TRANSFORMATION OF ROUND-TRIP WEB APPLICATION TO USE AJAX

by

JASON CHU

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Science (Engineering)

Queen’s University
Kingston, Ontario, Canada
June 2008

Copyright © Jason Chu, 2008
Abstract

AJAX is a web application programming technique that allows portions of a web page to be loaded dynamically, separately from other parts of the web page. This gives the user a much smoother experience when viewing the web page. This technique also conserves bandwidth by transmitting only new data relevant to the user, keeping all other content on the web page unchanged. The migration from traditional round-trip web application to AJAX-based web application can be difficult to implement due to the many details required by AJAX.

In this thesis, an approach is presented to automate the process of AJAX conversion using source transformation and backward slicing techniques. The result is an AJAX-based web page that will enhance the user experience and also conserve bandwidth.
Acknowledgments

I would like to thank my supervisor, Dr Thomas R. Dean, for his support and guidance during the work of this thesis. This thesis is possible because of his supervision and directions.

I thank A&L Computer Software Limited for their financial support of my studies. This thesis would not have been possible without their support.

Lastly, I would also like to thank my family and my fiancée for their continual support during my studies.
# Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iii

List of Tables vi

List of Figures vii

Glossary ............................................................... 1

Chapter 1:

Introduction ......................................................... 2

1.1 Motivation ......................................................... 3

1.2 Thesis Outline ................................................... 4

Chapter 2:

Background ............................................................. 6

2.1 TXL ................................................................. 6

2.2 Backward Slicing ................................................. 10

2.3 AJAX ............................................................... 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 Web Transformations</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Migration To AJAX</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 3:</td>
<td></td>
</tr>
<tr>
<td>AJAX Transformation Overview</td>
<td>15</td>
</tr>
<tr>
<td>3.1 AJAX Table Handling</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Transformation Process</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>25</td>
</tr>
<tr>
<td>Chapter 4:</td>
<td></td>
</tr>
<tr>
<td>XML Generation</td>
<td>26</td>
</tr>
<tr>
<td>4.1 AJAX Transformation Grammar</td>
<td>27</td>
</tr>
<tr>
<td>4.2 User Markup</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Source Preparation</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Slicing Preparation</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Slicing</td>
<td>39</td>
</tr>
<tr>
<td>4.6 XML Tagging</td>
<td>44</td>
</tr>
<tr>
<td>4.7 Clean Up</td>
<td>45</td>
</tr>
<tr>
<td>Chapter 5:</td>
<td></td>
</tr>
<tr>
<td>Web Page Transformation</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Source Preparation</td>
<td>50</td>
</tr>
<tr>
<td>5.2 Other Dynamic Data Replacement</td>
<td>51</td>
</tr>
<tr>
<td>5.3 Exterior Print Replacement</td>
<td>54</td>
</tr>
<tr>
<td>5.4 Interior Print Removal</td>
<td>55</td>
</tr>
<tr>
<td>5.5 Clean Up</td>
<td>60</td>
</tr>
</tbody>
</table>
5.6 AJAX Base Code Merge ............................................. 62
5.7 Other Dynamic Data Outputting .................................. 64

Chapter 6:

Examples ................................................................. 66
6.1 GotoCode Applications ............................................ 67
6.2 JSPBLOG ............................................................... 81

Chapter 7:

Conclusion ................................................................. 88
7.1 Summary ............................................................... 88
7.2 Thesis Contribution .................................................. 90
7.3 Future Work .......................................................... 91
7.4 Conclusion ............................................................ 96

Bibliography .............................................................. 98

Appendix A:

GotoCode Forum User-Marked Front Page ...................... 103

Appendix B:

GotoCode Forum XML Web Service ............................. 117

Appendix C:

GotoCode Forum AJAX Page ....................................... 126

Appendix D:

GotoCode Forum Supporting Javascript ........................ 139
List of Tables

7.1 Survey of websites’ paging control . . . . . . . . . . . . . . . . . . . . . 94
List of Figures

2.1 TXL transformation overview ........................................ 7
2.2 Basic TXL grammar ...................................................... 7
3.1 Dynamic and static data differences ................................. 17
3.2 Dynamic and static data merging in JavaScript .................... 18
3.3 Process flow of XML generation ...................................... 19
3.4 Process flow of transformation to AJAX ............................. 22
4.1 Example of marked code ................................................. 27
4.2 Grammar of markup tags ............................................... 28
4.3 Example of annotated code ............................................. 29
4.4 Grammar of annotation tags .......................................... 29
4.5 Before extract interesting elements .................................. 32
4.6 After extract interesting elements .................................... 32
4.7 Before JSP expression to print statement ............................ 33
4.8 After JSP expression to print statement .............................. 34
4.9 Combining disjoint scriptlet sections ................................. 34
4.10 Before JSP scriptlet to block ......................................... 35
4.11 After JSP scriptlet to block ........................................... 35
4.12 One print statement with variables and strings .................... 36
6.1 GotoCode Forum before transformation ............................ 68
6.2 GotoCode Forum after transformation ............................ 69
6.3 GotoCode Classifieds marked code ............................... 70
6.4 GotoCode Classifieds web service code ........................... 71
6.5 GotoCode Classifieds web service data ............................ 72
6.6 GotoCode Classifieds dynamic data removed ....................... 72
6.7 GotoCode Classifieds support JavaScript method .................. 73
6.8 GotoCode Classifieds before transformation ....................... 74
6.9 GotoCode Classifieds after transformation ......................... 75
6.10 Marked and annotated sections of GotoCode Events ............... 76
6.11 GotoCode Events web service code ............................... 77
6.12 GotoCode Events web service data ............................... 78
6.13 GotoCode Events dynamic data removed ......................... 79
6.14 GotoCode Events support JavaScript method ..................... 79
6.15 GotoCode Events before transformation ........................... 80
6.16 GotoCode Events after transformation ............................ 81
6.17 JSPBLOG marked code ........................................... 82
6.18 JSPBLOG web service code ....................................... 83
6.19 JSPBLOG web service data ....................................... 84
6.20 JSPBLOG dynamic data removed .................................. 85
6.21 JSPBLOG support JavaScript method ................................ 86
6.22 JSPBLOG before transformation .................................... 87
6.23 JSPBLOG after transformation ..................................... 87
Glossary

AJAX  Asynchronous JavaScript and XML

API  Application Programming Interface

ASP  Active Server Pages

CPU  Central Processing Unit

HTML  HyperText Markup Language

JSP  JavaServer Pages

PHP  Hypertext Preprocessor

SQL  Structured Query Language

TXL  A source code transformation tool

URL  Uniform Resource Locator

XML  Extensible Markup Language

XMLVM  A XML-based programming language
Chapter 1

Introduction

The World Wide Web was introduced in 1990. Since then, there have been many advances in web technologies. A web application is similar to any traditional desktop application, except that it is executed from a browser through the interfaces of web pages. The application code of a web application is located on the server which hosts the application. Users of the application access it using a web browser, browsing at the URL of the application.

Web applications can be implemented with many different web application languages such as JSP, ASP, or PHP. Each of these languages is considered a server-side language, since the source code of the application is compiled and executed on the server. In general, an application written in a server-side language generates an HTML web page which is to be viewed by the user. In return, the user interacts with the application by sending responses using a web form or selecting a link on the generated pages. The server uses the submission of the user to generate another web page, and the cycle continues.

JavaScript is a language which operates on the user’s browser, and is considered
a client-side language. Unlike HTML, which is used to format and present a web page layout to the user, JavaScript is a scripting language capable of most of the basic programming language capabilities. One of the common uses of JavaScript is to check forms, which checks the values provided by the user prior to submitting them to the server.

Around the year 2000, the ability to read and write XML and to communicate with the server independently of the browser get/submit actions was added to JavaScript, which is now called Asynchronous JavaScript and XML (AJAX). This new approach allows many new behaviors of web pages which had not been possible. The main use of AJAX is to update a portion of the web page without reloading the entire page. This gives the user a much smoother web browsing experience and in some cases also conserves bandwidth.

Migrating legacy round-trip web applications to utilize AJAX is a difficult task due to the many details involved. An automated transformation would make this task easier and more reliable. As a first step in this process, we propose an approach which automates the transformation of a class of round-trip web page to utilize AJAX.

1.1 Motivation

As the business logic of a web application becomes complicated, representing that flow of logic in a web page interface can become very difficult without confusing the user. Due to the large amount of information needed to be shown to the user, the web page can become crowded with information. For some users, the most jarring part comes when the user submits their request to the server, which triggers a reload of the entire page. This reload causes the user to disorient from the flow of the application,
making the application difficult to use. The reload also reloads many unchanged parts of the web page which wastes bandwidth and server CPU processing time. By utilizing AJAX, the user can experience a much smoother experience, keeping the user in flow with the application. AJAX allows the parts of the web page that require updating to be updated, while leaving the other sections unchanged.

The way that AJAX is incorporated into a web page varies between developers; however, all of the different ways require many steps, and details are needed to ensure that the content of the page remains the same after the transformation. An automated transformation can ensure that all the web pages in an application are transformed consistently and completely. Doing the same migration manually may lead to missing lines. In the case where there are multiple developers performing the transformation, the way AJAX is incorporated into a web page may even vary. By automating the migration, consistency and accuracy are ensured.

1.2 Thesis Outline

This thesis consists of seven chapters. Chapter 2 provides background information on the various technologies and techniques involved in this research. An introduction to TXL is provided along with Island Grammars and Agile Parsing used in TXL. The backward slicing technique and the AJAX technology are explained. Finally, some related works involving web transformations and AJAX migration are presented.

Chapter 3 gives an overview of the transformation process: Identification of data that qualifies for our AJAX transformation, the way that data is transformed into XML, and the web page transformation to use JavaScript in parsing and presenting the previously generated XML data. An overview is also given on the way AJAX is
used in handling table data.

Chapter 4 discusses the implementation of the transformation with the use of TXL. The generation of AJAX XML data is broken down into source code preparation, slicing preparation, the actual backward slicing, and the post-slicing processes which generate the XML data.

Chapter 5 explains the transformation of the original web page to use AJAX. This involves the replacement of the dynamic data with a SPAN tag, the merging of static data with the XML data, and the adding of JavaScript code to execute the AJAX functions.

Chapter 6 gives the results from running the transformation process through several web pages. Chapter 7 concludes the thesis with a summary of the entire research. A list of future improvements is given, showing how this research can be expanded.
Chapter 2

Background

In the research and development of the proposed migration approach, there are many different technologies and tools that make this possible. This chapter summarizes each of the technologies and the transformation tool involved. We have also summarized related research to give the reader a better understanding of the thesis.

2.1 TXL

TXL is a source transformation tool that takes a piece of source code and transforms it based on a given grammar(s) and transformation rule(s) [3]. The grammar refers to the grammar of the language of the input source code. In this thesis, the grammars used are the grammars of web languages such as HTML, JSP and JavaScript. The grammar provides TXL a guide to parse the input source code. The rule defines how the transformation of the source code will proceed.

A TXL program operates in three parts. First, the input source code is parsed according to its grammar. A parse tree is generated from the input. Second, the parse
tree is transformed according to the rules of the program. Lastly, the transformed
parse tree is unparsed, producing the output of the transformation. Figure 2.1 shows
an overview of the TXL transformation process.

![Figure 2.1: An overview of the transformation process of TXL. (from [3])](image)

The TXL grammar is written mainly with the “define” statement. Each define
statement gives one nonterminal type a set of alternative patterns. Each alternative is
separated by the vertical bar “|”. Figure 2.2 shows an example of a define statement.
This define statement defines the pattern of a simple expression. The definition defines
that the expression is composed of either a number, an expression plus a number, or
an expression minus a number. The alternatives are matched in succession, and the
second and third alternatives are matched after the first pattern is matched and failed.

```
define expression
    [number]
    | [expression] + [number]
    | [expression] - [number]
end define
```

![Figure 2.2: A basic TXL grammar showing the expression definition.](image)
The TXL transformation rules operate using a pattern and replacement method. A pattern defines a particular way a piece of code is constructed and searched for. This pattern is replaced with a replacement code. The replacement code has to be the same type of tree (as defined by the grammar) as the code being replaced. This ensures the transformation does not generate erroneous code. There are two types of transformation rules offered by TXL. A TXL function does a basic search and replace as described above. A TXL rule does a search and replace recursively. Finally, a TXL program contains a combination of rules and functions, which together make up a single TXL transformation. A TXL program contains a main rule or function which may execute other rules and functions.

2.1.1 Island Grammars

Island grammars are a concept introduced by Van Deursen et al. [26]. Synytskyy et al. made use of an island grammar for parsing web applications which contain multiple languages. Island grammars separate code into two categories. An island is a piece of code which is of interest to the transformation. Water is any pieces of code that is not an island, which would be of no interest to the transformation. With islands, the full detail of its grammar is defined, as these parts of the code will be transformed. With waters, grammars are not needed. This will speed up the transformation process by only focusing on elements of the source code which are of interest.

With TXL, the implementation of island grammar is easy with its sequential grammar matching. As mentioned above, each alternative pattern to a define statement is matched sequentially. For an island grammar, any islands’ patterns are placed first above the water pattern. Each of the island patterns is attempted to be matched
before the water pattern (which would accept any code not matched by the islands pattern). In the past, web migration [11], clone detection [25] [24], and software architecture recovery [26] [19] [20] [30] [31] have used island grammars with TXL.

2.1.2 Agile Parsing

Agile parsing refers to the ability to use a customized version of the input language grammar for each particular transformation task [6]. For web applications, there are many different languages involved. For each different transformation task, depending on the target of the transformation, a different set of languages may be used. In this thesis, the target of the transformation is JSP web pages, which involve other web languages such as HTML and JavaScript. With agile parsing, such a combination of languages can be used together without the need to rewrite the grammar.

With TXL, agile parsing uses the “redefine” and the “include” statement to allow the customization of grammars. The redefine statement, as the name implies, redefines an already defined non-terminal symbol. A redefine may append additional alternative patterns or completely overwrite an already defined non-terminal symbol. The include statement is used in a TXL program, which specifies which grammar(s) is/are to be used. In the case of web applications, the base grammar would be the HTML grammar. Other languages such as JavaScript, JSP, or PHP would be additions to the HTML grammar. When used together, the additional grammars to the base grammar would make use of the redefine statement which appends additional alternatives to a non-terminal definition. In the case with island grammar, the base HTML grammar would include definitions of islands and waters. With the additional grammars, the definition of islands would be redefined, allowing the parsing of more
CHAPTER 2. BACKGROUND

islands for the additional language.

2.2 Backward Slicing

Program slicing is a method introduced by Weiser [28]. It takes a subset of a program’s behavior and reduces the program based on that subset, such that the program would be at its smallest form and still produces that behavior. The reduced form of the program would be called a “slice”. A slicing criterion specifies a window for observing the subset behavior, and the window is a statement and a set of variables. A slicing algorithm determines which statements belong to a slice, and starts searching from the slicing criterion.

Backward slicing is a method to find a slice by tracing a program’s code backwards. From a given slicing criterion, the program is traced backwards, finding statements which may influence the set of variables. With each found statement, the variable set changes depending on whether or not a variable is of concern at that point of the program. Each of the found statements is added to the slice, until no more statements can be found. The result would be a slice of the program which only produces the behavior of the slicing criterion.

A new concept introduced in this thesis is the use of multiple slicing criteria to initiate the automatic backward slicing. This is achieved by manipulating the variable sets of each of the statements during the backward slicing process. The variable sets of the different statements are continually updated during the slicing process, such that the variable sets resulting from the different slicing criteria will be merged and be processed with only one slicing algorithm.
CHAPTER 2. BACKGROUND

2.3 AJAX

AJAX stands for Asynchronous JavaScript And XML. It is not a new language, but a programming technique utilizing JavaScript’s XMLHttpRequest object [18]. The XMLHttpRequest is an API that is used to transfer data between scripted client and the server transferring XML or other text-based data [27]. In another words, the AJAX technique allows web pages to retrieve data from the server without requiring the user to make a web form submission. This allows web pages to update the page contents asynchronously with new data from the server.

The practical use of this technique widely varies. For example, the Google Labs introduced their experimental search suggestions feature (Google Suggest) using AJAX [12]. This feature offers search query suggestions as the user types their search query in the textbox. Another example would be the chatting feature added to Gmail [9]. This allows live text message chatting with another online Gmail user. The asynchronous events involved with this chatting feature are possible with the help of AJAX. The last example is the Google Maps, which its popularity has increase significantly since its debut [10]. AJAX is used in Google Maps to asynchronously transfer new map data to the user upon requested. Without the use of AJAX, the performance and user-friendliness of the application would drop greatly.

