Impact Factor (SJIF): 3.632

International Journal of Advance Research in Engineering, Science & Technology

e-ISSN: 2393-9877, p-ISSN: 2394-2444

Volume 3, Issue 6, June-2016

Java Server Pages in Modernization

Prajapati Keyur P
Asst.Prof, MBICT
New V.V.Nagar-388120

Abstract

One way to improve the maintainability of a web applica-tion is to separate its presentation from the business logic. Such separation not only makes a web application easier to evolve and maintain, but also allows individual devel-opers with different skills to cooperate more efficiently. Custom tags are a recent addition to the JSP standard that helps to facilitate this separation. In this paper, we present a transform to restructure JSP pages by moving embedded Java code into custom tags without changing the original functionality or user interface. This reduces the complexity of the web applications, makes them more maintainable and makes best use different skill sets.

1. Introduction

Many web applications are developed using a mixture of HTML code and application code written in languages such as Visual Basic, Java, and JavaScript. The HTML code is used to layout the data, while the application code is embedded in the HTML, supporting the business func-tionality of the web application.

JavaServer Pages (JSP) is one of the popular tech-nologies for building web applications that serve dynamic contents. As indicated by the name, the scripting lan-guage in JSP is, by default, Java. In particular, scriptlets, which take the form of "`<% Java code %>`", are used to embed Java code within HTML. The explicit use of scriptlets facilitates rapid prototyping but introduces com-plexity into the implementation. Such scriptlets interweave HTML with Java code, and make code author-ing and debugging tricky, make software maintenance and evolution difficult. Moreover, scriptlets are not reusable and cause frequent cut-and-pastes edits between pages, leads to further code duplication and a more error-prone environment.

Custom tag libraries are a recent addition to JSP to separate HTML from Java code, but have not been widely used yet. One long-term solution to separate the presenta-tion and business logic of JSP-based web applications is to transform the scriptlets into tag libraries. In this paper, we present a set of automated transforms to implement the separation. These transforms restructure JSP pages by changing embedded Java code into custom tags without altering the functionality or user interface of the web ap-plication.

The remainder of the paper is organized as follows. The transforms are presented in Section 4 and give a short overview of the implementation in Section 5. Section 6 presents related work and Section 7 concludes the paper.

2. JSP Basics and Custom Tags

A JSP page contains template text and JSP elements. All content that is not a JSP element is called template text. The template text can be any text, such as HTML, XML, WML, or even plain text, which is passed directly through to the browser. The JSP elements, which are used to gen-erate dynamic content in the page, include directives, standard actions, custom actions, standard tag library tags, (JSTL) scripting elements and JavaBean components .

Standard actions use the prefix *jsp*, such as the <jsp:useBean> and <jsp:getProperty> actions, which are use to create beans, access bean properties, and invoke other pages. But there are still many actions involved in manipulating page content not covered by standard actions, but covered by custom actions (also called custom tags) and JSTL.

When developing a custom tag library, two components must be developed: the implementation of the tags in Java and an XML file called the tag library descriptor. Each individual tag is implemented as a Java class called a tag handler. A tag handler defines the tag's behavior, which must implement one of the interfaces defined in the pack-age

javax.servlet.jsp.tagext.

Sun provides a standard tag library which implements statement level tags for control flow and database opera-tions. Our approach produces higher level tags abstracting the business logic of the web application. As well, JSP 2.0 provides an alternate interface for custom tags, the sim-pleTag interface. It simplifies the implementation, and tags need only implement a single method. Our technique works with both interfaces.

3. Restructuring Through Transformation

This section presents our approach to restructuring JSP pages by transforming interweaved Java code into tags to modernize existing JSP web applications.

3.1. Major Requirements

In the implementation of the restructuring, there are three major requirements that must be considered, which are described as follows.

- The transformational restructuring should not change the functionality of web applications. To meet this requirement, static analysis information, such as con-trol flow and data dependence, must be collected. The complex interactions among different parts of a

