A Code Migration Framework for AJAX Applications

A. Puder

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
arno@sfsu.edu

Abstract. AJAX (Asynchronous JavaScript and XML) defines a new paradigm for writing highly interactive web applications. Prominent web sites such as Google Maps have made AJAX popular. Writing AJAX applications requires intimate knowledge of JavaScript since it is difficult to write cross-browser portable JavaScript applications. In this paper we first discuss the benefits of AJAX compared to other technologies such as Java applets. Then we propose a code migration framework that allows the programmer to write AJAX applications in Java. The Java application is automatically translated to JavaScript and migrated to the browser for execution. Our approach requires no knowledge of JavaScript. As web applications are written in Java, the developer benefits from powerful debugging tools that are not available for JavaScript. We have implemented a prototype that demonstrates the feasibility of our ideas. The prototype is available under an Open Source license.

1 Motivation

The initial intend of the World-Wide Wide (WWW) was to give access to remote documents. This document centric view soon proved to be insufficient as eCommerce recognized the potential of the new media. Subsequently, HTML was extended to allow the description of user interfaces based on web forms. The web browser thus assumed the role of a generic client that is capable to render *a priori* unknown user interfaces. The technologies of the WWW therefore have changed from being document centric to operational interaction centric. Numerous technologies came into existence to facilitate the development of web applications. Java Server Pages (JSP), PHP Hypertext Processor (PHP), and Struts are only few of those technologies.

Despite these new technologies, the user is very much aware of latencies because web applications are still based on web pages (i.e., user interfaces) being loaded from a remote web server. A light-weight scripting language called JavaScript was introduced by Netscape in 1995 mainly for doing some user input validation that does not require interaction with a remote web server. This can

already be seen as the first step towards migrating part of the application logic to the web browser.

Other technologies such as Java applets have attempted to become a standard for client-side processing, but they could not establish themselves mostly because of political issues between different vendors. The lowest common denominator today for writing client-side applications that can run inside any web browser without requiring any additional browser plugins is thus JavaScript. It is in this context that AJAX (Asynchronous JavaScript and XML) has emerged as a new paradigm for writing highly-interactive web applications.

At the core of AJAX is JavaScript and writing an AJAX application thus requires intimate knowledge of JavaScript. Matters become more complicated by the fact that writing portable JavaScript that runs in all major browsers such as Internet Explorer (IE) or Firefox is a daunting task. One of those problems is the lack of powerful development tools for JavaScript. This paper introduces a new approach for facilitating the creation of AJAX application based on a code migration framework. The outline of this paper is as follows: Section 2 gives a proper definition of AJAX and also discusses the difficulties in writing an AJAX application. Section 3 introduces our code migration framework. Section 4 discusses our prototype implementation while in Section 5 we discuss related work. Section 6 finally provides a conclusion and an outlook.

2 AJAX

In this section we first provide an introduction to AJAX (Section 2.1), explain the benefits of AJAX (Section 2.2), and finally why it is so difficult to write AJAX applications (Section 2.3).

2.1 Overview of AJAX

The term AJAX was first coined in [5]. The author of this article attempted to describe a new class of web applications that differ significantly from previous technologies such as PHP, JSP, or Struts. Figure 1 demonstrates this difference. The left side of this figure shows the traditional way of implementing web applications. The web browser is used for rendering the user interface, typically a web form that the user can populate. Apart simple input validation, no processing happens during this phase. Once the user presses the submit button, the form is sent via an HTTP request to the server. Upon unmarshalling the data, the web application running on the side of the web server computes a new HTML page that is sent back to the browser. While the browser is waiting for that response, the user cannot use the interface.

The right hand side of Figure 1 shows how AJAX changes this picture. The main difference is that AJAX application make use of the JavaScript interpreter that is contained in every popular web browser. Part of the application logic is thus implemented in JavaScript and executed on the side of the client. All browsers support the so-called `XMLHttpRequest` object that allows JavaScript

