SIGCSE Bulletin | inroads
Paving the Way Towards Excellence in Computing Education

Enabling XML Storage from Java Applets in a GUI Programming Course

Jesse M. Heines
Department of Computer Science
University of Massachusetts Lowell
One University Ave., Lowell, MA 01854
<heines@cs.uml.edu>

Abstract

XML, the Extensible Markup Language, is widely used in graphical user interface (GUI) programming today to both specify user interfaces and to hold the data displayed in visual components. It is relatively straightforward for Java applets to read and process XML documents over the Web, but security restrictions make it complex to store those documents back on the server after they have been modified. This paper describes a set of cooperating programs and their underlying algorithms that allow Java applets to read XML documents from – and, more importantly, to store those documents back to – a Java-enabled Web server. The author uses this approach in a GUI programming course to provide students who implement their projects as Java applets with the ability to use the full power of XML and its related technologies.

This paper was published in the June 2003 issue of
Inroads – The SIGCSE Bulletin Vol. 35 No. 2 pages 88-93.

1 Teaching GUI Programming

Graphical User Interface (GUI) programming is part art and part computer science. Students often come into a GUI programming course expecting it to be easier than courses in, say, operating systems and compiler construction. They don't expect to have to wrestle with sophisticated algorithms, and they certainly don't expect to have to deal with sophisticated data structures. They expect to put a widget here, trap an event there, call some methods in an application programming interface (API), bring up a few dialog boxes, and, just to emphasize GUI's "graphical" heritage, demonstrate that they can display user error messages in color. Entering students may recall having heard the term "object-oriented" used once or twice in reference to GUI programming, but they generally have no clue about the extensive data structures and algorithms that underlie programs with well-designed, adaptable, and extensible graphical user interfaces.

In addition, computer science students seem to check their programming skills at the door when they sign up to take GUI programming. Despite the admonition posted prominently on my Web site – "don't forget your programming and don't forget you're programming!" – most do, completely. Their image of GUI programming is extremely shallow. They expect the course to require only simple assignments that they can easily complete before dinner.

The challenge, then, is to get students to appreciate the depth of GUI programming and understand how standard programming principles apply to it. They need to see GUI programming not as a computer science subfield, but as an umbrella-like superfield that has far-reaching applicability. Good graphical user interfaces not only make the power of computing technology accessible to naïve users, they also allow sophisticated users to focus on their areas of interest by minimizing input/output concerns and allowing them to work more quickly.

1.1 GUI Programming at UMass Lowell

In our department, GUI programming is taught as a two-semester senior project course. Since the Web is the de facto universal GUI platform today and students enthusiastically flock to courses that feature it, everything is taught

from a Web perspective. The synopsis of topics covered in this course sequence is provided below. For further details, including lecture notes and assignments, please see courses 91.353 and 91.461 on the author's Web site at teaching.cs.uml.edu/~heines.

The first semester begins with dynamic HTML, moves into forms and their validation, and then introduces Java applets. (We even do client-side form processing by using JavaScript's ability to access a page's location object (its URL) and parse the search portion (the part following the question mark) when the form is posted using the GET method.) We cover the main controls in the Abstract Windowing Toolkit (AWT) and their complementary event handlers, staying on the client side to avoid the headache of rogue student programs crashing shared servers, but also to focus on the GUI without having to learn the intricacies of Java Servlets or JavaServer Pages.

The second semester moves into Swing components, focusing on the power of the Model-View-Controller (MVC) architecture. Here we introduce XML, the Extensible Markup Language, and the ability to both populate and specify GUI controls from XML documents. Assignments specifically related to these technologies are "light," so as not to intrude too heavily on students' time when their main responsibility is to their senior projects. However, students are encouraged to use XML in those projects, and they receive considerable coaching on how to do this.

2 Reading and Processing XML Documents With Applets

2.1 Enabling XML Processing in Applets

A Java applet has read access to all publicly accessible documents on the server from which that applet was loaded (see Figure 1). Therefore, all you need to do to make it possible for a Java applet to read and process XML documents from its server is to make Java's XML classes available to the Java Runtime Environment (JRE) on your client machine. The easiest way to do this is to place the Java archive (JAR) files containing the XML parsers and Document Object Model (DOM) into the JRE directory that is used by your browser.

> *Note:* The following discussion applies to Internet Explorer 6.0 and Sun JRE Version 1.4.0 (java.sun.com/getjava/download.html). Procedures for using Netscape or Microsoft's JRE will be somewhat different.

