspect module, and another hour was spent doing the same to the filters. The remaining two hours were needed for compiling, testing and deploying.

Table 3. Metrics after applying AOP solution

	Duke's Bank 1 (AOP)	Duke's Bank 2 (OO)	Duke's Bank 3 (Struts)	Mall (AOP)	Pet Store (AOP)
# modules with nav. rout.	3	3	12	4	4
LOC modules with nav. rout.	200	324	929	637	339
# methods	3	3	3	27	11
LOC methods	120	120	154	3175	2016
Modification Time (in hours)	3	2.5	1.5	5	5.5

As evidence that our solution improves maintainability, we first look at the number of files that contain navigation routing code before and after applying our solution. For example, in PetStore originally there were twenty-one files containing navigation routing code: 16 web pages, 2 filters, 2 servlets and web.xml. After applying our solution there are only the three aspect files and web.xml. Since the developer can focus on a smaller number of files, any maintenance involving navigation routing is easier to perform. Duke's Bank and Virtual Shopping Mall have similar improvements: from 12 to 3 and from 44 to 4 files containing navigation routing code. In Table 2, Mall has one additional file containing navigation routing because it uses a Struts configuration file.

If we look the size of these files before and after applying our AOP solution, we find a 5- to 25-fold reduction in the LOC. The main reason for the large decrease is that originally many files contain a small portion of navigation routing code along with other code. After conversion the files with navigation routing code contain only navigation routing code (except web.xml, which still contains other deployment information). This clearly reduces the amount of time a developer spends in locating and understanding navigation routing code.

Reviewing the modification time for the applications, we find that a developer needs about three hours to apply our solution for an existing small web application and five to six hours for an existing medium-size web application. For a 4- to 11-fold reduction in the number of files containing navigation routing code and 5- to 25-fold reduction in the lines of code in these files, the cost is not high. For the AOP variant, since the aspect modules are woven into the other source code before executing, performance is not affected.

4 Related Work

It is clear that researchers believe navigation is an important part of web applications. Rossi [12] identifies a number of common navigation patterns used in web applications. Reina [11] states that it is necessary to separate the navigation concerns into aspect modules but does not present a specific solution.

We studied other web application frameworks besides Struts. Web application frameworks like WAF[15], Tapestry[7], Expresso[2], Maverick[8] are close to Struts when considering only the navigation routing. A table is used to define part of the navigation routing and a front controller is used. However, Tapestry provides more separation of navigation routing code at the JSP side since it separates the client side HTML and the server side code in the JSPs.

Another web application framework, Barracuda [1], uses a more complicated event model, which is closer to the event model in swing applications. The events are hierarchically organized. The request from the web pages is first received by a servlet class that converts them to events. Then the listener classes process the events and may send more events. Finally, a response request indicates the corresponding web page to be sent back to the client. In Barracuda, the navigation routing can include more than one server component as in our approach while the Struts cannot. However, Barracuda does not separate the navigation routing code and navigation routing is harder to understand because of the hierarchical event model.

To the best of our knowledge, there is no other similar approach for separating navigation routing code. AspectJ has been applied to many different kinds of applications to separate other kinds of concerns, such as [6], [13]. Most of the results are considered a success and better than a pure Java implementation.

Kienzle [6] 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 routing is not simulated by objects and is not difficult to aspectize. We do expect developers to have some knowledge of AOP.

Murphy et.al. [9] present a study on applying AOP to separate concerns for 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: 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 routing code into aspect modules.

5 Conclusion

Our approach can be applied to both new application design and for refactoring existing applications. To show its benefit, we evaluated it by refactoring three commercial applications with a variety of navigation routing patterns: 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 an XML table for part of the navigation routing. While it took 3-6 hours to refactor these small- to medium-sized applications, the number of files containing navigation routing code and the lines of code in these files was greatly reduced (a 4- to 11-fold reduction in number of files and a 5 -to 25-fold reduction in lines of code in these files).

We found a large improvement in maintainability as a result of separating the navigation routing code into a handful of modules. Clearly most of this was the result of gathering together the navigation routing code, making it easier to locate and understand. But our solution also forces a more standard way of encoding navigation routing, which facilitates maintenance.

However, separation of concerns alone is not sufficient for solving the maintainability problem. Because server components are implicitly invoked by the J2EE framework, the navigation routing is difficult to follow even after locating the code. A model that explicitly describes the navigation routing can help the developer during maintenance. Our solution to this problem, described in [4], is a formal navigation

routing model and a set of tools to analyze the model, generate implementation code, and extract a model from the code. An implementation-level solution such as the one described in this paper enables this model to be automatically extracted from the code. We have also proposed the Navigation Routing Diagram as a visualization of this model. In addition, we can use the navigation routing model as an indication of the complexity of an application's navigation routing.

References

1. Barracuda Presentation Framework. http://barracudamvc.org/Barracuda/index.html
2. Expresso Web Services. http://www.jcorporate.com/
3. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns. Addison-Wesley Professional. 1995.
4. Han, M. and Hofmeister, C. Modeling Navigation Routing in J2EE Web Applications. Lehigh University Technical Report, 2004.
5. 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.
6. 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.
7. Lewis Ship, H. M. Tapestry in Action. Manning Publications. 2004.
8. Maverick. http://mav.sourceforge.net/
9. 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.
10. Oracle J2EE sample code. Virtual Shopping Mall. http://www.oracle.com/technology/sample_code/tech/java/j2ee/vsm13/index.html
11. Reina, A.M. and Torres, J. Analysing the Navigational Aspect. *In Proceedings of Second Aspect-Oriented Software Development Workshop (AOSD '02)* (Germany, February 2002).
12. 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.
13. 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.
14. Struts. http://struts.apache.org/
15. Sun Java Blueprints: Guidelines, Patterns, and Code for End-to-end Applications. http://java.sun.com/reference/blueprints/index.html
16. Sun Java Center. The Duke's Bank Application. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Ebank.html
17. Sun Java Center. Java Petstore 1.1.2. http://java.sun.com/developer/releases/petstore/petstore1_1_2.html
18. Sun Java Center. Java Servlet Technology. http://java.sun.com/products/servlet/