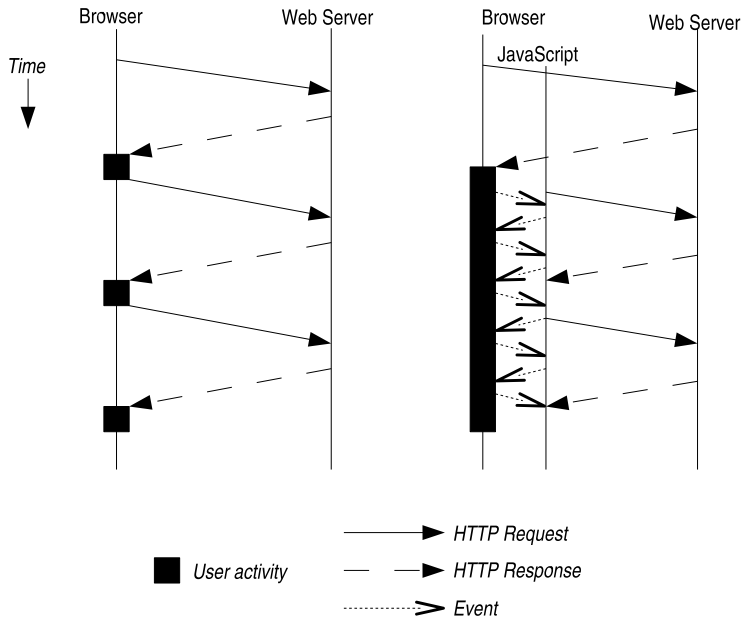

Fig. 1. Traditional web application vs. AJAX.

to issue a HTTP request to the remote web server. The user can continuously interact with the application as shown in the figure. Event handler invoke appropriate JavaScript functions that use the DOM (Document Object Model) to make fine-grained updates to the user interface without requiring complete page reloads as in the traditional model.

The asynchronous nature of AJAX applications refers to the fact that the JavaScript code may issue HTTP requests independent of user interaction. This makes it possible to do processing in the background without the user having to wait for a response from the web server. All parameters and responses have to be marshalled in a way so they can be piggy-backed on HTTP requests and responses. One obvious choice is XML as all popular browsers include XML parsers that make the marshalling and unmarshalling of parameters relatively simple.

2.2 AJAX vs. other technologies

AJAX allows the execution of application logic inside the browser. This increases interactivity of web application dramatically compared to the submit-and-page-reload paradigm. By doing so, AJAX applications get closer to the look-and-feel of desktop applications and some analysts already foresee the browser as the next-generation desktop replacement that could even threaten Microsofts

monopoly. Whether or not this vision will come true, it can certainly be expected that more and more web applications will want to employ AJAX technologies.

AJAX makes use of the fact that JavaScript interpreters are ubiquitous in all popular web browser. Moving application logic to the client side is not new. One of the promises of the Java programming language was to enable web applications in a similar way as AJAX through Java applets. A Java applet is a Java application running inside the browser. It is therefore possible to achieve the same effect with applets as with AJAX applications. While Java is a much more mature language than JavaScript with a more powerful GUI library, the major downside of applets is that they require a Java Runtime Environment (JRE) plugin for the respective browser. This requires the end-user to download the plugin which creates an additional burden for the end-user.

Given a choice, end-users either intentionally or unintentionally choose not to install additional software if they have an already existing solution and the benefits of the alternative are not immediately apparent. As a specific example of the reluctance of end-users to explicitly install software can be seen by the proportion of Windows users who use IE, despite security issues compared to other browsers. IE currently owns more than 85% of the market share (see [9]), primarily because it is bundled along with Windows. If end-users are reluctant or simply do not bother to use easy-to-install software such as alternate web browsers, they are usually not willing to install a Java Runtime Environment. This accounts for the fact that AJAX has become so popular because it only uses the lowest common denominator available in virtually all web browsers.

2.3 Writing AJAX Applications

As outlined above, writing AJAX applications therefore requires JavaScript to achieve the interactiveness desired by the latest generation of web applications. JavaScript was created by Netscape and was first incorporated in Netscapes browser version 2.0. The rationale behind JavaScript was to make Navigator's newly added Java support more accessible to non-Java programmers. The design goals of JavaScript therefore focused on a loosely-typed scripting language suited the environment and audience, namely the few thousand web designers and developers back in 1995 who needed to be able to tie into page elements without a bytecode compiler or knowledge of object-oriented software design.

Microsoft released a port of JavaScript called JScript with IE 3.0. JScript was one revision behind Navigator's JavaScript that made it difficult already back then to write cross-browser portable JavaScript. In 1997, the European Computer Manufacturers Association (ECMA) standardized a universally supported core functionality called ECMAScript (see [3]). Despite this standardization effort, support for JavaScript is not as homogeneous as one might wish. Writing portable JavaScript for all major browsers still requires intimate knowledge of the different object models.