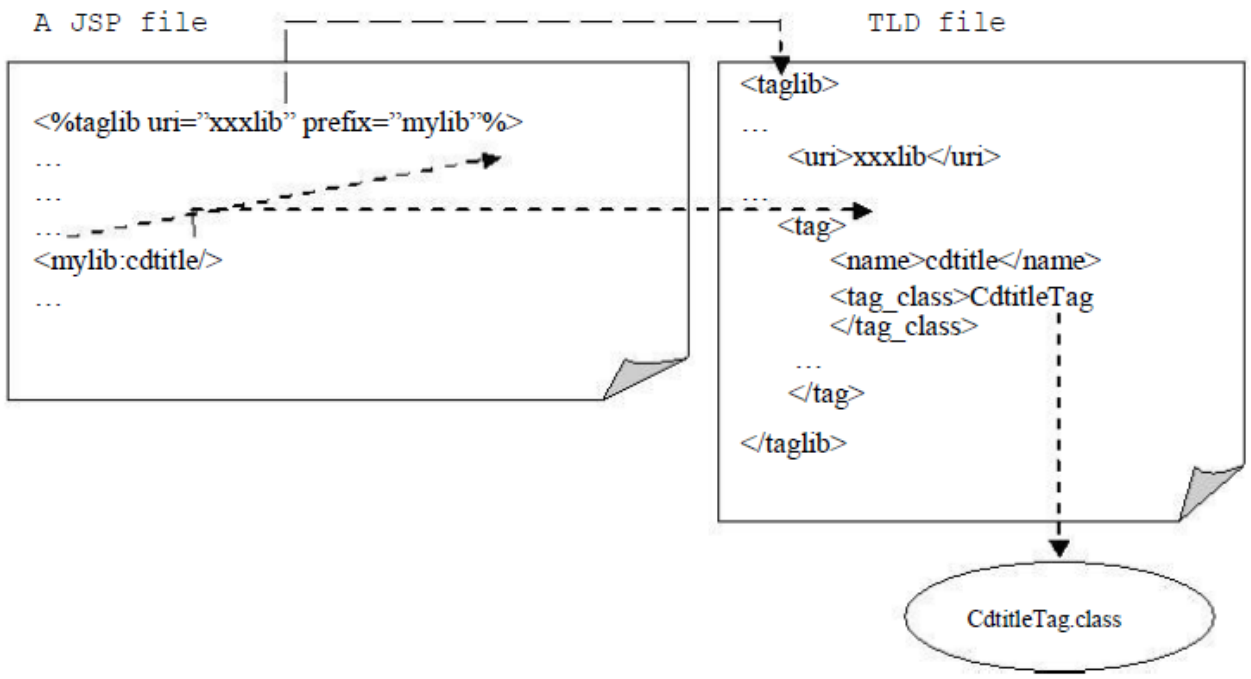


Figure 1. Relation among the taglib directive, the TLD and the tag handler class

JSP page should still be captured even after the parts written in Java are moved into custom tags.

• The appearance of web applications should not be affected by the transformation. The user interface of the applications, which is defined in part by the HTML and in part by the Java and JavaScript, should remain unchanged.

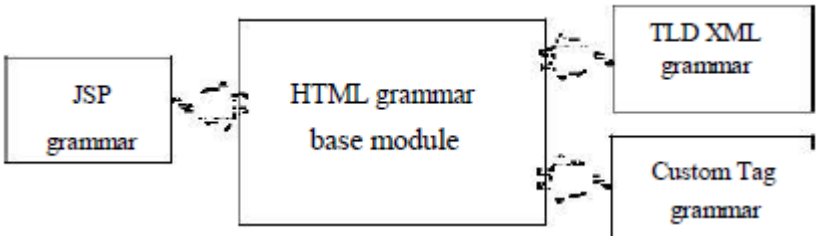
• All code comments should be kept for future mainte-nance. It is inevitable that the application will need to be maintained and evolved after it is transformed, such as adding new features and making further en-hancements. Thus the transformed application code with comments stripped out of the original application may not be acceptable. The code comments should be kept in the same locations relative to the original source code in order to ensure that the code is still readable.

3.2. A Multilingual Parser

Before we can transform the source code, we must be able to parse it. In addition to the mixing of multiple languages, the source code may contain minor syntax errors or other unexpected contents that are ignored by most browsers. To address these challenges, we adopt the parsing tech-nique developed by N. Synytskyy et al. This is a robust multilingual parsing technique based on island grammars for ASP applications and extended to JSP by Arial Li. One of the important properties of a multilingual parser using island grammars is the ability to capture complex interactions among different interesting parts of web applications written in different languages by simultaneously parsing multiple languages into a single parse tree, which benefits code analysis, fact extraction and transformation.

Figure 2 shows the relationship between the grammars. The base grammar is the HTML grammar. This grammar defines the structural elements of HTML that are of inter-est. In this grammar the elements of interest are tables, anchors and forms. The JSP grammar, the TLD XML grammar and the custom tag grammar are written in modular form.