The particular use of AJAX focus in this thesis is when it is used to retrieve an iteration of data from the server. This is seen most often in the situation where a query result returns from a database and this result needs to be presented on a web page. Sometimes a paging feature is added to the web page due to the large result set. Traditional round-trip web applications can only retrieve one page of data for each loaded web page. Any further data request would require another submission to
the server and the reloading of the web page. With AJAX, this reloading of the web page is avoided, making the transition much smoother and in some cases faster.

### 2.4 Web Transformations

A number of researchers in recent years have focused on the transformation of web applications. Dean et al. introduced the concept of agile parsing [6], and later applied the concept to the transformation of web applications [8]. Many other researchers have used a similar web application transformation technique and presented different approaches to add new web technologies to older web applications. S. Xu et al. used TXL to restructure JSP pages to move Java code into custom tags [30] [31]. A. Mao used TXL along with island grammars and agile parsing to transform HTML TABLE tags into DIV tags [16]. The similar technique is also used to migrate entire web applications from different frameworks. Hassan et al. used TXL along with the island grammar technique and proposed an approach which migrates web frameworks [11]. Using their approach, they migrated a web application written in the ASP framework to the NSP (Netscape Server Pages) framework. Ping et al. used their own transformation approach and migrated the IBM Net.Data based application to enterprise Java environments [21]. Lau et al. also used their own transformation technique and migrated the IBM Net.Commerce to WebSphere Commerce Suite [13]. With the emerging new web technologies, the transformation of web applications has become an important area of research. These researches help the maintainance tasks of old web applications and aid the adoption of the newer technologies.

Program slicing has been introduced a long time ago. With the growing use of web applications, the technique of slicing has also been adopted. Ricca et al. used
the concept of program slicing and applied it to web applications [23]. With their tool, ReWeb, the developer can visualize the relationships between web pages in a web application. This helps the developer understand a web application, and makes maintenance much easier. L. Xu et al. also used the program slicing technique and applied it to web applications; however, their purpose is for software testing [29]. They used slicing to determine changes to the web application caused by a modification. Much research relating program slicing and web applications mainly deals with the maintenance task and the understanding of the web application. In this thesis, the slice generated from the slicing process is actually used as a part of the transformation result, rather than just for viewing and understanding. The slicing also focuses on a single web page and the subset slice generated more on the entire web application and the relationship between web pages, which give the developer a better understanding of the application.

2.5 Migration To AJAX

In recent years, there are a few projects that focus on using AJAX. A. Puder proposed a migration framework that allows AJAX applications to be written in Java [22]. A Java desktop application is written and compiled. The generated .class file is then translated to XMLVM. XMLVM is similar to assembly language but formatted in XML. The XMLVM application is then translated to JavaScript. AJAX is used to transfer the JavaScript application from the server to the user. The transformation is not automated, and the target of the migration is Java application. For our approach presented in this thesis, the transformation is automated, and is targeted on JSP web pages.
Mesbah et al. presented an approach to migrate multi-paged web applications to single-paged AJAX interfaces [17]. A tool called RETJAX is implemented to carry out the approach. The main purpose of this tool is to identify web elements which could be candidates for AJAX transformation. Their presented approach does not produce an AJAX enabled product, but serves as a starting point to the transformation process. Our approach differs in that we focus on a single web page that offers multiple possible views through different database query results. Their approach focuses on multiple web pages, and searches for dissimilar web elements between the web pages.
Chapter 3

AJAX Transformation Overview

This chapter starts by briefly explaining how AJAX is used to handle multi-paged table data. The process of transforming a JSP round-trip web page into an AJAX enabled web page can be separated into two main phases. The first phase is the extraction and generation of XML data, and the second phase is the transformation of the web page to use the generated XML data. Both phases will be briefly described.

3.1 AJAX Table Handling

In a web page, when a table of data is shown in a multi-paged setting, the data usually comes directly from a database of some sort. In a round-trip web page, each time the user presses a link that changes the page relating to the table of data, the entire web page will be refreshed, lowering the user experience. In order to give the user a fluidity, AJAX allows the browser to update the table data when a different page data is selected. Any other area of the web page that is not a part of the table data (such as the banner, side menu) will remain intact and unchanged. This removes the
flashing of web page due to reloading and gives the user a better experience. It will also help preserve the user’s time and bandwidth due to the reduced amount of data transferred.

AJAX, as introduced in Chapter 2, is a technique that uses the XML handling feature of JavaScript. There is no standard way in which one uses AJAX to handle a particular situation. The method of handling multi-paged tabled data in a web page we present here is commonly used by many AJAX tutorial websites. [5]

The table that is to be transformed starts with a \( <table> \) tag, and ends with a \( <table> \) tag. Due to the difficulty and inability in some cases, of using JavaScript to insert data in the middle of a table tag while keeping the layout of the table correct, the entire content of the table is usually generated by JavaScript. The contents inside the table tag contain two types of data which are separated. First is the dynamic data that is relevant to the user when the page of the table is changed. This is usually the result of an SQL statement. The second type is the static data that remains after removing the dynamic data. This is any text related to the formatting of the table and any table data that does not change with page changes. The table formatting text consists of any table related HTML tags used to format the data and any other style parameters associated with the table tags. Figure 3.1 shows the distinction of the two types of data.

Once the identification of the two types of data is complete, the first type of data, the dynamic data, is changed into an XML representation. The format of the XML text usually reflects the table being transformed, such that the rows and columns of the table are preserved. The XML tag names are also named in such a matter that reflect the original purpose of the data. The XML data is generated, then a
CHAPTER 3. AJAX TRANSFORMATION OVERVIEW

```java
out.println("" + "<tr>");
out.print("" + "<td >");
out.print("" + "<font face="arial" size="2"">" + fldField1 + "&nbsp;&lt;/font &gt;" + fldField2 + "&nbsp;&lt;/font &gt;");
out.println("" + "</td>");
out.print("" + "<td >");
out.print("" + "<font face="arial" size="2"">" + toURL((String) rsHash.get("m_message_id")) + "&lt;" + "font face="arial" size="2"">" + toHTML(fldtopic) + "</font &gt;&lt;/a &gt;");
out.println("" + "</td>");
out.print("" + "<td >");
out.print("" + "<font face="arial" size="2"">" + toHTML(fldauthor) + "&nbsp;&lt;/font &gt;" + toHTML(fldtopic) + "&nbsp;&lt;/font &gt;" + toHTML(flddate_entered) + "&nbsp;&lt;/font &gt;" + toHTML(flddate_joined) + "&nbsp;&lt;/font &gt;" + toHTML(fldname) + "&nbsp;&lt;/font &gt;" + toHTML(fldusername) + "&nbsp;&lt;/font &gt;");
out.println("" + "</td>");
out.println("" + "</tr>");
```

Figure 3.1: The difference between dynamic and static data. Dynamic data is bolded in the figure.

JavaScript function is used to read the XML data through the XMLHttpRequest API as explained in Chapter 2. The second type of data, the static data, is then changed and stored as a JavaScript string data. Both the dynamic data (read in by JavaScript’s XMLHttpRequest API) and the static data (stored as JavaScript’s String data) are finally merged by JavaScript with simple string concatenation. Figure 3.2 shows the merging of the two types of data in JavaScript.

Lastly, the merged data is inserted into the web page through a placeholder (a DIV tag or a SPAN tag). The placeholder is placed in the web page at the same location as the table in the original web page. After placing the data into the placeholder, the resulting web page looks exactly the same as the original web page, while the data retrieval method is completely different from the original web page.

To handle the events of page changes, the original web links which point to URLs of other pages have to be modified. All of the links in the original web page pointing
strTable += " <tr>");
strTable += " <td >";
strTable += " <font face="arial" size="2">";
strTable += "<img src=images/Thread_small.gif>");
strTable += "&nbsp;</font >";
strTable += " </td>");
strTable += " <td >";
strTable += "<a href="viewthread.jsp?mid=";
strTable += row.getElementsByTagName("toURL−rsHash−get−m_message_id") [0].firstChild.nodeValue;
strTable += "&\""><font face="arial" size="2">";
strTable += row.getElementsByTagName("toHTML−fldtopic") [0].firstChild.nodeValue;
strTable += "</font"><a">");
strTable += " </td>");
strTable += " <td >";
strTable += "<font face="arial" size="2">";
strTable += row.getElementsByTagName("toHTML−fldauthor") [0].firstChild.nodeValue;
strTable += "</font>");
strTable += " </td>");
strTable += " <td >";
strTable += "<font face="arial" size="2">";
strTable += row.getElementsByTagName("toHTML−flddate_entered") [0].firstChild.nodeValue;
strTable += "&nbsp;</font >";
strTable += " </td>");
strTable += " </tr>");

Figure 3.2: The merging of dynamic and static data in JavaScript. The dynamic data is bolded in the figure.

to another URL of the new page are changed to execute a JavaScript method that initiates the action of updating the table data. This function takes in a parameter that contains the URL to the new page of XML data. It then retrieves the new XML data, merges the new dynamic data with the static data, and places the updated merged data in the placeholder, overwriting the old data. With the modified web links, when the new links are clicked, only the contents of the placeholder are changed. Everything else in the web page will remain intact and unchanged.
3.2 Transformation Process

The entire transformation process is executed using TXL which was introduced in Chapter 2. The process is broken down into two phases. The first phase, XML generation, deals with extracting a web service for the XML data from the source code of the original web page. The second phase transforms the source code of the original web page into the AJAX enabled web page which uses this web service. Each of the phases is a pipeline architecture. The web page is passed through many small transformations in succession which make up the entire transformation of the phases.

3.2.1 XML Generation

During the first phase, the original web page source goes through a pipeline of transformations. At the end of all the transformations, the result is a web service that provides an XML representation of the dynamic data. Figure 3.3 shows an overview of the transformation processes that take place to make the web page source output XML data.

![Figure 3.3: An overview pipeline process flow of XML generation](image-url)
Source Preparation

In source preparation, the original web page source is cleaned, providing a more uniform representation of the source. This makes it easier to slice later in the process. One of the major changes made to the source is the removal of any HTML related code. This leaves only the JSP source, which is where the dynamic data is produced. Another change made is to sort the code such that global declarations (which include method declarations) occur before scriplets. Scriplets are sections of JSP code, which are similar to the code that occurs inside of a normal Java method. The end effect is that the entire source will consist of two main sections, the declarations section and the scriplets section.

Slicing Preparation

Slicing preparation prepares the source code in a way that is essential for the proper execution of the slicing process. The major purpose of slicing preparation is to give the slicing process slicing criteria. These slicing criteria identify the starting point to the slicing process. Before the original source code is placed through the first phase, identification tags are manually entered into the code indicating the location of the looping table data, and the locations of any other dynamic data. This identification tag is discussed in detail in Section 4.2. In slicing preparation, the slicing criteria are found within the sections of code specified by the manually added tags, which give the following slicing process a starting point.
CHAPTER 3. AJAX TRANSFORMATION OVERVIEW

Slicing

The slicing process consists of three main steps. The first step is to annotate any part of the source code that belongs to the slice. As aforementioned, the slicing criteria are used as the starting point of the slicing process. Any pieces of the source code that are related to those slicing criteria are annotated. The second step is to remove any source code that is not annotated, indicating that the piece of code is not relevant to the slicing criteria being looked at. Lastly, all the annotation tags are removed, leaving only source code which is needed for the proper execution relating to the slicing criteria.

XML Tagging

After the slicing process, the source code now generates the dynamic data as specified by the slicing criteria. This data is generated as a single long string. We insert XML tags into the source code, formatting and separating the long string of data into separate XML tagged data. During this XML tagging process, these XML tags and a JSP header that specifies the document type as XML are added to the source code generating a valid XML instance. Thus the web page has been transformed into an XML based web service.

Clean Up

In the final step to transform the original source code to generate XML data, some cleaning up is done to return the source code into a more conventional format. Although some of the transformations done in the source preparation step make the slicing process easier, it may not be a conventional way which code is written. In
this step, these changes are reverted to keep the source code as close to the original format as possible.

### 3.2.2 Web Page Transformation

In the second phase, the same web page source that was used for the first phase goes through another pipeline of different transformations. At the end of this set of transformations, the web page will use the web service produced by the first phase as the source of dynamic data, making this an AJAX web page. Figure 3.4 shows an overview of the transformation processes that take place to change the web page to an AJAX web page.

![Figure 3.4: An overview of the pipeline process flow of the transformation to an AJAX-enabled web page](image)

Source Preparation

The source preparation step in the second phase is similar to that of the first phase. The purpose of this step is to prepare the code for the transformations that will follow. In JSP, there are three major methods to output text to the web page. The first is to place text directly on the JSP source page, outside of the JSP code brackets. The second is by using the JSP expression, which produces the result of an expression which is enclosed in the JSP expression tab (`<%= %>`). The third is through a
simple print method within the JSP code. Since most of the transformations are changes made to the JSP code, the print method is the recognized method in which text is produced for the web page. The other two methods are not recognized by the transformations that will follow. In this source preparation step, the major change is to change the other two methods of text outputting to the print method. Hassan et al. have used a similar method in the preprocessing stage in their transformation [11]. In our approach, instead of applying the change to the entire web page, it is only applied to the section of the source code containing the looping data.

**Other Dynamic Data Replacement**

Dynamically changing links are links such as page changing links. When the user uses them, the page to the table of data changes, and the links themselves are also changed to navigate to the next page of data. These links are manually marked by the user using custom tags in the original source file. In this step, these dynamic links and other dynamic data are replaced by placeholders. These placeholders will become the entry points for each of the dynamic data. AJAX JavaScript code will place each of the data relating to each link in their own placeholder. The placeholders are HTML SPAN tags with a unique ID relating to each of the dynamic link.

**Exterior Print Replacement**

In this transformation step, if the table tag is found outside the looping table data, all the print statements in between and including the one that prints out the table tag are replaced with string variable concatenation statements. Two variables are used for the replacement, one above and one below the looping table data. These
two variables will contain all of the data required to reproduce the same table as the original web page. The variables are used later by the AJAX JavaScript code to put the separate pieces of table data together.

**Interior Print Removal**

In the previous step, static data that appear above and below the looping data of the table are replaced with string variable concatenation statements. The looping data are removed from the original web page in this step. Each of the print statements that appears in the loop is removed in order for AJAX to handle its replacement. Static data that appear inside the loop (data that are not generated by the web service) are exported into a data file used later for merging with dynamic data.

**Clean Up**

The function of the clean up step in this phase is the same as the one in the first phase. Any changes made to the code in order to facilitate the transformations, but are not of conventional coding style, are undone in this step. The main change that is undone is the removal of brackets that are added in the beginning and end of a scriptlet section. A typical scriptlet section does not start and end with brackets, as they are added to generalize the TXL parsing of JSP grammar.

**AJAX Base Code Merge**

For the final step, the data file exported from the Interior Print Removal step is merged with a template AJAX JavaScript code designed for this class of transformation. The template code contains all the JavaScript methods needed for enabling
AJAX in the web page. A method is also included in the template code that iterates over an XML file, but the actual content of the loop is empty. In this step, the data file is merged with this base code. After the merge, the JavaScript loop will generate the static data values and the dynamic data values retrieved from the web service XML data.

3.3 Summary

This chapter gave an overview of the entire transformation system. The XML generation phase transforms the original web page to generate an XML of the dynamic looping data. The page transformation phase transforms the page to an AJAX-enabled web page using the XML generated in the first phase. In the following two chapters, an in depth description of each phase is given, showing the details involved in implementing the needed transformations.
Chapter 4

XML Generation

This chapter describes the extraction of a web service that provides the data of the page in XML format. The generation of the XML service is done by using 17 transformations in a pipeline structure as mentioned in Chapter 3. With the exception of the first transformation, all of the transformations are TXL programs. The transformations are based on the HTML/JavaScript/ASP grammars originally created by Mykyta Synyskyy et al. [25] [24] and modified for JSP by Xinzheng Li [14] and Andy Mao [16]. We have corrected some remaining minor problems in both the JSP and HTML grammars. Following the agile parsing technique [2], we have also introduced a set of overrides that provide additional syntax specific to the transformations. This set of overrides provides the set of markup and annotations for JSP code. More details about this grammar override are discussed in the following section.
CHAPTER 4. XML GENERATION

4.1 AJAX Transformation Grammar

TXL programs use grammars to correctly parse the source file to guide the transformation. The HTML, JavaScript, and JSP grammars provide the needed syntax to parse and transform web pages. Our transformations use and add annotations to the source code which are then used by later transformations. These extra markups and annotations do not conform to the syntax of the language they reside in. The AJAX transformation override grammar is an extension to the standard JSP grammar to allow the markups and annotations to be placed on the JSP code.

Two distinct extensions are provided by the AJAX override grammar. First is the addition of markup tags. A markup tag is manually added by the user to mark a section of code, which contains the loop that generates the dynamic data. The markup consists of an opening tag and a closing tag which contains a section of Java code. Figure 4.1 shows an example of a marked section of code, and Figure 4.2 shows the grammar overrides behind the markup.

```
<ajax::mark>
  while ( (iCounter < RecordsPerPage) && rs.next() ) {
    ...
    out.println("<tr>");
    ...
    out.println("</tr>");
    iCounter++;
  }
</ajax::mark>
```

Figure 4.1: An example of marked code.