As of this writing (October 29, 2002), two JAR files provide all the functionality needed: xercesImpl.jar and xmlParserAPIs.jar. These files are freely available from xml.apache. org/dist/xerces-j. When you go to that URL, look for a file with a .zip extension labeled "Latest stable binaries." As of this writing, that file was named Xerces-J-bin.2.2.0.zip and was dated 26-Sep-2002.

The default location for the Sun JRE on Windows platforms is a subdirectory of C:\Program Files\Java identified by the JRE version. As of this writing, the latest version is 1.4.0_02, and its default location is C:\Program Files\Java\j2re1.4.0_02. Under that you'll find a lib subdirectory, and under that another named ext. This is where Internet Explorer will look for xercesImpl.jar and xmlParserAPIs.jar. On my system, the full path to the directory in which these files are stored is C:\Program Files\Java\j2re1.4.0_02\lib\ext.

This should work fine on your own system for running applets that access XML documents, but what about Joe Public somewhere out on the Web whose system is completely out of your control? For such people, you need to use the archive attribute of the applet tag to identify a location from which the .jar files can be automatically downloaded if they are not already cached on the user's system. On my system I set up a virtual directory named javajars in which to store archive files needed by my applications. I then specify the location of the .jar files as follows:

```
<APPLET code="MyXMLApplet.class"
    width="800" height="400"
    archive="/javajars/xercesImpl.jar,
        /javajars/xmlParserAPIs.jar">
</APPLET>
```

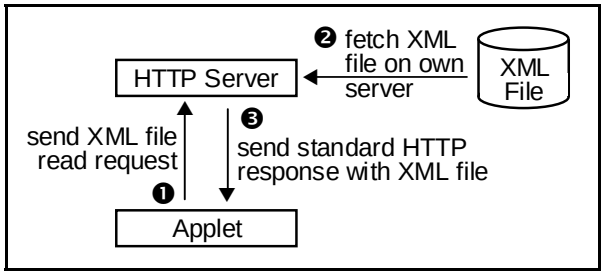

Figure 1. An applet requesting an XML document from the server from which that applet was loaded.

2.2 Implementing XML Processing in Applets

With access to Java's XML classes on the client side, an applet can read an XML file from the server on which that applet resides. The goal of the discussion that follows is to make a Document object available to the applet so that it can use the various classes and methods of the XML Standard API (see xml.apache.org/xerces2-j/javadocs/api/index.html). The classes we define will reside in the UmlGui package, imported using the statement:

```
import edu.uml.gui.* ;
```

The method that opens an XML document can encounter a number of errors. The most obvious of these are that the specified file may not exist or the document may not be "well-formed" in an XML sense and therefore the method cannot parse the document. This and all other methods in the UmlGui package are therefore written to throw a pro-

grammer-defined exception (UmlGuiException) when an error occurs.

Opening an XML document requires three basic steps:

(1) Get an instance of the DocumentBuilderFactory class:

```
DocumentBuilderFactory dbf =
  DocumentBuilderFactory.newInstance() ;
```

(2) Use the instance of the DocumentBuilderFactory class to get an instance of the DocumentBuilder class:

```
DocumentBuilder db = null ;
// declare outside of try block
try {
  db = dbf.newDocumentBuilder() ;
catch ( ParserConfigurationException pce ) {
  throw new UmlGuiException( pce.toString() + ... ) ;
}
```

