The Journal of Systems and Software 86 (2013) 72–88

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Refactoring legacy AJAX applications to improve the efficiency of the data exchange component

Ming Ying*, James Miller

Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada

ARTICLE INFO

Article history:
Received 3 September 2011
Received in revised form 28 June 2012
Accepted 6 July 2012
Available online 25 July 2012

Keywords:
XML
JSON
JavaScript
AJAX
Efficiency
Refactoring

ABSTRACT

The AJAX paradigm encodes data exchange XML formats. Recently, JSON has also become a popular data exchange format. XML has numerous benefits including human-readable structures and self-describing data. However, JSON provides significant performance gains over XML due to its light weight nature and native support for JavaScript. This is especially important for Rich Internet Applications (RIA). Therefore, it is necessary to change the data format from XML to JSON for efficiency purposes. This paper presents a refactoring system (XtoJ) to safely assist programmers migrate existing AJAX-based applications utilizing XML into functionally equivalent AJAX-based applications utilizing JSON. We empirically demonstrate that our transformation system significantly improves the efficiency of AJAX applications.

1. Introduction

Rich Internet Application (RIA) technologies provide richer, faster and more interactive experiences by updating data without reloading the entire page. They break the old click-and-wait user experience mode and make web applications behave more like desktop applications (Paulson, 2005). AJAX (Asynchronous JavaScript and XML) is one of the most popular RIA technologies. Examples of AJAX applications include: Google Maps,[1] Gmail,[2] Google Suggest,[2] and Facebook.[3] AJAX applications are designed to be more responsive which implies that the waiting time for users is significantly reduced. Card's research (Card et al., 1991) demonstrates that for a system to be seen as responsive, it has to respond rate of 0.1 s or less.

To achieve a more responsive user experience, data transmission rates and performance (on both the client and server) characteristics for AJAX applications are quite crucial. Though XML is the standard way to exchange data, the X in AJAX standing for XML, it does have significant performance overheads. Hence, replacing it by JSON which is a lightweight data exchange format has the promise of producing faster applications. Changing data format from using XML to JSON for AJAX applications can improve the efficiency of an AJAX application (up to one hundred times faster) and enhance user experience by reducing the network transfer time and client JavaScript processing time. Nurseitov et al. (2009) have demonstrated this possibility by manually transforming a small number of legacy AJAX applications into (technically) non-AJAX applications (replacing XML by JSON), where the functionality is preserved but the performance has been enhanced.

One way to implement this transformation is through refactoring – a process of restructuring code without changing its behavior to improve code quality (Fowler, 1999). Refactoring allows a small behavior preserving change to be made to the source code. Once all the refactorings are applied, a significant structural change to the code can be seen. Clear, cleaner, simpler and more reusable code is the main benefits of refactoring (Thompson and Reinke, 2002). Unlike traditional refactoring which is usually for readability, extendibility and maintainability purposes; our refactoring process is for efficiency purpose. In addition, our transformation process is more likely to be used to transform pre-existing legacy AJAX systems rather than transforming their own code as the write. Programmers can use our proposed refactoring technique to transform existing XML-based AJAX web applications into JSON-based AJAX web applications to increase the efficiency of their applications. To avoid tedious, error-prone and omission-prone manual refactoring (Dig et al., 2009), in this paper, we introduce a refactoring tool, XtoJ to aid with the transformation process. Specifically, the system assists programmers who maintain, or adapt, existing RIAs with the refactoring of systems that are still using XML data

* Corresponding author.
E-mail address: mying@ualberta.ca (M. Ying).

[1] http://maps.google.ca/.
[2] http://www.google.com/.
[3] http://www.facebook.com/.

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.07.019

structures into JSON data structures, which are known to be more efficient.

Although JSON has been gaining in popularity with new AJAX-style applications, XML still remains very common. According to ProgrammableWeb.com,[4] which is a website specializing in the categorization of Web APIs, there are 58% more XML-based Web APIs than JSON-based Web APIs as of August 2011. Additionally, many popular Web APIs from corporations such as Yahoo and Google initially only offered XML-based APIs before releasing JSON-based APIs at a much later date. Hence, legacy AJAX web applications developed using XML-based APIs can now be refactored to take advantage of the available JSON-based APIs. As the popularity of JSON-based APIs increases, it is important for programmers to refactor their AJAX web applications from XML-based APIs because Web 2.0 websites may stop supporting XML-based APIs in the future. For example, Twitter stopped supporting XML-based Streaming API as of December 2010 (Irani, 2010).

This paper makes the following contributions.

(1) It introduces a refactoring approach to convert legacy XML-based AJAX applications to JSON-based AJAX applications. This approach provides programmers with a structured method to transform their AJAX applications. This transformation technically breaks the traditional AJAX paradigm, by removing XML; however, it achieves this without changing the functionality. It is believed that this process is the first structured process to allow programmers to safely update their AJAX legacy code without requiring them to learn about alternative data exchange formats, in this case JSON.
(2) A proof of concept tool called XtoJ is produced to demonstrate that the transformation can be automated thus saving developers valuable time and ensuring a defect-free transformation. XtoJ includes an XML to JSON Converter, a JavaScript Code Transformer and a JavaScript Code Generator.
(3) It empirically demonstrates that JSON-based AJAX applications are more efficient than XML-based AJAX applications; and that programmers can use the introduced method to rapidly access this efficiency gain.

The remainder of the paper is as follows: Section 2 introduces RIA technologies, specifically AJAX, XML and JSON. Section 3 presents our refactoring approach to transform XML-based AJAX applications to JSON-based AJAX applications. Section 4 describes the three components of our refactoring system. Section 5 evaluates the performance of our refactoring system by utilizing it to refactor a number of real-world applications; Section 6 introduces related work and Section 7 concludes the paper.

2. Background

2.1. AJAX

AJAX (Asynchronous JavaScript and XML) is a set of web development technologies to make web applications more interactive and dynamic. AJAX improves the user experience by updating the content within a web application without reloading the entire page (Paulson, 2005).

AJAX combines the following technologies (Garrett, 2005):

(1) standard-based presentation using XHTML and CSS;
(2) dynamic display and interaction using DOM trees;
(3) data interchange and manipulation using XML (or any other data interchange standards);
(4) asynchronous data retrieval using XMLHttpRequest; and
(5) JavaScript binding everything together.

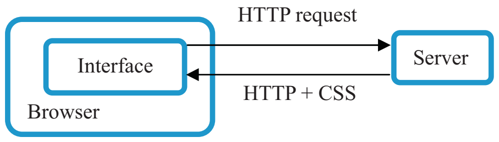


Fig. 1. Classic web application model.

In the classic web application model (Fig. 1), users on the browser (client) side trigger an HTTP request to the server side. The server side deals with the request and returns the updated page to the user. Within the AJAX web application model (Fig. 2), an AJAX engine runs within the browser, communicates with the server, performs interactions, and displays requested information in the browser. If the AJAX engine requires more data, it sends requests asynchronously to the server in the background to retrieve updated data (and potentially additional code) without interfering with the users' interaction with the application (Zakas et al., 2006).

Four aspects influence the performance of an AJAX application:

(1) Network transfer time: Network transfer time depends upon the amount of data to be transferred, and the available bandwidth. This latter factor is clearly outside of the programmer's control.
(2) Network latency: Network latency is a combination of:
    - The average delay of a zero-byte transfer from the client to the server or visa versa.
    - The number of transfers required to complete the request/response cycle. The number of transfers required is dependent on a number of factors including the version of HTTP(S) that is utilized.
(3) Server processing time: The server processing time reflects the server's capability in handling requests and processing application logic. Data processing time on the server has great influence on the server processing time.
(4) Client processing time: JavaScript processing time on the client is also crucial for AJAX applications, because AJAX technology relies upon a JavaScript-based AJAX engine to interact with the browser. See Fig. 2.

Although AJAX is used to perform partial updates of the user interface, a partial update does not mean that only a small amount of data is retrieved. In addition, a partial update does not imply that only a small section of the application is updated. For example, similar to Microsoft Excel, online spreadsheet applications such as EditGrid,[5] Google Spreadsheets[6] and Zoho Sheet[7] require most of the user's browser window to be updated with hundreds of cells (objects or properties) when the user switches between workbooks. Online mapping applications such as Google Maps and Yahoo Maps[8] also require that the majority of the user's browser window be updated each time the user pans, zooms, or performs any other activity. Other AJAX-enabled websites such as Facebook (switching between most pages is done using AJAX); Dell's Online Store[9] (customization of the entire computer system is done

4 http://www.programmableweb.com/.

5 http://www.editgrid.com/.
6 http://www.google.com/.
7 http://www.zoho.com/.
8 http://ca.maps.yahoo.com/.
9 http://www.dell.com/.

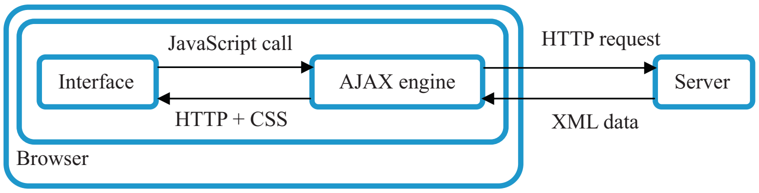


Fig. 2. AJAX web application model.

```
<movie>
   <title>A</title>
   <rating>6.5</rating>
</movie>
```

Fig. 3. An example of XML data structure.

```
{
 "movie":{
     "title": "A",
     "rating":"6.5"
  }
}
```

Fig. 4. An example of JSON data structure.

through AJAX-enabled technology); and Gmail (most navigation within Gmail is via AJAX) require hundreds of objects or properties be loaded as quickly as possible to provide users with an uninterrupted user experience similar to that achieved in desktop applications. As more companies start transforming existing traditional web applications into AJAX-enabled applications and as AJAX applications grow in complexity, the volume of AJAX applications that will request a large number of objects will also grow.

2.2. XML

Typically, the format for retrieving AJAX data is XML, as the "X" in the name of AJAX indicates. XML, an acronym for Extensible Markup Language, is a subset of the Standard Generalized Markup Language (SGML). XML describes self-describing data in standardized ways, and allows user-defined document markups to be created and formatted (Allen et al., 2008). The primary uses for XML are data interchange and storage in web environments. XML represents data using a hierarchical structure. Fig. 3 shows a simple example of an XML data structure.[10]

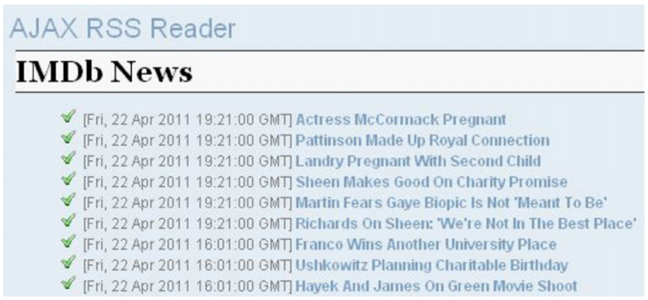
The advantages of XML include flexibility and readability. However, XML is not optimal for data interchange between machines because it is overly verbose.