The second extension refers to the annotations used by the transformation rules.
CHAPTER 4. XML GENERATION

Figure 4.2: The extension of JSP grammar to allow markup tags containing a JSP declaration or statement.

Annotations are used by the rules to preserve information between transformations; particularly, they are used by the rules that implement slicing. The annotation is a pair of tags that surround a single statement. The annotation tags are similar to the XML structure, but are not XML. This single statement may enclose other statements in the case of an if or while statement. The annotations have a list of parameters stored on the tag. These parameters store information needed for the transformations. Figure 4.3 shows an example annotation of an if statement and an annotation of the statement inside the if statement. The list of **keywords** in the tag stores a list of variable names representing the variable set of the slicing algorithm. Figure 4.4 shows the grammar behind the annotations.

### 4.2 User Markup

To start off the transformation, the user manually marks the section of Java code in the JSP page. This section contains the loop generating the dynamic data that
is transformed into a web service. The dynamic data generation is not limited to a table of data. Other page elements such as menus, headers, and footers may also be dynamically generated. The transformation only supports one page element transformation; therefore, one of these elements is marked for transformation. Within a web page, the identification of the code associated with a particular table data is beyond the scope of this thesis. The section of code identified by the user is usually a loop statement contained inside a table tag, but this may not always be the case. Once
the section is identified, the user adds the markup tags around the loop statement as described in Section 4.1 and shown in Figure 4.1.

4.3 Source Preparation

The first step of the extraction is source preparation. It cleans up and normalizes the code without changing the meaning. This limits the number of cases that must be dealt with by the rest of the transformations.

4.3.1 Sed Transformation

This transformation, adapted from Mykyta et al., uses the Unix sed command to apply a set of regular expressions to transform the source code. Several minor bugs are corrected from the original version of the transformation. The two main tranformaions done by the sed script are to normalize the end of line conventions to the Unix norm (single newline), and to normalize the various forms of comments to a single commenting convention. A minor transformation is also to normalize any single quote found inside the HTML code to the HTML friendly version.

4.3.2 Extract Interesting Elements

In the extended grammar used for the transformation, the definition of “interesting element” is limited to Java elements. HTML elements that were considered interesting elements in the previous version of the grammar are removed by the grammar overrides for this step of this transformation. The TXL transformation removes any existing non-interesting elements in the source code. This essentially removes all the
HTML and JavaScript portions of the web page, which leaves only Java code.

The JSP include directive can be used to merge an external JSP file with the current file at the location of the include directive. This can be used to include HTML code such as a common header or footer, or to include Java scriptlets and global definitions. A simplifying assumption made by our implementation in this step is that any JSP include directive involving Java code (such as global definitions or initialization codes) are placed on the first line of the web page. Include directives found elsewhere on the web page would be considered non-interesting elements and removed. Properly processing all of the include directives in the transformation would require the transformation to inline each of the include directives before the transform and then to separate the results after the transform. Previously published research [15] [7] has shown how this can be accomplished. This technique is time consuming to implement in TXL and therefore left outside the scope of this thesis. In each of the case studies, however, the include element containing global definitions to the web page is found at the start of the web page.

Figure 4.5 shows the source code before extracting the interesting elements, Figure 4.6 shows the source code after the transformation where the HTML codes are removed.

4.3.3 JSP Expression Normalization

To limit the number of variants in the source code, the ways in which the JSP code can generate text are limited. There are three main ways in which JSP outputs text to the web page: HTML text in the JSP page, JSP expressions, and Java print statements. The first case was eliminated by removing uninteresting elements from
CHAPTER 4. XML GENERATION

Figure 4.5: The source code before extracting the interesting elements.

```html
<% Search_Show(request, response, session, out, sSearchErr, sForm, sAction, conn, stat); %>
</td>
</tr>
</table>
<table>
<tr>
<td valign="top">
<% Messages_Show(request, response, session, out, sMessagesErr, sForm, sAction, conn, stat); %>
```

Figure 4.6: The source code after extracting the interesting elements, HTML code is removed.

the web page in the previous step. In this step, any JSP expressions that are found in the section of code marked by the user are changed into JSP scriptlets that contain a print statement. Figure 4.7 shows the JSP expression being used to generate data; Figure 4.8 shows the JSP expression changed to use print statements.

4.3.4 Code Reorganization

In order to make the code more uniform, global declarations are moved to the top of the page. Two main types of Java code appear at the top lexical level of a JSP file.
while (myResultSet.next())
{
    String body = myResultSet.getString("body");
    String headline = myResultSet.getString("headline");
    String author = myResultSet.getString("author");
    String date = myResultSet.getString("date");
%
<%= headline %> <%= date %> <%= author %> <%= body %>
<%}

Figure 4.7: The source code showing the JSP expression used to output data.

These are formal declarations and scriptlets. JSP formal declarations are declarations of variables that will become instance fields in the class generated by the JSP server and Java methods. The code in the scriptlets will be combined by the JSP server into a single method to handle the request; thus, changing the placement of these declarations does not change the semantics of the code. During the reorganization, adjacent sections of code are also merged. After moving the formal declarations and merging adjacent sections of code, the web page results in two main sections instead of an interweaving of formal declarations and scriptlets. Figure 4.6 shows the adjacent disjoint sections of JSP scriptlets. In Figure 4.9, these two disjoint sections are joined together.

4.3.5 Converting JSP Scriptlets to Java Blocks

Inside the JSP grammar definitions, different non-terminals are used to parse scriptlets and blocks. Since both contain Java statements, it is safe to change scriptlet to blocks. This simplifies the slicing step. After the previous reorganizational transformation
while (myResultSet.next())
{
    String body = myResultSet.getString("body");
    String headline = myResultSet.getString("headline");
    String author = myResultSet.getString("author");
    String date = myResultSet.getString("date");
    out.print(headline);
    out.print(date);
    out.print(author);
    out.print(body);
}

Figure 4.8: The JSP expression changed to print statements.

Search_Show(request, response, session, out, sSearchErr, sForm, sAction, conn, stat);
Messages_Show(request, response, session, out, sMessagesErr, sForm, sAction, conn, stat);

Figure 4.9: The excess opening and closing scriptlet symbols are removed to join the two scriptlet sections.

and application of the scriptlets to blocks transformation, a single block of Java code now generates the dynamic data for the page. Figure 4.10 and Figure 4.11 show the scriptlet section before and after the transformation.

4.4 Slicing Preparation

In the above sections, the JSP code has been prepared to make it easier to slice. The user has identified the main block of code in the page that generates the dynamic content. In this section, we describe some transformations in preparation for the slicing algorithm.
\begin{figure}
\begin{verbatim}
... 
  boolean bDebug = false;
...
  if (conn != null)
    conn.close();
\end{verbatim}
\end{figure}

Figure 4.10: The one scriptlet section before brackets are added.

\begin{figure}
\begin{verbatim}
...
{
  boolean bDebug = false;
...
  if (conn != null)
    conn.close();
}
\end{verbatim}
\end{figure}

Figure 4.11: Brackets added to the JSP scriptlet section.

4.4.1 Identifying Slicing Criteria

With the section of code marked up by the user, lines of the code which produce dynamic data need to be identified. Since the dynamic data has to be printed out onto the web page, for the first condition to this identification, the statement has to be a print statement. As previously transformed, all of the ways in which JSP outputs text to the web page are limited to only print statements. For the second condition, this print statement has to contain dynamic data and not just static text. One simple assumption made is that any variables being printed will contain dynamic data. The worst scenario occurs when the variable contains static text, but it would not affect the result of the transformation.
A problem faced when identifying slicing criteria is that print statements often contain a concatenation of both strings and variables. Before the identification, a simple transformation is performed to split a single print statement into multiple print statements if it contains concatenations of strings and variables. This is shown in Figure 4.12 and Figure 4.13. The resulting code will have a single print statement for each variable that is printed out.

\[
\text{out.print("<font face="arial" size="2">" + fldField1 +"&nbsp;" + "</font>"});}
\]

Figure 4.12: One print statement where both variables and strings are concatenated together.

\[
\text{out.print("<font face="arial" size="2">" + fldField1 +");}
\text{out.print(fldField1);
out.print("&nbsp;" + "</font>"));}
\]

Figure 4.13: The one print statement splits into multiple print statements for each string and variable that was concatenated.

After the splitting of print statements, the slicing criteria are identified. All the print statements that contain a variable are annotated with an annotation tag as shown in Figure 4.14.

\[
\text{out.print("<font face="arial" size="2">" + fldField1 +");
<ajax:annotate>
out.print(fldField1);
<\ajax:annotate type=[data]>
out.print("&nbsp;" + "</font>"));}
\]

Figure 4.14: The one print statement that was split off gets annotated because it prints a variable.
4.4.2 Other Dynamic Data Handling

As an addition to looping table data, there might be dynamic data outside of the loop which gets updated when a page change event occurs. URL links which initiate a page change are examples of these other dynamic data. The user wishing to include these dynamic data in the AJAX transformation may use annotation tags to enclose these sections of code. The annotation tag has an extra parameter identifying the annotation as a special annotation. Figure 4.15 shows an example of a dynamic URL link section annotated by the user.

```xml
<ajax : : annotate>
if ( iPage == 1 )
{
    out.print("\n    <a href="#">\n        <font face="arial" size="2" style="font : bold" color="#0033cc">Previous</font>
    </a>\n") ;
}
else
{
    out.print("\n    <a href="#FileName+?"+formParams+sSortParams+"FormMessages.Page="+(iPage - 1)+"#Form">\n        <font face="arial" size="2" style="font : bold" color="#0033cc">Previous</font>
    </a>\n") ;
}
</ajax : : annotate type = [dlink] >
```

Figure 4.15: User annotation of the dynamic URL link section.

Two changes are made to the annotation tags in this step. First, a unique identification ID is generated for each of the other dynamic data annotation added by the user. This ID is generated by a concatenation of all the variable names inside any print statements, which are found within the section of code enclosed by the annotation tags. This gives the ID a meaning to the user. The ID is also concatenated with a unique number at the end to ensure uniqueness. Second, the user-added annotation
tags are removed; for each print statements found inside the section, annotation tags are added. The added tags are similar to the ones in Section 4.4.1. A parameter is added to each of the tags with the generated ID, which is used later when naming the XML tags. Figure 4.16 shows the user-added annotation tags being replaced.

```c++
if ( iPage == 1 ) {
    <ajax : : annotate >
    out.print("\n <a href="#" >
    <font face="arial" size ="2" style="font : bold" color ="#0033cc"">Previous</font>
    </a> ");
    < /ajax : : annotate tagname = [ sFileNameformParamssSortParamsiPage0 ] type = [ dlink2 ] >
} else {

    <ajax : : annotate >
    out.print("\n <a href="" + sFileName + "?" + formParams + 
    sSortParams + "FormMessages_P" + ( iPage - 1 ) + "#Form" + 
    "<font face="arial" size ="2" style="font : bold" color ="#0033cc"">Previous</font>
    </a> ");
    < /ajax : : annotate tagname = [ sFileNameformParamssSortParamsiPage0 ] type = [ dlink2 ] >
}
```

Figure 4.16: User annotation removed, and annotations added on print statements.

### 4.4.3 Variable Set Extraction

Slicing criteria are identified statements that must be included in the slice. In variable set extraction, any variables that are found in these statements belong to the set. These variable sets are used in slicing to identify which other statements are relevant to the slice. In this step, a list of variables are extracted from the annotated statements and added to the annotation tag as a list of parameters. Figure 4.17 shows the list of variables (called `keywords` in the figure). Variable ‘fldField1’ is the first
item in the list which will get populated during the slicing process.

```xml
<ajax : : annotate >
  out.print (fldField1);
< / ajax : : annotate type = [data] keyword = [fldField1] >
```

Figure 4.17: Variable fldField1 extracted from the print statement to be in the variable set.

### 4.4.4 Direct String Removal

As a means to reduce the number of irrelevant slicing criteria, any criteria that contains a variable which has an assignment of pure string data are removed. This variable has to originate directly from a string variable declaration, where the value assigned to the variable is purely string. This case is found often when the author of the code uses the variable assignment of pure string as a means for easier manageability of the code.

### 4.5 Slicing

After all the preparations, the major transformation of slicing can begin. In this section, details of slicing are explained. The source code is annotated in the two slicing steps. Any statements in the code that are relevant to the slicing criteria are annotated. The unannotated statements are removed from the source code. Finally the annotations are removed, which leave only a slice of the source code relevant to the slicing criteria.
4.5.1 Main Slicing

The method of backwards slicing is used in this step. The structure of the TXL backwards slicing used is based on the TXL backwards slicing code of the Tiny Imperative Language by Cordy [4]. A main rule continuously loops and a set of smaller backwards slicing rules are executed in each loop. The loop continues until no further modification is made to the code (no new annotations are made). The list of backwards slicing rules includes rules such as slicing of declaration statements, assignment statements, and various control flow statements. Each annotation tag contains a variable set. Each variable from the set is of interest to the slice. The general structure of a slicing rule has two parts. The first part finds a variable in a statement that is being modified by the statement. For example, the variable assignment statement would have a variable that is being modified. The second part looks for that modified variable in the variable set from the annotation that is found closest following the current statement. If it is found in the variable set, the current statement would be significant to the slice; hence the statement is annotated. The variable set is copied to the new annotation tag. Any variables found in the current statements that relate to the modification of the variable are also added to the set.

Two major changes are made to Cordy’s code in order to support JSP: Support for cross method slicing and the support for multiple slicing criteria. For supporting methods, a method-call rule is added. An assumption made to simplify the problem is that any variable going inside a method as its parameter would be considered a modified variable. Therefore, if any variables found in a method-call parameter list are found in the variable set, that method-call statement would be sliced. Two other rules are added to slice in and out of a method declaration. To slice into a method
declaration, any method calls that are annotated will trigger the annotation of that method’s declaration. To slice out of a method declaration, any method declarations that contain annotations will have their method names searched in the source code for places where they have been called; that statement will be annotated.

To support multiple slicing criteria, a rule is created that sorts and cleans up the variable sets. This rule will maintain the variable set in each annotation, ensuring the earlier executed (higher) statement’s annotation (within a block of code) will have the most updated variable set up to that point. Variables from the later executed (lower) statement’s annotation are copied to the one higher up, and any duplicates are removed. In the case of declarations, where the declared variable would not be found anywhere above the declaration statement, the variable will be labeled \textit{unkeyword} in the annotation tag. This variable will be searched and removed from any variable sets above this annotation. With this sorting rule, any number of slicing criteria would not matter to this slicing operation, as the highest slicing criterion’s annotation would have the variable set inclusive of all the other slicing criteria’s variable sets. Any slice made will include all of the slicing criteria’s need. Due to this rule, any slicing rule only needs to look at the next annotated statement’s annotation to find the most updated variable sets. Figure 4.18 shows a part of the looping dynamic data after slicing. The variable “fldtopic” is being sliced. The corresponding name of the variable “fldtopic” is added to the variable set in the annotation. Tracing up the code, the declaration of “fldtopic” is found and included in the slice. The variable “fldtopic” is set to \textit{unkeyword}, which indicates that this variable will not be found anywhere above this point.
4.5.2 Global Declaration Slicing

Globally declared variables are not included in the slice from the main slicing step. This is because it is easier to identify them after completing the main slicing step. Any global variables that need to be sliced would be found at the top annotation’s variable set. Due to the sorting and cleaning rule used to clean out the variable sets, any declared variables in a block would be removed from the variable set; therefore, the set should be empty at the top of a block. Any remaining variables would be either variables from the parameters of a method or global variables. In this step, a rule is used to find these global variables and to annotate their declaration statements. Figure 4.19 shows a global declaration being sliced.
4.5.3 Removing Unannotated Code and Annotations

After both slicing steps, all of the statements that are relevant to the slicing criteria are identified by their annotations. In this step, any statements that are not annotated are removed from the source code. Finally, the annotation tags are removed, leaving a slice of source code relevant to just the slicing criteria. Annotation tags on the dynamic data are left on the code. They help the following transformation in locating the dynamic data. Figure 4.20 shows the same parts of Figure 4.18, but now with the irrelevant parts and annotations removed.
String fldtopic = (String) rsHash.get("m_topic");

...  

<ajax:annotate>
out.println(toHTML(fldtopic));
</ajax:annotate>

Figure 4.20: Unannotated code and the annotation tags are removed except for the one that is on the dynamic data.

