Transforming Embedded Java Code into Custom Tags

Shannon Xu and Thomas Dean
Queen's University
Kingston, Canada
{xus,dean}@cs.queensu.ca

Abstract

When a new technology is introduced, the migration of existing applications to the new technology must be carefully considered. Automation can make some migrations feasible that otherwise may be too risky or expensive to be worth the advantages of the new technology. In this paper we describe a technique for migrating web applications using embedded Java code into a custom tag implementation. The technique uses source code transformation techniques to analyze and separate the Java from the web pages. The code is then automatically transformed to custom Java classes which are invoked from the modified web pages containing custom tags. The result is an web application with identical function and appearance, but where the business logic (the Java code) has been separated from the presentation (the web pages).

1. Introduction

When new technology is introduced, it is not always effective to migrate older applications to use the new technology. Software transformation can reduce both the risk and the cost of the migration.

Java Server Pages (JSP) is one of the popular technologies for building web applications that serve dynamic contents [1]. In the initial incarnation, JSP-based web applications consist of web pages that contain Java code embedded in the HTML. This provides a direct and simple way of implementing dynamic web pages. However, while an improvement over previous technology (program generating HTML in I/O statements), it still mixed application logic expressed as Java code with the presentation provided by the HTML code and JavaScript. Even if most of the logic is separated into a servlet and communicated to a page using a Data Bean, application logic to traverse the results in the bean must be inserted into the page.

The explicit use of scriptlets facilitates rapid prototyping but introduces more complexity in implementation. Such scriptlets interweave all sorts of HTML with Java code, and make code debugging and authoring tricky, and software maintenance and evolution difficult.

The most recent version of JSP provides a functionality called custom tags. HTML is extended with XML markup tags that link to a library of Java classes. These tags may be given abstract names and incorporate all of the decision and iterative logic. This permits the almost total separation of application logic from presentation.

In this paper we examine an analysis and transformation technique which allows us to migrate web applications with embedded Java code to a custom tag implementation. The result is an application with identical function and appearance, but where the application logic expressed in Java code has been removed and replaced with appropriately named custom tags. The Java code is moved to a set of custom classes invoked by the custom tags.

The remainder of the paper is organized as follows. In Section 2, we introduce some related basic concepts and previous work. In Section 3, we present the proposed restructuring technique for Java code transformation. Section 4 presents some preliminary results and in Section 5, we conclude the paper.

2. Background and Related Work

One of the most challenging aspects in parsing JSP pages is code intermingling. In JSP web applications, Java code can be inserted into HTML/JavaScript and HTML/JavaScript can be nested within the Java code. The actual implementation of Java Server pages treats all of the HTML and JavaScript as print statements. Some analysis and transformation of web page takes the same approach. Hassan et al. [6] translate the HTML in web applications to print statements before migrating applications to different server side

translations. Several techniques to analyze the structure of web applications including transforms have been successful [10,11,12]. However these focus on the HTML code that is produced by the server.

The information we are interested in is contained not only in the individual languages, but also in the way they are interwoven. All these characteristics require our JSP parser to have the ability to analyze all the languages in a multilingual file together, rather than separately. In addition, the JSP parser must also have the ability to process the web pages containing errors or other unexpected contents. To address these challenges, we use the grammar presented by N. Synytskyy et al. [13] and modified to recognize Java instead of Visual Basic [7]. This grammar unifies all three languages (JavaScript, HTML and Java) in a single language definition allowing transforms that cross the boundary between Java, HTML and JavaScript.

The grammar is based on island grammars [4] and robust parsing [8,9]. The main strength of island grammars is the ability to separate the incoming text to be parsed into two general categories: interesting islands and uninteresting water. Therefore, 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. This ability will benefit code analysis and fact extraction for the structure transformation. We have further extended the grammar to allow the XML based custom tags as well as other markup.

Most of our implementation is based on the TXL programming language. TXL is a pure functional programming language particularly designed to support rule-based source-to-source transformation [2]. Each TXL program has two parts: a structure specification of the input to be transformed, which is expressed as an unrestricted context-free grammar; and a set of one or more transformation rules, which are specified as pattern/replacement pairs to define what actions will be performed on the input. Each pattern/replacement pair in a transformation rule is specified by example, and may be arbitrarily parameterized for more rule flexibility.

3. Process

Figure 1 illustrates the JSP transformation process implementing our proposed restructuring technique. The whole process is divided into five phases: preprocessing, grouping, tag naming, code transformation, and post-processing. The process is automated except for the tag naming phase where human assistance is required.