There are many pitfalls that a JavaScript programmer has to deal with today. First and foremost, there are no powerful development tools available for JavaScript. Mozilla offers a debugger, but IE merely indicates by an alert icon in

the status bar when something went wrong. Other issues in creating JavaScript applications has to do with differences in the JavaScript object model supported by various browsers. Sometimes events such as mouse events are offered from inner-most nested elements to top-level elements (called *Event Bubbling* and supported by IE); sometimes events are offered elements in the reverse order (called *Event Capturing* and supported by Netscape/Mozilla). Advanced event models such as Event Listeners that allow the registration of multiple listeners for one particular event are not supported sufficiently in IE. The author of [6] gives a more comprehensive list of issues.

This is only a short list of the problems that one will likely encounter when developing AJAX applications. These issues combined with the fact that JavaScript supports object-oriented programming only through conventions and clever programming tricks (e.g., to achieve the effect of inheritance one has to change the prototype of the derived class) will place a high burden on anyone interested in creating AJAX applications. The main idea of this paper is that a programmer can write an AJAX application without requiring any knowledge of JavaScript. Our approach is outlined in the following.

3 Framework

This section gives a detailed description of our framework. At its core is a code migration framework that shields the programmer from the complexities of writing JavaScript applications. A developer can write an AJAX web application in Java benefiting from powerful and mature tools and then migrate the code to JavaScript. In Section 3.1 we briefly state our assumptions that guide the design of our framework. Section 3.2 then introduces XMLVM, an XML-based programming language that is at the core of our code migration framework. Section 3.3 then shows how to create JavaScript out of XMLVM. In Section 3.4 we finally describe the underlying architecture of our framework.

3.1 Assumptions

Before describing the details of our approach, we first explicitly state the assumptions that will influence some design decisions of our framework:

Universal access: We assume that potentially any user in the WWW might be using the web application.

No special browser plugins: In order to support universal access, we do not assume any special browser plugins such as the Java Runtime Environment.

Web applications using Java: We assume that the programmer is using Java (not JavaScript) as the programming language of choice to write his or her web application.

Self-contained applications: For now, we only consider self-contained applications that have no dependencies to external resources such as databases.

No JavaScript knowledge necessary: The programmer does not need to know any JavaScript in order to develop AJAX-enabled web applications.

The reason for assuming universal access to a web application is that it is generally much simpler to develop a web application for a closed environment. Corporate intranets for example typically enforce the use of a particular desktop configuration. AJAX should only be considered in heterogeneous environments. The assumptions stated above basically lead to a development environment where the programmer is shielded from JavaScript. Since we do not assume any special browser plugins, but yet allow the programmer to implement his or her program in Java, we need a code migration framework that can translate and migrate the Java application to JavaScript.

3.2 XMLVM

As a first step towards our code migration framework for AJAX applications, we begin by defining an XML-based programming language. In this section we focus on describing the details of this language and defer the usage of this language to a subsequent section. Since this XML-based programming language is based on the Java virtual machine, we call this language XMLVM. XMLVM basically allows us to represent the contents of a class file (i.e., the output generated by a Java-compiler) through XML. Another way to look at XMLVM is that it defines an assembly language for the Java virtual machine using XML for the syntax. The object model of XMLVM is consequently based on the object model of Java. The virtual machine model of XMLVM is shown in Figure 2.

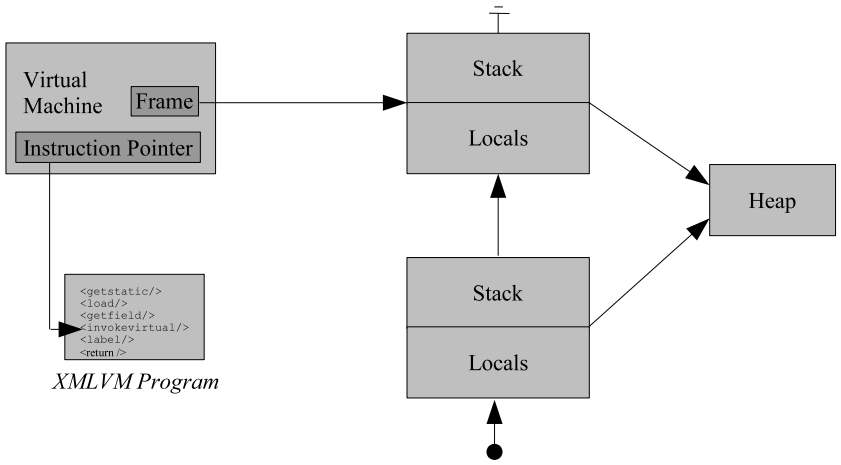

Fig. 2. XMLVM Virtual Machine Model.