(3) Use the DocumentBuilder object to open and parse the document, returning the Document object if successful:

```
try {
  return db.parse( strFilePath ) ;
} catch ( SAXException saxe ) {
  throw new UmlGuiException( saxe.toString() + ... ) ;
} catch ( FileNotFoundException fnfe ) {
  throw new UmlGuiException( fnfe.toString() + ... ) ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

These steps are encapsulated into an openXMLDocument method that takes a single String argument, strFilePath, which is the full URL of the XML file to open on the server from which the applet was loaded. (Applet method getCodeBase().getHost() is very handy for getting the name of that server.)

Once students have a Document object available to their applets, they can use the various methods in the XML Standard API to navigate the document and extract data with which to populate GUI controls (see Figure 2). They have the full power of the XML DOM available, and they can manipulate the data in memory in any way they wish. The only problem is that they can't store that data back onto the server and thus make it persistent. To do that, they need to make a different type of request.

3 Storing XML Documents on an Applet's Server

It is commonly understood that Java applets run in a "sand box" and security restrictions explicitly prohibit them from reading and writing files on the client system. However, security restrictions also prohibit applets from writing data on the server. Therefore, the only way to get XML data back to the server is to send it to a Java servlet or JavaServer Page (JSP) and request that program to act as a surrogate for writing it to a persistent file (see Figure 3).

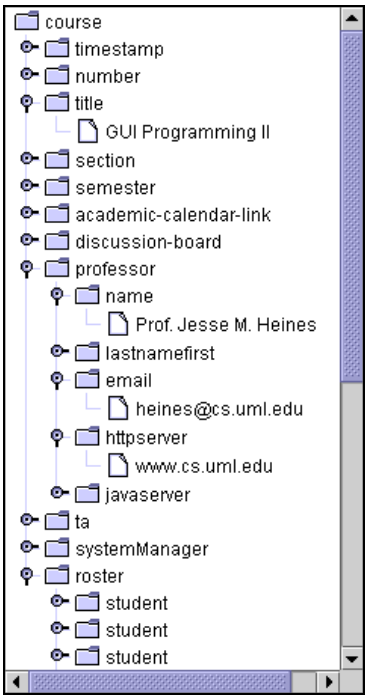

Figure 2. A Java Swing JTree control populated with data extracted from an XML file.

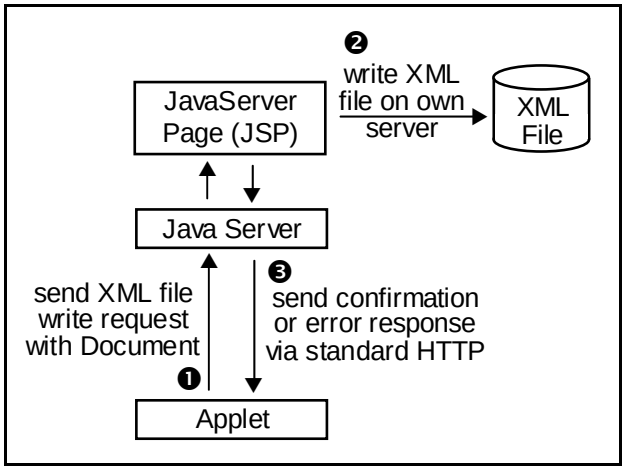

Figure 3. An applet requesting an XML document from the server from which that applet was loaded.

Note: We use JSPs rather than servlets simply for ease of development, because the server automatically recompiles JSPs after any revision and there is no need to deal with deployment descriptors.

Given the size of an XML file, it is not practical to send it via a GET request. You could send it via a POST request, but that means that the applet would have to call a page other than the one that loaded it, thus exiting the applet. This is a significant paradigm shift from reading the XML document, which can be done with straightforward state-

ments within the context of the applet itself.

To send the XML document without exiting the applet, we use HTTP tunneling [2]. This technique requires the structure shown in Figure 3. Most importantly, the applet must be loaded from a Java Web server that supports JSPs because a Java program is required on the server to receive the XML document sent by the applet.

3.1 Client-Side Algorithm

To make it as easy as possible for students to store their XML documents, all the necessary steps are encapsulate into a method named storeXMLDocument that takes three required parameters:

(1) a reference to the calling applet, specified simply by passing this
(2) a reference to the Document object to be stored
(3) a name to be used for storing the XML document

The reference to the calling applet is used to determine two critical pieces of information:

- the host from which the applet was loaded by calling *reference*.getCodeBase().getHost()
- the student's user name by parsing the String returned by calling *reference*.getCodeBase().toString() (when running an Apache server on Linux, the student's user name is the string between the ~ character and the first / that follows it)

Given these data, storeXMLDocument proceeds as follows [2].

(1) Construct a URL object for the server-side program to be called that will store the user's document (app is the parameter that gets the reference to the user's applet (this), and strServletURL is the relative URL (without the host name) to the storage JSP on the server side):

```
URL urlData ;
String strProtocol = "http" ;
String strHost = app.getCodeBase()).getHost() ;
int intPort = -1 ;
try {
   urlData =
       new URL( strProtocol, strHost, intPort, strServletURL ) ;
} catch ( MalformedURLException murle ) {
   throw new UmlGuiException( murle.toString() + ... ) ;
}
```

(2) Establish a URLConnection to the URL just constructed:

```
URLConnection urlcConn ;
try {
  urlcConn = urlData.openConnection() ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

(3) Disable browser data caching and enable the connection to send as well as receive data:

```
urlcConn.setUseCaches( false ) ;
urlcConn.setDoOutput( true ) ;
```