4.6 XML Tagging

At this point of the transformation, the web page is transformed to only output dynamic data. From the browser, it would look like a stream of data with no breaks between each item. In this step, XML structure is added to the source code. This will split up each dynamic data item as separate XML data values. The first addition made is to add the JSP directive to set the “content-type” of the web page to be an XML document. This lets the browser reading the web page know that an XML document is being read. Next, a main XML tag labeled “ajaxdata” is put in the beginning and end of the web page, surrounding any dynamic data that will be produced by this web page. A set of XML tags are added to the looping dynamic data, which separates each data item and iteration of the loop. For each dynamic data item, a unique ID is given for the tag’s label. This ID is generated similarly to the method described in the generation of unique ID for the other dynamic data. The name of the variables and methods used to produce the dynamic data is concatenated...
to generate the unique ID. To help locate the dynamic data items inside the loop, annotations were left unremoved from the last step. The annotations on the dynamic data are removed in this step.

Besides the looping dynamic data, the other dynamic data also needs to be tagged. These data are tagged and placed just inside the “ajaxdata” tag. The tags used to surround the other dynamic data use the ID generated in the Other Dynamic Data Handling step as its label. In the case where the other dynamic data is a URL link, a JSP method is used to dynamically modify the link’s URL instead of going to another web page, to executes a JavaScript AJAX method that will update the table data. Figure 4.21 shows the resulting source code with the extra print statements giving structure to the data. Figure 4.22 shows the other dynamic data transformed to include the XML tags and the URL changing JSP method. The labels of the two other dynamic data tags are the same, because both of these print statements are placed in the same location on the web page.

To aid the transformations of the next phase, a data file is exported with information about things that are done in this phase. This file lists out the unique IDs of the dynamic data that are handled in this phase and that they can be expected to appear in the web service. This reduces the work needed in the next phase to identify which data has been handled in the first phase.

4.7 Clean Up

At this stage, all of the major transformations are finished. The remaining transformations are to clean up the code and to add in the support JSP page to the web page. One of the changes made to the source code that needs to be reverted is the addition
out.print("<ajaxlist">
while ((iCounter < RecordsPerPage) && rs.next())
{
    out.print("<ajaxitem">
    getRecordToHash(rs, rsHash, aFields);
    String fldauthor = (String) rsHash.get("m_author");
    String flddateentered = (String) rsHash.get("m_date_entered");
    String fldtopic = (String) rsHash.get("m_topic");
    out.print("<toURL-rsHash-get-m_message_id>");
    out.print(HTMLFilter(toURL((String) rsHash.get("m_message_id"))));
    out.print("</toURL-rsHash-get-m_message_id>");
    out.print("<toHTML-fldtopic>");
    out.print(HTMLFilter(toHTML(fldtopic)));
    out.print("</toHTML-fldtopic>");
    out.print("<toHTML-fldauthor>");
    out.print(HTMLFilter(toHTML(fldauthor)));
    out.print("</toHTML-fldauthor>");
    out.print("<toHTML-flddate_entered>");
    out.print(HTMLFilter(toHTML(flddate_entered)));
    out.print("</toHTML-flddate_entered>");
    iCounter ++;
    out.print("</ajaxitem>");
}
out.print("</ajaxlist>");

Figure 4.21: XML structure is added in between the dynamic data print statements.

of brackets around a section of JSP scriptlet code. The brackets are removed from the code making the scriptlet code look more normal. The support JSP page is added to the web page as a JSP include directive. This JSP page contains all the support methods that will be used by the AJAX enabled web page. For example, the method changing the dynamic URL link’s URL to the execution of JavaScript method is in this JSP page. Figure 4.23 shows the addition of the include directive to include the support JSP page. Finally, Figure 4.24 shows the output of the finished web service as seen by a browser.
if (iPage == 1) {
    out.print("<sFileNameformParamssSortParamsiPage0>");
    out.print(HTMLFilter(AJAXLinkConvert("\n <a href_ 
    ="#\">\font face="arial" size="2" style="font:bold"
    color="#0033cc">Previous</a></font>")));
    out.print("</sFileNameformParamssSortParamsiPage0>");
} else {
    out.print("<sFileNameformParamssSortParamsiPage0>");
    out.print(HTMLFilter(AJAXLinkConvert("\n <a href="" + 
    sFileName + "?" + formParams + sSortParams + " 
    FormMessages_Page=" + (iPage - 1) + "#Form\">\font face="arial" 
    size="2" style="font:bold" color="#0033cc">Previous</a></font>")));
    out.print("</sFileNameformParamssSortParamsiPage0>");
}

Figure 4.22: Print statements of other dynamic data are added with XML tags.

<%@ include file = "AJAXCore.jsp" %>
<%@ page contentType = "text/xml" %>
<%@ include file = "Common.jsp" %>
<%! static final String sFileName = "index.jsp";
...

Figure 4.23: An include directive is added to the top of the page to include the support JSP page.
Figure 4.24: The web service as seen by the Firefox web browser showing the XML data.
Chapter 5

Web Page Transformation

This chapter describes the transformations used to transform the original page into one that uses the generated XML service. The same pipeline architecture and similar structure flow of the transformation as the one used in Chapter 4 is used for this phase. The same user marked-up web page is used as the input to the pipeline transformations. The grammars used by the TXL transformations are also the same as the ones used previously. At the end of the pipeline, two outputs are produced: the AJAX-enabled web page and some data extracted from the transformation processes. In the final transformation, the extracted data are used to merge with a pre-made JavaScript method placed in an external file. The JavaScript method in the external script file is used to support the AJAX functionalities.

Some of the transformation rules implemented in this phase were used in the previous generation of XML service phase. Mainly the preparation and clean up transformations are reused, as they prepare the code for the main transformations in the pipeline to ease their implementations.
5.1 Source Preparation

In source preparation, the source code is prepared for the main transformations that follow. Sed transformation, JSP expression to print statement, and JSP scriptlet to block transformations are reused to prepare the code. Details of these transformations can be found in Section 4.3.1, 4.3.3, and 4.3.5 respectively. One transformation that is new to this phase is the HTML Tags to Print Statement transformation which is explained below.

5.1.1 HTML Tags to Print Statement

Unlike the XML generation transformations, the extraction of interesting elements transformation is not used here because the web page’s HTML text contents need to be preserved in the output. As mentioned in Section 4.3.3, JSP expressions are changed to print statements to limit the ways in which JSP code can output text. This lowers the number of variants in the source code. Since the HTML tags are not removed, they appear as plain text in the JSP code. Therefore, they need to be transformed to print statements in order for the following transformations to recognize them. This transformation is applied only to the section of code containing the looping dynamic data. For example, if the looping data appears inside a method, any HTML tags that appear as plain text inside the method will be transformed into print statements. This transformation helps the transformations in the Exterior Print Replacement and the Interior Print Removal steps that will follow. HTML text that appears within a JSP code begins with a “%>” symbol followed by the HTML codes, and ends with a “<%” symbol. This transformation takes the entire section of HTML text between the “%>” symbol and the “<%” symbol, and transforms it into
one print statement. Figure 5.1 shows the sections of code that will be transformed to a print statement and Figure 5.2 shows the result of the transformation.

```java
while (myResultSet.next())
{
    String body = myResultSet.getString("body");
    String headline = myResultSet.getString("headline");
    String author = myResultSet.getString("author");
    String date = myResultSet.getString("date");
    <table border='1' align="center" class="blog">
        <tr><td><b>% out.println(headline); %</b></td></tr>
        <tr><td><b>% out.println(date); %</b></td></tr>
        <tr><td><i>% out.println(author); %</i></td></tr>
        <tr><td>% out.println(body); %</td></tr>
    </table>
}
```

Figure 5.1: HTML tags inside of the marked section that will be changed to print statements.

### 5.2 Other Dynamic Data Replacement

As mentioned in Section 4.4.2, the user may annotate sections of the source code to include them in the AJAX transformation. These annotated sections of code will be replaced in this transformation. Any text outputs that are generated by JSP in this section of code are already included in the XML web service. In this transformation,
while (myResultSet.next()) {
    String body = myResultSet.getString("body");
    String headline = myResultSet.getString("headline");
    String author = myResultSet.getString("author");
    String date = myResultSet.getString("date");
    out.print("<table border='1' align="center" class="blog" ×
                tr×td×b>");
    out.print( headline );
    out.print("</b></td></tr></td></b>");
    out.print( date );
    out.print("</b></td></tr></td></i>");
    out.print( author );
    out.print("</i></td></tr></td>");
    out.print( body );
    out.print("</i></td></tr></table>");
}

Figure 5.2: HTML tags inside of the marked section that are replaced with print statements.

The annotation tags along with the section of code that they contain are replaced with a print statement, which prints out an HTML SPAN tag. This SPAN tag includes an ID which is generated in the same way as explained in Section 4.4.2. The SPAN tag along with its unique ID becomes a placeholder for the dynamic data that is retrieved by a JavaScript method from the web service.

Two HTML tags are possible candidates to be used for the dynamic data’s placeholder, the SPAN tag or the DIV tag. The value stored in a DIV tag gets rendered on the web page starting on a new line, while the SPAN tag’s stored value is rendered inline with its surrounding HTML elements. Since this AJAX transformation should not change the look of the web page, the SPAN tag is used.

Figure 5.3 shows the three sections of code annotated by the user to be handled by AJAX. Figure 5.4 shows the sections of code replaced with print statements, which
prints out the SPAN tag along with its unique ID as the placeholder.

```plaintext
<ajax::annotate>
if ( iPage == 1 )
{
    out.print("\n    <a href="#">font face="arial" size ="2" style="font: bold" color="#0033cc"">Previous</font></a>\n");
}
else
{
    out.print("\n    <a href="#"+sFileName+"?"+formParams+sSortParams+"FormMessages_Page=+(iPage - 1)+"#Form"awi}><font face="arial" size ="2" style="font: bold" color="#0033cc">Previous</font></a>\n");
}
</ajax::annotate type=[dlink]>
<ajax::annotate>
out.print("\n    [ "+iPage+ " ]\n");
</ajax::annotate type=[dlink]>
<ajax::annotate>
if (!bNext)
{
    out.print("\n    <a href="#">font face="arial" size ="2" style="font: bold" color="#0033cc"">Next</font></a>\n<br>\n");
}
else
{
    out.print("\n    <a href="#"+sFileName+"?"+formParams+sSortParams+"FormMessages_Page=+(iPage + 1)+"#Form"awi}><font face="arial" size ="2" style="font: bold" color="#0033cc">Next</font></a>\n<br>\n");
}
</ajax::annotate type=[dlink]>

Figure 5.3: The other dynamic data sections that will be changed to print statements, which prints out SPAN tags.
CHAPTER 5. WEB PAGE TRANSFORMATION

5.3 Exterior Print Replacement

The SPAN tag, as mentioned in the section above, is used as a placeholder to hold the dynamic data; however, there is a limitation to where any placeholder (SPAN tag or DIV tag) can be placed. The placeholder cannot be placed inside the HTML table tag structure outside of the TD tag, because the browser will format the table incorrectly. Any data inside the placeholder will appear on top of the table instead of inside the table. The reason is caused by the browser expecting to find a HTML table structure tag, but finds the placeholder tag instead and does not know how to handle it. Due to this limitation, an HTML TABLE tag structure cannot be taken apart and joined together with separate placeholders. Therefore, if the dynamic data does not contain the entire TABLE tag structure but is a loop of a table row, work is needed to search and include the rest of the table structure in the transformation.

To include the extra static data so that JavaScript can handle the entire table structure, two JSP variables are introduced in the AJAX JSP support page. The two string variables are used to collect all the static data from before and after the looping dynamic data. In this transformation, any print statements found outside of the looping dynamic data are transformed into a string concatenation statement. The print statements found above the looping data are assigned to one of the variables, and the ones found below are assigned to the other. The values collected by these two variables will be transferred into a JavaScript method, which will put together the

```
out.print("<span id="sFileNameformParamssSortParamsiPage0"></span>");
out.print("<span id="iPage1"></span>");
out.print("<span id="sFileNameformParamssSortParamsiPage2"></span>");
```

Figure 5.4: The other dynamic data sections that are replaced with print statements, which prints out SPAN tags along with their unique ID.
entire table structure and output it to the placeholder.

This transformation is only performed if an opening TABLE tag is found above the looping data and a closing TABLE tag below the looping data. The check may overlook other cases where the transformation may not be needed. This transformation also goes beyond the opening and closing TABLE tag including other static data in the two variables. However, these two situations will not hinder the result of the AJAX transformation. These two situations will only let more data be handled by the JavaScript method, but the end result of the web page will appear the same.

Figure 5.5 shows some of the print statements found above and below the looping dynamic data. Figure 5.6 shows the result after the transformation, where the print statements are turned into string concatenation statements.

5.4 Interior Print Removal

After handling the print statements outside the looping dynamic data, the print statements inside the loop need to be handled. They need to be removed such that the JavaScript method can put in the AJAX version of the data in its place.

5.4.1 Print Statement Splitting

In order to distinguish the different types of data the print statements are printing out, print statements that are printing out a concatenation of static and dynamic data need to be split up. This operation of splitting the print statements is exactly the same as the print splitting explained in Section 4.4.1, where the print statements are split up in order to identify dynamic data for slicing.
...  
out.println("""<table border=""""0"""" cellspacing=""""1"""" cellpadding=""""3"""">")
out.println("""<tr><td bgcolor=""""#c2c2c2"""" colspan=""""4""""><a name="Messages">"""
<font face=""""arial"""" style=""""font : bold color =""""#0033cc"""">
Discussion</font></a></td>
</tr>")
out.println(tableHeader);
...

<ajax : : mark>
while ((iCounter < RecordsPerPage) & rs.next ( ))
{
...
}
</ajax : : mark>
...
out.println("""</table>"""");
...

Figure 5.5: Print statements where the TABLE tags are found above and below the looping dynamic data.

5.4.2 Print Statement Removal

There are two types of data that are removed from this transformation. The dynamic data is the data that was sliced and transformed into the XML web service in the first phase. The static data is the data that appears in between the dynamic data. Similar to the static data found outside the looping dynamic data as explained in Section 5.3, these data have to be handled by the JavaScript method in order for it to reproduce the entire table structure. In this transformation, a data file is exported ("merge list"), listing the print-out value of each print statement as they are all removed. This list preserves the order in which the print statements appear as it gets populated. For static data, the string value is exported. For the dynamic data,
Figure 5.6: Print statements where the TABLE tags are found and transformed into string concatenation statements.

the unique ID for each dynamic data is exported, and the value of the data will be retrieved from the web service. This merge list is used later in the final transformation where all the separated parts of the table data are combined.

This transformation uses the data file exported in the XML Tagging transformation in Section 4.6 to help identify dynamic data. Each print statement found in the loop is removed and a merge data is exported into the merge list. If it is a dynamic data, a “+” symbol is added in front of the merge list data. If the print statement is not a dynamic data, then it must be a static data. In that case, the print statement is removed, and the merge list data would be quoted with “’” if it is a string text; otherwise, it would be exported unmodified. When the static data is not a string but
is a variable, a special case occurs where the variable value is purely string. This case is mentioned in Section 4.4.4 and handled in the following transformation.

Figure 5.7 shows the merge list from this transformation. Note the highlighted section where variable “fldField1” is not a dynamic data nor a string text, but a variable that contains pure string value. Figure 5.8 shows the section of marked code where the print statements are removed.