One of the main characteristics of the grammar is that it preserves the relationships between the scopes of the multiple source languages. For example a loop statement that contains HTML text within the block is not parsed as two separate scriptlets (one for the beginning of the loop and one for the end), but as a single scriptlet, treating the



HTML text within the loop as if it was a Java statement. In this case the meaning of the start and end scriptlet tags are reversed. That is the '%>' tag starts HTML text and the '<%' tag ends the HTML text.

3.3 General Approach

In practice, HTML code and Java code are tightly inter-weaved in existing JSP web applications. Java code is embedded in HTML code and HTML tags are embedded in Java code. Before we discuss the code transformation, we must first examine the elements that are affected by the transform.

JSP scripting elements. Scripting elements allow us to add pieces of Java code in a JSP page, which have three types: *scriptlets*, *expressions*, and *formal declarations*. The scripting elements in a JSP page will be executed each time the page is requested. It is better to move and encapsulate Java code implementing business logic into appropriate custom tags.

HTML content embedded in *out.print()*. The Java code sequence `out.print(…);` is a commonly-used simple method to generate HTML content from within a scriptlet. The use of this code has two disadvantages. One is that it needs extra effort to create and maintain HTML pages for application programmers. The other is that content authors (or web page designers) must under-stand the embedded code or must ask the application programmers to make any changes to the scriptlet. Before we perform the transformation, we extract the HTML con-tent from the `out.print()` and `out.println()` statements and put the content directly into the page.

This changes the code to an HTML segment that is nested within the scriplet. It starts with a JSP end tag (`%>`) and ends with a JSP start tag (`<%`), allowing HTML seg-ments to appear inside interesting JSP elements and is used to handle "the lake inside an island" parse. As a re-sult, the extracted HTML code can be included directly into the web pages without `out.print()` calls and page designers can make changes to the HTML contents with-out risk of breaking the Java code.

JavaBean action elements. A JavaBean component is a Java class that has a no-argument constructor and con-forms to the JavaBean coding conventions. There are three kinds of JSP standard action elements which allow developers to use JavaBean components : *<jsp:useBean>* instantiates a Java Bean and makes it available in a page; *<jsp:getProperty>* gets a property value from a JavaBean and adds it to the response; *<jsp:setProperty>* sets all properties value in a JavaBean with names matching the names of the parameters re-ceived from the request.

These three action elements along with other scriptlets are all functions which are more suitable for an applica-tion programmer. The web page designer need not know

how to instantiate a Java Bean or how to set properties of the bean. The page designer only needs to know the prop-erties of a Java bean component in applications where a separate servlet instantiates a bean and passes it to a JSP page for display. To this end, we change the three action elements into valid Java code enclosed within scriptlet tags (<%…%>) before the general transformation.

JSP page directives. JSP page directives are usually found at the top of a JSP page and always enclosed within directive tags (<%@ … %>). There can be any number of page directives within a JSP page, but the attribute/value pair must be unique. The cases that affects the transform is the case where the page imports a Java package. For example,

```
<%@ page import ="java.sql.ResultSet" %>
```

This directive imports the class ResultSet from the package `java.sql` into the page. As this class should also be imported into our newly created custom tags, we change the page directive into a *scriptlet*. For example, the directive above becomes `<% import java.sql.Re-sultSet; %>`. We can then transform this valid import statement as part of the general transform.

4. The General Transformation

This section looks at the general transformations needed to convert embedded Java into custom tags. We present the four general cases that comprise the transforms.

Before the general transformation, the original source code must be normalized by modifying non-scripting ele-ments (bean, page directives) that need to be migrated into JSP scripting elements, extracting HTML content from `out.print` statements, and merging adjacent *scriptlets*.

After the normalization transform, all Java code is em-bedded inside JSP scripting elements. The next goal is to eliminate JSP scripting elements from the normalized source code by replacing them with JSP custom tags.

The intermingling of HTML and code presents a chal-lenge not only for parsing but also for transformation. In particular, data flow analysis through various software pieces is complicated when using multiple programming languages and technologies. To transform embedded Java code into appropriate custom tags, we need to determine how to break down the processes of JSP pages into smaller components, and what the relationship is among the components. We also need to determine:

(1) how to use new custom tags for better control of dy-namic content,

(2) how to name the tags,

(3) how to pass input through by placing data either in tag attributes or between opening and closing tags,

how to devise the tags that focus on the "*what*" and hide the "*how*" to make the transformed web pages and resulted tags maintainable.

Our transformation strategy breaks down into four ba-sic cases.

Case 1 Only one custom tag is needed. Figure 3 shows that one block of Java code is inserted into HTML but no HTML segment is nested within the block. In this simple case, only one custom tag needs to be created, into which all Java code is migrated, and this tag has an empty body. In the example, the block of Java code is mainly about user session management. Accordingly, we can name this tag as <mylib:userSession> and name the Java class implementing this tag as UserSessionTag. Figure 4 shows the transformed page after Java code is migrated into the custom tag <mylib:userSession>.