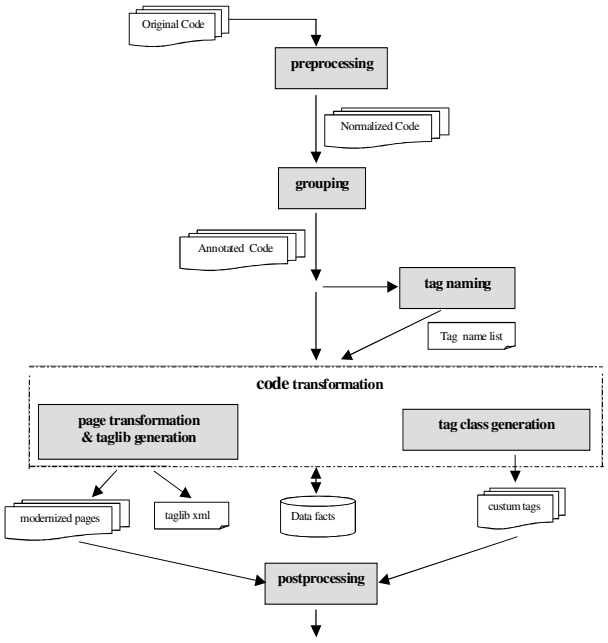

Figure 1. Transformation process

1. *Preprocessing*. The preprocessing phase takes the original source code as its input and produces normalized code by modifying non-scripting elements that need restructuring into scripting elements, extracting HTML content out of out.print() or out.println() statements, merging adjacent scriptlets, encoding code comments, and Java scriptlets embedded in strings. All procedures during the preprocessing phase except the last two are implemented using TXL. The last two procedures are implemented using shell scripts.
2. *Grouping*. The grouping phase takes the normalized code produced in the preprocessing phase as its input and produces annotated code by marking each line of code with a tag id. The tag id indicates that the line of code is going to be moved into a tag that has the same id. This phase is implemented using TXL.
3. *Tag naming*. The tag naming phase takes the annotated code produced in the grouping phase as its input and produces a list of meaningful tag names with the assistance of a human operator. This phase is implemented using TXL and JSP.
4. *Code transformation*. The transformation phase takes each of the annotated source code files and the tag name list produced in the tag naming phase as its inputs and produces three types of outputs: the modernized JSP pages, the tag library XML file, and the custom tag classes. There are two

```
<html ><body >
<%octs.ShoppingCart cart = new octs.ShoppingCart ();
  octs.CDStoreDB storeDB = new octs.CDStoreDB ();
  cart.setItem (request.getParameter ("item"));
  String title;
  String price;
  String par = request.getParameter ("submit");
  if (par.equals ("Purchase")){
    //more Java code to handle the case when customer
    //makes a purchase
    …
    updateNum = DBtest.makeCart(orderNum, custNum);
    String[] items = cart.getItems();
    int[] UPCitems = new int[items.length];
    …
    %>
          Purchase Processing!
<%} else { %>
   <br> You have the following items in your cart :
     <table border>
       <tr>
         <td> Title </td>
         <td> Price </td>
       </tr>
        <%
           String [] items = cart.getItems ();
           for (int i = 0; i < items.length; i ++){ %>
            <tr>
            <%
              ResultSet rs =
                storeDB.cdPrice(Integer.parseInt(items[i]));
              while (rs.next()){
                title = rs.getString (1);
                price = rs.getDouble (2); %>
                <td > <%= title %> </td >
                <td > <%= price %> </td >
            </tr>
              <%}
         }%>
         </table>
<%}%>
</body ></html >
```

Figure 2. Intermixing of HTML and Java code

branches in this phase: Page Transformation and taglib Generation branch, and Custom Tag Generation branch, which share the same fact base extracted from the annotated source code during this phase. This phase is implemented using TXL.

5. *Post-processing*. The post-processing phase takes the modernized JSP pages and custom tag classes from the code transformation phase and re-inserts all comments stripped in the pre-processing phase into appropriate locations in the source code. This phase is implemented using shell scripts.

Consider the JSP page shown in Figure 2, which illustrates the original mixed Java and HTML to be processed. After the transformation process, the same page is modified by inserting newly created custom tags into the page, which is shown in Figure 3.

There are two other options for migrating to custom tags in JSP. The first is the standard tag library. These tags are relatively low level, corresponding to individual java statements with tags for conditional loops, simple conditional statements (such as if statements) and database access. Translating to standard tag libraries is straightforward but still leaves the application code in the web page, just in a different language. The other option is the new SimpleTag interface which is a less complex interface than the CustomTag interface. Our technique can be easily adapted to the SimpleTag interface.

```
<html ><body >
<%@ taglib uri=http://xyzzy.org/oct01-taglib
    prefix="oct01" %>
<oct01:checkout attr_item="item" attr_submit="submit">

  <oct01:purchase>
    Purchase Processing !
  </oct01:purchase>

  <oct01:notPurchase>
    <br> You have the following items in your cart :
    <table border>
      <tr>
         <td> Title </td>
         <td> Price </td>
      </tr>
      <oct01: shoppingCartItems>
       <tr>
        <oct01:eachItem>
         <td> <oct01:ex_title/> </td>
         <td> <oct01:ex_price/> </td>
       </tr>
       </oct01:eachItem>
      </oct01: shoppingCartItems>
    </table>
  </oct01:notPurchase>

</oct01:checkout>
</body ></html >
```

Figure 3. Modified page with custom tags

In the following sections, we will examine each of the phases of the process in more detail.

3.1 Preprocessing

There are several components to the preprocessing phase. The first of these is lexical normalization, which has two elements: scriplets in strings and comments.