```
"' ,  <tr>' ,
"' ,  <td>' ,
"'<font face="arial" size="2">'' ,
"fldField1",  "&nbsp;/font>' ,
"'<td>' ,
"'<a href="viewthread.jsp?" ,
"'mid=' ,
"+toURL−rsHash−get−m_message_id",  "&" ,
"<font face="arial" size="2">'' ,
"+toHTML−fldtopic",  "&nbsp;/font>' ,
"' </font></a>' ,
"'</td>' ,
"' ,  <td>' ,
"'<font face="arial" size="2">'' ,
"+toHTML−fldauthor",  "&nbsp;/font>' ,
"' </font>' ,
"'</td>' ,
"' ,  <td>' ,
"'<font face="arial" size="2">'' ,
"+toHTML−flddate_entered",  "&nbsp;/font>' ,
"' </font>' ,
"'</td>' ,
"' ,  </tr>'
```

Figure 5.7: Merge list from print removal that highlights the direct string assignment variable. Each item is split into a new line for clarity.
```java
while ((iCounter < RecordsPerPage) & & rs.next ( ) )
{
    getRecordToHash ( rs , rsHash , aFields ) ;
    String fldauthor = (String) rsHash . get ( "m_author" ) ;
    String flddate_entered = (String) rsHash . get ( "m_date_entered" ) ;
    String fldmessage_id = (String) rsHash . get ( "m_message_id" ) ;
    String fldtopic = (String) rsHash . get ( "m_topic" ) ;
    String fldField1 = "<img src=images/Thread_small.gif >" ;
    iCounter ++ ;
}
...
```

Figure 5.8: Print statements removed from the section of marked code after the print statement removal transformation.

### 5.4.3 Direct String Replacement

In Section 4.4.4, any dynamic data variables that are found with their values assigned as purely string text, are removed from the slicing process. These variables reappear in the merge list from the previous transformation. To reduce the work of retrieving these data from the server as dynamic data, the string text value found in these variables is used directly instead of going through the use of the variable. For the merge list as mentioned in the previous section, dynamic data that is handled in the first phase has a “+” in front of its ID, and string data is quoted with “’”. If an item appears in the merge list that is not started with a “+” or not quoted with “’”, it would be a variable of direct string assignment. These items are replaced from the merge list with the variables’ string values. Figure 5.9 shows the same merge list from Figure 5.8, but the “fldField1” variable is now replaced with its string value.
5.5 Clean Up

In this clean up section, two of the same transformations are reused from the first phase: The removal of brackets on the scriptlets, and the addition of the include directive for the support JSP page. They are described in Section 4.7. One of the new transformations used in this section removes the user added markup from the source code. This transformation was included within the XML Tagging step in the last phase; in this phase, it is treated separately. The following section will describe the other new transformation used in this clean up section.
5.5.1 Adding Method-Call for AJAX Support Method

In this transformation, a method-call statement is added to the source code. This method-call calls a support method that is defined in the support JSP page. The purpose of this method is to initiate the entire data retrieval process for AJAX. There are four things that this method does. First, it prints out the placeholder SPAN tag needed for the data placement. This placeholder will hold the main looping dynamic data that is handled by AJAX (section marked by the user). Secondly, it includes the two supporting JavaScript script files on the web page. One script file contains the core methods to enable AJAX, and the second script file will be generated in the next transformation described in the following section. Thirdly, the two variables that are used to collect print statement values from above and below the dynamic looping data are transferred into JavaScript for the support methods to handle. Lastly, the support JavaScript method is executed, which will initiate the dynamic data retrieval from the web service using AJAX.

In the case where Exterior Print Replacement transformation is performed, the method call will be placed at the end of the section of code containing the replaced print statements. In the case where Exterior Print Replacement transformation is not performed, the method call is placed directly following the looping dynamic data. Figure 5.10 shows the added method call, which is placed at the end of the section of code containing the replaced print statements.
... if (rs != null)
    rs.close();

postData += "  </table>");

ajaxExData (out, request.getRequestURI ());

} catch (Exception e) {
  out.println (e.toString ());
}
...

Figure 5.10: The AJAX support method call added to the end of the section of code containing the replaced print statements.

5.6 AJAX Base Code Merge

At this point, the web page has already been transformed to support the AJAX functionality. The web page will make the appropriate JavaScript call to start the AJAX data retrieval process. A JavaScript file that contains a method to iterate through the retrieved data is prepared beforehand. This method does not contain any static or dynamic data for which it iterates through, but it is a base file prepared for this transformation. The method is executed by the JavaScript AJAX core method, which is executed by the JSP support method. In this transformation, the merge list generated from Section 5.4.2 and 5.4.3 is used to merge with the prepared base file. Figure 5.11 shows the base file prepared for this transformation. The two bolded statements that concatenate to the variable “strTable” are replaced with concatenations of static and dynamic data as listed in the merge file. The string concatenation of variables “ajaxPreCont” and “ajaxPostCont” refers to the two JSP variable used in Exterior Print Replacement in Section 5.3. The collected static data from before and after the looping dynamic data are in the two JavaScript variables.
function tableMerge(ajaxlist)
{
    var strTable = "";
    strTable += ajaxPreCont;

    for (var rowIndex = 0; rowIndex < ajaxlist.childNodes.length; rowIndex++)
    {
        var row = ajaxlist.childNodes.item(rowIndex);
        strTable += row.childNodes.item(0).childNodes.item(0).data;
    }
    strTable += ajaxPostCont;
    return (strTable);
}

Figure 5.11: The base JavaScript file used to merge with the merge list. The highlighted sections are placed with the merged data.

The JavaScript variable “strTable” is the variable that will contain the entire HTML table data. Once the complete table is pieced together, the value of “strTable” will be assigned to the placeholder. This merging transformation iterates through the merge list. A static data found in the merge list is transformed into a string concatenation statement, which adds the static value to the JavaScript variable “strTable”. A dynamic data is transformed into a data retrieval statement, which retrieves the dynamic data from a JavaScript variable containing the entire web service XML data downloaded from the server. The retrieved dynamic data value is also concatenated to “strTable”. The resulting statements transformed from the merge list are added to the JavaScript base file. The end result will be a collection of string concatenation statements, which pieces the entire table structure together. The resulting “strTable” variable will contain the entire table text which gets placed inside the placeholder.
Figure 5.12 shows the base file after the merging transformation.

```
function tableMerge (ajaxlist)
{
    var strTable = "";
    strTable += ajaxPreCont;
    for (var rowIndex = 0; rowIndex < ajaxlist.childNodes.length; rowIndex++)
    {
        var row = ajaxlist.childNodes.item(rowIndex);
        strTable += " <tr>");
        strTable += " <td >";
        ...
        if (row.getElementsByTagName("toURL-rsHash-get-m_message_id")[0].firstChild != null)
            strTable += row.getElementsByTagName("toURL-rsHash-get-m_message_id")[0].firstChild.nodeValue;
        ...
        strTable += "</td>");
        strTable += " </tr>");
    }
    strTable += ajaxPostCont;
    return (strTable);
}
```

Figure 5.12: The base JavaScript file merged with the static and dynamic data from the merge list.

### 5.7 Other Dynamic Data Outputting

In Section 5.2, the other dynamic data is replaced with placeholders of SPAN tags. The merged JavaScript method generated in Section 5.6 is executed by a core JavaScript method created to facilitate the AJAX data retrieval process. The merged JavaScript
method’s function is to only merge data together to be placed in the placeholder. The core method is the method that places the merged data into the placeholder. After placing the main table data into the placeholder, the core method looks inside the web service XML for any extra data. Any extra data would be the other dynamic data generated in Section 4.4.2. Any of the extra data found in the web service is placed into its own placeholder according to the generated ID. Since the other dynamic data is extracted from the first phase transformation without separating the static and dynamic parts, the data from the web service does not need to be merged or processed by JavaScript before being placed into their placeholders.
Chapter 6

Examples

To demonstrate the effectiveness of the AJAX transformation, four different web applications were used. Three of the web applications are from GotoCode.com [32]. These three separate web applications were created using a web application development environment called CodeCharge, by YesSoftware. CodeCharge is an integrated development environment that provides an interface for generating a web application skeleton based on a given database, attaching snippets of Java code to HTML form elements, authoring JSP pages and automatically handles some of the low level details such as generating code for database connectivity. From the GotoCode website, a large variety of web applications can be found with their source codes available in various languages. The three applications chosen from GotoCode to transform are the forum application, the classifieds posting application, and the events management application. The last application to transform is a web application called JSPBLOG from SourceForge [1]. JSPBLOG is a simple blogging web application. This forth application shows the transformation’s ability to operate on a range of applications other than the ones written with CodeCharge. All of the web applications contain a
database of populating data which needs to be displayed back to the user. This type of application would contain a web page that has a looping of data retrieval from the database, which is the target of our AJAX transformation.

6.1 GotoCode Applications

6.1.1 Forum

The first case transform is the forum application from GotoCode. For the transformation, the front page of this application is chosen. The front page contains a listing of the forum threads that are in the database. The listing contains a paging mechanism for when the number of threads to display becomes too big. Most of the figures from Chapter 3 to Chapter 5 are taken from this application.

Before starting the transformation, the user needs to mark-up the source code, indicating the location of the looping dynamic data. This is shown in Figure 4.1. For each of the paging control link, it has to be annotated as shown in Figure 4.15. After finishing the first phase’s transformation, code is generated to produce the XML web service as shown in Figure 4.21 and Figure 4.22. Figure 4.24 shows the XML data generated from the transformed code. Note the XML data for the other dynamic data. The HTML link now executes a JavaScript method which updates the web page instead of pointing to a different URL.

After the second phase transformations, an AJAX enabled web page is generated along with a JavaScript file which helps its functionalities. Figure 5.8 shows the code of the AJAX enabled web page where the print statements are removed. Figure 5.12 shows the JavaScript file generated from the second phase transformation. This file
compiles the pieces of the transformed table data together.

Figure 6.1 shows the front page of the forum before the transformation, and Figure 6.2 shows the front page after the transformation.

![Front page of the forum before transformation](image1)

**Figure 6.1: GotoCode Forum before transformation.**

There is no difference in the appearance of the two web pages, but the mechanism of the data retrieval is different. The page change operation of the transformed web page is also faster. This is due to the less data transferred between the server and the client, and the less work needed to reparse the entire web page by the browser. Note that the link URL in the XML web service (Figure 4.24) points to execute a method to update the page data instead of a new page.

### 6.1.2 Classifieds

The second case to transform from the GotoCode website is the classifieds web application. This application allows users to post advertisements on the website. The
advertisements are categorized into different categories. For the AJAX transformation, one of the pages where the user can see a listing of all of their own advertisements is chosen. This page has a loop which outputs the advertisements posted by the user. This loop is marked by the marking tags to initiate the transformation. Figure 6.3 shows the portion of the code that is marked.

After the first phase transformation, the web page outputs XML data as the web service. Figure 6.4 shows the transformed code. Figure 6.5 shows the XML data as seen from a browser.

The second phase takes the same marked source code and transforms it into an AJAX enabled web page. This consists of the web page itself with the dynamic data removed and the JavaScript supporting code. Figure 6.6 shows the web page with the dynamic data removed. Figure 6.7 shows the supporting JavaScript method which puts together the table for outputting.

The transformed web page as seen in the browser looks identical to the original
...<ajax::mark>
while ( rs.next() ) {
    getRecordToHash( rs, rsHash, aFields );
    String fldcategory_id = (String) rsHash.get("c_name");
    String flddate_posted = (String) rsHash.get("a_date_posted");
    String fldname = (String) rsHash.get("a_name");
    String fldprice = (String) rsHash.get("a_price");
    out.println("<tr>");
    out.print("<td bgcolor="#EEEEEE">" + toHTML(flddate_posted) + "&nbsp;<font face="arial" size="2">" + toHTML(fldcategory_id) + "+" + toHTML(fldname) + "+" + toHTML(fldprice) + "+" + toHTML(fldprice) + "+" + toHTML(fldprice) + ");
    out.println("</td>");
    out.println("</tr>");
    iCounter++;
}
</ajax::mark>
...

Figure 6.3: GotoCode Classifieds source code marked for transformation.
out.print("<ajaxlist>");
while (rs.next())
{
    out.print("<ajaxitem>");
    getRecordToHash(rs, rsHash, aFields);
    String fldcategory_id = (String) rsHash.get("c_name");
    String flddate_posted = (String) rsHash.get("a_date_posted");
    String fldname = (String) rsHash.get("a_name");
    String fldprice = (String) rsHash.get("a_price");
    out.print("<toHTML−flddate_posted>");
    out.print(HTMLFilter(toHTML(flddate_posted)));
    out.print("</toHTML−flddate_posted>");
    out.print("<toURL−rsHash−get−a_ad_id>");
    out.print(HTMLFilter(toURL((String) rsHash.get("a_ad_id"))));
    out.print("</toURL−rsHash−get−a_ad_id>");
    out.print("<toHTML−fldname>");
    out.print(HTMLFilter(toHTML(fldname)));
    out.print("</toHTML−fldname>");
    out.print("<toHTML−fldcategory_id>");
    out.print(HTMLFilter(toHTML(fldcategory_id)));
    out.print("</toHTML−fldcategory_id>");
    out.print("<toHTML−fldprice>");
    out.print(HTMLFilter(toHTML(fldprice)));
    out.print("</toHTML−fldprice>");
    out.print("</ajaxitem>");
}
out.print("</ajaxlist>");
...

Figure 6.4: GotoCode Classifieds source code transformed to produce the web service XML data.
while ( rs.next () )
{
    getRecordToHash ( rs , rsHash , aFields );
    String fldcategory_id = ( String ) rsHash . get ("c_name");
    String flldate_posted = ( String ) rsHash . get ("a_date_posted");
    String fldname = ( String ) rsHash . get ("a_name");
    String fldprice = ( String ) rsHash . get ("a_price");
iCounter ++;
}

Figure 6.5: GotoCode Classifieds web service data

...
function tableMerge (ajaxlist)
{
    var strTable = "";
    strTable += ajaxPreCont;
    for (var rowIndex = 0; rowIndex < ajaxlist.childNodes.length; rowIndex++)
    {
        var row = ajaxlist.childNodes.item(rowIndex);
        strTable += "<tr>
        strTable += "<td bgcolor="#EEEEEE">
        ...
        if (row.getElementsByTagName("toHTML-field-date-posted")
            [0].firstChild != null)
            strTable += row.getElementsByTagName("toHTML-field-date-posted")
                [0].firstChildnodeValue;
        ...
        strTable += "</td>
        strTable += "</tr>
    }
    strTable += ajaxPostCont;
    return (strTable);
}
web page. Figure 6.8 shows the original web page. Figure 6.9 shows the transformed web page.

![GotoCode Classifieds](image)

**Figure 6.8: GotoCode Classifieds before AJAX transformation**

Even though the two web pages appear identical, they use different mechanism to retrieve their data.

### 6.1.3 Events

The third case to transform from the GotoCode website is the events posting web application. This web application allows the user to post different events’ details to the website. The website keeps a database of the events posted and displays them on the front page along with a paging feature. For the AJAX transformation, the front page is chosen as it contains a loop that generates the list of events posted.
Figure 6.9: GotoCode Classifieds after AJAX transformation

Figure 6.10 shows both the user-marked tags on the loop and the annotations on the paging controls.

After the first phase transformation, the web page transforms into a web service outputting XML data. Both the dynamic data from the main table loop and paging controls data are included in the web service. Figure 6.11 shows the transformed code. Figure 6.12 shows the XML data as seen from a browser.

The second phase transforms the marked and paging control annotated source code into an AJAX enabled web page. The AJAX enabled web page and the JavaScript supporting code are produced after the transformation. Figure 6.13 shows the web page with the dynamic data removed from the loop and the paging controls changed into placeholders. Figure 6.14 shows the supporting JavaScript method which puts together the static and dynamic data of the table for outputting.
CHAPTER 6. EXAMPLES

Figure 6.10: GotoCode Events web application where the loop is marked with tags and paging controls annotated.
... out.print("<ajaxlist>");
while ((iCounter < RecordsPerPage) & & rs.next ())
{
    out.print("<ajaxitem>");
    ...
    out.print("<transitParams>");
    out.print(HTMLFilter(transitParams));
    out.print("</transitParams>");
    ...
    iCounter ++;
    out.print("</ajaxitem>");
}
out.print("</ajaxlist>");
...
if (iPage == 1)
{
    out.print("<sFileNameParamssSortParamsiPage0>");
    out.print(HTMLFilter(AJAXLinkConvert("\n <a href="\n<font style="font-size: 10pt; color: #FF00FF; font-family: Arial, Tahoma, Verdana, Helvetica; font-weight: bold
">Previous</font></a>")));
    out.print("</sFileNameParamssSortParamsiPage0>");
}
else
{
    ...
}
out.print("<iPage1>");
out.print(HTMLFilter(AJAXLinkConvert("\n [" + iPage + "]")));
out.print("</iPage1>");
...

Figure 6.11: GotoCode Events source code transformed to produce the web service XML data.
--<ajaxdata>
--<ajaxlist>
--<ajaxitem>
  <transitParams>category_id=&name=&</transitParams>
  <toURL-rsHash-get-e_event_id>18</toURL-rsHash-get-e_event_id>
  <toHTML-fldname>tid</toHTML-fldname>
  <toHTML-flldate_start>2007-12-17 03:01:00</toHTML-flldate_start>
  <toHTML-flldcategory_id>Business &amp; Economy</toHTML-flldcategory_id>
  <toHTML-flldpresenter/>
</ajaxitem>
++<ajaxitem></ajaxitem>
++<ajaxitem></ajaxitem>
</ajaxlist>
--<sFileNameformParamssSortParamsiPage0>
  <a href="#"><font style="font-size: 10pt; color: #FFFFFF; font-family: Arial, Tahoma, Verdana, Helvetica; font-weight: bold">Previous</font></a>
</sFileNameformParamssSortParamsiPage0>
<iPage1>[ 1 ]</iPage1>
++<sFileNameformParamssSortParamsiPage2></sFileNameformParamssSortParamsiPage2>
</ajaxdata>

Figure 6.12: GotoCode Events web service data
while ((iCounter < RecordsPerPage) && rs.next()) {
    getRecordToHash(rs, rsHash, aFields);
    String fldcategory_id = (String) rsHash.get("c_category_desc");
    String flldate_start = (String) rsHash.get("e_date_start");
    String fldname = (String) rsHash.get("e_name");
    String fldpresenter = (String) rsHash.get("e_presenter");
    iCounter ++;
}

...  

postData += "<span id="sFileNameformParamssSortParamsiPage0"></span>";
postData += "<span id="iPage1"></span>";

...  

Figure 6.13: GotoCode Events source code transformed to enable AJAX; dynamic data removed from the web page.

function tableMerge (ajaxlist) {
    var strTable = "";
    strTable += ajaxPreCont;
    for (var rowIndex = 0; rowIndex < ajaxlist.childNodes.length; rowIndex ++) {
        var row = ajaxlist.childNodes.item( rowIndex);
        ...
        if ( row.getElementsByTagName("transitParams") [0].firstChild != null )
            strTable += row.getElementsByTagName("transitParams") [0].firstChild.nodeValue;
        ...
        strTable += " <tr><td colspan="2" &nbsp <td colspan="2" &nbsp </td><tr>
    }
    strTable += ajaxPostCont;
    return ( strTable);
}

...  

Figure 6.14: GotoCode Events supporting JavaScript method; static and dynamic data merged to produce table data.
The paging controls data are placed into the placeholders by the core JavaScript method as explained in Section 5.7.

The transformed web page looks identical to the original web page. The paging controls use AJAX to update the page contents, and hence provide a seamless page change. The page changes also operate faster than the original page. Figure 6.15 shows the original web page. Figure 6.16 shows the transformed web page.

Figure 6.15: GotoCode Events before AJAX transformation
Figure 6.16: GotoCode Events after AJAX transformation

6.2 JSPBLOG

The last case to transform is the JSPBLOG from the SourceForge. This simple blogging application allows the user to post blog items on the website. The blog items are listed on the front page of the web application. The purpose of this case is
to ensure that the AJAX transformation method works for other web applications, and not just the ones from GotoCode. For the AJAX transformation, the front page is chosen. The loop in the front page is marked by the user to initiate the transformation. Figure 6.17 shows the portion of the code that is marked.