2.3. JSON

An alternative data format to XML is JSON; JSON is short for JavaScript Object Notation. JSON is a lightweight data interchange format based on a subset of the array and object literal notations of JavaScript (Standard ECMA-262). Fig. 4 shows an example of a JSON data structure, which can be considered equivalent to the previous XML data structure (in Fig. 3).

JSON has the advantage of being compact and directly supported by JavaScript. The biggest disadvantage of JSON is that the format is not very readable for humans.

The process for transmitting JSON data between the browser and server is as follows (Webucator, 2009):


AJAX RSS Reader

IMDb News

✓ [Fri, 22 Apr 2011 19:21:00 GMT] Actress McCormack Pregnant
✓ [Fri, 22 Apr 2011 19:21:00 GMT] Pattinson Made Up Royal Connection
✓ [Fri, 22 Apr 2011 19:21:00 GMT] Landry Pregnant With Second Child
✓ [Fri, 22 Apr 2011 19:21:00 GMT] Sheen Makes Good On Charity Promise
✓ [Fri, 22 Apr 2011 19:21:00 GMT] Martin Fears Gaye Biopic Is Not 'Meant To Be'
✓ [Fri, 22 Apr 2011 19:21:00 GMT] Richards On Sheen: 'We're Not In The Best Place'
✓ [Fri, 22 Apr 2011 16:01:00 GMT] Franco Wins Another University Place
✓ [Fri, 22 Apr 2011 16:01:00 GMT] Ushkowitz Planning Charitable Birthday
✓ [Fri, 22 Apr 2011 16:01:00 GMT] Hayek And James On Green Movie Shoot

Fig. 5. The AJAX RSS Reader webpage.

(1) On the client side: The client creates a JavaScript object and then serializes the JavaScript data structures into JSON text by using JSON stringifier[11] (for JavaScript). After that, the client uses GET or POST methods to trigger an HTTP request, which contains the encoded JSON string.
(2) On the server side: After receiving the request, the server deserializes the JSON string into an object by using a JSON parser for the language used by the server. For example, Argo[12] is a JSON parser for the Java programming language. Subsequently, the server manipulates this object for different purposes.

3. Refactoring XML into JSON in AJAX applications

In Section 3.1, we provide an example of an XML-based AJAX application which will be used to demonstrate our refactoring process. We will then discuss the performance comparison between utilizing XML and JSON in Section 3.2. Finally, Section 3.3 describes our approach to implement the transformation from XML-based AJAX applications to JSON-based AJAX applications.

3.1. Example

We use the AJAX RSS Reader[13] and the RSS XML file from IMBb[14] as an example. Fig. 5 shows the webpage of the AJAX RSS Reader (index.html), it displays the information of the channel and each feed in the channel.

The structure of the RSS XML file is shown in Fig. 6.

Fig. 7 shows the JavaScript code used to retrieve data from the RSS XML file. In Fig. 7, the responseXML property of the XML-HttpRequest object (RSSRequestObject) gets the response (XML) from a server and returns an XML DOM object (node). To get the value of the node, two XML DOM properties: firstChild (returns the first child of a node) and data (returns the value of a node), and one XML DOM method: getElementsByTagName(name) (returns all elements by a tag name) are used.

10 http://www.w3schools.com.

11 http://www.json.org/json2.js.
12 http://argo.sourceforge.net/.
13 http://AJAX.phpmagazine.net/2005/11/AJAX_rss_reader_step_by_step_t.html.
14 http://www.imdb.com/.

```
<channel>
      <title>IMDb News</title>
      <link>http://www.imdb.com/rg/rss/news-channel/news/</link>
      <description>IMDb News</description>
      <language>en</language>
      <copyright>Copyright (C) 2011 IMDb.com, Inc.
              http://www.imdb.com/conditions</copyright>
      <image>
          <title>IMDb News</title>
          <url>http://i.imdb.com/logo.gif</url>
          <link>http://www.imdb.com/rg/rss/news-channel/news/</link>
      </image>
      <item>
          <title>Actress McCormack Pregnant</title>
          <pubDate>Fri, 22 Apr 2011 19:21:00 GMT</pubDate>
          <link>http://www.imdb.com/rg/rss/news/news/ni9872103/</link>
      </item>
      <item>
          <title>Pattinson Made Up Royal Connection</title>
          <pubDate>Fri, 22 Apr 2011 19:21:00 GMT</pubDate>
          <link>http://www.imdb.com/rg/rss/news/news/ni9872102/</link>
      </item>
    ...
</channel>
```

Fig. 6. The structure of the RSS XML file.

```
var node = RSSRequestObject.responseXML;
var channel = node.getElementsByTagName('channel').item(0);
var title = channel.getElementsByTagName('title').item(0).firstChild.data;
var link = channel.getElementsByTagName('link').item(0).firstChild.data;

content = '<div class="channeltitle"><a href="'+link+'">'+title+'</a></div><ul>';

var items = channel.getElementsByTagName('item');
for (var n=0; n < items.length; n++) {
   var itemTitle = items[n].getElementsByTagName('title').item(0).firstChild.data;
   var itemLink = items[n].getElementsByTagName('link').item(0).firstChild.data;
   try {
      var itemPubDate = '<font color=gray>['+items[n].getElementsByTagName('pubDate').
                        item(0).firstChild.data+'] ';
   } catch (e) {
      var itemPubDate = '';
   }
   content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
}
```

Fig. 7. JavaScript code for XML-based AJAX RSS Reader.

3.2. Performance comparison

The performance differences between utilizing XML and JSON is obvious, Nurseitov et al. (2009) designed and implemented scenarios to measure and compare the transmission time and resource utilizations between JSON and XML. They utilized massive data sets during this exploration. In the first scenario, a client sends 1,000,000 objects to a server using XML and JSON encoding; and in the second scenario, a client sends a series of smaller number of objects (20,000, 40,000, 60,000, 80,000 and 100,000) to a server using XML and JSON encoding separately.

Table 1 describes the performance differences between JSON and XML for these two scenarios. To compare their performances, the relative mean difference (RMD) of the transmission time of accessing XML data and JSON data is calculated as:

$$\frac{(\text{Response Time for XML} - \text{Response Time for JSON})}{(\text{Response Time for XML} + \text{Response Time for JSON})/2} \qquad (1)$$

It can be seen from the table that sending JSON encoded data is much faster than sending XML encoded data. This is because JSON is more compact than XML, which results in smaller documents. Thus, less bandwidth will be consumed and the data transmission between the client and server side will be significantly faster (Nurseitov et al., 2009). On the other hand, XML documents are usually accessed and manipulated by constructing an XML DOM (Jacobs, 2006). According to the definition by W3C (World Wide Web Consortium), "The W3C Document Object Model (DOM)[15] is a platform and language-neutral interface (API) that allows programs and scripts to dynamically access and update the content, structure, and style of a document". To facility the node navigation, a DOM-based parser reads the entire XML document and transforms the XML document or XML string into an XML DOM object (a tree structure) in memory. When parsing large XML documents, it can be slow and resource-intensive (Jacobs, 2006).

However, JSON is a subset of JavaScript, retrieving data from a JSON object is "identical" to retrieving information from another JavaScript object. Hence, processing JSON encoded data is much faster than processing XML encoded data, which makes the browser's response faster.

Therefore, based on the testing results in Table 1, we can see that a strong argument exists: AJAX applications requiring high performance should utilize JSON rather than XML, because JSON improves the efficiency of an AJAX application by reducing network transfer time and client processing time.

[15] http://www.w3.org/DOM/.

Table 1
Results for different scenarios.

Scenario	Measurement	XML (ms)	JSON (ms)	RMD
Scenario 1	Total time	4546694.78	78257.9	1.93
	Average time per object	4.55	0.08	
Scenario 2	Total time for 20,000 objects	61333.68	2213.15	1.86
	Average time per object	3.07	0.11	
	Total time for 40,000 objects	123854.59	3127.99	1.90
	Average time per object	3.10	0.08	
	Total time for 60,000 objects	185936.27	4552.38	1.90
	Average time per object	3.10	0.08	
	Total time for 80,000 objects	247639.81	6006.72	1.90
	Average time per object	3.10	0.08	
	Total time for 100,000 objects	310017.47	7497.36	1.91
	Average time per object	3.10	0.07	

3.3. *Methodology*

The performance gains for AJAX applications described by Nurseitov et al. (2009) were made by human experts manually undertaking a transformation from using XML data structure to JSON data structure. Thus, to automatically implement such transformation, we produce a tool, called XtoJ to transform XML-based AJAX applications (such as AJAX RSS Reader) to JSON-based AJAX applications. However, transforming existing applications is not straightforward, for an existing AJAX application (a common form of RIAs) changing the data format involves two procedures:

(1) Converting the data from XML to JSON.
(2) Changing the JavaScript code – from the code which manipulates the XML data to the code which manipulates the JSON data.

However, programmers knowledgeable about XML may know nothing about JSON. In addition, programmers who implement the transformation may not be the original authors of the system; therefore, they require an understanding of all the interactions between the JavaScript code and the XML data. Such a process can clearly introduce defects; thus, we have designed a refactoring tool, XtoJ, to assist with the transformation of AJAX applications from utilizing XML to utilizing JSON.

We used refactoring to implement this transformation. Refactoring can be performed using two automated approaches (Mens and Tourwé, 2004), either semi-automatic (which requires the user's participation) or fully automatic (which does not require user's interaction). Our system is a semi-automated system as the transformation will often encompass functionality which cannot be automatically inferred by a program.

Our refactoring process contains three principle steps:

(1) XML to JSON conversion. Converting an XML file into a JSON file is the first step in our refactoring process; the system is capable of achieving this conversion automatically. This transformation requires no "judgments calls" and is completely safe; hence, a completely automated approach is the best option.
(2) JavaScript code transformation. After the XML file is converted into JSON, JavaScript code transformation can be automatically completed to transfer the JavaScript code which accesses the XML data to the functionally equivalent JavaScript code which processes the semantically identical (to the XML) JSON data. This step is fully automated.
(3) JavaScript code generation. The JavaScript code transformation only can convert the already existing JavaScript code which manipulates the existing XML data to manipulate the newly created JSON data. If more functionality is required, JavaScript code generation is invoked, in a semi-automated fashion, to produce the JavaScript code skeletons in accordance with users' requirements and the user is asked to provide the "bodies" for these skeletons. *Note*: As nothing is known about this new additional functionality, it is clearly impossible for the system to produce these bodies. In addition, as the user is only working with automatically produced skeletons, which they can rename (traditional refactoring pattern "change name"), we believe that the automatically produced code does not introduce a significant readability issue into the update process.