TXL has an initial scanning phase which recognizes, among other things, string and character literals. In most cases this is desirable. However sometimes, the string literals may in fact contain scriptlets. If we leave these elements inside of the strings, they will be ignored during parsing. Consider the following two examples:

```
<input type=”text” name=”username” value=
’<%=StringFormat.toHTMLString(request.getPara
meter(“username”))%>’
>

<a href=
”check-
out.jsp?submit=add&item=<mylib:itemi/>”>
Buy this CD!
</a>
```

```
<!-- html comments -->
<html ><body >
<%-- jsp comments --%>
<%
    //one line of java comments
    /* multiple lines of java comments
       java comments again  */
    octs.ShoppingCart cart = new octs.ShoppingCart ();
    octs.CDStoreDB storeDB = new octs.CDStoreDB
...
```

Figure 4. Before comment encoding

```
<commentHandling:htmlComment
          attr1="<!-- html comments -->"/>
<html ><body >
<commentHandling:jspComment attr1=" jsp comments "/>
<%
    CommentCall("//one line of java comment");
    CommentCall("/* multi lines of java comments");
    CommentCall("   java comments again  */");
    octs.ShoppingCart cart = new octs.ShoppingCart ();
    octs.CDStoreDB storeDB = new octs.CDStoreDB
...
```

Figure 5. After comment encoding

In the first piece of code a JSP expression is embedded in the string and a simple custom tag is enclosed within double quotes in the second piece of code. Both of them are interesting elements but would be ignored as part of the string during the parsing. Similar to the preprocessing introduced by Li [7], we use double square quote brackets (`[[...]]`) to replace the single quotes in the first example and the double quotes in the second example before parsing, and then change them back after the transformation. The double square brackets can be chosen because they are not meaningful in the HTML and JSP grammars and we modify the JSP grammar to define the double square brackets as delimiters for attribute values in HTML. The two examples become:

```
<input type="text" name="username" value=
[[<%=StringFormat.toHTMLString(request.getPar
ameter("username"))%>]]
>

<a href=
[[check-
out.jsp?submit=add&item=<mylib:itemi/>]]>
Buy this CD!
</a>
```

Another issue is that different languages have different commenting conventions. In a multilingual JSP page, there are generally three types of comments: HTML comments enclosed within HTML comment tags (`<!-...-->`) , JSP comments enclosed within JSP comment tags (`<%--...--%>`), and normal Java comments (`//...` or `/*...*/`) embedded inside JSP scripting elements. Comments in one language may be perfectly legitimate and meaningful content in another.

We take a similar to the approaches taken by Synytskyy [13] and Hassan et al. [6]. We use a lexical preprocessor to transform HTML and JSP comments

```
<html ><body >
<%
   octs.ShoppingCart cart = new octs.ShoppingCart ();
   octs.CDStoreDB storeDB = new octs.CDStoreDB ();
   cart.setItem (request.getParameter ("item"));
   String title, price;
   String par = request.getParameter ("submit");
   <ifelse id="Block1"> if (par.equals ("Purchase"))
    { %>
      Purchase Processing !
 <%}else{%>
   <br> You have the following items in your cart :
   <table border>
   <tr > <td > Title </td> <td > Price  </td> </tr>
    <%
      String [] items = cart.getItems ();
      int i = 0;
      <whileloop id="Block2"> while (i < items.length) {%>
         <tr >
         <%
            ResultSet rs = storeDB.cdPrice (Integer.parseInt (items [i]));
            <whileloop id="Block3"> while (rs.next ()){
               title = rs.getString (1);
               price = rs.getDouble (2);
               %>
                 <td > <% = title %></td >
                 <td > <% = price %></td ></tr >
               <%} </whileloop>
                i ++;
           } </whileloop>
        %>
   </table>
  <%}</ifelse>
%>
</body ></html >
```

Figure 6. Intermediate annotated code

into an attribute value of new custom tag, and to transform Java comments into the parameter of a function call (`CommentCall()`).

Figure 4 shows part of a JSP page before comments encoding, and Figure 5 shows the same page after comments encoding. Once the code modernization is performed, we can use the comment-handling custom tags and Java function calls as place holders to re-insert all comments that have been encoded in the preprocessing phase.

We also perform some simple normalization of the code to make the final transform easier. For loops are converted to equivalent while loops, and some if statements with else blocks are separated into two if statements (one for the then part, one for the else part).

3.2 Grouping

The grouping phase identifies which part of the Java code in a JSP page is to be migrated into each custom tag. It does this by annotating each Java statement with a tag id. The tag id associated with each statement indicates that the line of code is to be migrated into a tag with the same id. This phase can be divided into the following two steps.