Case 2 Two nested custom tags are needed. Figure 7 shows a JSP page in which there is a JSP expression, <%=userName%>, following a block of Java code. Note that the variable used in the expression is defined in the Java code block. In this case, two nested custom tags must be created where one tag is nested within the body of another tag. The Java code in the block will be migrated into the outer custom tag, whereas the expression will be migrated into the simple tag.

Figure 8 shows the transformed page after the scriptlet and expression are migrated into the custom tags `<mylib:userSession>` and `<mylib:userName>`, re-

```
 String userName = entry.getUser (userId);
 s.putValue ("userName", userName);        UserSessionTag
%>
Welcome!
</body></html>
```

Figure 3. Block of Java Code in HTML

```
<html><body>
  <mylib:userSession></mylib:userSession>
Welcome!
</body></html>
```

Figure 4. Block After Transformation

spectively. While the tags are nested, the classes that implement the tags are not. To preserve the explicit par-ent-child relationship, the class implementing the parent tag must provide get and set methods for the variables that are accessed by the child. The appropriate code is gener-ated in the child tag class to obtain a reference to the instance of the class for the parent tag, and the child can then access the variables defined in its parent using the parent's *get* and set methods.

The development of the outer custom tag is the same as the tag in Case 1 except that this outer tag has a body which contains not only a segment of HTML code but also another tag (*<mylib:userName>*).

Case 3 HTML content depends on Choice Figure 9 shows a JSP page in which a choice statement determines if some HTML content is displayed. A choice statement may be an if-else statement or a switch-case statement. This HTML content is nested within the choice statement and enclosed between a end tag (%>) and a start tag (<%).

As mentioned in section 3, the multilingual grammar parses the HTML nested within Java statements correctly. Thus, the nested HTML content will be considered by our grammar as an entity within the if-else statement. As a result, our parser will detect one block of Java code at the top scope in Figure 10, which starts from the first "<%" tag to the last "%>" tag. The *then* part of the if-statement contains HTML content which is parsed as if it was a statement.

For this case, three tags are created, including one parent tag and two children tags. The two children tags are nested within the body of their parent tag and are deployed side by side in the body of the parent tag because they have a sibling relationship. One child will handle the *then*

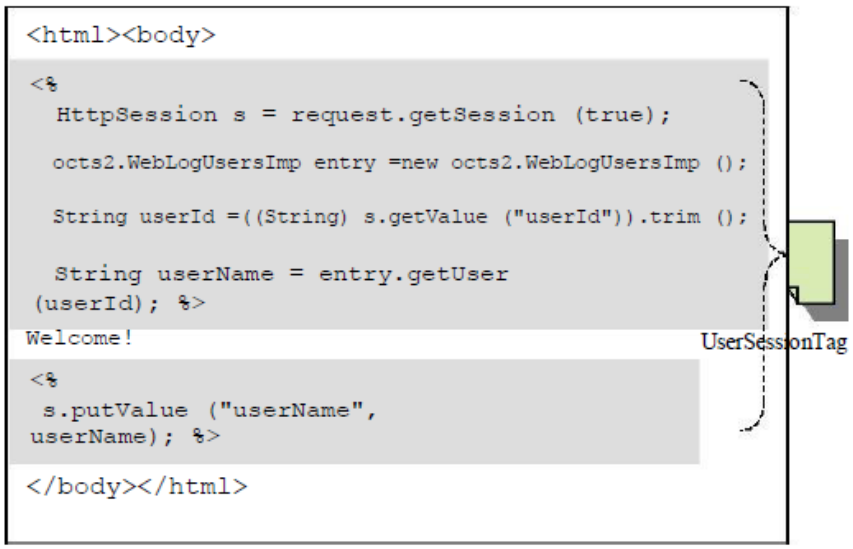


Figure 5. Two Blocks of Java Code in HTML

```
<html><body>
<mylib:userSession>
Welcome!
</mylib:userSession>
</body></html>
```

Figure 6. Two Blocks After Transformation

```
<html><body>
<%
  HttpSession s = request.getSession (true);
  octs2.WebLogUsersImp entry = new octs2.WebLogUsersImp ();
  String userId = ((String) s.getValue ("userId")).trim ();
  String userName = entry.getUser (userId);
  s.putValue ("userName", userName);
%>
Welcome!  <%= userName %>
</body></html>
```

UserSessionTag

UserNameTag

Figure 7. A Page with an Expression

```
<html><body>
<mylib:userSession>
Welcome!  <mylib:userName/>
</mylib:userSession>
</body></html>
```

Figure 8. Expression Page After Transformation

part of the choice, while the other tag handles the *else* part. The then child will only allow the HTML content to be included if the condition is true. In order to keep the ex-clusive relationship between the two child tags, we add a simplification step as shown in Figure 10.