4. System components

Our system has three components to perform the transformation; we now discuss each of them in detail.

4.1. *XML to JSON Converter*

XML to JSON Converter converts XML files into JSON files automatically. It is not necessary for programmers to know the syntax of XML or JSON, and the transformation rules between XML and JSON. We extended the grammar of JsonXml.js[16] to implement such transformations automatically.

Though XML and JSON have different syntax and structures, there are seven basic options for converting XML into JSON (in Appendix A). As these seven options provide complete coverage, the conversion between languages is automatic and does not require user intervention, unless the user elects to change naming conversions. For each option, an XML version of the code and the corresponding JSON version of code are given, as well as the methods required to access the JSON version of the code as an example. JavaScript provides an eval( ) function to convert a JSON string into an object. However, it is not secure since it can execute any piece of JavaScript code including malicious scripts. Hence, unless the client trusts the source of the data, it is advisable to use a JSON parser instead. We use a JSON parser[17] to recognize and accept only valid JSON strings, stopping malicious scripts from executing. Finally, grammatical statements covering the transformational options are provided as general statements of the cases each transformational situation covers. The grammatical statements are EBNF (Extended Backus–Naur Form) (Attenborough, 2003) for XML 1.0 (W3C, 2008), namespace in XML 1.0 (W3C, 2009) and JSON.[18] To simplify the grammar, we only include directly utilized symbols and rules in the grammars.

[16] http://michael.hinnerup.net/blog/2008/01/26/converting-json-to-xml-and-xml-to-json/.
[17] https://github.com/douglascrockford/JSON-js/blob/master/json2.js.
[18] http://www.json.org/.

4.1.1. Example of system in operation

We use the XML to JSON Converter to convert the RSS XML file in Section 3.1 to the RSS JSON file (rss.js). The structure of the created JSON file is shown in Fig. 8.

4.2. JavaScript Code Transformer

The JavaScript Code Transformer automatically transforms JavaScript code used to access XML data into the JSON equivalent. It does this without requiring the programmers to have knowledge of how to access XML or JSON data. Despite the changes to the data formats accessed, both the original code and the transformed version also require the same functionality.

To ensure the safety of the refactorings, the pre- and post-conditions assertions (Opdyke, 1992; Roberts, 1999) are established. Pre- and post-conditions are what required to be satisfied before and after refactoring has been performed. The JavaScript Code Transformer is able to check the following pre- and post-conditions:

Pre-conditions:

(1) The syntax of the XML documents and the JavaScript code (DOM APIs) used to access the XML nodes are functionally correct.
(2) The hierarchical depth of the XML documents is less than five. Based on our experience with XML documents in AJAX applications, this level of depth is sufficient to cover many existing XML documents. However, the tool can be easily modified to accommodate XML documents at greater depths if this is required.

Post-condition: The syntax of the converted JSON files and the JavaScript code used to access those JSON files are functionally correct.

Unit testing (Coelho et al., 2006) is essential for refactoring; it is adopted to ensure the behavior of a program before and after the refactoring is preserved. A unit is the smallest piece of a program that is testable, such as a method. Hence, refactoring – incorporating unit testing – is designed to assist the programmer with the transformation, while providing safeguards to ensure that defects are not introduced during the transformational process. RhinoUnit,[19] a framework for performing unit testing of JavaScript programs is used to ensure that the refactoring preserves the existing behavior of the program.

We used Another Tool for Language Recognition (ANTLR),[20] a recursive parser generator for constructing interpreters, translators, compilers and recognizers to implement the JavaScript Code Transformer. Our refactoring cycle contains several steps: (1) the input of the Lexer is a character stream; (2) the Lexer converts a character stream into a token stream with vocabulary symbols. (3) Subsequently, the Parser generates an abstract syntax tree (AST) from the token steam; and (4) the Tree Parser processes the AST. Followed by (5), when a bad smell is detected, the Tree Parser rewrites the code according to rewriting rules and generates output. And finally (6), the Tree Parser continues to process the other bad smells until all the bad smells are considered. Fig. 9 shows the translation data flow for ANTLR (Parr, 2007).

The JavaScript Code Transformer adopts a fully automated approach to detect bad smells and rewrite the code.

(1) Bad smells detection

For traditional refactoring, bad smells are blocks of source code with bad designs in the existing software, which offers opportunities to perform refactoring (Fowler, 1999; Srivisut and Muenchaisri, 2007). The bad smells in our refactoring system are the inefficient set of XML DOM APIs used by JavaScript to manipulate XML nodes and attributes. XML DOM APIs includes: XML DOM properties (such as x.firstChild – x stands for any node), XML DOM methods (such as x.getElementsByTagName("tagName")) and XML attribute node methods (such as x.getAttribute(name)). The XML DOM API allows a variable to be used as a parameter for the methods to retrieve nodes and attributes; the bad smell detection phase can detect these statements to allow the code rewriting phase to successfully transform them into semantically equivalent JSON statements.

The XML DOM APIs allow many different approaches to access nodes or attributes. For example, there are two approaches to access XML nodes, using the x.getElementsByTagName(tagName) method which returns all nodes with a specified tag name or using XML DOM properties (such as x.childNodes, x.lastChild and x. nextSibling) to traverse the tree of the XML document. For example, the node "channel" in the rss.xml can be accessed by:

```
var node = RSSRequestObject.responseXML;
var channel = node.getElementsByTagName('channel').item(0);
or
var node = RSSRequestObject.responseXML;
var channel = node.documentElement.childNodes[0];
```

Additionally, three approaches can be used to access XML attributes, using the x.attributes property and x.getAttribute(name) or x.getAttributeNode(name) methods.

Since there is more than one approach to access XML nodes and attributes, programmers can use many different combinations of approaches to retrieve XML nodes or attributes they want to access. The bad smell detection phase can detect all of these different combinations.

(2) Code rewriting

Unlike XML, JSON allows objects to contain other objects or arrays. Arrays can also contain other objects or arrays. JSON structure is accessed through dot or subscript operators from object to the member of the object to be retrieved. The name of the object or array is required to access its members. For example, the node "channel" in the rss.js can be accessed by:

```
var jsonObject = JSON.parse(RSSRequestObject.responseText);
var channel = jsonOjbect.channel;
```

Thus, the code rewriting phase transforms the bad smells – combination of XML DOM APIs that programmers use to access XML nodes or attributes – to the JavaScript statements used to access JSON structure.

When a bad smell is detected, the JavaScript Code Transformer performs the following steps to rewrite code statements.

Step 1: Retrieve all the XML DOM APIs to determine the relationship between the nodes and the location of the XML node that is being accessed within the XML DOM.
Step 2: Based on the location determined in Step 1, the JavaScript Code Transformer traverses the XML document to retrieve the name of the node being accessed and the name of all the parent nodes of that node.
Step 3: Produce JavaScript statement(s) to access the node in JSON format based upon the information obtained in Step 2.

4.2.1. Grammar for the JavaScript Code Transformer

In this section, we provide the EBNF for the JavaScript which accesses the XML and the JSON data respectively. The grammar for accessing XML nodes and XML attributes are different, thus, we show them separately. We extended the grammar from

[19] http://code.google.com/p/rhinounit/.
[20] http://www.antlr.org/.

JavaScript.g[21] and again, to make the grammar simpler, we only include the directly utilized symbols and rules.

(1) EBNF for JavaScript to access XML nodes:

```
statement::= forStatement | variableStatement
forStatement::= 'for(' forInStatementInitialiserPart  ';' forControl ';'
expression ')' statement
forControl::=   Identifier   '<'  XMLHttpRequestName  '.'  responseXML   '.'
(documentElement  '.')?  (((getElementsByTagName  '("'  Identifier  '")'  ('['
NumericLiteral ']'| item '(' NumericLiteral  ')')  '.' )* getElementsByTagName
'("'  Identifier  '")')  |  ((((childNodes  ('['  NumericLiteral  ']'|  item  '('
NumericLiteral ')')) | firstChild) '.')* childNodes)) '.' length
variableStatement::= 'var' LT* variableDeclarationList (LT | ';')!
variableDeclarationList::= variableDeclaration (LT* ',' LT*
variableDeclaration)*
variableDeclaration::=Identifier LT* initialiser?
Initialiser::= '=' LT* XMLhttprequestName'.'responseXML '.' (documentElement
'.') ? (getElementsByTagName '("' Identifier '")' ('[' (NumericLiteral |
Identifier) ']'| item '(' (NumericLiteral | Identifier) ')')  '.' )*
(((childNodes ('[' (NumericLiteral | Identifier) ']'| item '(' (NumericLiteral
| Identifier) ')')) | firstChild) '.')+ (nodeValue | data)
```

(2) EBNF for JavaScript to access XML attributes: All the grammar rules for accessing XML attributes are the same as accessing XML nodes except the rule "Initialiser", shown as follows:

```
Initialiser::= '=' LT* XMLhttprequestName'.'responseXML '.' (documentElement
'.')? (getElementsByTagName '("' Identifier '")' ('[' (NumericLiteral |
Identifier) ']'| item '(' (NumericLiteral | Identifier) ')')  '.' )+ |
(((childNodes ('[' (NumericLiteral | Identifier) ']'| item '(' (NumericLiteral
| Identifier) ')')) | firstChild) '.')+ (((getAttributeNode '("' Identifier
'")')| (attributes('[' (NumericLiteral | Identifier) ']'| item '('
(NumericLiteral | Identifier) ')')) | (attributes '.' getNamedItem '("'
Identifier '")')) '.' nodeValue ) | (getAttribute '("' Identifier '")')
```

(3) EBNF for JavaScript to access JSON:

```
statement::= forStatement | variableStatement
forStatement::= 'for(' forInStatementInitialiserPart  ';' forControl ';'
expression ')' statement
forControl::=  Identifier  '<' XMLHttpRequestName '.' responseText  '.'
(Identifier '.')+ length
variableStatement::= 'var' LT* variableDeclarationList (LT | ';')!
variableDeclarationList::= variableDeclaration (LT* ',' LT*
variableDeclaration)*
variableDeclaration::=Identifier LT* initialiser?
Initialiser::= '=' LT* JSON '.' parse ( XMLHttpRequestName '.' responseText )
'.' (Identifier | Identifier '[' NumericLiteral |('"' (#Text | ('@' Identifier)
| #cdata | (Identifier ':' Identifier)) '"')']')+
```

4.2.2. Example of system in operation

Again, we use the example in Section 3.1 to illustrate how the JavaScript Code Transformer works. After an XML file is successfully converted into a JSON file by the XML to JSON Converter, the JavaScript Code Transformer takes an HTML or JavaScript file that retrieves the data from the XML file as input and outputs an HTML or JavaScript file that retrieves data from the JSON file. The process of the transformation is as follows:

Source code preparation: The transformation is on JavaScript code, thus, if the input file is HTML, all the HTML elements are removed. However, the HTML elements embedded in the JavaScript snippet will stay the same. In the example, the input file is an HTML file.