Step 1. We assign a unique tag id to each control or loop block that contains at least one segment of HTML code using scope rules presented elsewhere

```
<html ><body >
<%
   <tag id="Block0"> octs.ShoppingCart cart = new octs.ShoppingCart ();
   <tag id="Block0"> octs.CDStoreDB storeDB = new octs.CDStoreDB ();
   <tag id="Block0"> cart.setItem (request.getParameter ("item"));
   <tag id="Block0"> String title, price;
   <tag id="Block0"> String par = request.getParameter ("submit");
   <tag id="Block0"> <ifelse id="Block1"> if (par.equals ("Purchase"))
   <tag id="Block1_Then">{%>
      Purchase Processing !
  <%} else
     <tag id="Block1_Else">{ %>
       <br > You have the following items in your cart :
       <tag id="Block2"> < table  border >
       <tr ><td > Title </td> <td > Price </td> </tr>
        <%
         <tag id="Block2_Block1_Else">String [] items = cart.getItems ();
         <tag id="Block2_Block1_Else">int i = 0;
         <tag id="Block2_Block1_Else"><whileloop id="Block2"> while (i < items.length)
         <tag id="Block2_Block1_Else"> {%>
           <tr >
            <%
              <tag id="Block2_Block1_Else">ResultSetrs =storeDB.cdPrice(Integer.parseInt(items[i]));
              <tag id="Block3_Block2_Block1_Else"> <whileloop id="Block3"> while (rs.next ())
              <tag id="Block3_Block2_Block1_Else"> {
              <tag id="Block3_Block2_Block1_Else"> title = rs.getString (1);
              <tag id="Block3_Block2_Block1_Else"> price = rs.getDouble (2);
            %>
           <td > <% = title %></td ><td > <% = price %></td ></tr >
          <%}</whileloop>
         <tag id="Block2_Block1_Else"> i ++;
       }</whileloop>
     %>
   </table>
  <%}</ifelse>
%>
</body ></html >
```

Figure 7. Annotated code

[15]. These segments may only occur in sub-scopes. That is, inside constructs such as loops, switch statements, and subroutines. The annotation for each block has the form of

```
<block-type id=tag-id> ... </block-type>
```

The block-type tag is `ifelse` for a control block and `whileloop` for a loop block. We use a tag id of the form of "`Block#`", and "`#`" is a number starting from one and increased by one for each block, which is automatically generated in this step. Figure 6 shows the output of for the example code in Figure 2. Some formatting has been changed to better fit the proceedings.

Step 2. Based on the tag id assigned and block type recognized during the first step, we annotate each line of code with an appropriate tag id if this line of code belongs to the same scope as the tag id. During the annotation, we preserve the nesting relationship in the assignment of each id. For example, if a line of code in "`Block2`" is nested inside "`Block1`", we will assign this line of code a tag id equal to "`Block2_Block1`". Figure 7 gives the annotated code for the same example. Note that "`Block0`" represents the topmost scope of this page, and we call it the main block for each JSP page. These annotations not only identify which tag but also are used to hold extra attribute information as part of the transformation process.

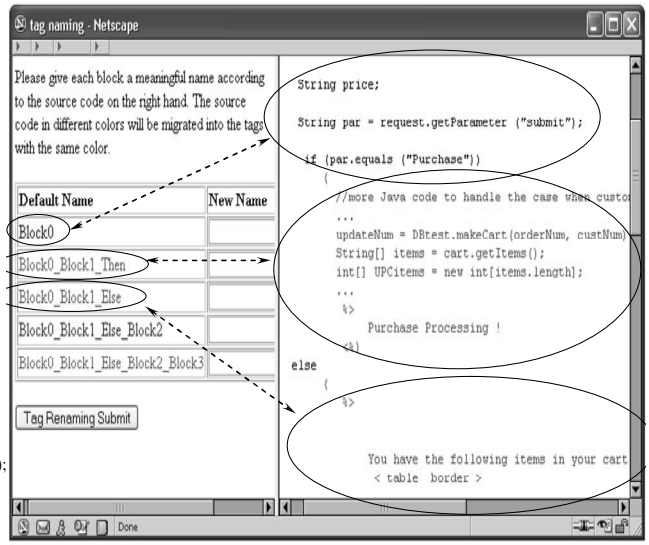

Figure 8. Tag naming interface before a renaming

According to the annotated code shown in Figure 7, we can tell which line of Java code will be migrated into each tag, how each custom tag should be used, and what kind of relationship holds between the custom tags. This approach also provides for future development of our technique. The current grouping transform takes a greedy approach, maximizing the code that should be put into each tag. Data Flow analysis combined with clone detection can be used to identify and mark code that should be put in separate tags such as the code for session management.

3.3 Tag naming

Up until now, each tag has a default name (i.e. the tag id shown in Figure 7), which is less than meaningful for application programmers. In order to determine a meaningful name for each tag, we allow human beings to make the decision using their business and application knowledge. To make the tag naming simple, this phase takes the annotated code as its input and uses a web interface to assist application programmers to decide on meaningful tag names.