A new boolean variable is introduced which will store the evaluation of the choice expression is declared in the top scope or the parent tag. The if statement is replaced by two if statements using the boolean variable. A `get` method for this variable should be declared in the parent tag at the same time. Finally, all Java code in the top scope including the declaration of the new variable is mi-grated into the parent tag (i.e. `<mylib:checkLogin>)`, and the Java code in the sub-scopes wrapped with an ap-propriate if-condition expression should enter its appropriate children tags (`<mylib:invalidLogin>` and `<mylib:valid Login>`, respectively). Figure 11 shows the transformed page after Java code is moved into custom tags.

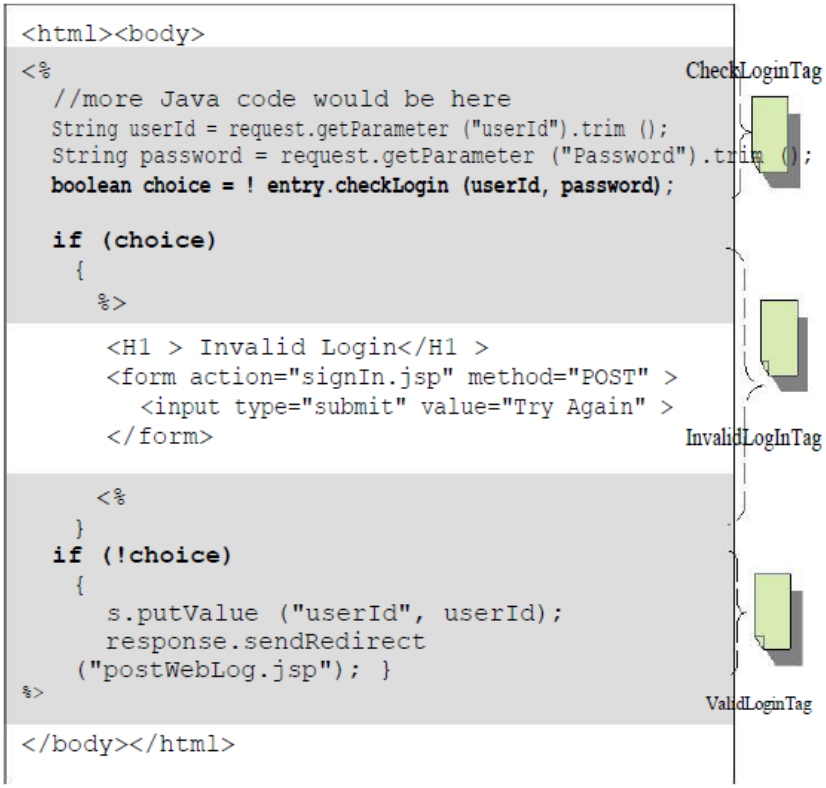
Figure 11, also shows another feature of the transform. Two of the Java statements within the code that will be-come part of the parent tag retrieve parameters from the page request. This is information provided by the browser, possibly by a form on the previous page. Since this infor-mation is part of the web design, we wish to make it visible to the web page designer. So the tag is parameter-ized by the attributes. This parameterization is not limited to the choice case, but can be used in all cases, where the code to be moved to the custom tag accesses the page re-quest.

Case 4 HTML Content Inside a Loop. Figure 12 shows a JSP page in which a block of Java code is fol-lowed by a table and the table body content is controlled

Figure 9. Page With a Choice Statement

by a loop. A loop can be a while-loop statement or a for-loop statement. In the figure, the outermost layer of the table is a HTML table tag. It has an opening tag `<table border>` and a closing tag `</table>`. A block of Java code containing a while -loop is contained inside the table tag. The code generates the table body. Moreover the body of the loop (the table body) is made up of HTML text, scriplets and expressions.