Copy propagation transformation: This transformation "eliminates cases in which values are copied from one location or variable to another" (Hagen, 2006). To prepare for the transformation, statements that are used to retrieve the node properties are combined with statements that are used to access the value of the node. For example, the variable "node" and "channel" are eliminated in the following statements:

```
var node =        RSSRequestObject.responseXML;
var channel =     node.getElementsByTagName('channel').item(0);
var channel =     node.getElementsByTagName('channel').item(0);
                  firstChild.data;
```

and are changed to:

```
var titleRSSRequestObject.responseXML.
        getElementsByTagName('channel').item(0).
        getElementsByTagName('title').item(0).firstChild.data;
```

JavaScript code transformation: The JavaScript parser generated by ANTLR is used to parse the JavaScript code for accessing the XML files. If a bad smell (the combination of XML DOM APIs that programmers use to access XML nodes or attributes) is found; the system rewrites the code according to the transformation rules. Fig. 10 shows the JavaScript code to process the JSON file after refactoring.

The JavaScript code transformation is comprise of three steps:

Step 1: Change responseXML into responseText: in the JavaScript code for JSON (Fig. 10), the responseXML property of the XMLHttpRequest object (RSSRequestObject) is changed to the

[21] http://www.antlr.org/grammar/1206736738015/JavaScript.g.

```
{
 "channel":{
      "title":"http://www.imdb.com/rg/rss/news/news/ni9872099/fff",
      "link":"http://www.imdb.com/rg/rss/news-channel/news/",
      "description":"IMDb News",
      "language":"en",
      "copyright":"Copyright (C) 2011 IMDb.com, Inc. http://www.imdb.com/conditions",
      "image":{
            "title":"IMDb News",
            "url":"http://i.imdb.com/logo.gif",
            "link":"http://www.imdb.com/rg/rss/news-channel/news/"
      },

      "item":[{
             "title":"Actress McCormack Pregnant",
             "pubDate":"Fri, 22 Apr 2011 19:21:00 GMT",
             "link":"http://www.imdb.com/rg/rss/news/news/ni9872103/"
               },
             {
             "title":"Pattinson Made Up Royal Connection",
             "pubDate":"Fri, 22 Apr 2011 19:21:00 GMT",
             "link":"http://www.imdb.com/rg/rss/news/news/ni9872102/"
              },
              …
             ]
      }
}
```

Fig. 8. The converted RSS JSON file.

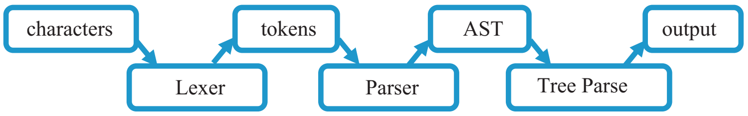


Fig. 9. Translation data flow.

```
var jsonObject = JSON.parse(RSSRequestObject.responseText);
var title = json.channel.title;
var link = json.channel.link;

content = '<div class="channeltitle"><a href="'+link+'">'+title+'</a></div><ul>';

if(typeof(json.channel.item[0]) == 'undefined'){
   var itemTitle = json.channel.item.title;
   var itemLink = json.channel.item.link;
   try{
      var itemPubDate = '<font color=gray>['+json.channel.item.pubDate+']';
   }catch (e){
       var itemPubDate = '';
   }
   content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
   }
else{
   for(var n=0; n < json.channel.item.length; n++){
      var itemTitle = json.channel.item[n].title;
      var itemLink = json.channel.item[n].link;
      try{
          var itemPubDate = '<font color=gray>['+json.channel.item[n].pubDate+'] ';
      }catch (e){
          var itemPubDate = '';
      }
      content += '<li>'+itemPubDate+'</font><a href="'+itemLink+'">'+itemTitle+'</a></li>';
   }
}
```

Fig. 10. JavaScript code for accessing the JSON file after the transformation.

responseText property, which gets the response (non-XML) and returns a string (JSON).

Step 2: Changing the JSON string into an object: the JSON parser[22] is used to change the JSON string into an object (jsonObject).

Step 3: Changing value accesses: JavaScript code that accesses the value of the nodes <title>, <link> and <item> (including child nodes <title>, <link> and <pubDate>) in the RSS XML file are transformed to access the corresponding members of the object in the RSS JSON file for this example. The <item> node is converted to an object containing three members “title”, “link” and “pubDate”. If the “item” object is an array, an alternative mechanism is required to access its members. Thus, the value of typeof(json.channel.item[0]) is

[22] https://github.com/douglascrockford/JSON-js/blob/master/json2.js.

used to check whether the “item” object is an array. The value equals to “undefined” indicates that the “item” object is not an array, its members are accessed without iteration. The value equals to a number indicates that “item” object is an array, iteration is required to access the members.

Cleanup: If the input file is HTML file, the HTML elements (add the corresponding CSS elements, if they exist) are added back and the output file is generated. That is, the transformation process does not affect these components directly. Hence, the final output consists of the original (if existing) HTML and CSS elements and the automatically produced JavaScript code.

4.3. JavaScript Code Generator

The JavaScript Code Generator is an optional component in our transformation system. After refactoring the existing application, the JavaScript Code Generator can be used to generate JavaScript code skeletons to access a JSON file when more functionality is required. Unlike the JavaScript Code Transformer which is fully automated, the JavaScript Code Generator is semi-automated as the user must supply several items of information to allow code constructed to safely processed, specifically, the user:

(1) Selects an XML file to be converted to an JSON file;
(2) Selects groups of data (node values or attributes) required to be accessed;
(3) Provides an explanation of the conditions under which the data can be safely accessed;
(4) Selects the format for output:
  - Print the data;
  - Store the data into an array;
  - Display the data through a form.

Given this information, the system creates the JavaScript code for processing the JSON structures produced in Step 1. The generated code is considered a skeleton of the final program as the system has no way of understanding the domain of the application. The final (manual) step is for the programmers to add any such domain specific components.

For each node selected by the programmers, the JavaScript Code Generator performs the following steps to generate the JavaScript code for accessing the converted JSON file.

Step 1: Retrieve the node selected by the programmers and determine the location of the XML node within the XML DOM.
Step 2: Based upon the location determined in Step 1, the JavaScript Code Generator traverses the XML document to retrieve the name of the node being selected and the name of all the parent nodes of that node.
Step 3: Produce JavaScript statement(s) to access the node in JSON format based upon the information obtained in Step 2.

4.3.1. Patterns for generating JavaScript code

The JavaScript Code Generator creates code for:

(1) Accessing node values or attributes for a single node and
(2) Traversing nodes to get values or attributes for multiple nodes.

The following section discusses different patterns to generate JavaScript code for accessing a JSON file. As in previous sections, we provide the EBNF for XML 1.0 (W3C, 2008) and the EBNF for the generated JavaScript code for each pattern. We only include the directly utilized symbols and rules of the grammar. The following EBNF explicit definitions are required for the generated JavaScript Code:

`JsonObject`: the object name after converting a JSON string to an object;
`nodeMembers`: the names of the node(s) in an XML file;
`attributeMember`: the attribute names in an XML file;
`conditions`: the conditions under which it is safe to access the data.

(1) EBNF for XML 1.0:

```
document::= prolog element Misc*
element::= EmptyElemTag | STag content ETag
STag::='<' Name (S Attribute)* S? '>'
Attribute::= Name Eq AttValue
ETag::='</' Name S? '>'
content::= (element | CharData | Reference | CDSect | PI | Comment)*
```

(2) EBNF for generated JavaScript code to access a single node:

- Accessing the value of a node:

```
output::= JsonObject nodeMembers+
JsonObject::= Identifier
nodeMembers::= (('.' Identifier) |( '.' Identifier '[' Identifier |('"' (#Text
| ('@' Identifier) | #cdata | (Identifier ':' Identifier)) '"')']'));
```

- Accessing an attribute of a node:

```
output::= JsonObject nodeMembers+ '.' attributeMember)
JsonObject::= Identifier
nodeMembers::= (('.' Identifier) |( '.' Identifier '[' Identifier |('"' (#Text
| ('@' Identifier) | #cdata | (Identifier ':' Identifier)) '"')']'))
```

(3) EBNF for generated JavaScript code to traverse nodes:

- Accessing the value of nodes

```
output::= forStatement | ifStatement
forStatement::= 'for(' forInStatementInitialiserPart  ';' forControl ';'
expression ')' statement
statement::= forStatement | ifStatement | accessStatement
ifStatement::= 'if(' ifExpression ')' statement 'else' statement
```

```
forControl::=  Identifier  '<' JsonObject nodeMembers+ '.length'
ifExpression::= ('typeof(' JsonObject nodeMembers+ '[0] == undefined)')
accessStatement::= 'document.write(' JsonObject nodeMembers+ ')'
JsonObject::= Identifier
nodeMembers::=(('.' Identifier) |( '.' Identifier '[' Identifier |('"' (#Text
| ('@' Identifier) | #cdata | (Identifier ':' Identifier)) '"')']'))
```

- Accessing the attributes of nodes

```
output::= forStatement | ifStatement
forStatement::= 'for(' forInStatementInitialiserPart  ';' forControl ';'
expression ')' statement
statement::= forStatement | ifStatement | accessStatement
ifStatement::= 'if(' ifExpression ')' statement 'else' statement
forControl::=  Identifier  '<' JsonObject nodeMembers+ '.length'
ifExpression::= ('typeof(' JsonObject nodeMembers+ '[0] == undefined)') |
conditions
accessStatement::= 'document.write(' JsonObject nodeMembers+ '.'
attributeMember) ')'
conditions::= expression
JsonObject::= Identifier
nodeMembers::=(('.' Identifier) |( '.' Identifier '[' Identifier |('"' (#Text
| ('@' Identifier) | #cdata | (Identifier ':' Identifier)) '"')']'))
attributeMember::= Identifier
```

4.3.2. Example of system in operation

In this section, we reuse the RSS XML file from IMDb[23] to illustrate the operation of the JavaScript Code Generator. There are six steps (five input steps plus the code production step) for programmers to produce JavaScript code as shown in Fig. 11 from scratch.