Figure 8 shows the web interface that is composed of two columns. The right column displays the source code of a JSP page with each code block in a different color. The left column of this interface displays a table that shows a default name for each tag in the color corresponding to the code block. That is, the Java code in red will be migrated into the tag with a red tag id "`Block0_Block1_Then`". Since color is not

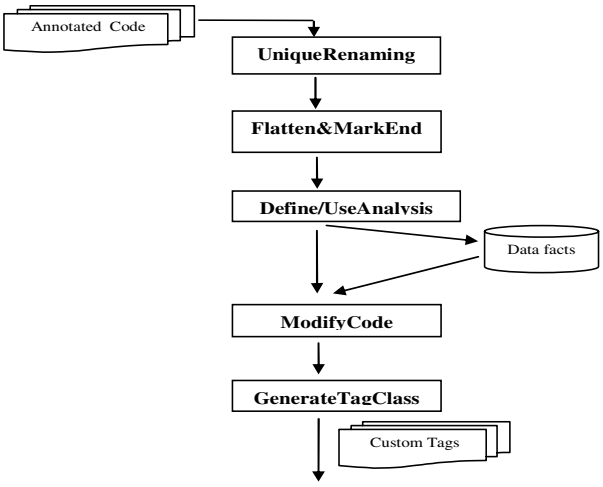

Figure 9. Custom tag generation

available in the proceedings, we have indicated corresponding colors by adding circles and lines to figure 8. Based on the unchanged source code presented in the right column and his/her own business and system knowledge, an application developer may choose a new name for each tag, and fill the form in the left column. A list of meaningful tag names for each JSP page can be produced based on the application developers' knowledge, and the list is added to a fact base for future use.

3.4 Code transformation

The code transformation phase takes each of the annotated source code files and the tag name list as its inputs and produces three sets of results as well as some data facts. There are two branches in this phase. The modernized JSP pages and the tag library description xml file are generated by the Page Transformation and *taglib* Generation branch. The custom tag classes are generated by the Custom Tag Generation branch. Now let us take a closer look at these two branches.

3.4.1 Tag class generation. Figure 9 shows the structure of the tag class generation branch, which is composed of five sub-phases: UniqueRenaming, Flatten&MarkEnd, Define/UseAnalysis, ModifyCode, and GenerateTagClass.

1) UniqueRenaming In the first sub-phase, a unique id (UID) is assigned to the declaration of each entity of each JSP file and all references to the same entity are annotated with the same assigned UID. Here the UID of an entity has the similar form as that used by Guo et al. [5] and Li [7] except that our UID includes the tag id. Our UID is in the form:

```
...
  <tag id="Block2_Block1_Else"> String[]
            <uid id="items ex.jsp Block2_Block1_Else">items</uid>
                = <uid id="cart ex.jsp Block0"> cart </uid>.getItems ();
...
```

Figure 10. Partial output from UniqueRenaming

```
...
  <tags id="Block3_Block2_Block1_Else">
    <tag id="Block3_Block2_Block1_Else">
        <whileloop id="Block3_Block2_Block1_Else">
            <uid id="rs ex.jsp Block2_Block1_Else"> rs </uid>.next ();
    <tag id="Block3_Block2_Block1_Else" location="inloop">
            <uid id="title ex.jsp Block0"> title </uid>
              = <uid id="rs ex.jsp Block2_Block1_Else"> rs </uid>.getString (1);
    ...
    %>
    <td >
      <% = <uid id="title ex.jsp Block0"> title </uid> %>
    </td>
    ...
  </tr>
  <%
    </tags id="Block3_Block2_Block1_Else" type="whileloop">
    <tag id="Block2_Block1_Else" location="afterBody">
<uid id="i ex.jsp Block2_Block1_Else"> i </uid> ++;
...
```

Figure 11. Partial output from Flatten&MarkEnd

```
"entity_name
enclosing_class_and_interface_name
package_name file_name tag_id"
```

The tag id included in the UID of an entity indicates the tag class in which the entity is first declared. This extra information about tag id will make the later analysis on the use/definition of an entity more convenient. For example, the line of code shown in Figure 10 declares the variable `items`, which has been assigned a unique identifier of "`items ex.jsp Block2_Block1_Else`". We can also see that the variable "`cart`" has a UID of "`cart ex.jsp Block0`".

2) Flatten&MarkEnd. This sub-phase is responsible for marking the starting point and ending point for each tag block and collecting more information about the migration decision for each line of code. As an example, this sub-phase replaces the opening and closing brace brackets of the marked conditional and loop blocks using `<tags>` and `</tags>` to reduce nesting complexity. This sub-phase also identifies the condition expressions of the statements with these marked blocks.