In the grammar, the interesting elements cover not only JSP elements such as JSP scripting elements, but also in-teresting HTML elements such as table tags, form tags and anchor tags. As a result, our parser will detect two inter-esting elements at the top-most scope level for this case.




```
<html><body>
<%
  //more Java code would be here
  String userId = request.getParameter ("userId").trim ();
  String password = request.getParameter ("Password").trim ();
  boolean choice = ! entry.checkLogin (userId, password);

  if (choice)
   {
    %>
     <H1 > Invalid Login</H1 >
     <form action="signIn.jsp" method="POST" >
       <input type="submit" value="Try Again" >
     </form>
     <%
   }
  if (!choice)
   {
     s.putValue ("userId", userId);
     response.sendRedirect
   ("postWebLog.jsp"); }
%>
</body></html>
```

Figure 10. Simplified If Statement

```
<html><body>
<mylib:checkLogin attr1="userId" attr2=”Password”>

   <mylib:invalidLogin>
        <H1 > Invalid Login</H1 >
        <form action="signIn.jsp" method="POST" > <input
            type="submit" value="Try Again" >
        </form>
   </mylib:invalidLogin>

   <mylib:validLogin redirectPage=”postWebLog.jsp”>
   </mylib:validLogin>

</mylib:checkLogin>

</body></html>
```

Figure 11. Simplified If Statement After Transformation

```
<%
String title;
String price;
while (rs1.next ())
{
 %>

<tr>
 <%
```

The child tag in this case is an iteration action, which means this tag will evaluate its body content (the table cells and table rows) repeatedly until some condition be-comes true (i.e. the collection rs1 is empty). This involves implementing the IterationTag interface by the tag class.

The code within the loop body makes a reference to the result set that is in the top level tag. This can be simplified by adding a local variable to the child tag for the loop that is initialized as the start of the loop (loopvar in Figure 14).

Then the tags created for the inner children can then easily reference the value from the tag that implements the loop.

Composing Cases. Obviously not all JSP pages fit into simple choice or loop cases. The cases must compose if we are to handle JSP web pages in general. The example used for case four (the loop case) also illustrated the prob-lem, having simple code nested within the loop, and the loop itself nested within the top level code. Nested code may refer to variables in higher up in the scope hierarchy.

Just as the scopes are nested, the resulting tags will also be nested. So if we were to modify the example shown in Figure 10 to include a loop generating a table inside of the form, the tags generated as part of the loop solution would be nested within the *invalidLogin* tag. Similarly, if a choice existed within a loop, the tags generated as part of the choice would be nested within the loops.

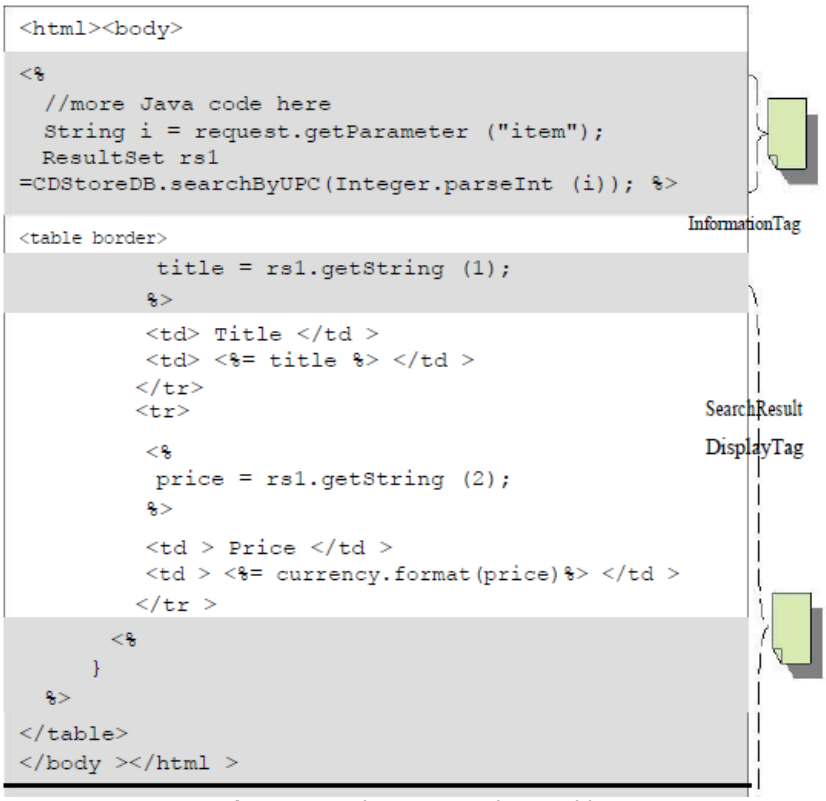


Figure 12. A loop generating a table

```
<html><body>
<mylib:information attri="item"> <table
border>
      <mylib:searchResultDisplay>
                        <tr  >
                            <td >  Title  </td  >
                            <td    <mylib:CDTitle/    </td  >
                        </tr  >
                        <tr  >
                            <td >  Price  </td  >
                            <td >   <mylib:CDPrice/>    </td  >
                        </tr  >
      </mylib:searchResultDisplay>
</table>
</mylib:information>
</body></html>
```

Figure 13. Page after transformation

Our transform performs data flow analysis between the scoping levels, identifying the scope to which a referenced variable belongs. Code is generated using the JSP API to find the instance of the class implementing the parent (or ancestor) tag and invokes the appropriate get or set method to access the variable.