Step 1: Select the XML file to be converted. The RSS XML file is converted into the RSS JSON file (rss.js) by the XML to JSON Converter;
Step 2: Select the groups of data (node values or attributes) to be accessed. In this example, we have selected to access the values of the nodes <title> and <link> whose parent node is <channel> and all the values of the nodes whose parent node is <item>;
Step 3: The user provides the conditions for accessing the data (for example, only the feeds published in the morning are retrieved). This is optional; hence we have omitted it here for the sake of brevity;
Step 4: Select the format for output. We have chosen to print the retrieved data (by using `document.write`);
Step 5: The system, using these inputs (Steps 1–4), automatically generates JavaScript code for accessing the new JSON object, as shown in Fig. 11.

To explain Step 5 further, we provide a brief overview of the generated code:

Lines 1–7: Creates an XMLHttpRequest object for different browsers:

- if web browsers are Internet Explorer 5 or 6, an XMLHttpRequest object is created by using ActiveX controls.
- if web browsers are Internet Explorer 7 or 8, Mozilla, Chrome, Opera and Safari, an XMLHttpRequest object is created using a native object;

Lines 8 and 28: Sends a request to the server; Line 8 makes a GET request for the URL: “rss.js” and line 28 sends the request to the server using the send ( ) function; see Step 1.

Lines 9 and 10: Handles properties of the XMLHttpRequest; ultimately these lines indicate when the response is completed and all the data has been received;

Line 11: Checks the HTTP status, the request is completed “correctly” when the value is 200;

Line 13: Converts the JSON string (rss.js) to a JSON object named “jsonObject” by using the JSON parser[24];

Lines 14 and 15: Retrieves the members of “title” and “link” in the object “channel” (see Step 2); all the JSON objects are retrieved by the names of their objects; these objects are actually the nodes names or attributes names in the RSS XML file. After retrieval these objects are output via `document.write` (see Step 4).

Line 16: Checks whether the “item” object is an array.

Lines 17–19: If the “item” object is not an array, the members of “title”, “link” and “pubDate” are retrieved.

Lines 21–24: If the “item” object is an array, the code traverses the “item” object and accesses the members of “title”, “link” and “pubDate”, for every object in the “item” object. After retrieval, these objects are output via `document.write` (see Step 4).
Step 6: Add domain specific components. In the JavaScript code in Fig. 11, the variable “content” is created and used to output HTML elements with the value of the nodes retrieved from the RSS JSON file (rss.js). To produce the final program, programmers modify the generated code (Fig. 11) according to the specific requirements. Thus, instead of outputting the different members of the “channel” object and the “item” object, a variable “content” is used to provide the output.

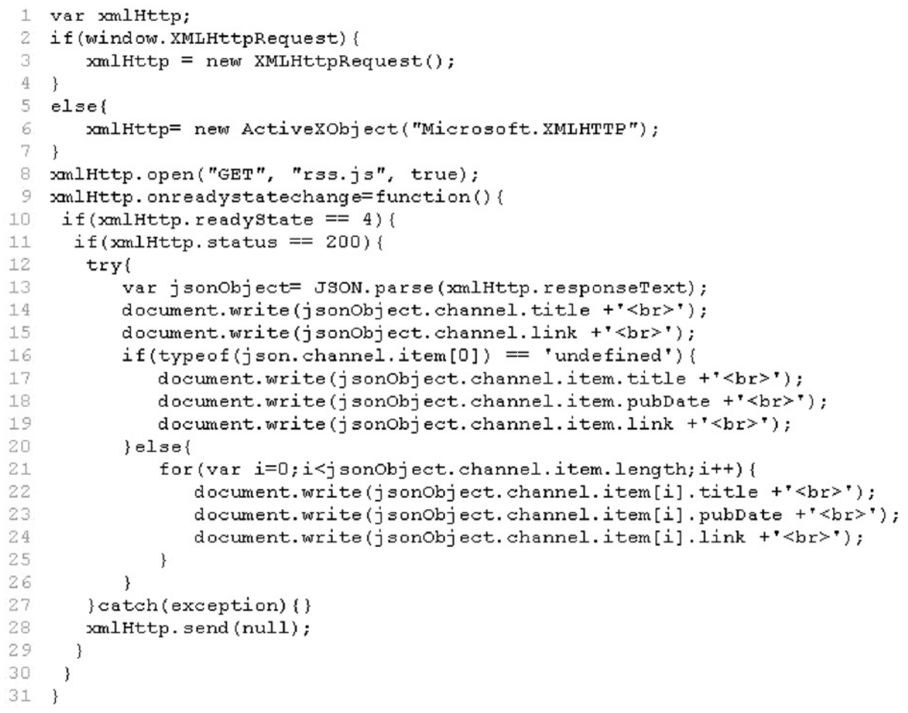
5. Evaluation

In this section, we provide an initial evaluation of the system from both an efficiency viewpoint (a comparison between the runtime characteristics of the original and refactored versions) and a usability viewpoint (can the average I.T. professional utilize our system).

5.1. Performance evaluation

To evaluate our technique, we randomly selected ten XML-based, AJAX applications from the Internet to test the effectiveness of the refactoring system. Because no new functionality is added to the existing ten applications, the JavaScript Code Transformer is used to automatically change the ten applications from XML-based to JSON-based. Subsequently, we tested the “response time” of each AJAX application utilizing XML and utilizing JSON.

[23] http://www.imdb.com/.
[24] https://github.com/douglascrockford/JSON-js/blob/master/json2.js.



```
var xmlHttp;
if(window.XMLHttpRequest){
   xmlHttp = new XMLHttpRequest();
}
else{
   xmlHttp= new ActiveXObject("Microsoft.XMLHTTP");
}
xmlHttp.open("GET", "rss.js", true);
xmlHttp.onreadystatechange=function(){
 if(xmlHttp.readyState == 4){
  if(xmlHttp.status == 200){
   try{
      var jsonObject= JSON.parse(xmlHttp.responseText);
      document.write(jsonObject.channel.title +'<br>');
      document.write(jsonObject.channel.link +'<br>');
      if(typeof(json.channel.item[0]) == 'undefined'){
         document.write(jsonObject.channel.item.title +'<br>');
         document.write(jsonObject.channel.item.pubDate +'<br>');
         document.write(jsonObject.channel.item.link +'<br>');
      }else{
         for(var i=0;i<jsonObject.channel.item.length;i++){
            document.write(jsonObject.channel.item[i].title +'<br>');
            document.write(jsonObject.channel.item[i].pubDate +'<br>');
            document.write(jsonObject.channel.item[i].link +'<br>');
         }
      }
   }catch(exception){}
   xmlHttp.send(null);
  }
 }
}
```

Fig. 11. The generated JavaScript code for accessing the RSS JSON file.

5.1.1. *Methodology*

We used the Client/Server architecture to send XML or JSON data from a server to a client. The process of transferring data is as follows:

(1) After a TCP connection is created, the client creates an XMLHttpRequest object and sends an HTTP request by using the XMLHttpRequest object to the server. In addition, the client specifies what type of data is to be retrieved (XML or JSON).
(2) The server processes the request and creates an HTTP response message. The responseXML and responseText properties of the XMLHttpRequest object are used to retrieve the requested XML or JSON data from the HTTP response on the client side.

We measure the "response time" as an amalgamation of the network transfer time, the network latency time, the server response time and the client processing time. It starts from the time that the XMLHttpRequest object is created on the client side to the time that all the data has been retrieved from the server and processed by the client. Removing components such as (network latency) from our measurement is not possible. However, we can argue that the largest contributor to the execution time is the processing of the data description language components based on the following facts.

(1) When measuring the response time for both XML and JSON data structures, the invariants are the network devices, network interfaces, client machine and server machine.
(2) The variants are the data structures (XML vs JSON), and the code used to process the data structures.
(3) Each test scenario is executed repeatedly and an average is obtained. This average is used for the analysis.

(1) means both XML and JSON data are subjected to the same network environments which means both are exposed to normal variations within the network environments. (3) Further minimizes the impact thanks to the averaging effect. The only remaining factors which have a significant impact on the result are the variants 2. The results from our experience show the response time for JSON is consistently lower than XML which demonstrates the significant impact (2) has.

Tests were executed in the following environment:

- Server: Intel(R) Core (TM) 2 Quad CPU Q6600 @2.4 GHz, with 4 GB of RAM running Microsoft Windows XP Professional, Service Pack 3;
- Client: Intel(R) Core (TM) 2 CPU 6300 @1.86 GHz 1.87 GHz with 2 GB of RAM running Microsoft Windows 7 Professional.

In addition, we used two different browsers: Firefox 9.0 and Internet Explorer 8.0 to ensure that no browser-specific bias was introduced. Finally, each test was executed 100 times to ensure that any extraneous timing issues were minimized.

5.1.2. *Testing results for ten AJAX applications*

To determine the increase in performance, we explored different amounts of objects transferred. As stated above, many large AJAX-based applications regularly transferred more than 100 objects, hence we elected to start our investigation from 100 objects (lower end); and provide values up to 1000 (upper end), which perhaps represents the maximum volume of transfers that are likely to be witnessed from the current generation of AJAX-based applications. Table 2 lists the size information of the XML documents used in our trails.

The size of these documents is in many ways arbitrary, as we know of no reliable information on the (average) size of XML documents in web applications. However, a quick search of the Internet can find significant numbers of applications using XML documents which are several orders of magnitude larger than the sizes used in our trails. Table 3 provides some examples of web applications that utilize "large" XML documents.

For the sake of brevity, Tables 4 and 5 shows the response time (ms) for retrieving the XML and the JSON data from the server for three different amounts of objects (100, 500, and 1000 objects). Again, this task was repeated 100 times allowing calculation of the

Table 2
The XML documents size for the ten tested applications.

	App	100 objects (kB)	500 objects (kB)	1000 objects (kB)
1	w3schools.com[a]	19	94	187
2	ibm.com[b]	7	34	68
3	developer.com[c]	79	391	781
4	captain.at[d]	8	39	78
5	JavaScriptkit.com[e]	117	583	1165
6	Understanding AJAX: using JavaScript to create rich internet applications (Eichorn, 2007)	6	27	53
7	ibm.com[f]	9	43	86
8	sitepoint.com[g]	7	32	63
9	xml.com[h]	8	40	79
10	Brainjar.com[i]	56	277	554

[a] https://www.ibm.com/developerworks/xml/.
[b] http://www.w3schools.com.
[c] http://www.developer.com/.
[d] http://www.captain.at/.
[e] http://www.javascriptkit.com/.
[f] https://github.com/douglascrockford/JSON-js/blob/master/json2.js.
[g] http://sitepoint.com/.
[h] http://www.xml.com/.
[i] http://www.brainjar.com/.