The XMLVM program shown in Figure 2 contains the instructions of a method to be executed. These instructions are essentially the byte code instructions supported by the Java virtual machine. The virtual machine maintains

an instruction pointer to the next instruction to be executed. Upon entering a method, a new frame consisting of a stack and local variables is created. This frame will be deleted upon exiting the method. The virtual machine maintains a pointer to the current frame (which represents the most nested method call). A method has only access to its own stack and local variables as well as the global heap. The actual parameters of a method are automatically stored in the local variables. Besides the stack frames, the virtual machine maintains a garbage collected global heap where a program can allocate new objects. The following template shows the general structure of any XMLVM program:

```
<xmlvm>
  <class ...>
    <field .../>
    <method ...>
      <signature>...</signature>
      <code>...</code>
    </method>
  </class>
</xmlvm>
```

An XMLVM program consists of one class. Every class can have one or more fields and methods. The attributes of the XML-tags, that are not shown in the template above, give more details such as identifiers or modifiers. A method is defined through a signature and the actual implementation, denoted by the tags <`signature`> and <`code`> respectively. Consider the following simple Java-class:

```
// Java
class Calc {
    int x;
    void add(int y)
    {
        x += y;
    }
}
```

Class `Calc` has one field called `x` and one method called `add`. The method adds the actual parameter given to it to the field `x`. Although this is a very simple example, it allows us to show all basic aspects of an XMLVM program. The following XML shows the representation of class `Calc` in XMLVM:

```
<xmlvm>
  <class name="Calc">
    <field name="x" type="int"/>
    <method name="add" stack="3" locals="2">
      <signature>
```

```
6         <return type="void"/>
7         <parameter type="int"/>
8       </signature>
9       <code>
10        <load type="Calc" index="0"/>
11        <dup/>
12        <getfield class-type="Calc" type="int" field="x"/>
13        <load type="int" index="1"/>
14        <add/>
15        <putfield class-type="Calc" type="int" field="x"/>
16        <return/>
17      </code>
18    </method>
19  </class>
20 </xmlvm>
```

It should be emphasized again that the above XMLVM program is essentially an XML-representation of the contents of the `Calc.class` class file. The top-level tags are identical to the XML-template shown earlier. The <`method`>-tag has two attributes: `stack` and `locals`. `stack` tells the virtual machine the maximum stack-size needed for this method. In this example, method `add` will never push more than 3 elements at the same time onto its stack. The `locals` attribute tells the virtual machine how many local variables are needed for this method. The first local variable always represents the `this`-pointer. The next local variables represent the actual parameters. Since method `add` has only one input parameter and no additional local variables, the `locals` attribute is 2. Note that the Java compiler computes the values for `stack` and `locals` and stores them in the class file.

The more interesting part of the XMLVM-program shown above is the actual implementation of method `add`. The <`load`> instruction pushes the `this`-pointer referred to by local variable with index 0 onto the stack. Instruction <`dup`> duplicates the top of the stack so that the `this`-pointer now is pushed twice on the stack. <`getfield`> pushes the current value of field `x` onto the stack. Since every instance of class `Calc` has its own field `x`, <`getfield`> needs a reference to the instance whose field `x` should be pushed onto the stack. This reference has to be on the top of the stack. <`getfield`> pops off the reference and replaces it with the value of field `x`. After this instruction, the stack contains the `this`-pointer and the value of field `x`.

The next instruction <`load`> pushes the actual parameter `y` (referenced through local variable index 1) onto the stack. The top two elements of the stack are now the values to be added. The following instruction <`add`> pops off the last two values and pushes their sum back onto the stack. At this point, the stack contains the `this`-pointer as well as the sum. The <`putfield`> instruction works similarly as the <`getfield`> instruction, except that a value is written back to a field. After this instruction, the stack is empty. The final instruction <`return`> exits the method.