5. Implementation

In this section, we briefly describe the implementation of the transformation. Most of the process is implemented using the TXL language, which is a pure functional pro-gramming language particularly designed to support rule-

```
<table border>
    <%
       String title;
       String price;
       ResultSet loopvar = rs1;
       while (loopvar.next ())
          {
            %>
              <tr >
            <%
               title = loopvar.getString (1);
            %>
                 <td > Title </td >
                 <td > <%= title %> </td >
              </tr >
              <tr >
             <%
               price = loopvar.getString (2);
             %>
                 <td > Price </td >
                 <td > <%= currency.format(price)%> </td >
              </tr >
            <%
          }
    %>
</table>
```

Loop body

Figure 14. Modified Table Generation Code

based source-to-source transformation. Figure 15 illus-trates our transformation process, which includes five phases. The preprocessing phase normalizes the source code as described in section 3. It also performs some comment and lexical preprocessing.

The grouping phase performs an analysis and annotates each line of normalized source code with a tag id identify-ing the custom tag to which the source code will belong. The first part of the analysis identifies the cases that we have identified in this paper. It identifies control state-ments that contain HTML/JavaScript and the first statement of Java sequences within the template text that must be given their own tags. Thus, the basic structure of the tags is identified. The second phase of grouping as-signs the remaining statements to one of the identified tags.

The tag id generated in the group phase is mapped to a reasonable user name such as *invalidLogin* or *CDTitle* by a web interface in the tag naming phase.

The code transformation phase uses the markup from the group phase and the mapping from the tag naming phase to generate the three outputs of the process. These are the modernized JSP pages, the tag library description file, and the custom tag classes. The final post -processing phase deals with final touchups such as fixing comments.

The whole process is automated, except the tag naming phase where the human assistance is required. The details of the implementation, particularly, the markup approach and the transformations are described elsewhere.

We have tested our system on 3 small systems to date consisting of an online music store, a mini weblog applica-tion and a guest book application. Two were obtained from within Queen's, the other is a sample system.