Table 3
The size of some large XML documents (Ng et al., 2006).

Data sets	Size (MB)
XMark (Schmidt et al., 2002)	97
DBLP[a]	42
Shakespeare[b]	7.8
SwissProt[c]	21
TPC-H[d]	34
Weblog[e]	30

[a] http://dblp.uni-trier.de/.
[b] http://www.cs.wisc.edu/niagara/data/shakes/shakspre.htm.
[c] http://www.expasy.ch/sprot/.
[d] http://www.tpc.org/tpch/default.asp.
[e] http://httpd.apache.org/docs/logs.html.

mean and standard deviation for each task. We also performed *t*-test to the response time for retrieving the XML and the JSON data to display the difference between the two groups of data. In addition, Cohen's d value is calculated to measure the strength of the relationship between the response time for retrieving the XML and the JSON data. Values of Cohen's d above 0.8 are normally considered "large"; that is, correspond to a large effect size between the attributes of comparison. In this situation, all tasks result in large differences. However, it is important to exercise caution in interpreting these results as it is known that some aspects of the response timing are out with the control of the experiment; and hence, it is impossible to guarantee that the standard deviation estimation is robust under these circumstances.

Therefore, we believe that the testing results illustrate that our transformation process is successful in refactoring AJAX code for efficiency. These results show the browser version does have an impact on the response time. Firefox 9.0 generally outperforms IE 8 in both XML and JSON versions thanks to a more modern JavaScript

Table 4
Testing results for ten AJAX applications (in Firefox).

	Number of objects	XML (ms)		JSON (ms)		*T*-Test (*P*-value)	Cohen's *d* value
		Mean	Standard deviation	Mean	Standard deviation		
Firefox 9.0							
1	100	5.32	0.62	1.77	0.79	$P<0.01$	5.00
	500	22.37	1.36	6.41	0.99	$P<0.01$	13.42
	1000	43.02	1.25	12.17	1.81	$P<0.01$	19.83
2	100	4.31	0.46	0.97	0.81	$P<0.01$	5.07
	500	18.07	1.07	2.31	1.15	$P<0.01$	14.19
	1000	36.19	1.32	3.90	1.18	$P<0.01$	25.79
3	100	14.52	2.10	5.80	1.61	$P<0.01$	4.66
	500	47.76	4.16	14.35	2.32	$P<0.01$	9.92
	1000	86.53	5.51	20.9	2.25	$P<0.01$	15.59
4	100	2.78	0.46	1.26	0.44	$P<0.01$	3.38
	500	10.81	0.63	3.22	0.52	$P<0.01$	13.14
	1000	21.04	0.76	5.48	0.59	$P<0.01$	22.87
5	100	18.05	0.80	5.63	0.75	$P<0.01$	16.02
	500	87.95	4.08	27.69	1.40	$P<0.01$	19.76
	1000	123.51	4.41	50.12	1.51	$P<0.01$	22.27
6	100	2.37	0.56	0.84	0.47	$P<0.01$	2.96
	500	7.97	0.72	1.97	0.52	$P<0.01$	9.55
	1000	15.05	0.88	3.69	0.86	$P<0.01$	13.06
7	100	2.25	0.59	1.03	0.54	$P<0.01$	2.16
	500	6.53	0.63	3.28	0.59	$P<0.01$	5.33
	1000	11.73	0.72	5.85	0.74	$P<0.01$	8.05
8	100	2.20	0.43	1.05	0.44	$P<0.01$	2.64
	500	7.81	0.65	2.61	0.55	$P<0.01$	8.64
	1000	15.31	0.91	4.81	0.60	$P<0.01$	13.62
9	100	5.19	0.60	0.71	0.52	$P<0.01$	7.98
	500	20.48	0.67	1.93	0.57	$P<0.01$	29.82
	1000	40.53	1.30	3.20	0.80	$P<0.01$	34.59
10	100	22.51	1.47	2.85	0.58	$P<0.01$	17.59
	500	105.29	4.07	13.11	1.63	$P<0.01$	29.73
	1000	212.55	7.58	26.82	1.78	$P<0.01$	33.73

Table 5
Testing results for ten AJAX applications (in IE).

	Number of objects	XML (ms)		JSON (ms)		*T*-Test (*P*-value)	Cohen's *d* value
		Mean	Standard deviation	Mean	Standard deviation		
IE 8.0							
1	100	5.56	0.89	2.23	0.51	$P < 0.01$	4.59
	500	23.80	2.31	13.72	1.94	$P < 0.01$	4.73
	1000	40.37	2.50	27.06	2.90	$P < 0.01$	4.92
2	100	6.71	0.48	2.66	0.48	$P < 0.01$	8.44
	500	19.55	0.69	10.71	0.62	$P < 0.01$	13.48
	1000	36.09	1.07	15.33	0.68	$P < 0.01$	23.16
3	100	10.40	1.15	4.98	0.88	$P < 0.01$	5.29
	500	44.61	3.62	21.45	2.80	$P < 0.01$	7.16
	1000	96.12	6.97	38.37	3.86	$P < 0.01$	10.25
4	100	7.51	0.85	2.85	0.36	$P < 0.01$	7.14
	500	38.47	2.91	5.83	0.40	$P < 0.01$	15.71
	1000	88.83	3.89	10.17	0.45	$P < 0.01$	28.41
5	100	36.07	1.49	10.09	1.44	$P < 0.01$	17.73
	500	240.25	4.06	48.57	1.68	$P < 0.01$	61.69
	1000	576.48	9.92	101.6	3.28	$P < 0.01$	64.28
6	100	4.12	0.36	2.96	0.28	$P < 0.01$	3.60
	500	11.21	0.54	6.20	0.49	$P < 0.01$	9.72
	1000	19.02	1.95	9.99	0.82	$P < 0.01$	6.04
7	100	3.99	0.63	2.70	0.46	$P < 0.01$	2.34
	500	13.28	2.16	8.25	0.48	$P < 0.01$	3.21
	1000	20.53	2.23	13.57	0.83	$P < 0.01$	4.14
8	100	5.65	0.58	3.58	0.59	$P < 0.01$	3.54
	500	17.46	1.33	10.32	1.39	$P < 0.01$	5.25
	1000	30.66	1.56	18.12	2.43	$P < 0.01$	6.14
9	100	12.15	0.67	2.94	0.31	$P < 0.01$	17.64
	500	58.61	2.86	6.11	0.40	$P < 0.01$	25.71
	1000	102.13	3.05	9.83	0.60	$P < 0.01$	41.99
10	100	44.95	4.73	8.46	0.58	$P < 0.01$	10.83
	500	218.72	14.09	34.29	2.93	$P < 0.01$	18.12
	1000	462.17	19.37	65.51	4.39	$P < 0.01$	28.24

engine. However, both browser versions benefit from the switching from XML to JSON.

5.1.3. Variables influencing the response time

During the testing of the ten AJAX applications, we found there are three additional variables that significantly affect the response time for accessing XML and JSON data and thus the efficiency improvement rate of our transformation system. We take the AJAX RSS Reader in Section 3.1 as an example to demonstrate their impacts.

(1) The number of objects and properties

To test the impact of the number of objects and properties, we investigated different amounts of feeds in an RSS file for popular websites. The number of feeds in an RSS file varies from website to website, however, a quick search of the Internet can find many websites using 50 or more feeds in their RSS files. Some websites have fixed number of feeds, such as IMDb[25] (50 feeds), The New York Times-books[26] (50 feeds), investopedia[27] (60 feeds) and TUAW[28] (40 feeds), Engadget[29] (40 feeds). Some websites change the number of feeds every day. For example Gizmodo[30] has up to 100 feeds according to our observations. We tested the XML and JSON version of the AJAX RSS Reader code by retrieving 50, 100, 150 and 200 feeds (objects) from the RSS XML file of IMDb and the (converted) RSS JSON file. For each trial, we measured the response time for the XML version and JSON version of the program, using the methodology described in Section 5.1.2. For the XML version, the response time for accessing different volumes of properties (<title>, <link> and <pubDate>) from the parent node (<item>) is measured. For the JSON version, the response time for accessing the same numbers of properties as the XML version of the code is measured. Table 6 indicates the influence of the different number of objects and properties tested in Firefox 9.0. As more objects and properties are accessed, the response time for both XML and JSON version of the code increase.

(2) The structure of the XML and JSON data

XML documents have a hierarchical structure; accessing different nodes in different hierarchical depths affects the response time. We tested the execution time for accessing the text of the node <title> whose parent node is <channel> and the text of the node <title> whose parent node is the <image> node. These nodes have different depths.

The JSON data is accessed by the dot operator. Accessing an object and accessing the objects within that object results in different response time. We tested the execution time for the JSON version of the code to access the `jsonObject.channel.title` property and the `jsonObject.channel.image.title` property. Each trial was run 10,000 times in Firefox 9.0 to produce a stable mean value. Table 7 clearly shows that for the XML version of the code, accessing nodes in deeper structures increases

[25] http://www.imdb.com/.
[26] http://www.nytimes.com/pages/books/index.html.
[27] http://www.investopedia.com/.
[28] http://www.tuaw.com/.
[29] http://www.engadget.com/.
[30] http://ca.gizmodo.com/.

Table 6
Testing results for different number of objects and properties.

Number of objects	Number of properties	XML (ms)		JSON (ms)		*T*-Test (*P*-value)	Cohen's *d* value
		Mean	Standard deviation	Mean	Standard deviation		
50	1	2.71	0.50	0.86	0.49	$P<0.01$	3.74
	2	3.48	0.76	0.95	0.52	$P<0.01$	3.89
	3	3.91	0.68	0.99	0.44	$P<0.01$	5.10
100	1	4.45	0.59	1.09	0.45	$P<0.01$	6.40
	2	5.38	0.58	1.19	0.53	$P<0.01$	7.54
	3	6.11	0.51	1.25	0.56	$P<0.01$	9.07
150	1	6.34	0.67	1.46	0.52	$P<0.01$	8.14
	2	7.55	0.69	1.53	0.56	$P<0.01$	9.58
	3	8.95	0.64	1.59	0.57	$P<0.01$	12.14
200	1	8.31	0.9	1.76	0.62	$P<0.01$	8.48
	2	10.00	1.02	1.94	0.83	$P<0.01$	8.67
	3	11.7	0.96	2.13	0.79	$P<0.01$	10.89

Table 7
Testing results for accessing different nodes in different structures.