```java
Class.forName("org.gjt.mm.mysql.Driver");
Connection myConn = DriverManager.getConnection("jdbc:mysql://localhost/test");
java.sql.Statement stmt = myConn.createStatement();
String query="select * from news";
ResultSet myResultSet = stmt.executeQuery(query);
if (myResultSet != null) {
    <ajax::mark>
    while (myResultSet.next()) {
        String body = myResultSet.getString("body");
        String headline = myResultSet.getString("headline");
        String author = myResultSet.getString("author");
        String date = myResultSet.getString("date");
        <table border='1' align="center" class="blog">
        <tr><td><b><%= headline %></b></td></tr>
        <tr><td><%= date %></td></tr>
        <tr><td><%= author %></td></tr>
        <tr><td><%= body %></td></tr>
        </table>
    }</ajax::mark>
}
stmt.close();
myConn.close();
```

Figure 6.17: JSPBLOG source code marked for transformation.

Figure 6.18 shows the transformed code after the first phase transformation. Figure 6.19 shows the XML data as seen from a browser.
... out.print("<ajaxlist>");
while (myResultSet.next()) {
    out.print("<ajaxitem>");
    String body = myResultSet.getString("body");
    String headline = myResultSet.getString("headline");
    String author = myResultSet.getString("author");
    String date = myResultSet.getString("date");
    out.print("<headline>");
    out.print(HTMLFilter(headline));
    out.print("</headline>");
    out.print("<date>");
    out.print(HTMLFilter(date));
    out.print("</date>");
    out.print("<author>");
    out.print(HTMLFilter(author));
    out.print("</author>");
    out.print("<body>");
    out.print(HTMLFilter(body));
    out.print("</body>");
    out.print("</ajaxitem>");
}
out.print("</ajaxlist>");
...

Figure 6.18: JSPBLOG source code transformed to produce the web service XML data.
Figure 6.19: JSPBLOG web service data
Figure 6.20 shows the AJAX enabled web page with the dynamic data removed after the second phase transformation. Figure 6.21 shows the supporting JavaScript method which puts together the blog items for outputting.

```java
... if (myResultSet != null) {
    while (myResultSet.next()) {
        String body = myResultSet.getString("body");
        String headline = myResultSet.getString("headline");
        String author = myResultSet.getString("author");
        String date = myResultSet.getString("date");
    }
    ajaxExData(out, request.getRequestURI());
}
...```

Figure 6.20: JSPBLOG source code transformed to enable AJAX with the dynamic data removed.

Figure 6.22 shows the original web page. Figure 6.23 shows the transformed web page.

The two web pages are almost identical. The one difference seen is the extra space found after the apostrophe in the title. This is due to the formatting rules in the grammar that we used. A space is added between every parsed element unless specified otherwise in the grammar; hence, a space is seen after the apostrophe but not before it.
function tableMerge (ajaxlist)
{
    var strTable = "";
    strTable += ajaxPreCont;
    for (var rowIndex = 0; rowIndex < ajaxlist.childNodes.length; rowIndex++)
    {
        var row = ajaxlist.childNodes.item(rowIndex);
        strTable += row.getElementsByTagName("headline")[0].firstChild != null
            ? row.getElementsByTagName("headline")[0].firstChild.nodeValue
            : "";
        strTable += row.getElementsByTagName("date")[0].firstChild != null
            ? row.getElementsByTagName("date")[0].firstChild.nodeValue
            : "";
        strTable += row.getElementsByTagName("author")[0].firstChild != null
            ? row.getElementsByTagName("author")[0].firstChild.nodeValue
            : "";
        strTable += row.getElementsByTagName("body")[0].firstChild != null
            ? row.getElementsByTagName("body")[0].firstChild.nodeValue
            : "";
        strTable += ajaxPostCont;
    }
    return (strTable);
}

Figure 6.21: JSPBLOG supporting JavaScript method; static and dynamic data merged to produce table data.
Figure 6.22: JSPBLOG before AJAX transformation

Figure 6.23: JSPBLOG after AJAX transformation
Chapter 7

Conclusion

7.1 Summary

With the increasing complexity of the web applications, it becomes difficult to keep track with the web page in the mist of browsing. AJAX provides a seamless web browsing experience to the user such that it decreases the amount of tracking needed. However, the manual transformation of source code from the traditional round-trip to AJAX could be erroneous due to the details required in the process. In this thesis, an approach is presented to automate this transformation process. The transformation is accomplished by two phases, each in a pipeline structure. The first phase transforms the original web page into a web service. This web service provides the dynamic data needed by the web page in an XML format. The second phase transforms the web page to enable the AJAX feature. The AJAX enabled web page has the specified portion of the web page removed, and replaced by an AJAX method. The AJAX method retrieves the dynamic data from the web service and outputs the data to the web page. The following sections will summarize the transformation process of each
7.1.1 XML Generation

The XML Generation phase has five sections of transformations. Source preparation prepares the source code creating a more uniform representation of the source. Slicing preparation prepares the source code for the proper execution of the slicing process. The slicing section slices the source code creating a subset of the source code. The slice consists of codes essential to the execution of the slicing criteria. XML tagging adds XML tags to the output of the source code, which formats the dynamic data for the web service. Clean up reverts the source code into a more conventional format, as the previous transformations added unconventional changes to the code.

7.1.2 Web Page Transformation

The Web Page Transformation phase has six sections of transformations. Source preparation once again prepares the source code for a uniform representation of the source. Other dynamic data replacement replaces web page elements into placeholders that the user wants to be transformed. The placeholder will be filled with dynamic data on the web page’s execution. Exterior print replacement replaces print statements surrounding the dynamic data with string variable concatenation statements. The replaced statements allow AJAX to handle the web page elements surrounding the dynamic data. This retains the original format for the transformed web page. Interior print removal removes the specified web page element from the web page. A placeholder will be used in its place for the element’s placement during the web page execution. Clean up is similar to the previous clean up section and reverts the
source code. AJAX base code merge merges all the web page elements removed from the web page with a base JavaScript method. This JavaScript method mixes static data and dynamic data together and outputs the result to the placeholders in the transformed web page.

### 7.2 Thesis Contribution

The contribution of this thesis is a semi-automated system in transforming a round-trip web application to use AJAX. The transformation approach had performed successfully on four different web applications. Three of the applications come from GotoCode.com [32], and the last from SourceForge [1]. The three applications from GotoCode.com are the forum application, the classifieds posting application, and the events management application. The last one from SourceForge is a blogging application. A web page from each of the application is chosen to be transformed. The chosen web page has the best characteristic of a typical web page that utilizes AJAX, which includes a table of dynamic data that is retrieved from the database. A small amount of user intervention is needed to identify and to mark sections of the web page, where transformations are wanted. The remaining transformation processes are done automatically without requiring the user’s intervention.

The resulting web pages from the transformation execute as meant. They look identical to the original web pages with the exception of a spacing problem in the blogging application. The transformation should conserve as much of the original look of the web page as possible, as the usage of the page should not change. A performance change is seen from the resulting web pages where paging controls are implemented. Due to the use of AJAX, the paging control will only refresh the changed data on the
web page of the new page. This causes the transformed web page to perform faster than the original web page, which shows that AJAX is operating.

7.3 Future Work

The presented approach transformed four web pages from their traditional round-trip design to one that uses AJAX. Although the transformation performed correctly, there are enhancements that could be made to further improve the approach. The following sections will outline the various improvements that can be made.

7.3.1 Other Web Languages

The current approach uses TXL along with the JSP grammar; therefore, the current setup will only work with web applications written in JSP. There are various little implementations throughout the entire transformation where its purpose is geared towards an JSP application. For example, the support JSP page as described in Section 5.5 contains methods which make the AJAX transformation possible. To make this transformation approach possible for other web languages, this support page will have to be translated.

Among the JSP grammar, there are some specific language structures or implementation which other languages lack; such as the JSP scriptlets structure and the JSP directives. While other languages may have something similar in their place, their grammatical format may be different. These grammatical differences will affect the way some of the TXL transformation rules are implemented.
CHAPTER 7. CONCLUSION

Having a generic transformation that will perform for multiple languages is difficult due to the language differences. One may implement a generic transformation that will perform for multiple languages, and have a language specific transformation to make up for the differences. This type of implementation, however, is outside the scope of this thesis.

7.3.2 Mark Up Identification

With the presented approach, the user is required to identify and mark up a section of source code for the transformation. Although the amount of user intervention required for the transformation is small, the approach can be further automated if the marking up of source code can be done automatically. The general identification guideline is to find a listing with an iteration of data retrievals from the database, but the way to implement this structure varies between different developers. Things such as the database access method or the way iteration is made could be largely varied between different applications. Furthermore, multiple structures of such could be found within a web page, while the approach only supports the transformation of one of them. Therefore, user intervention would be required to choose which structure to transform. Due to the variance in performing such identification, and the need for the user’s decision of when multiple transformable structures are found, the identification of the transformable structure is performed by the user in our approach. Mesbah et al. did present an approach which could identify candidates for AJAX transformation [17], however their targets for AJAX transformation are changed web elements between multiple web pages. Our targets are iterations of database data on a single web page.
7.3.3 Transformation of Multiple Elements

As mentioned briefly in the above section, the presented approach only supports the transformation of one element in the web page. The approach can be modified to support the transformation of multiple elements, but careful attention is needed in making the modification. This modification will also remove the obstacle for the automatic mark up identification. For this modification, many of the supporting elements of the transformed AJAX web page will have to be renamed and modified to support multiple AJAX elements in the web page. The naming convention used in the transformation may also have to be changed to support multiple AJAX elements. The work required for this modification is large and is outside the scope of the thesis.

7.3.4 Other Types of Paging Control

For our approach, the support for transforming paging controls is allowed for controls that are based on HTML links. This type of paging control changes the page number of the web page by varying a parameter on the link URL which indicates the current page number. Another form of paging control also used in web applications uses the HTML form submission. With this type of paging control, the page number is sent to the web server through the form submission data instead of the URL parameter. The support of other types of paging controls implementation allows the approach to be used on a wider range of web applications.

Through an informal survey of major websites, table 7.1 shows 18 out of 20 websites use HTML links based paging controls.

The informal survey shows that although there are other methods of implementing the paging controls of which our approach does not handle, they correspond to a small
### Table 7.1: An informal survey of major websites regarding the type of paging control used.

<table>
<thead>
<tr>
<th>Website</th>
<th>Paging control type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>link</td>
</tr>
<tr>
<td>Yahoo</td>
<td>link</td>
</tr>
<tr>
<td>IBM</td>
<td>link</td>
</tr>
<tr>
<td>Sun</td>
<td>link</td>
</tr>
<tr>
<td>Microsoft</td>
<td>link</td>
</tr>
<tr>
<td>Dell.ca</td>
<td>link</td>
</tr>
<tr>
<td>Staples.ca</td>
<td>link</td>
</tr>
<tr>
<td>The Source</td>
<td>link</td>
</tr>
<tr>
<td>Government of Canada Official Web Site</td>
<td>link</td>
</tr>
<tr>
<td>TD Canada Trust</td>
<td>link</td>
</tr>
<tr>
<td>CIBC</td>
<td>link</td>
</tr>
<tr>
<td>RBC Royal Bank</td>
<td>link</td>
</tr>
<tr>
<td>Bank of Montreal</td>
<td>link</td>
</tr>
<tr>
<td>Bell.ca</td>
<td>link</td>
</tr>
<tr>
<td>Rogers</td>
<td>link</td>
</tr>
<tr>
<td>TXL Forum</td>
<td>link</td>
</tr>
<tr>
<td>Elsevier</td>
<td>link</td>
</tr>
<tr>
<td>Canada411.ca</td>
<td>link</td>
</tr>
<tr>
<td>YellowPages.ca</td>
<td>non-link</td>
</tr>
<tr>
<td>IEEE Xplore</td>
<td>non-link</td>
</tr>
</tbody>
</table>
population of websites. Many other websites not included in this survey not using the HTML link based paging controls are those that already use AJAX.

### 7.3.5 Further Retaining of Original Format

While the approach retains most of the original format of the transformed web page, there are possible cases where some of the original formats are not retained. Such a case is seen from the transformation of JSPBLOG, where an extra space is added after the apostrophe in the transformed source code. Another issue deals with the source code comments which are lost in the midst of transformations. In further retaining the original format of the source code, an approach was presented by Malton et al. [15]. In their presented structure, the source code is factorized into a common structural decomposition. This will allow the transformed source code to be merged and integrated with the original source code. The resulting source code will retain the original format and comments, because those parts of the source code would not be modified by the transformations.

### 7.3.6 Additional Language Features Support

Most web development languages are based from other desktop application languages, but they have additional features which are added specifically for web purposes. Some of these features are commonly found between different languages and some are language specific. For example, the ability to compose a single web page using multiple smaller web pages can be found in JSP and several other languages such as PHP or ASP. JSP has the feature to make custom tags where the user can design their own tags containing their own functionalities, but this only applies to JSP.
For the presented approach, the most basic and common form of a web page is supported. The web page would have to be a single file, where all of the source codes required for the generation of the web page are contained in the one file (with exceptions to globally declared variables or methods). Any support of the language features would potentially require major changes to the slicing procedure and other transformation rules, and would be outside the scope of the thesis. Lastly, the additional support for any of the language specific features to the transformation would make it difficult to be adapted for other languages.

7.4 Conclusion

AJAX is widely adopted and used by many developers and companies to enhance their web application. Many of these AJAX driven web applications are coded manually, and there are no automated tools that can transform an existing round-trip web application to utilize AJAX. To manually transform the web application is a time consuming and difficult task, which could lead to an erroneous result due to the complicated process. An automated tool is needed to make the task easier.