The sub-phase collects the information about which statements work as part of loop-body for an iteration tag, suggesting that the code be migrated into both `doStartTag()` and `doAfterBody()` methods for the tag. It also determines which line of code

```
…
  <tags id="Block3_Block2_Block1_Else">
  <tag id="Block3_Block2_Block1_Else"> ex_Block2_Block1_ElseTag parent_Block2_Block1_Else =
      (ex_Block2_Block1_ElseTag) findAncestorWithClass (this, ex_Block2_Block1_ElseTag.class);

  <tag id="Block3_Block2_Block1_Else"> if (parent_Block2_Block1_Else == null)
  throw new JspTagException ("Block3_Block2_Block1_ElseTag not in ex_Block2_Block1_ElseTag");
  …
  <tag id="Block3_Block2_Block1_Else" location="inclass"> ResultSet
       <uid id="loopvar_rs ex.jsp Block3_Block2_Block1_Else"> loopvar_rs </uid>;

  <tag id="Block3_Block2_Block1_Else">
       <uid id="loopvar_rs ex.jsp Block3_Block2_Block1_Else"> loopvar_rs </uid>
           = parent_Block2_Block1_Else.getrs ();

  <tag id="Block3_Block2_Block1_Else"> <whileloop id="Block3_Block2_Block1_Else">
       <uid id="loopvar_rs ex.jsp Block3_Block2_Block1_Else"> loopvar_rs </uid>.next ();

  <tag id="Block3_Block2_Block1_Else" location="inloop"> parent_Block0.settitle
      (<uid id="loopvar_rs ex.jsp Block3_Block2_Block1_Else"> loopvar_rs </uid>.getString (1));
  %>
  <td >
       <% = <uid id="title ex.jsp Block0"> title </uid> %>
  </td>
  …
  </tr>
 <%
  </tags id="Block3_Block2_Block1_Else" type="whileloop">
  <tag id="Block2_Block1_Else" location="afterBody">
     <uid id="i ex.jsp Block2_Block1_Else"> i </uid> ++;
…
```

Figure 12. Output from ModifyCode

must be migrated into `doAfterBody()` method for a tag and which will be marked up with "`afterBody`" location. the collected migration location information is stored in the annotation of each line of code as shown in Figure 11. A line of code containing the "`inloop`" annotation should be migrated into both `doStartTag()` and `doAfterBody()` methods as part of a loop body for while-loop blocks or into `doStartTag()` method as part of if-statement body for if-else blocks.

A statement containing the "`afterBody`" annotation will be migrated into the `doAfterBody()` method only. Statements containing the "`inclass`" annotation will be migrated into tag classes as a class member. A line of code without a location annotation will just be migrated into the `doStartTag()` method.

3) Define/UseAnalysis. This sub-phase ensures that each variable used by other tags will have a get method in the tag class, and each variable that works as an attribute for its declaration tag class will have a set method. Each variable that is related to a `get` or `set` method will be declared as an instance variable and not as a local variable of any method in the tag class. This sub-phase also collects names and types for all of the tags in each JSP source file. All of the collected information is stored in the fact base for future use. Although not yet implemented, JSP declarations (<%! … *variable or class definition* … %>) can be handled by assigning them to an appropriate tag class and ensuring that the other tag classes have access to them using get and set methods.

```
…Java import statements…
public class eachItemTag extends BodyTagSupport {
  ResultSet loopvar_rs;
  public int doStartTag () throws JspTagException {
   int action = SKIP_BODY;
   try {
    shoppingCartItemsTag parent_shoppingCartItems =
        (shoppingCartItemsTag)findAncestorWithClass(this,
                        octs.shoppingCartItemsTag.class);
    if (parent_shoppingCartItems == null)
throw new JspTagException("eachItemTag not in shoppingCartItemsTag");
    checkoutTag parent_checkout=(checkoutTag)
        findAncestorWithClass(this,octs.checkoutTag.class);
    if (parent_checkout == null)
     throw new JspTagException("eachItemTag not in checkoutTag");
    loopvar_rs = parent_shoppingCartItems.getrs();
    if (loopvar_rs.next()){
      parent_checkout.settitle(loopvar_rs.getString(1));
      parent_checkout.setprice(loopvar_rs.getDouble(2));
      action = EVAL_BODY_INCLUDE;
    }else{ action = SKIP_BODY; }
   } catch(Exception e){
       System.out.println("error in eachItemTag "+e);}
   return action;
  }
  public int doAfterBody () throws JspTagException {
   int action = SKIP_BODY;
   try {
    checkoutTag parent_checkout=(checkoutTag)findAncestorWithClass(
                                        this,
octs.checkoutTag.class);
    if (parent_checkout == null)
     throw new JspTagException ("eachItemTag not in checkoutTag");
    if (loopvar_rs.next ()){
      parent_checkout.settitle (loopvar_rs.getString (1));
      parent_checkout.setprice (loopvar_rs.getDouble (2));
      action = EVAL_BODY_AGAIN;
    }else{ action = SKIP_BODY; }
   } catch(Exception e){
     System.out.println ("error in eachItemTag "+e);}
   return action;
  }
```

Figure 13. A tag class with an iteration action

4) ModifyCode. This sub-phase modifies the code of each JSP file by inserting the declarations for the `get` and `set` methods of each tag based on the data facts extracted in the previous sub-phase, and annotates the inserted code with the appropriate positions. This sub-phase also replaces each variable that is not declared but is used in the current tag using its parents' `get` method, and adds and annotates its parent validation code for the variable.