Element	Parent node	Depth	XML (ms)		JSON (ms)		*T*-Test (*P*-value)	Cohen's *d* value
			Mean	Standard deviation	Mean	Standard deviation		
Title	Channel	1	1.90	0.39	0.06	0.14	$P<0.01$	6.28
Title	Image	2	2.94	0.46	0.08	0.22	$P<0.01$	7.93
Attribute	Channel	1	1.77	0.65	0.06	0.14	$P<0.01$	3.64

the response time of the browsers; for the JSON version, accessing an object directly is faster than accessing any objects inside the object.

In addition, accessing the attribute of a node and accessing the text of a node in XML files also leads to different response times. To test the execution time for accessing the attribute of a node, we manually added an attribute to the node <title> whose parent node is <channel>. Subsequently, we tested the execution time for accessing the attribute of the node <title> in the XML file and the property of the object "title" in the JSON file, which is converted from the added attribute in the XML file. Again, each trial was run 10,000 times in Firefox 9.0 to produce stable mean value. As can be seen from Table 7, accessing the attribute of a node is faster than accessing the text of a node for the XML version; however, for the JSON version, accessing the converted attribute takes a similar time (in absolute terms) to accessing other properties.

(3) The length of the text in a node

To analyze the influence of the length of the text in a node, we tested the response time of accessing the text of the node <title> in the XML file and the property of "title" in the JSON file by manually changing the length of the text. We used the same methodology as in Section 5.1.2 and the results can be seen in Table 8. As the text length is increases, the longer it takes for the browser to respond to both the XML and the JSON version of code.

As can be seen from all the results, a number of factors, beyond the number of object retrieved, significantly impact the efficiency of Rich Internet Applications using either XML or JSON. Given almost any combination of these factors, implies that significant performance differences will exist between semantically equivalent implementations based upon either data format. However, in every situation, utilizing JSON is more efficient than utilizing XML. Hence, these figures provide an empirical proof that a large number of situations exist where it is worthwhile, from a performance viewpoint, transforming an existing application from an XML-based application to a JSON-based application.

5.2. User survey

To further evaluate our system, we conducted a brief usability trial. The first step is to determine the sample size. Various research works on the sample size for usability testing have been discussed. For example, Nielsen and Landauer (1993) show that five users can detect approximately 85% of the problems in an interface if the probability of discovering usability problems by single test user is about 31%. Faulkner (2003) did research on whether five participants were sufficient for usability testing and the results showed that five users can detect as few as 55% or as much as 99%. If the number of participants is increased to ten, the lowest percentage of problems detected is increased to 80% and if the number of participants is increased to twenty, the number is increased to 95%. Lewis (2006) shows that when the probability of detecting a problem is low, larger numbers of participants is required.

The proper sample size depends on many factors, such as the tasks of usability testing, iterations of the testing, the range of the knowledge of the participants. Our sample size for usability test is eight participants, which is the minimum sample size as recommended in the Common Industry Format for Usability Test Reports.[31] In addition, the use of our system is straightforward – the system is highly automated and requires very little interaction with the user. Hence, we believe that, based upon the above evidence and the simplicity of the interaction between the system and the user, an argument can be constructed that the probability of detecting a problem is between 18% and 30%, which means five to eight participants are required to detect 85% of the problems.

Our trail used simple acceptance sampling for selecting our participants. A pre-trial interview with each trial-ist, separately, elucidated the following information:

(1) Sample size: eight graduate students
(2) Range of refactoring knowledge or experience:
  - One user: intermediate experience with refactoring

[31] http://www.idemployee.id.tue.nl/g.w.m.rauterberg/lecturenotes/common-industry-format.pdf.

Table 8
Testing results for accessing a node with different length.

Length of the text	XML (ms)		JSON (ms)		*T*-Test (*P*-value)	Cohen's *d* value
	Mean	Standard deviation	Mean	Standard deviation		
50 chars	2.5	0.54	0.79	0.41	$P < 0.01$	3.57
1000 chars	3.05	0.56	1.21	0.43	$P < 0.01$	3.69

Table 9
The results for the questionnaire.

Question	Answer
1	8 yes
2	8 no
3	15 min
4	4.75
5	4.25
6	8 yes
7	4.125
8	8 yes
9	4.25
10	7 yes/1 no

- Four users: a basic knowledge and experience with refactoring
- Three users: no experience with refactoring

(3) Range of technical knowledge with JavaScript, XML and JSON:
- Two users: expert
- Four users: intermediate
- Two users: basic

Given our trial-ists, we designed different tasks for them to test the system:

(1) Task 1: Convert an XML file into a JSON file using the XML to JSON Converter.
(2) Task 2: Transform JavaScript code for accessing the XML file to accessing the JSON file converted in the Task1 using the JavaScript Code Transformer.
(3) Task 3: Generate JavaScript code for accessing selected nodes or attribute of another XML file.

Finally, after completing their task, the participants were required to fill in the following questionnaire:

(1) Did you successfully complete all the tasks? (yes/no)
(2) Did you encounter any bugs in the system? (yes/no)
(3) How much time did you spend using the system?
(4) Compare to other software you used, was the system easy to use? (1–5 from hardest to easiest)
(5) Was the system screen easy to navigate? (1–5 from hardest to easiest)
(6) Did you find all the features you expect in the system? (yes/no)
(7) Were you able to discover the features easily? (1–5 from hardest to easiest)
(8) Was the refactored code functionally correct? (yes/no)
(9) What is your overall impression of the system? (1–5 from very negative to very positive)
(10) Would you use the tool on your production system? (yes/no)

According to the results from Table 9, the users have little trouble using the system. They were able to use the system to successfully complete the three tasks. Additionally, most of them would have no hesitation to use the system in the future. One trial-ist expressed reservations about using XtoJ for production systems. Upon further discussions with this person, we discovered that the reason for his hesitation is that XtoJ currently does not conform to the coding style standard currently used by his organization. However, XtoJ can easily be modified to resolve this concern. Although the sample size is small, it does provide evidence that our system can easily be used by other users to help improve the refactoring process.

6. Threats to validity

As with all experiments, our experimental design is far from perfect. Here we discuss some of the design decisions and their potential impact.

6.1. Internal threats

Clearly, the potential for experimenter bias exists. As the authors of the paper are the same as the authors of the approach, a risk exists that the experimental design somehow favours the approach. The risk has been tackled by randomly selecting (first applications found in an Internet search) the materials (applications) that are utilized. That is, we utilize real-world applications which are independent of the authors. As such, the problem cannot be redefined (the performance of these applications has a unique definition) ensuring its independence from our approach. However, we clearly cannot guarantee that these applications are representative samples of web applications; instead, this is tackled by comparing our results to another impendent source – that of the manually produced results by Nurseitov et al. (2009). The fact that we demonstrate that these two sets of independent results as consistent provides evidence that the presented results have a degree of independence from the experimental set-up.

The experiment treats the applications effectively as black-boxes as the question is only about changing the data exchange language; that is, it investigates mainly at the system level. However, clearly, these languages possess internal structures which may have some noticeable impact on the results. This possibility is investigated in Section 5.1.3 which does indeed find that different XML structures have significant impact on performance. While, these impacts are not found at a level to invalidate the system-level question; it is important to note that they are covariates to the question. And further research is required to fully resolve their relationships.

6.2. Construct threats

We know of no ideal mechanism to answer the question about the relative performance. Performance is this content has constraints which need to be recognized as limitations of the experiment. Clearly, the performance is any system is impacted by the input it receives. In general, the delay introduced by humans entering data would swap the execution time of the other components; and hence, the experiment explicitly avoids timing across data entry statements. Other variations are handled by repeatedly sampling an applications performance in an attempt to produce a robust estimate of its average behavior.

6.3. Content validity

The principal threat here is that the applications may be of differing quality standards. The study may include systems which are coded in such a poor fashion that the “coding style” has a significant impact upon performance. Clearly, this impact also exists for systems which are coded well above average; however, it is believed that “poor coding” is more likely to have a more significant impact than “good coding”. Unfortunately, we know of no mechanism to estimate if an application has an “average-quality coding style”. The authors undertook a manual inspection of the systems and can report, in their opinion, that no obvious “extremely poor coding styles” were in evidence; however, this clearly remains a threat to the validity of the results.

6.4. External validity

Clearly, the results of the experiment cannot be safely generalized to other contexts. In fact, as the numbers of applications are finite, the study has a low statistical power in this regard. In addition, it is unsafe to generalize these findings into scenarios outside the characteristics of the (randomly) selected applications. For example, the selected applications, and (in general) the current generation of web applications spend a massive amount of their time dealing transfer, encoding and storage of data. However, in the future applications may emerge which have different characteristics, such as being computational bound. Clearly, under such scenarios, further research in required to evaluate the impact of the approach on these different types of systems.

7. Related work

Refactoring for improving the quality of software is not new. For example, Opdyke (1992) defines different refactoring patterns to automatically restructure object-oriented programs. Fowler (1999) provides a comprehensive catalog of refactorings, the principles of refactorings and when and where to implement the refactorings for object-oriented programs. Bulka and Mayhew (1999) presents many optimization techniques for coding and designing of the C++ programming language to improve the code efficiency and performance. Tokuda and Batory (2001) discuss how to design object-oriented applications to improve software design. Dudziak and Wloka (2002) provide a method to detect structural weaknesses to improve code structure. Tahvildari and Kontogiannis (2004) propose a reengineering framework to detect potential design flaws, by using object-oriented metrics and applying transformations, to improve the specific qualities of a software system.

Various refactoring proposals for web applications have also been discussed: Harold (2008) shows how to refactor HTML to improving the design of existing web applications; Ricca and Tonella (2001) present an automatic and semi-automatic restructuring tool (ReWeb) to implement the analysis on the architecture and evolution of a website. They (Ricca et al., 2002) also present transformation rules on HTML to improve the quality of web applications; Olsina et al. (2007) and Garrido et al. (2011) present a Web Model Refactoring (WMR) approach on the navigation and presentation models, aimed at improving the quality of web applications. They also demonstrate how to use WebQEM, a quality evaluation method, to test the impact of refactoring.

Recently, refactoring has started to be applied for different purposes, such as migration: Matthews et al. (2004) show how to automatically transform traditional interactive programs into CGI programs; Rossi et al. (2008) present an approach to refactor legacy web applications into RIAs; Schäfer et al. (2011) present a refactoring tool (Relocker) to assist programmers with the refactoring of synchronized blocks into ReentrantLocks and ReadWriteLocks; Dig et al. (2009) and Dig (2011) present a refactoring tool (CONCURRENCER) to refactor sequential code into concurrent code; Lindvall et al. (2003) describe a process to restructure an existing experience management system (EMS) to improve the architecture of the system; Mesbah and Deursen (2007) propose a migration process using reverse engineering techniques to transform multi-paged web applications into single-paged AJAX.