The XMLVM instruction set feature a mix of low-level and high-level virtual machine instructions. Next to the low-level instructions mentioned above, there exist high-level instructions such as `new` (for instantiating new objects) and `invokevirtual` (invoke a virtual method). These instructions go beyond the capabilities of normal (hardware) machine languages and therefore require substantial runtime support. Table 1 gives an overview of some of the instructions found in XMLVM. The table shows how the instructions introduced in this section affect the stack by showing the stack before and after the respective instruction.

Instr.	**Stack**
<add>	$\ldots, value_1, value_2 \Rightarrow \ldots, result$
<getfield>	$\ldots, objref \Rightarrow \ldots, value$
<putfield>	$\ldots, objref, value \Rightarrow \ldots$
<load>	$\ldots \Rightarrow \ldots, value$
<new>	$\ldots \Rightarrow \ldots, objref$
<invokevirtual>	$\ldots, objref, [arg_1, [arg_2, \ldots]] \Rightarrow \ldots$

Table 1. Representative XMLVM instructions.

3.3 Language transformation

As stated earlier, XMLVM can be seen as an assembly language for the Java virtual machine. The difficult part is done by a Java compiler. Once a class file has been created as the result of the compilation process, it can be easily translated to XMLVM simply by analyzing the contents of the class file. The next step consists in translating XMLVM to JavaScript. This translation can be done by an XSL-stylesheet that maps XMLVM-instructions one-to-one to the target language. Since XMLVM is based on a simple stack-based machine, we simply mimic a stack-machine in the target language. An example helps to illustrate this approach. The XMLVM instruction <add> introduced earlier pops off two values and pushes the sum back onto the stack. Here is the XSL-template that creates JavaScript code for this instruction:

```
1 <xsl:template match="add">
2         <xsl:text>
3             __op2 = __stack[--__sp];
4             __op1 = __stack[--__sp];
5             __stack[__sp++] = __op1 + __op2;
6         </xsl:text>
7 </xsl:template>
```

We mimic the virtual machine of XMLVM via the variables `__locals` (for local variables), `__stack` (for the stack), and `__sp` (for the stack pointer). Vari-

ables __op1 and __op2 are used as temporary variables needed by some XMLVM-instructions. Those variables are declared for every method. The code below represents the JavaScript version of the class Calc introduced in Section 3.2:

```
1 // JavaScript generated by stylesheet
2 function Calc()
3 {
4   this.x = null;
5
6   this.add = function( __arg1)
7   {
8     var __locals = new Array(2);
9     var __stack = new Array(3);
10    var __sp = 0;
11    var __op1;
12    var __op2;
13    __locals[0] = this;
14    __locals[1] = __arg1;
15    __stack[__sp++] = __locals[0];
16    __op1 = __stack[__sp - 1];
17    __stack[__sp++] = __op1;
18    __op1 = __stack[--__sp];
19    __stack[__sp++] = __op1.x;
20    __stack[__sp++] = __locals[1];
21    __op2 = __stack[--__sp];
22    __op1 = __stack[--__sp];
23    __stack[__sp++] = __op1 + __op2;
24    __op2 = __stack[--__sp];
25    __op1 = __stack[--__sp];
26    __op1.x = __op2;
27    return;
28  }
29 }
```

The JavaScript code was generated automatically by applying an appropriate XSL-stylesheet to the XMLVM version of class Calc. As can be seen, there is a natural mapping from XMLVM to JavaScript. The intention is not to generate readable code, but correct code that uses the API of the target language. It should also be obvious that the above JavaScript code will be less efficient than the original Java program. Our assumption is that we do not migrate computational heavy applications to the browser. By carefully designing the XSL-stylesheet one can generate portable JavaScript.

3.4 Architecture

The description of the architecture that is to follow in this section, explains how XMLVM is embedded in an infrastructure for the code migration framework.