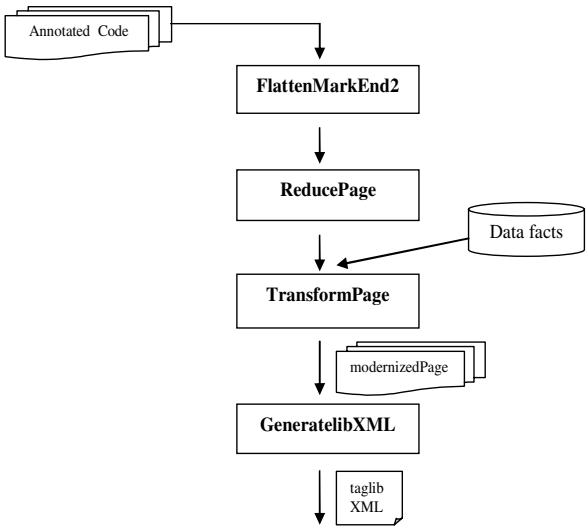

Figure 14. Page transformation and *taglib* xml generation

This sub-phase also inserts new statements to implement iteration actions for some of the marked loops, such as the marked *while loop* with an id of "Block3" shown in Figure 7. The variable "rs", which is used in the while loop condition expression rs.next() in the block "Block3_ Block2_ Block1_Else", must be declared as an instance variable in its declaration tag class and initialized in the doStartTag() method of the tag class. Therefore, the variable "rs" will be renamed as "loopvar_rs", and two new lines of Java code will be inserted into the page (shown in Figure 12) to declare "loopvar_rs" as an instance variable of type "ResultSet", and to initialize this variable in the doStartTag() method of the class. The prefix "loopvar_" is used as a simple technique to ensure a unique name. We must also rename all references to this variable.

5) GenerateTagClass. This last sub-phase takes the results from the previous sub-phase and generates the tag classes for each JSP file. To generate a tag class, we build the class body in the order from the innermost scope to the outermost scope.

For example to build a tag class without an iteration action, we first collect all statements and declarations annotated as "inloop" to build an if-else statement element for the doStartTag() method, and collect all other elements without any location annotation and use them to build the try-catch block for the doStartTag() method. Second, we build the doStartTag method body and header, and annotate this newly created method as "inclass". Then, we collect all statement and declaration elements annotated as "inclass" to build the class body. Finally, we build the class by creating its class header encapsulating the constructed class body.

```
<html><body>
<%
   <tags id="Block0">
   <tags id="Block1_Then">
%>
Purchase Processing !
<%
   </tags id="Block1_Then">
   <tags id="Block1_Else">
%>
<br>
You have the following items in your cart :
<tag id="Block2"> <table  border>
   <tr>
      <td > Title </td>
      <td > Price </td>
   </tr>
    <%
      <tags id="Block2_Block1_Else">
    %>
   <tr >
    <%
      <tags id="Block3_Block2_Block1_Else">
    %>
    <td > <%=title %> </td>
    <td > <%=price %>  </td>
   </tr>
    <%
      </tags id="Block3_Block2_Block1_Else">
      </tags id="Block2_Block1_Else">
    %>
</table>
<%
   </tags id="Block1_Else">
   </tags id="Block0">
%>
</body></html>
```

Figure 15. Output of sub-phase ReducePage

The difference between building a tag class without iteration action and building a tag class with iteration action is that we need to build the doAfterBody() method as well as the doStartTag() method and also annotate it as "inclass". To build the doAfterBody() method body, we collect all elements annotated as "afterBody" and combine them with the if-else statement that has been constructed for the doStartTag() method to build try-catch block for doAfterBody() method.

To automatically generate tag classes based on the syntax and the collected information from the previous sub-phases, we have four templates based on JSP custom tag cases[15]: a tag class for main block / block without iteration action, a tag class for if-then/if-else block without iteration action, a tag class with iteration action and a simple tag class for JSP-expression. Figure 13 shows one of the tag classes generated for the example. The name of the class is taken from the facts generated by the tag naming phase..

3.4.2 Page transformation and *taglib* xml generation. Once the classes implementing the tags have been generated, the next step is to generate the new web application pages where the embedded Java code

has been replaced by custom tags. The process, shown in Figure 14 takes 4 steps.

1) FlattenMarkEnd2. The first sub-phase is different from the Flatten&MarkEnd sub-phase in the custom tag generation branch. This sub-phase identifies and marks only the ending point for the main block of each JSP file to make sure all of the children tags are properly nested. It removes the opening brace brackets of the marked control or loop blocks, and replaces the closing brace brackets of the same marked blocks by `</tags>` markup.

2) ReducePage. This sub-phase identifies and marks the starting point, and also adds a `<tags>` markup for each tag block in each page. At the same time, it removes all Java elements from the page except for the `<tags>` and `</tags>` markup. Figure 15 shows the example output of this sub-phase for the example page of Figure 2. From the output, we can see that all Java statements or declarations code have been removed from the page, and the starting point and the ending point for each tag block has been indicated in the page.

3) TransformPage. This sub-phase takes the output of ReducePage in combination with the data facts extracted by the Custom Tag Generated Branch to produce a migrated JSP page for each original page. This is the final result that was shown in Figure 3. From the output, we can see that the *taglib* directive has been added to the page, and for each tag block the `<tags>` and `</tags>` markup has been replaced by opening and closing tags. If there are any existing attributes for a tag, all attribute/value pairs of this tag are also added to the open tag.