The approach presented in this thesis uses TXL to transform a round-trip web page to utilize AJAX. The transformation is done in two parts. The first transformation transforms the web page into a web service that generates dynamic data for AJAX in an XML format. The second transformation transforms the web page to use AJAX, which takes the dynamic data from the web service from the first transformation. The user initiates the entire transformation by marking the source code, indicating the web element to be replaced by AJAX. The rest of the transformation is done automatically. The approach has been performed on four different web pages, all of
which have been successfully transformed to utilize AJAX.
Bibliography


Appendix A

GotoCode Forum User-Marked Front Page

<%@ include file="Common.jsp" %>

//
//   Filename: index.jsp
//   Generated with CodeCharge v.1.2.0
//   JSP.ccp build 05/21/01
//
static final String sFileName = "index.jsp";

static final String PageBODY = "link=\"#000000\" vlink=\"#0033cc\" alink
   =\"red\"\";
static final String FormTABLE = "border=\"0\" cellspacing=\"1\"
   cellpadding=\"3\"\";
static final String FormHeaderTD = "bgcolor=\"#c2c2c2\"\";
static final String FormHeaderFONT = "face=\"arial\" style=\"font:bold\"
   color=\"#0033cc\"\";
APPENDIX A. GOTO CODE FORUM USER-MARKED FRONT PAGE

```java
static final String FieldCaptionTD = "bgcolor="#f2f2f2";
static final String FieldCaptionFONT = "face=\"arial\" size=\"2\" style =\"font:bold\" color=\"#0033cc\"";
static final String DataTD = "";
static final String DataFONT = "face=\"arial\" size=\"2\"";
static final String ColumnFONT = "face=\"arial\" size=\"2\" style=\"font :bold\" color=\"#0033cc\"";
static final String ColumnTD = "bgcolor="#f2f2f2";%
%
boolean bDebug = false;
String sAction = getParam( request , "FormAction");
String sForm = getParam( request , "FormName");
String sMessagesErr = "";
String sSearchErr = "";
java.sql.Connection conn = null;
java.sql.Statement stat = null;
String sErr = loadDriver();
conn = cn();
stat = conn.createStatement();
if ( ! sErr.equals("")) {
  try {
    out.println(sErr);
  } catch (Exception e) {}%
}
<html>
<head>
<title>Forum</title>
```
<meta name="GENERATOR" content="YesSoftware CodeCharge v.1.2.0 / JSP ccp build 05/21/01"/>
<meta http-equiv="pragma" content="no-cache"/>
<meta http-equiv="expires" content="0"/>
<meta http-equiv="cache-control" content="no-cache"/>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
</head>
<body link="#000000" vlink="#0033cc" alink="red">
<jsp:include page="header.jsp" flush="true"/>
<table>
<tr>
<td align="top">
<% Search_Show(request, response, session, out, sSearchErr, sForm, sAction, conn, stat); %>
</td>
</tr>
</table>
<table>
<tr>
<td align="top">
<% Messages_Show(request, response, session, out, sMessagesErr, sForm, sAction, conn, stat); %>
</td>
</tr>
</table>
This dynamic site was generated with CodeCharge.

```java
void Messages_Show (javax.servlet.http.HttpServletRequest request,
    http.HttpSession session, javax.servlet.jsp.JspWriter out, String
sMessagesErr, String sForm, String sAction, java.sql.Connection
conn, java.sql.Statement stat) throws java.io.IOException {

    String sWhere = "";
    int iCounter = 0;
    int iPage = 0;
    boolean bIsScroll = true;
    boolean hasParam = false;
    String sOrder = "";
    String sSQL="";
    String transitParams = "";
    String sQueryString = "";
    String sPage = "";
    int RecordsPerPage = 20;
    String sSortParams = "";
```
String formParams = "";
String ps_topic="";
transitParams = "s_topic=" + toURL(getParam( request, "s_topic")) + 
"&";
formParams = "s_topic=" + toURL(getParam( request, "s_topic")) + 
"&";
// Build WHERE statement
// Check Messages parameter and create a valid sql for where clause

ps_topic = getParam( request, "s_topic");
if ( ! ps_topic.equals("" ) ) {
    hasParam = true;
    sWhere = "m_topic like '%" + replace(ps_topic, " ", " ") + "%' or " + "m_message like '%" + replace(ps_topic, " ", " ") + "%'";
    if (hasParam) {
        sWhere = " WHERE (message_parent_id is null) AND (" + sWhere + ") ";
    }
    else {
        sWhere = " WHERE message_parent_id is null";
    }
} // Build ORDER statement
sOrder = " order by m.date_entered Desc";
String sSort = getParam( request, "FormMessages_Sorting");
String sSorted = getParam( request, "FormMessages_Sorted");
String sDirection = "";
String sForm_Sorting = "";
int iSort = 0;
try {
    iSort = Integer.parseInt(sSort);
}

catch (NumberFormatException e) {
    sSort = "";
}

if ( iSort == 0 ) {
    sForm_Sorting = "";
}

else {
    if ( sSort.equals(sSorted) ) {
        sSorted="0";
        sForm_Sorting = "";
        sDirection = " DESC";
        sSortParams = "FormMessages_Sorting=" + sSort + "&
                        FormMessages_Sorted=" + sSort + "&";
    }
    else {
        sSorted=sSort;
        sForm_Sorting = sSort;
        sDirection = " ASC";
        sSortParams = "FormMessages_Sorting=" + sSort + "&
                        FormMessages_Sorted=" + &";
    }
}

if ( iSort == 1 ) { sOrder = " order by m.topic" + sDirection; }
if ( iSort == 2 ) { sOrder = " order by m.author" + sDirection; }
if ( iSort == 3 ) { sOrder = " order by m.date_entered" +
                    sDirection; }
APPENDIX A. GOTOCODE FORUM USER-MARKED FRONT PAGE

// Build full SQL statement

sSQL = "select m.author as m_author, " +
"m.date_entered as m_date_entered, " +
"m.message as m_message, " +
"m.message_id as m_message_id, " +
"m.topic as m_topic " +
" from messages m ";

sSQL = sSQL + sWhere + sOrder;
String sNoRecords = "<tr><td colspan="4">" +
"<font face="arial" size="2">No records</font></td></tr>";
String tableHeader = "";

tableHeader = "<tr><td bgcolor="#f2f2f2"><a>&lt;font face="arial" size="2" style="font : bold" color ="#0033cc">Thread</font>&gt;</a></td><td bgcolor="#f2f2f2"><a>&lt;font face="arial" size="2" style="font : bold" color ="#0033cc">Author</font>&gt;</a></td><td bgcolor="#f2f2f2"><a>&lt;font face="arial" size="2" style="font : bold" color ="#0033cc">Date Entered</font>&gt;</a></td></tr>";
try {
    out.println("<table border="0" cellspacing="1" cellpadding="3">
    <tr><td bgcolor="#c2c2c2" colspan="4"><a name="Messages"><font face="arial" style="font:bold color="#0033cc">Discussions</a></font></td></tr>
    
    " + tableHeader + "
    
    " + iPage + "
    " + RecordsPerPage + ");
    out.println(tableHeader);
}

} catch (Exception e) {}

try {
    // Select current page
    iPage = Integer.parseInt(getParam(request, "FormMessages.Page"));
}

} catch (NumberFormatException e) {
    iPage = 0;
}

if (iPage == 0) { iPage = 1; }

RecordsPerPage = 3;

try {
    java.sql.ResultSet rs = null;
    // Open recordset
    rs = openrs(stat, sSQL);
    iCounter = 0;
    absolute(rs, (iPage-1)*RecordsPerPage+1);
    java.util.Hashtable rsHash = new java.util.Hashtable();
String[] aFields = getFieldsName(rs);
// Show main table based on recordset
<ajax::mark>
while (iCounter < RecordsPerPage && rs.next()) {
    getRecordToHash(rs, rsHash, aFields);
    String fldauthor = (String) rsHash.get("m_author");
    String flddate_entered = (String) rsHash.get("m_date_entered");
    String fldmessage_id = (String) rsHash.get("m_message_id");
    String fldtopic = (String) rsHash.get("m_topic");
    String fldField1 = "<img src=images/Thread_small.gif>";
    out.println("<tr>");
    out.print("<td>");
    out.print("<font face="arial" size="2">" + fldField1 + "\n\n</font>");
    out.println("</td>");
    out.print("<td>");
    out.print("<a href="viewthread.jsp?mid=" + toURL((String) rsHash.get("m_message_id")) + ";&">" + toHTML(fldtopic) + "</a>");
    out.println("</td>");
    out.print("<td>");
    out.print("<font face="arial" size="2">" + toHTML(fldauthor) + "\n\n</font>");
    out.println("</td>");
    out.print("<td>");
    out.print("<font face="arial" size="2">" + toHTML(flddate_entered) + "\n\n</font>");
    out.println("</td>");
    out.println("</tr>");
    iCounter++;
}
<ajax::mark>
if (iCounter == 0) {
  // Recordset is empty
  out.println(sNoRecords);

  out.println("<tr><th colspan="4" bgcolor="#f2f2f2">n
  <font face="arial" size="2" style="font:bold" color="#0033cc">n
  </font><a href="/newthread.jsp?+formParams+">n
  <img border="0" src="/images/thread.gif"></a>&nbsp;&nbsp;
  </th></tr>" );

  iCounter = RecordsPerPage+1;
  bIsScroll = false;
}
else {

  // Parse scroller
  boolean bInsert = false;
  boolean bNext = rs.next();
  if ( !bNext && iPage == 1 ) {

  out.println("<tr><th colspan="4" bgcolor="#f2f2f2">n
  <font face="arial" size="2" style="font:bold" color="#0033cc">n
  </font>");

  out.println("<tr><th colspan="4" bgcolor="#f2f2f2">n
  <font face="arial" size="2" style="font:bold" color="#0033cc">n
  </font>");
out.println("\n"+a.href="newthread.jsp?"+formParams+"\""><font face="arial" size="2" style="font:bold" color ="\"#0033cc\""> <img border="0" src="images/thread.gif"></font></a><br> out.println("\n\n</td>\n</tr>\n")
} else {
    out.print("\n<tr><td colspan="4" bgcolor="#f2f2f2"><font face="arial" size="2" style="font:bold" color ="\"#0033cc\""> Previous\n</font></td></tr>\n")
}

out.println("\n"+a.href="newthread.jsp?"+formParams+"\""><font face="arial" size="2" style="font:bold" color ="\"#0033cc\""> <img border="0" src="images/thread.gif"></font></a>&nbsp; Previous\n";

bInsert = true;

<ajax:annotate>
if ( iPage == 1 ) {
    out.println("\n"+a.href=""+sFileName+"?"+formParams+sSortParams+"FormMessages_Page="+(iPage - 1)+"#Form\""><font face="arial" size="2" style="font:bold" color ="\"#0033cc\"">Previous</font></a>\n")
} else {
    out.println("\n"+a.href=""+sFileName+"?"+formParams+sSortParams+"FormMessages_Page="+(iPage - 1)+"#Form\""><font face="arial" size="2" style="font:bold" color ="\"#0033cc\"">Previous</font></a>\n")
}
APPENDIX A. GOTOCODE FORUM USER-MARKED FRONT PAGE

```java
{
}  
</ajax::annotate type=[dlink]>

<ajax::annotate>
out.print("\n ["+iPage+" ]"));
</ajax::annotate type=[dlink]>

<ajax::annotate>
if (!bNext) {
    out.print("\n <a href="#"">font face="arial" size ="2" style="font:bold" color="#0033cc">Next</font></a>
    <br>");
}
else {
    out.print("\n <a href=""+sFileName+"?"+formParams+
    sSortParams+"FormMessages_Page="+(iPage + 1)+"#Form"">font
    face="arial" size ="2" style="font:bold" color="#0033cc">Next</font></a><br>");
}
</ajax::annotate type=[dlink]>

if (!bInsert) {
    out.print("\n <tr><td colspan="4" bgcolor="#f2f2f2"
    "><font face="arial" size ="2" style="font:bold" color="#0033cc">";
    out.print("\n <a href="/newthread.jsp?"+formParams+""><font
    face="arial" size ="2" style="font:bold" color="#0033cc">";
    
    <img border="0" src="/images/thread.gif"/>
</font></a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;");
```
APPENDIX A. GOTOCODE FORUM USER-MARKED FRONT PAGE

```java
    try {

        String flds_topic = "";
        String sSQL = "";
        String transitParams = "";
        String sQueryString = "";
        String sPage = "";

        out.println("<table border="0" cellspacing="1" cellpadding="3" >");
    }
    catch (Exception e) { out.println(e.toString()); }
}
```
APPENDIX A. GOTOCODE FORUM USER-MARKED FRONT PAGE

```java
out.println("<form method="get" action="/searches.jsp"
           name="Search">
<tr>
// Set variables with search parameters

flds_topic = getParam( request, "s_topic");
// Show fields

out.println("<td bgcolor="#f2f2f2"><font face="arial"
            size="2" style="font:bold" color="#0033cc"></font></td>");
out.println("<td>"); out.println("<input type="text" name="
            s_topic" maxlength="50" value=""+toHTML(flds_topic)+""
            size=""/>");
out.println("</td>");

out.println("<td><input type="submit" value="Search
            "/></td>");
out.println("</tr>
</form>
</table>
"

} catch (Exception e) { out.println(e.toString()); }

%>
```
Appendix B

GotoCode Forum XML Web Service

```java
<%@ include file = "AJAXCore.jsp" %>
<%@ page contentType = "text/xml" %>
<%@ include file = "Common.jsp" %>
<%! static final String sFileName = "index.jsp"; %>


    String sWhere = "";
    int iCounter = 0;
    int iPage = 0;
    boolean hasParam = false;
    String sOrder = "";
    String sSQL = "";
```
String transitParams = "";
int RecordsPerPage = 20;
String sSortParams = "";
String formParams = "";
String ps_topic = "";
transitParams = "s_topic=" + toURL (getParam (request, "s_topic")) + 
"&";
formParams = "s_topic=" + toURL (getParam (request, "s_topic")) + 
"&";
ps_topic = getParam (request, "s_topic");
if (! ps_topic.equals (""))
{
  hasParam = true;
  sWhere = "m_topic like '%" + replace (ps_topic , " '", "'") + 
"%' " or " + "m_message like '%" + replace (ps_topic , 
" '", "'") + 
"%";
}
if (hasParam)
{
  sWhere = " WHERE (message_parent_id is null) AND (" + sWhere 
+ ")";
}
else
{
  sWhere = " WHERE message_parent_id is null";
}
sOrder = " order by m_date_entered Desc";
String sSort = getParam (request, "FormMessages_Sorting");
String sSorted = getParam (request, "FormMessages_Sorted");
int iSort = 0;
try {
    iSort = Integer.parseInt(sSort);
} catch (NumberFormatException e) {
    sSort = "";
}
if (iSort == 0)
{
}
else
{
    if (sSort.equals(sSorted))
    {
        sSorted = "0";
        sSortParams = "FormMessages_Sorting=" + sSort + "&
                           FormMessages_Sorted=" + sSort + "&";
    }
    else
    {
        sSorted = sSort;
        sSortParams = "FormMessages_Sorting=" + sSort + "&
                           FormMessages_Sorted=" + "&";
    }
}
sSQL = "select m.author as m_author, " + "m.date_entered as 
m_date_entered, " + "m.message as m_message, " + "m.message_id
as m_message_id, " + "m.topic as m_topic " + " from messages m ";
sSQL = sSQL + sWhere + sOrder;
try {
    iPage = Integer.parseInt(getParam(request, "FormMessages_Page"));
} catch (NumberFormatException e) {
    iPage = 0;
}
if (iPage == 0) {
    iPage = 1;
}
RecordsPerPage = 3;
try {
    java.sql.ResultSet rs = null;
    rs = openrs(stat, sSQL);
    iCounter = 0;
    absolute(rs, (iPage - 1) * RecordsPerPage + 1);
    java.util.Hashtable rsHash = new java.util.Hashtable();
    String [] aFields = getFieldsName(rs);
    out.print("<ajaxlist>");
    while ((iCounter < RecordsPerPage) && rs.next()) {
        out.print("<ajaxitem>");
        getRecordToHash(rs, rsHash, aFields);
        String fldauthor = (String) rsHash.get("m_author");
        String flddate_entered = (String) rsHash.get("m_date_entered");
        String fldtopic = (String) rsHash.get("m_topic");
        out.print("<toURL-rsHash-get-m_message_id>");
    }
}
out.println(HTMLFilter(toURL((String)rsHash.get("m_message_id"))));
out.println("</URL−rsHash−get−m_message_id>");
out.println("<toHTML−fldtopic>");
out.println(HTMLFilter(toHTML(fldtopic)));
out.println("</toHTML−fldtopic>");
out.println("<toHTML−fldauthor>");
out.println(HTMLFilter(toHTML(fldauthor)));
out.println("</toHTML−fldauthor>");
out.println("<toHTML−flddate_entered>");
out.println(HTMLFilter(toHTML(flddate_entered)));
out.println("</toHTML−flddate_entered>");
iCounter ++;
out.println("</ajaxitem>");
}
out.println("</ajaxlist>");
if (iCounter == 0)
{

}
else
{
    boolean bNext = rs.next();
    if (!bNext && iPage == 1)
    {
    
    }
    else
    {
        if (iPage == 1)
        {
        
    }
out.print ("<
    sFileNameformParamssSortParamsiPage0
>");
out.print (HTMLFilter (AJAXLinkConvert
("\n    <a href="#">font face="arial" size="2" style="font: bold" color="#0033cc">Previous</font></a>")));
out.print ("</
    sFileNameformParamssSortParamsiPage0
>");
}
else
{
    out.print ("<
    sFileNameformParamssSortParamsiPage0
>");
out.print (HTMLFilter (AJAXLinkConvert
("\n        <a href="" + sFileName + 
        "?" + formParams + sSortParams + "
        FormMessages_Page=" + (iPage - 1) + 
        "#Form">font face="arial" size 
        ="2" style="font: bold" color 
        ="#0033cc">Previous</font></a>")));
out.print ("</
    sFileNameformParamssSortParamsiPage0
>");
}
out.print("<iPage1>");
out.print(HTMLFilter(AJAXLinkConvert("\n
[ " + iPage + "]\n")));
out.print("</iPage1>");
if (!bNext)
{
    out.print("<
    sFileNameformParamssSortParamsiPage2
>");
    out.print(HTMLFilter(AJAXLinkConvert("\n    <a href="#"><font face="arial" size="2" style="font: bold" color="#0033cc">Next</font><br/>
</a></br>")));
    out.print("</
    sFileNameformParamssSortParamsiPage2
>");
}
else
{
    out.print("<
    sFileNameformParamssSortParamsiPage2
>");
    try {
        String flds_topic = "";
        flds_topic = getParam (request, "s_topic");
    } catch (Exception e) {
    }
}