As shown in Figure 3, the main component is a Web Container that serves as an HTTP server towards the web browser. The URL used to contact the Web Container encodes a bootstrap web page as well as the application that is to be executed as an AJAX application. As shown in Figure 3, the web page `index.html` will be returned to the browser. This page has the following simple structure:

```
1 <html>
2   <head>
3     <script type="text/javascript" src="xmlvm.js"/>
4   </head>
5   <body onLoad="bootXMLVM()">
6     <div id="AJAX_APP"/>
7   </body>
8 </html>
```

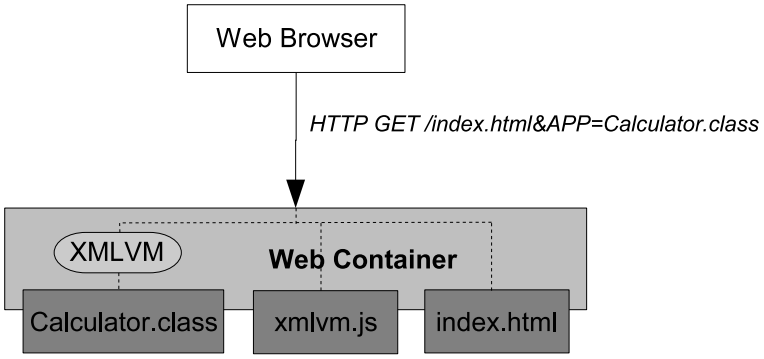

Fig. 3. Architecture.

The header of this page includes a JavaScript file called `xmlvm.js` that contains a JavaScript version of the runtime library required by the application. Without this library, running the application would result in unresolved externals. The <`body`> tag will invoke function `bootXMLVM()` once `index.html` has been successfully loaded. This function, which is defined in `xmlvm.js`, will parse the URL and retrieve the `APP` parameter (`Calculator.class` in this example). `bootXMLVM()` will then issue an HTTP request containing the application to be loaded to the Web Container. Upon receiving this request, the Web Container uses XMLVM to create a JavaScript version of `Calculator.class` that is being returned to the browser where it will be executed. The application will render itself in a visual placeholder denoted by the <`div`> element with ID `AJAX_APP` in the `index.html` page shown above.

4 Prototype Implementation

We have implemented a prototype based on the ideas outlined in the previous section to show the feasibility of our approach. We have leveraged as many Open Source tools as possible. The Web Container is implemented using a light-weight HTTP engine called Simple (see [4]). We use the Byte Code Engineering Library (BCEL) from the Apache Foundation (see [2]) to inspect the contents of a Java class file. Using BCEL, it is relatively easy to translate a class-file to XMLVM. We have implemented an XSLT stylesheet to translate XMLVM to JavaScript. Furthermore we have implemented a rudimentary JavaScript library for certain Java-API that are used in the example explained below.

To test our framework, we have implemented a calculator using Sun's Abstract Windowing Toolkit (AWT). The calculator, which is shown on the left side of Figure 4, allows simple mathematical operations. The source code of the application is 322 lines of Java. The screenshot on the left-hand side of Figure 4 shows the desktop version of the calculator. Even this simple application makes use of several external classes such as widgets (e.g., Buttons, Labels), Layout Managers, and utility classes (e.g., String and Float).

The class file of this Java application results in 1920 lines of XMLVM. After applying the stylesheet, the resulting JavaScript is 1693 lines of code. The `xmlvm.js` library, which implements all the API that is needed by the calculator, adds up to another 1210 lines of JavaScript code. The right side of Figure 4 shows the calculator as an AJAX application running inside Firefox. The buttons shown on the right side are HTML-buttons created by the AJAX application.

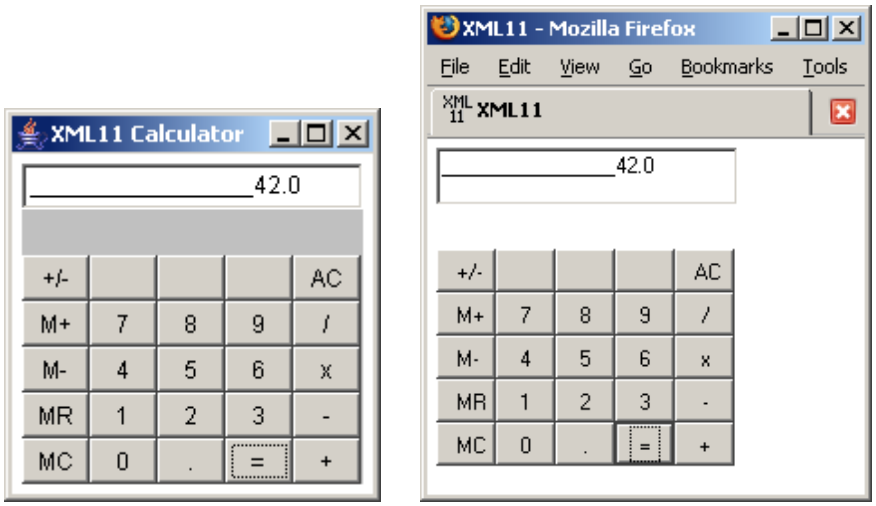

Fig. 4. Calculator as an AWT and AJAX application.