4) GeneratelibXML. The last sub-phase takes the output of TransformPage and produces an xml file, which is the tag library description file for the web application system. Figure 16 shows part of the output of this sub-phase for the same page. From the output, we can see that the custom tag `checkout` is implemented by the Java class `octs.checkout.class`, whose body content is JSP type and two required attributes are `attr_submit` and `attr_item`, respectively.

4. Preliminary Results

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

```
<tag>
   <name>checkout</name>
   <tag-class>octs.checkoutTag</tag-class>
   <body-content>JSP</body-content>
   <description></description>
   <attribute>
     <name>attr_submit</name>
     <required>true</required>
     <rtexprvalue>false</rtexprvalue>
     <description></description>
   </attribute>
   <attribute>
     <name>attr_item</name>
     <required>true</required>
     <rtexprvalue>false</rtexprvalue>
     <description></description>
   </attribute>
</tag>

<tag>
   <name>purchase</name>
   <tag-class>octs.purchaseTag</tag-class>
   <body-content>JSP</body-content>
   <description></description>
</tag>
…
```

Figure 16. Output of sub-phase generatelibXML

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 generated.

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 classess.

5. Future Work and Conclusions

One of the main opportunities in future work is to improve the java grouping phase. The current approach uses a greedy algorithm to group as much code together as possible. However this is not always optimal. In many web applications there may be Java code to deal with session management at the beginning of each page. Ideally this code could be identified by the use of method invocations on the session object and placed in a single tag class that can be shared among all pages.

Near miss clone detection [3,14] can be used to identify common scriptlets that can also be shared. The differences between the similar scriplets can be resolved to produce a tag class library invoked from parameterized custom tags. Both of these possibilities are currently under investigation.

In this paper, we have presented a technique that restructures a JSP-based web application by transforming embedded Java code of JSP pages into custom tags, without changing the original functionality, user interface or code comments of the application. To implement this technique, we used a robust multilingual

parser technique using island grammars for program understanding, design recovery, unique renaming, and source-to-source TXL transformation techniques for program analysis and transformation. After the transformation process, the restructured pages with eliminated scriptlets are easier to debug and test without affecting run-time performance. Moreover, all business logic intensive Java code in the JSP pages has been moved and encapsulated into custom tag classes, and all elements for presentation are kept in the pages, which reduces the complexity of web applications and makes the restructured applications more maintainable.

References

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

[2] J.R. Cordy, "TXL – A Language for Programming Language Tools and Applications", *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications, Electornic Notes* in *Theoretical Computer Science 110,* Dec. 2004, pp. 3-31.

[3] J.R. Cordy, T. Dean, N. Synytskyy, "Practical Language-Independent Dectection of Near Miss Clones", *Proc. CASCON 2004, 14th IBM Center for Advanced* Studies *Conference*, Toronto, October 2004, pp. 29-40.

[4] van Deursen and T. Kuipers, "Building Document Generators", *Proc. Int.* Conference *on Software Maintenance (ICSM 99)*, Oxford, England, pp. 40-49.

[5] X. Guo, J.R. Cordy and T.Dean, "Unique Renaming of Java Using Source Transformation", *IEEE 3rd International Workshop on Source Code Analysis and Manipulation*, Amsterdam, p151-160, September 2003.

[6] A. E. Hassan and R.C. Holt, "Migrating Web Frameworks Using Water Transformations", *Proceedings of COMPSAC 2003: International Computer Software and Application Conference*, Dallas, Texas, USA, p296-303, November 2003.

[7] X. Li, *Defining and* Visualizing *Web Application Slicing Using Design Recovery*, M.Sc. Thesis, Queen's University, 2004.

[8] L. Moonen, "Generating Robust Parsers using Island Grammars", *Proc 8th International Workshop on Reverse Engineering*, Stuttgart, Germany, October 2001, pp 13–22.

[9] L. Moonen, "Lightweight Impact Analysis using Island Grammars", *Proc 10th International Workshop on Program Comprehension*, Paris, France, June 2002, pp 343–352.

[10] F. Ricca, P. Tonella, and Ira D. Baxter, "Web Application Transformations based on Rewrite Rules", *Information and Software Technology*., vol. 44, n. 13, pp. 811-825, October 2002

[11] F. Ricca, and P. Tonella, "Web Application Slicing", *IEEE* International *Conference on Software Maintenance*., Florence, Italy, November 2001, pp. 148-157.

[12] F. Ricca, "Analysis, Testing and Re-Structuring of Web Applications", *Proc. of ICSM'2004,* International *Conference on Software Maintenance*, pp. 474-478, Chicago, *Illinois, 11-14 September 2004*

[13] N. Synytskyy, J.R. Cordy and T.R.Dean, "Robust Multilingual Parsing Using Island Grammars", *Proc. 2003 13th IBM Centres for Advanced Studies Conference*, Toronto, p149-161, October 2003.

[14] N. Synytskyy, J.R. Cordy and T.R.Dean, "Resolution of Static Clones in Dynamic Web Pages", *Proc. WSE 2003, IEEE 5th International Workshop on Web Site Evolution*, Amsterdam, p49-58, September 2003.

[15] S. Xu and T. Dean, "Modernizing Java Server-Pages", being submitted to WSE 2005.