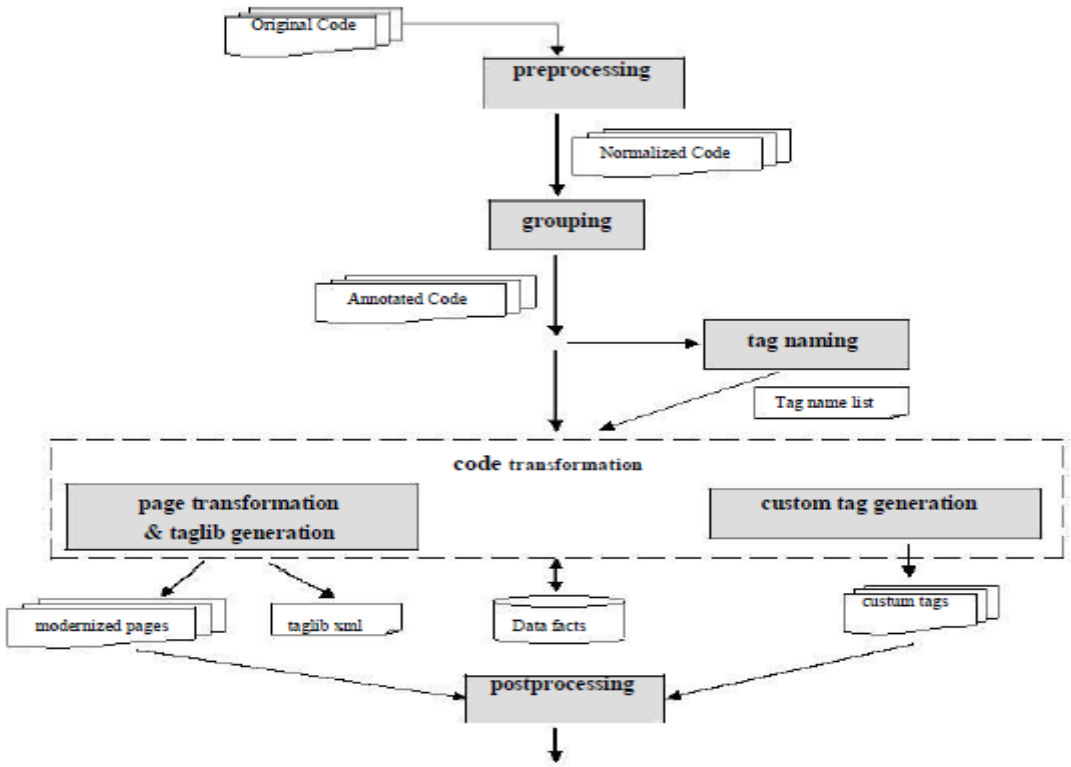


Figure 15. Transformation process

downloaded from the internet. The systems comprise a total of 14 JSP pages containing a total of 682 lines of mixed JSP and HTML. The resulting pages contain 362 lines of tags and HTML. 74 custom tag classes were gen-erated.

Currently each JSP expression is translated into its own custom tag. A simple optimization is to fold the simple JSP expressions into one simple tag. This would eliminate 23 custom tag classes.

6. Related Work

There has been considerable investigation into the evolu-tion of web sites. This includes results in program understanding and architectural modeling, clone detection and removal restructuring and refactoring and migration .

Ricca et al. present an approach of using rewrite rules to improve the quality of web applications. The HTML transformations cover both interpage and intrapage transformations and can identify six cases. Further work illustrates a semi-automatic process to identify static pages that can be transformed into dynamic pagesusing clustering techniques. Jiang et al.present a method to migrate a web application to a web service by examining the generated pages and using pattern matching techniques to infer the services.

Hassan et al. propose a framework for migrating web applications between different development frame-works based on water transformations, an extension of island grammars. Ping et al. present an approach for migrating web applications from IBM Net.Data into JSP, separating database access functionality from presentation logic. Lau et al. present a migration methodology, supported by a tool developed for the IBM WebSphere Commerce Suite and released on IBM's alpha Works. An alternate approach by Ping et al. restructures JSP based web applications by adapting them to a controller-centric architecture. Tilley et al. renovate web appli-cations by reengineering transactions with a user-centered approach.

7. Conclusions

The implementation of our transformation is a greedy approach. It attempts to group as many statements as pos-sible into each tag. Each web page is also processed independently. One potentially extension is to identify clones between pages, separating them in to separate tags. One example is session management code common to multiple pages.

In this paper, we have presented a set of transforms that can be used to implement the separation of the presenta-tion and business logic for existing JSP-based web applications. The transforms restructure the web applica-tions by moving Java code embedded in JSP pages into custom tags without changing the original functionalities and user interfaces of the applications. The interesting information required for this restructuring is contained not only in the multiple languages themselves but also in the way they are coupled.

An advantage of our Java code transformation is that all business logic intensive Java code in JSP pages is moved and encapsulated into custom tags and all elements for presentation are kept in pages, which helps to reduce the complexity of web applications and helps make the re-structured applications more reusable and maintainable.

References

[1] Hans Bergsten, *JavaServer Pages*, O'Reilly 2002.

[2] T. Bodhuin, E. Guardabascio, M.Tortorella, "Migrating COBOL Systems to the WEB by Using the MVC Design Pattern", Proc Working conference on Reverse Engineering, Richmond, Virginia, pp 329-338.

[3] C. Boldyreff, R. Kewish, "ReverseEngineering to Achieve Maintainable WWW Sites", Working Conference on Re-verse Engineering, Stuttgart, Germany, Oct. 2001, pp. 249-257

[4] F. Ricca, P. Tonella, "Using Clustering to Support the Mi-gration from Static to Dynamic Web Pages", International Workshop on Program Comprehension, Portland, Oregon, May 2003, pp 207-216.

[5] N. Synytskyy, J.R. Cordy and T.R.Dean, "Robust Multilin-gual Parsing Using Island Grammars", *Proc. IBM Centres for Advanced Studies Conference*, Toronto, p149-161, Oc-tober 2003

[6] S. Tilley, D. Distante, S. Huang, "Web Site Evolution via Transaction Reengineering", Proc International Workshop on Web Site Evolution, Chicago, 2004.

[7] S. Xu and T. Dean, "Transforming Embedded Java Code into Custom Tags", to appear in International Workshop on Source Code Analysis and Manipulation, Budapest, 2005.

[8] Y. Zou, T. Lau, K. Kontogiannis, T. Tong, R. McKegney, "Model-Driven Business Process Recovery", 11th Working Conference on Reverse Engineering, Delft, Nov. 2004, p 224-233.