APPENDIX B. GOTOCODE FORUM XML WEB SERVICE
<%
out.println("<ajaxdata>");
String sAction = getParam(request, "FormAction");
String sForm = getParam(request, "FormName");
String sMessagesErr = "";
String sSearchErr = "";
java.sql.Connection conn = null;
java.sql.Statement stat = null;
conn = cn();
stat = conn.createStatement();
Search_Show(request, response, session, out, sSearchErr, sForm, sAction, conn, stat);
Messages_Show(request, response, session, out, sMessagesErr, sForm, sAction, conn, stat);
out.println("</ajaxdata>");
%>
Appendix C

GotoCode Forum AJAX Page

```jsp
<%@ include file = "AJAXCore.jsp" %>
<%@ include file = "Common.jsp" %>
<%! static final String sFileName = "index.jsp";
    static final String PageBODY = "link=\"#000000\" vlink=\"#0033cc\" alink =\"red\"";
    static final String FormTABLE = "border=\"0\" cellspacing=\"1\"
        cellpadding=\"3\"";
    static final String FormHeaderTD = "bgcolor=\"#c2c2c2\"";
    static final String FormHeaderFONT = "face=\"arial\" style=\"font:bold\"
        color=\"#0033cc\"";
    static final String FieldCaptionTD = "bgcolor=\"#f2f2f2\"";
    static final String FieldCaptionFONT = "face=\"arial\" size=\"2\" style =\"font:bold\"
        color=\"#0033cc\"";
    static final String DataTD = ";
    static final String DataFONT = "face=\"arial\" size=\"2\"";
    static final String ColumnFONT = "face=\"arial\" size=\"2\" style=\"font
        :bold\" color=\"#0033cc\"";
    static final String ColumnTD = "bgcolor=\"#f2f2f2\"";"
```
boolean bDebug = false;
String sAction = getParam (request, "FormAction");
String sForm = getParam (request, "FormName");
String sMessagesErr = "";
String sSearchErr = "";
java.sql.Connection conn = null;
java.sql.Statement stat = null;
String sErr = loadDriver ();
conn = cn ();
stat = conn.createStatement ();
if (!sErr.equals (""))
{
    try {
        out.println (sErr);
    } catch (Exception e) {
    }
}
<html>
<head>
<title>Forum</title>
<meta name="GENERATOR" content="YesSoftware CodeCharge v.1.2.0 / JSP.ccp build 05/21/01" />
<meta http-equiv="pragma" content="no-cache" />
<meta http-equiv="expires" content="0" />
<meta http-equiv="cache-control" content="no-cache" />
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"
>
</head>
<body link="#000000" vlink="#0033cc" alink="red">
<jsp:include page="header.jsp" flush="true" />
<table>
<tr>
<td valign="top">
<%
Search_Show (request, response, session, out, sSearchErr, sForm, sAction, conn, stat);
%
</td>
</tr>
</table>
<table>
<tr>
<td valign="top">
<%
Messages_Show (request, response, session, out, sMessagesErr, sForm, sAction, conn, stat);
%
</td>
</tr>
</table>
<center>
<font face="Arial">
<small>
This dynamic site was generated with <a href="http://www.codecharge.com" >CodeCharge</a>
</small></font></center></body></html>
<% %>
<% 
  if (stat != null)
    stat.close();
  if (conn != null)
    conn.close();
%>
<%!
  String sWhere = "";
  int iCounter = 0;
  int iPage = 0;
  boolean bIsScroll = true;
  boolean hasParam = false;
  String sOrder = "";
  String sSQL = "";
String transitParams = "";
String sQueryString = "";
String sPage = "";
int RecordsPerPage = 20;
String sSortParams = "";
String formParams = "";
String ps_topic = "";

transitParams = "s_topic=" + toURL (getParam (request, "s_topic")) + "&";
formParams = "s_topic=" + toURL (getParam (request, "s_topic")) + "&";

ps_topic = getParam (request, "s_topic");
if (! ps_topic.equals (""))
{
    hasParam = true;
    sWhere = "m.topic like '%" + replace (ps_topic, "'" , "''") + "%'" + " or " + "m.message like '%" + replace (ps_topic , "'" , "''") + "%'";
}
if (hasParam)
{
    sWhere = " WHERE (message_parent_id is null) AND (" + sWhere + ")";
}
else
{
    sWhere = " WHERE message_parent_id is null";
}
sOrder = " order by m.date_entered Desc";
String sSort = getParam (request, "FormMessages.Sorting");
String sSorted = getParam (request, "FormMessages.Sorted");
String sDirection = "";
String sForm_Sorting = "";
int iSort = 0;
try {
    iSort = Integer.parseInt (sSort);
} catch (NumberFormatException e) {
    sSort = "";
}
if (iSort == 0)
{
    sForm_Sorting = "";
}
else
{
    if (sSort.equals (sSorted))
    {
        sSorted = "0";
        sForm_Sorting = "";
        sDirection = " DESC";
        sSortParams = "FormMessages.Sorting=" + sSort + ":& FormMessages.Sorted=" + sSort + "&";
    }
    else
    {
        sSorted = sSort;
        sForm_Sorting = sSort;
        sDirection = " ASC";
sSortParams = "FormMessages.Sorting=" + sSort + "& FormMessages.Sorted=" + ";
}
if (iSort == 1)
{
    sOrder = " order by m.topic" + sDirection;
}
if (iSort == 2)
{
    sOrder = " order by m.author" + sDirection;
}
if (iSort == 3)
{
    sOrder = " order by m.date_entered" + sDirection;
}
}
sSQL = "select m.author as m_author, " + "m.date_entered as m_date_entered, " + "m.message as m_message, " + "m.message_id as m_message_id, " + "m.topic as m_topic " + " from messages m ";
sSQL = sSQL + sWhere + sOrder;
String sNoRecords = " <tr><td colspan="4"> <font face ="arial" size="2">No records</font></td></tr>";
String tableHeader = "";
tableHeader = "<tr><td bgcolor="#f2f2f2"><a>&gt;&lt;font face="arial" size="2" style="font:bold" color="#0033cc">&lt;/font&gt;&lt;/a&gt;&lt;/td&gt;&lt;td bgcolor="#f2f2f2"&gt;&lt;a href="" + sFileName + "?" + formParams + " FormMessages_Sorting=1&amp;FormMessages_Sorted=" + sSorted + "&amp;FormMessages_Sorting=2&amp;FormMessages_Sorted=" + sSorted + "&amp;FormMessages_Sorting=3&amp;FormMessages_Sorted=" + sSorted + "&amp;">&lt;font face="arial" size="2" style="font:bold" color="#0033cc">Thread</font&gt;&lt;/a&gt;&lt;/td&gt;&lt;td bgcolor="#f2f2f2"&gt;&lt;a href="" + sFileName + "?" + formParams + " FormMessages_Sorting=1&amp;FormMessages_Sorted=" + sSorted + "&amp;&gt;&lt;font face="arial" size="2" style="font:bold" color="#0033cc">Author</font&gt;&lt;/a&gt;&lt;/td&gt;&lt;td bgcolor="#f2f2f2"&gt;&lt;a href="" + sFileName + "?" + formParams + " FormMessages_Sorting=1&amp;FormMessages_Sorted=" + sSorted + "&amp;FormMessages_Sorting=2&amp;FormMessages_Sorted=" + sSorted + "&amp;FormMessages_Sorting=3&amp;FormMessages_Sorted=" + sSorted + "&amp;">&lt;font face="arial" size="2" style="font:bold" color="#0033cc">Date Entered</font&gt;&lt;/a&gt;&lt;/td&gt;&lt;/tr&gt;"; try {
  preData += "&lt;table border="0" cellspacing="1" cellpadding="3"&gt;";
  preData += "&lt;tr&gt;&lt;td bgcolor="#c2c2c2" colspan="4"&gt;&lt;a name="Messages"&gt;&lt;font face="arial" style="font:bold" color="#0033cc">Discussions</font&gt;&lt;/a&gt;&lt;/td&gt;&lt;/tr&gt;";
  preData += tableHeader;
} catch (Exception e) {
}

try {
  iPage = Integer.parseInt(getParam(request, "FormMessages_Page"));
} catch (NumberFormatException e) {
iPage = 0;
}
if (iPage == 0)
{
    iPage = 1;
}
RecordsPerPage = 3;
try {
    java.sql.ResultSet rs = null;
    rs = openrs (stat, sSQL);
    iCounter = 0;
    absolute (rs, (iPage - 1) * RecordsPerPage + 1);
    java.util.Hashtable rsHash = new java.util.Hashtable();
    String [] aFields = getFieldsName (rs);
    while ((iCounter < RecordsPerPage) && rs.next ())
    {
        getRecordToHash (rs, rsHash, aFields);
        String fldauthor = (String) rsHash.get ("m_author");
        String flddate_entered = (String) rsHash.get ("m_date_entered");
        String fldmessage_id = (String) rsHash.get ("m_message_id");
        String fldtopic = (String) rsHash.get ("m_topic");
        String fldField1 = "<img src=images/Thread_small.gif>";
        iCounter ++;
    }
    if (iCounter == 0)
    {
        postData += sNoRecords;
    }
postData += " <tr><n t><td colspan="4" bgcolor="#f2f2f2"><font face="arial" size="2" style="font:bold" color="#0033cc">";  
 wildfire += " newthread.jsp?" + formParams + " ><font face="arial" size="2" style="font:bold" color="#0033cc">"<img border="0" src="images/thread.gif"></a>&nbsp;&nbsp;";  
 postData += " </td><n t><tr>";  
 iCounter = RecordsPerPage + 1;  
 bIsScroll = false;  
 }  
 else  
 {  
 boolean bInsert = false;  
 boolean bNext = rs.next();  
 if (! bNext && iPage == 1)  
 {  
 postData += " <tr><n t><td colspan="4" bgcolor="#f2f2f2">n t><font face="arial" size="2" style="font:bold" color="#0033cc">";  
 postData += " newthread.jsp?" + formParams + " ><font face="arial" size="2" style="font:bold" color="#0033cc">"<img border="0" src="images/thread.gif"></a>&nbsp;&nbsp;";  
 postData += " </td><n t><tr>";  
 iCounter = RecordsPerPage + 1;  
 bIsScroll = false;  
 }  
 else  
 {  
 boolean bInsert = false;  
 boolean bNext = rs.next();  
 if (! bNext && iPage == 1)  
 {  
 postData += " <tr><n t><td colspan="4" bgcolor="#f2f2f2">n t><font face="arial" size="2" style="font:bold" color="#0033cc">";  
 postData += " newthread.jsp?" + formParams + " ><font face="arial" size="2" style="font:bold" color="#0033cc">"<img border="0" src="images/thread.gif"></a>&nbsp;&nbsp;";  
 postData += " </td><n t><tr>";  
 iCounter = RecordsPerPage + 1;  
 bIsScroll = false;  
 }  
 else  
 {  
 boolean bInsert = false;  
 boolean bNext = rs.next();  
 if (! bNext && iPage == 1)  
 {  
 postData += " <tr><n t><td colspan="4" bgcolor="#f2f2f2">n t><font face="arial" size="2" style="font:bold" color="#0033cc">";  
 postData += " newthread.jsp?" + formParams + " ><font face="arial" size="2" style="font:bold" color="#0033cc">"<img border="0" src="images/thread.gif"></a>&nbsp;&nbsp;";  
 postData += " </td><n t><tr>";  
 iCounter = RecordsPerPage + 1;  
 bIsScroll = false;  
 }  
 else  
 {  
 boolean bInsert = false;  
 boolean bNext = rs.next();  
 if (! bNext && iPage == 1)  
 {  
 postData += " <tr><n t><td colspan="4" bgcolor="#f2f2f2">n t><font face="arial" size="2" style="font:bold" color="#0033cc">";  
 postData += " newthread.jsp?" + formParams + " ><font face="arial" size="2" style="font:bold" color="#0033cc">"<img border="0" src="images/thread.gif"></a>&nbsp;&nbsp;";  
 postData += " </td><n t><tr>";  
 iCounter = RecordsPerPage + 1;  
 bIsScroll = false;  
 }
else {

postData += "<tr>
    <td colspan="4" bgcolor="#f2f2f2">" + font + 
    size="2" style="font: bold" color="#0033cc">
</font><a href="newthread.jsp?
+ formParams + 
    <img border="0" src="images/thread.gif
"/></a>&nbsp;&nbsp;&nbsp;

bInsert = true;
postData += "<span id="sFileNameformParamssSortParamsiPage0">
    sFileNameformParamssSortParamsiPage0"</span>

postData += "<span id="iPage1"></span>

postData += "<span id="sFileNameformParamssSortParamsiPage2">
    sFileNameformParamssSortParamsiPage2"</span>

if (! bInsert) {

postData += "<tr>
    <td colspan="4" bgcolor="#f2f2f2">" + font + 
    size="2" style="font: bold" color="#0033cc">";

}
```java
    try {
        String flds_topic = "";
        String sSQL = "";
        String transitParams = "";
        String sQueryString = "";
        postData += "\n        <a href="" + formParams + "">";
        font face="arial" size="2" style ="font:bold" color="#0033cc"><img border="0" src="images/thread.gif"></font></a>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
        postData += "</td><tr>
                
                }
                if (rs != null)
                    rs.close();
                    postData += "</table>";
                    ajaxExData (out, request.getRequestURI ());
                    } catch (Exception e) {
                        out.println (e.toString ());
                    }
}
```
String sPage = "";
out.println ("<table border="0" cellspacing="1"
cellpadding="3"><tr>"
);
out.println ("<form method="get" action="searches.jsp"
name="Search"><tr>"
flds_topic = getParam (request, "s_topic");
out.println ("<td bgcolor="#f2f2f2"><font face="arial"
size="2" style="font: bold color="#0033cc">"+toHTML (flds_topic)+"</font></td>"
);
out.print ("<td >
out.print ("<input type="text" name="s_topic" maxlength
=50" value=""+toHTML (flds_topic)+"" size=""">
out.println ("</td>");
out.println ("</tr></form></table>
"
);
}
catch (Exception e) {
    out.println (e.toString ());
}
"
function tableMerge (ajaxlist )
{
    var strTable = "";
    strTable += ajaxPreCont;
    for ( var rowIndex = 0; rowIndex < ajaxlist.childNodes.length;
        rowIndex ++)
    {
        var row = ajaxlist.childNodes.item( rowIndex);
        strTable += "   <tr>");
        strTable += "   <td >";
        strTable += "<font face="arial" size="2">";
        strTable += "<img src=images/Thread_small.gif">");
        strTable += "&nbsp;</font>");
        strTable += "</td>");
        strTable += "   <td >";
        strTable += "";
    }
strTable += "<a href="viewthread.jsp?mid=";
strTable += if (row.getElementsByTagName("toURL-rsHash-get-m_message_id")
[0].firstChild != null )
strTable += row.getElementsByTagName("toURL-rsHash-get-m_message_id")[0].firstChild.nodeValue;
strTable += ";
strTable += if (row.getElementsByTagName("toHTML-fldtopic")[0].firstChild
!= null )
strTable += row.getElementsByTagName("toHTML-fldtopic")[0].
firstChild.nodeValue;
strTable += "</font></a>";
strTable += ";
strTable += "</td>";
strTable += "<font face="arial" size="2" >";
strTable += if (row.getElementsByTagName("toHTML-fldauthor")
[0].firstChild
!= null )
strTable += row.getElementsByTagName("toHTML-fldauthor")[0].
firstChild.nodeValue;
strTable += ";
strTable += ";
strTable += \&nbsp;</font>";
strTable += "</td>";
strTable += "<font face="arial" size="2" >";
strTable += if (row.getElementsByTagName("toHTML-flldate_entered")
[0].firstChild != null )
strTable += row.getElementsByTagName("toHTML-flldate_entered")
[0].firstChild.nodeValue;
strTable += ";
strTable += \&nbsp;</font>";
strTable += "</td>";
strTable += "</tr>
};

strTable += ajaxPostCont;
return ( strTable);
}