The `xmlvm.js` library contains implementation for all external references of our calculator application. These external dependencies include JavaScript implementations for `java.awt.Button`, `java.awt.Panel`, `java.awt.BagLayout`, `java.lang.String`, `java.lang.Float`, and several other classes. The JavaScript version of these classes is semantically equivalent to their Java counterparts. The JavaScript version of `java.awt.Button` for example has the same API as its Java counterpart, but will draw an HTML button inside the browser at the appropriate location using CSS.

Ideally, `xmlvm.js` should contain JavaScript implementations for the complete Java Runtime Library. Currently, `xmlvm.js` is hand-coded, specifically for the needs of our calculator application. Since the majority of the Java Runtime Library is itself written in Java, it is possible to automatically convert those class files via XMLVM to JavaScript. The only portions of the Java Runtime Library that would need to be hand-coded are JNI (Java Native Interface) calls. It is to be noted that while we believe that the majority of the Java Runtime Library can thus be automatically translated to JavaScript, this will not be possible for certain features. For example, threads cannot be supported in JavaScript since none of the JavaScript interpreters allow multithreaded applications.

5 Related Work

Several projects – commercial and Open Source – exist that aim at providing an easy migration path for legacy Java applications to web applications. WebCream is a commercial product by a company called CreamTec (see [1]). They have specialized in providing AWT and Swing replacements that render the interface of the Java application inside of a web browser. WebCream makes use of proprietary features of Microsoft's Internet Explorer and therefore only runs inside this browser.

Two Open Source projects, both hosted at SourceForge, follow the same idea of exposing Java desktop applications as web applications. The first one is called WebOnSwing (see [8]). Unlike WebCream, this project is not tailored for a particular browser. One feature offered by WebOnSwing are templates that allow to change the look-and-feel of the application that is rendered inside the browser. Another project with similar features, but not quite as mature, is SwingWeb (see [7]).

The major difference between these approaches and the one introduced in this paper is that none of them supports code migration. While the user interface rendered inside the browser looks similar, every event such as pushing a button, requires an HTTP request to the remote server. Migrating the application logic to the browser dramatically increases the responsiveness of the application while reducing the load on the remote server.

6 Conclusion and Outlook

AJAX applications have gained prominence as their interactiveness rivals that of desktop applications. Writing portable JavaScript is a difficult task due to

the fact of cross-browser incompatibilities as well as lack of powerful development tools for JavaScript. In this paper we propose a code migration framework that would allow a programmer to write a web application in Java. Using our framework, the Java application can be translated to JavaScript and executed inside any browser. The prototype implementation is released under the GNU GPL and is available at `www.xml11.org`. This web site also hosts the calculator demo discussed earlier.

As for the next step, we will investigate the dynamic translation of the Java Runtime Library in order to avoid hand-coding this complex library. We also plan to investigate the restriction of self-contained applications. Fixed resources such as databases can obviously not be migrated. We therefore need to investigate a way to keep part of the application on the server side and use proxies to communicate between the migrated and the stationary portions of the application.

References

1. CreamTec, LLC. *WebCream*. http://www.creamtec.com/webcream/.
2. Markus Dahm. Byte code engineering. Java Informations Tage, pages 267–277, 1999.
3. European Computer Manufacturers Association. *ECMAScript Language Specification*. http://www.ecma-international.org/publications/standards/Ecma-262.htm.
4. Niall Gallagher. *Simple - A Java HTTP engine*. http://sourceforge.net/projects/simpleweb/.
5. Jesse Garrett. *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.com/publications/essays/archives/000385.php.
6. Peter-Paul Koch. *Writing Portable JavaScript*. http://www.quirksmode.org/.
7. Tiong Hiang Lee. *SwingWeb*. http://swingweb.sourceforge.net/swingweb/.
8. Fernando Petrola. *WebOnSwing*. http://webonswing.sourceforge.net/xoops/.
9. WebSideStory. *U.S. Browser Usage Share*. http://www.websidestory.com/.