(4) Construct a ByteArrayOutputStream to buffer the data to be sent:

```
ByteArrayOutputStream baosByteStream =
   new ByteArrayOutputStream( 512 ) ;
```

(5) Construct an ObjectOutputStream:

```
ObjectOutputStream oosOut ;
try {
  oosOut = new ObjectOutputStream( baosByteStream ) ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

(6) Make a serialized version of the student's XML document (passed as parameter doc of type Document) [3]:

```
OutputFormat of = new OutputFormat( doc ) ;
of.setIndenting( true ) ;   // yields "pretty printing"
of.setIndent( 2 ) ;         // number of spaces indented per level
StringWriter sw = new StringWriter() ;
XMLSerializer xmlser = new XMLSerializer( sw, of ) ;
try {
  xmlser.serialize( doc ) ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

Note: Some readers may question why we don't serialize directly to the ObjectOutputStream. The answer is that doing so yields a java.io.OptionalDataException when reading the data with ObjectInputStream.readObject() on the server.

(7) Put the data to be sent into the output buffer:

```
try {
  oosOut.writeObject( strUserName ) ;
  oosOut.writeObject( strFileName ) ;
  oosOut.writeObject( strFilePath ) ;
  oosOut.writeObject( sw.toString() ) ;
  oosOut.flush() ;
  oosOut.close() ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

(8) Set the length of the output buffer:

```
urlcConn.setRequestProperty(
  "Content-Length", String.valueOf( baosByteStream.size() )) ;
```

(9) Do the actual data send:

```
try {
  baosByteStream.writeTo( urlcConn.getOutputStream() ) ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

The client-side applet now reads the server's response.

(10) Construct a BufferedReader from the URLConnection established in Step (2) above:

```
BufferedReader buffIn = null ;
try {
   buffIn = new BufferedReader( new InputStreamReader(
                urlcConn.getInputStream() ) ) ;
} catch ( FileNotFoundException fnfe ) {
   throw new UmlGuiException( fnfe.toString() + ... ) ;
} catch ( IOException ioe ) {
   throw new UmlGuiException( ioe.toString() + ... ) ;
```

```
}
```

(11) Read the server's response:

```
String strReturned = "" ;
String strLineRead = "" ;
try {
  while ( ( strLineRead = buffIn.readLine() ) != null )
    strReturned += strLineRead + "\n" ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

(12) Close the input buffer:

```
try {
  buffIn.close() ;
} catch ( IOException ioe ) {
  throw new UmlGuiException( ioe.toString() + ... ) ;
}
```

(13) Return the server's response to the calling statement:

```
return strReturned ;
```

3.2 Server-Side Algorithm

Even though server-side programs have fewer security issues than applets, there are still some. Most importantly, our Java Web server cannot write directly to student accounts. We therefore defined a special directory tree for uploaded student programs and defined a virtual path named students to point to it. This directory is owned by the same process that runs the Java Web server, so JSPs can write to it. Students can read from it directly by addressing files as http://*servername*/**students**/*username*/*filename*, and we provide facilities through standard HTML forms that allow them to upload files to or delete files from their personal subdirectories using the MultipartFormData class written by Walter Brameld [1].

When the storage JSP is connected to as described above, it

- reads the serialized XML document and other parameters
- determines where the document should be stored
- does the actual file write
- constructs and sends a response to the applet

The steps to accomplish this are as follows.

(1) Define variables so that they have global scope: (*Note:* Do not initialize variables here, as that them makes act like static variables, i.e., they are not reinitialized when the JSP is run repeatedly.)

```
<%!
  /** the XML document as a string */
  private String strDocument ;
  /** name of user to store data for */
  private String strUserName ;
  /** name of the file to write (no path allowed) */
  private String strFileName ;
  /** path to write file to */
  private String strFilePath ;
  /** response to return to caller */
  private String strResponse ;
  /** error message to return to caller */
  private String strException ;
  /** base for student uploads */
  private String strBase ;
  /** base where student accesses his/her uploaded files */
  private String strBaseURL ;
%>
```

(2) Initialize response to send back to the applet (this and all remaining steps except (12) are within <% %> pseudo tags):

```
strResponse = "" ;
strException = "" ;
```

(3) Construct an ObjectInputStream to read the incoming data:

```
ObjectInputStream oisIn = null ;
try {
  oisIn = new ObjectInputStream( request.getInputStream() ) ;
} catch ( IOException ioe ) {
  strException += "IOException ..." + ioe ;
}
```

From here on execution terminates if strException gets a value.

(4) Do the actual data read:

```
try {      // Hall & Brown, p. 1074
  strUserName = (String) oisIn.readObject() ;
  strFileName = (String) oisIn.readObject() ;
  strFilePath = (String) oisIn.readObject() ;
  strDocument = (String) oisIn.readObject() ;
  oisIn.close() ;
} catch ( ClassNotFoundException cnfe ) {
  strException = "ClassNotFoundException ..." + cnfe ;
} catch ( IOException ioe ) {
  strException += "IOException ..." + ioe ;
}
```

(5) Initialize the location of this student's uploads:

```
if ( System.getProperty( "os.name" ).
      equals( "Windows 2000" ) ) {
  strBase = "C:\\StudentUploads\\" + strUserName + "\\"  ;
  strBaseURL = "http://" + request.getServerName() +
        "/students/" + strUserName + "/" ;
}
else if ( System.getProperty( "os.name" ).equals( "Linux" ) ) {
  strBase = "/opt/JRun/servers/default/students/" +
        strUserName + "/" ;
  strBaseURL = "http://" + request.getServerName() +
        "/students/" + strUserName + "/" ;
}
```

(6) Declare variables for file writing:

```
String strUserPath = "" ;        // path to user file on server
File filePath = null ;       // File object for directory path
File fileXML = null ;            // File object for XML file
FileOutputStream fosStream = null ;     // for writing XML file
PrintWriter pwWriter = null ;   // for writing XML file
```

(7) Construct the path specified by the user:

```
StringTokenizer st = new StringTokenizer( strFilePath, "/" ) ;
while ( st.hasMoreTokens() ) {
  if ( System.getProperty( "os.name" ).equals( "Windows 2000" ) )
    strUserPath += st.nextToken() + "\\" ;
  else if ( System.getProperty( "os.name" ).equals( "Linux" ) )
    strUserPath += st.nextToken() + "/" ;
}
```

(8) Set the path specified by the user, creating it if it does not exist:

```
if ( ! filePath.exists() )
  if ( ! filePath.mkdirs() ) {
```

```
        strResponse += "Unable to create directory " ;
        strResponse += strBaseURL ;
        if ( strFilePath != null && strFilePath.length() > 0 )
            strResponse += strFilePath + "/" ;
    }
```

(9) Set the file specified by the user, creating it if it does not exist:

```
fileXML = new File( filePath, strFileName ) ;
try {
   fosStream = new FileOutputStream ( fileXML ) ;
} catch ( FileNotFoundException fnfe ) {
   strException += fnfe.toString() ;
} catch ( SecurityException se ) {
   strException += se.toString() ;
```

(10) Create a PrintWriter from the FileOutputStream and write to it:

```
pwWriter = new PrintWriter( fosStream ) ;
pwWriter.print( strDocument ) ;
pwWriter.close() ;
```

(11) Construct the URL to the created file to return to the user:

```
strResponse += strBaseURL ;
if ( strFilePath != null && strFilePath.length() > 0 )
   strResponse += strFilePath + "/" ;
strResponse += strFileName ;
```

(12) Send the response to the user:

```
<%= strResult.length() > 0 ? strResult : strException %>
```

4 FUTURE WORK

As currently implemented, the UmlGui package described in this paper has no security to prevent users from gaining access to and even overwriting other users' files if they choose to "fake" their user names. We did not consider such security critical to implement in this first phase of development since the package is being used only for educational purposes. Clearly, security would be required in a less collegial environment and should be added.

We know that the inability to write directly to a student's account is related to the fact that we deploy our Java server as a non-privileged process. We are currently investigating whether a switch from Macromedia's JRun to Apache's Tomcat will allow us to run privileged without compromising system integrity and thus make it possible to write directly to user accounts. Doing so would also solve the security problem just described.

Despite these issues, the software in its present form has proven invaluable in the author's GUI programming classes. The entire package is freely available for download from the author's Web site at teaching.cs.uml.edu/~heines.

5 REFERENCES

[1] Brameld, Walter. The MultipartFormData Java Class. users.boone.net/wbrameld/multipartformdata (2001).

[2] Hall, M., and Brown, L. *Core Web Programming*, pp. 1083-1090. Prentice Hall, Saddle River, NJ, 2001.

[3] Sosnoski, D.M. XML in Java: Java document model usage. www-106.ibm.com/developerworks/xml/library/ x-injava2/?dwzone=xml (2002).