Additionally, some of refactoring approaches for efficiency purposes exist: Demeyer (2002) refactored C++ programs by replacing conditionals in the programs with polymorphic method calls. The results from their work show that the refactored program is faster than the original program; Beyls and D’Hollander (2009) present a cache profiling tool (SLO). By analyzing runtime reuse paths, SLO finds the root cause of poor data locality that generates cache misses. The tool also performs the most promising optimizations through three levels: loop, iteration and function; Chu and Dean (2008) use a set of source level transformations to automatically migrate list based JSP web pages to AJAX. The process includes (1) extracting a web service (XML format) from a JSP page and (2) transforming the code of the JSP web page to use the service (XML data) instead of the data from a relational (or SQL) database.

8. Conclusion and future work

The Web 2.0 drives the growth of Rich Internet Application (RIA) technologies. These applications often impose significant requirements on run-time efficiency. Due to its light weight nature and the native support for JavaScript, JSON, an alternative data exchange format to XML, improves the efficiency of AJAX applications. Specifically, it improves the efficiency with respect to (1) network transfer time and (2) JavaScript processing time on client side. Changing the data format for an existing AJAX application involves: (1) Converting the data format from XML to JSON and (2) Changing the JavaScript code – from code which manipulates the XML version of the code to code which manipulates the JSON version of the code.

However, manual versions of this transformation tend to introduce defects because the transformation requires the understanding of XML, JSON and all the interactions between the JavaScript code and the XML data. Thus, we have designed a refactoring system (XtoJ) to achieve such transformations. This system is based around three components (XML to JSON Converter, JavaScript Code Transformer and JavaScript Code Generator) which aid programmers with the transformation from utilizing XML to utilizing JSON. We empirically demonstrate that our transformation system significantly improves the efficiency of AJAX applications; and the improvements are in-line with efficiency reports for manual construction [3]. We also analyze three additional variables that influence the performance of our system (the number of objects and properties, the structure of the XML and JSON data and the length of the text in the properties). These additional results imply that the benefits of transforming such systems will significantly improve large numbers of applications. Additionally, usability testing results provide evidence that our system can easily be used by other users to help improve the refactoring process.

Our future work includes gathering additional insights beyond the presented transform into further improve the efficiency of the refactored programs. For example, inappropriate interaction structures between AJAX code and ActionScript 3 add-ons. We will also fully develop XtoJ and release it to the public. Once the tool is released, we will perform a survey by gathering inputs from the programmers using our tool to determine the effectiveness of our tool using a much larger sample size with a variety of different configurations (Eaton and Memon, 2007).

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.jss.2012.07.019.

References

Allen, R., Qian, K., Tao, L., Fu, X., 2008. Web Development with JavaScript and AJAX Illuminated. Jones & Bartlett Learning, Sudbury, MA, USA.

Attenborough, M., 2003. Mathematics for electrical engineering and computing. In: Electronics & Electrical. Newnes, Boston, USA.

Beyls, K., D'Hollander, E.H., 2009. Refactoring for data locality. IEEE Computer 42 (2), 62–71.

Bulka, D., Mayhew, D., 1999. Efficient C++ Performance Programming Techniques. Addison-Wesley, Boston, MA, USA.

Card, S.K., Robertson, G.G., Mackinlay, J.D.,1991. The information visualizer, an information workspace. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM Press, New York, NY, USA, pp. 181–188.

Chu, J., Dean, T.,2008. Automated migration of list based JSP web pages to AJAX. In: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society, Los Alamitos, CA, USA, pp. 217–226.

Coelho, R., Kulesza, U., Staa, A.V., Lucena, C.,2006. Unit testing in multi-agent systems using mock agents and aspects. In: Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. ACM Press, New York, NY, USA, pp. 83–90.

Demeyer, S., 2002. Maintainability versus performance: what's the effect of introducing polymorphism? Technical report, Lab on Reengineering, Universiteit Antwerpe, Belgium.

Dig, D., Marrero, J., Ernst, M.D.,2009. Refactoring sequential Java code for concurrency via concurrent libraries. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, Washington, DC, pp. 397–407.

Dig, D., 2011. A refactoring approach to parallelism. IEEE Software 28 (1), 17–22.

Dudziak, T., Wloka, J., 2002. Tool-supported discovery and refactoring of structural weaknesses in code. Dissertation. Technical University of Berlin.

Eaton, C., Memon, A.M., 2007. An empirical approach to testing web applications across diverse client platform configurations. International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering 3 (3), 227–253.

Eichorn, J., 2007. Understanding AJAX: Using JavaScript to Create Rich Internet Applications. Prentice Hall, NJ, USA.

Faulkner, L., 2003. Beyond the five user assumption: benefits of increased sample sizes in usability testing. Behaviour Research Methods 35 (3), 379–383.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

Garrett, J., 2005. AJAX. A new approach to web applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php.

Garrido, A., Rossi, G., Distante, D., 2011. Refactoring for usability in web applications. IEEE Software 28 (3), 60–67.

Hagen, W.V., 2006. The Definitive Guide to GCC. Apress, Berkeley, CA, USA.

Harold, E.R., 2008. Refactoring HTML: Improving the Design of Existing Web Applications. Addison-Wesley, Upper Saddle River, NJ, USA.

Irani, R. JSON continues its winning streak over XML. http://blog.programmableweb.com/2010/12/03/json-continues-its-winning-streak-over-xml/.

Jacobs, S., 2006. Beginning XML with DOM and AJAX: From Novice to Professional. Apress, Berkeley, CA, USA.

Lewis, J.R., 2006. Sample sizes for usability tests: mostly math, not magic. Interactions 13 (6), 29–33.

Lindvall, M., Tvedt, R.T., Costa, P., 2003. An empirically-based process for software architecture evaluation. Empirical Software Engineering 8 (1), 83–108.

Matthews, J., Findler, R.B., Graunke, P.T., Krishnamurthi, S., Felleisen, M., 2004. Automatically restructuring programs for the web. Automated Software Engineering 11 (4), 337–364.

Mens, T., Tourwé, T., 2004. A survey of software refactoring. IEEE Transactions on Software Engineering 30 (2), 126–139.

Mesbah, A., Deursen, A.V.,2007. Migrating multipage web applications to single-page AJAX interfaces. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington, DC, USA, pp. 181–190.

Ng, W., Lam, W.Y., Cheng, J., 2006. Comparative analysis of XML compression technologies. World Wide Web 9 (1), 5–33.

Nielsen, J., Landauer, T.K.,1993. A mathematical model of the finding of usability problems. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM Press, New York, NY, USA, pp. 206–213.

Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C.,2009. Comparison of JSON and XML data interchange formats: a case study. In: The International Conference on Computer Applications in Industry and Engineering. ISCA, Cary, NC, USA, pp. 157–162.

Olsina, L., Rossi, G., Garrido, A., Distante, D., Canfora, G.,2007. Incremental quality improvement in web applications using web model refactoring. In: Proceedings of the 2007 International Conference on Web Information Systems Engineering Web Information. Springer-Verlag, Berlin, Heidelberg, pp. 411–422.

Opdyke, W.F., 1992. Refactoring object-oriented frameworks. Dissertation. University of Illinois.

Parr, T., 2007. The Definitive ANTLR Reference: Building Domain-specific Languages. Pragmatic, Raleigh, NC, USA.

Paulson, L.D., 2005. Building rich web applications with AJAX. Computer 38 (10), 14–17.

Ricca, F., Tonella, P., 2001. Understanding and restructuring web sites with ReWeb. IEEE MultiMedia 8 (2), 40–51.

Ricca, F., Tonella, P., Baxter, I.D., 2002. Web application transformations based on rewrite rules. Information and Software Technology 44 (13), 811–825.

Roberts, D.B., 1999. Practical analysis for refactoring. Dissertation. University of Illinois at Urbana-Champaign.

Rossi, G., Urbieta, M., Ginzburg, J., Distante, D., Garrido, A.,2008. Refactoring to rich internet applications. A model-driven approach. In: Eighth International Conference on Web Engineering. IEEE Computer Society, Los Alamitos, CA, USA, pp. 14–18.

Schäfer, M., Sridharan, M., Dolby, J., Tip, F.,2011. Refactoring Java programs for flexible locking. In: International Conference on Software Engineering. ACM Press, New York, NY, USA, pp. 71–80.

Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R., 2002. XMark: a benchmark for XML data management. In: Proceedings of the Very Large Database Conference. VLDB Endowment, pp. 974–985.

Srivisut, K., Muenchaisri, P.,2007. Defining and detecting bad smells of aspect-oriented software. In: 31st Annual International Computer Software and Applications Conference. IEEE Computer Society, Washington, DC, USA, pp. 65–70.

Tahvildari, L., Kontogiannis, K., 2004. Improving design quality using meta-pattern transformations: a metric-based approach. Journal of Software Maintenance and Evolution: Research and Practice 16 (4–5), 331–361.

Thompson, S., Reinke, C., 2002. A catalogue of function refactorings. Lab Report. University of Kent.

Tokuda, L., Batory, D., 2001. Evolving object-oriented designs with refactorings. Automated Software Engineering 8 (1), 89–120.

Webucator, 2009. JavaScript object notation (JSON). http://www.learn-AJAX-tutorial.com/Json.cfm.

W3C, 2008. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml/.

W3C, 2009. Namespaces in XML 1.0. http://www.w3.org/TR/REC-xml-names/#NT-LocalPart.

Zakas, N.C., McPeak, J., Fawcett, J., 2006. Professional AJAX. Wiley, Indianapolis, IN.

Ming Ying received the B.Sc. degree in Information Management and Information System and M.Sc. degree in Computer Application Technology from the Northwest A&F University, China. She received her Ph.D. degree in Software Engineering and Intelligent Systems from the Department of Electrical and Computer Engineering at the University of Alberta, Canada. Her interests include Web 2.0, Rich Internet Applications (Adobe Flash and Ajax), web application efficiency, software refactoring and software restructuring, scripting programming language (JavaScript and ActionScript), and web data format.

James Miller received the B.Sc. and Ph.D. degrees in Computer Science from the University of Strathclyde, Scotland. In 2000, he joined the Department of Electrical and Computer Engineering at the University of Alberta as a full professor. He has published over one hundred refereed journal and conference papers on Software and Systems Engineering (see www.steam.ualberta.ca for details on recent directions); and currently serves on the program committee for the IEEE International Symposium on Empirical Software Engineering and Measurement; the IEEE International Conference of Software Testing, the IEEEE International Symposium on Software Reliability, and sits on the editorial board of the Journal of Empirical Software Engineering.