The IStudio Environment: An Experience Report

Andrea H. Skarra, Karrie J. Hanson, Gerald M. Karam, and Jeff Elliott*

AT&T Labs – Research, Florham Park, NJ 07932

*Telepresence Systems, Inc., Toronto, Ontario M5A 3S5**

{ahs,karrie,karam}@research.att.com, jeffe@telepres.com

1. Introduction

IStudio is an application development environment based on Java^{TM} [1], XML, and XSL technologies, for end-user Internet applications (also called services). In particular, IStudio supports the development of reusable-component based applications that may be accessed through more than one kind of end-user device, such as a PC or laptop (XHTML), a wireless device (WML), or a telephone (VoiceXML^{TM} [2]).

The objective of the IStudio approach is to support service creation and to reduce development time through the definition of reusable and extensible application components. With IStudio, reuse of components can occur within a single application, among several different applications, or it may consist of reusing an entire application through a standardized interface. Further, IStudio supports a design methodology in which the definitions of presentation and business logic are largely separate, facilitating their implementation by separate groups of people as well as their reuse as separate, independent software entities. Finally, the approach represents the application of object-oriented design and other software engineering techniques to the development of Internet applications.

Commonly, presentation is interleaved with business logic, such that they cannot be easily reused separately. For example, in traditional CGI or servlet technology, presentation content is intertwined with code, making it difficult to separate the two and reuse either portion. Solutions like ePerl (www.engelschall.com/sw/eperl), PHP (www.php.net), and JavaServer Pages^{TM} (JSP^{TM}) produce cleaner versions of code intertwined with content, but still, the intermingling hinders separate reuse. Further, JSP effects only page generation; all backend software must be additionally developed. Enterprise JavaBeans^{TM} as a framework for business logic goes the furthest in separating the code from presentation, but even so, it does not structure presentation and business logic as reusable combinations. Moreover, EJB requires a significant programmer investment, as it's still relatively low level for building components.

Cocoon (xml.apache.org/cocoon) transforms XML data documents into presentation through a series of XSL transformations; structure and logic is isolated in the XSL transforms. It requires XSL expertise, which may constrain its ease of use, and XML interfaces to data sources, which are currently limited. Further, XSLT performance may affect scalability. Cocoon does not really address reusability or sharing of subsystems. Velocity (jakarta.apache.org/velocity) supports some separation of presentation and logic, but it still involves embedded scripting in presentations and provides little by way of an application environment, reusable parts, or data management. XMLC (xmlc.enhydra.org) provides a rudimentary form of presentation and logic separation: it transforms an XML or HTML document into a java class as a DOM object that can be modified through methods in the class. Presentation is transformed a priori, restricting what can be done dynamically at runtime. It provides no other support for page or application reuse. Other component-based approaches involving XML that separate presentation from business logic have been reported [1,2].

In our approach, a developer uses XML to specify a service (presentation, business logic, data, and configuration) and a suite of IStudio tools to generate a collection of objects that implements the service. The tools are based on XML's surrounding toolkit (XSL, parsers, and conventions). The power of XML and its toolkit made it easy to machine-process the specifications and generate Java code, to automatically create documentation, to intermix our specification language with other XML-based presentation languages (such as XHTML, WML, and VoiceXML), and to use a common syntax for all aspects of our environment.

2. System Overview

The IStudio system consists of the following:

- A specification language for service definitions
- A set of utilities for the following:
  - translation of service specifications into persistent objects with interpretable code

[1] Java, Enterprise JavaBeans, JavaServer Pages & JSP are trademarks of Sun Microsystems Inc.

[2] VoiceXML is a trademark of the VoiceXML Forum.

  - initialization, installation and maintenance of the service objects and code
  - creation of visual browser-displayable documentation
- A runtime engine that interprets service code and responds to user requests
- A collection of pre-defined components, applications, and presentation content that can be reused and extended.

Figure 1 illustrates the execution model of a service developed in IStudio. Clients access the service from a variety of devices via a web server, which contacts an application server. The application server contains the persistent objects that implement the presentation, data, and business logic of the service. The objects respond to service requests from the client web server, possibly sending messages to other objects or other systems to assemble the content of the response. The content is returned to the client for audio or visual display.

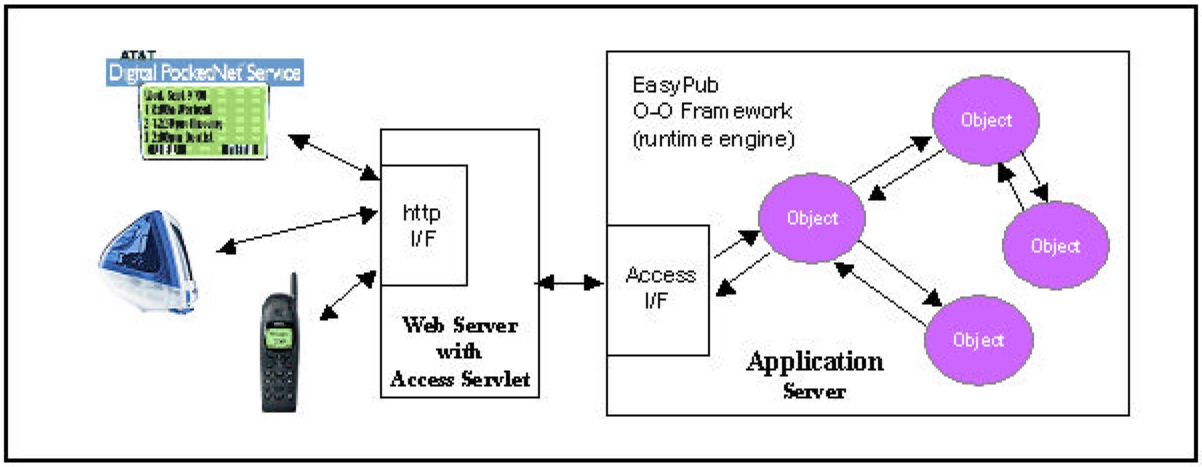
AT&T
Digital PocketNet Service
http I/F
Web Server with Access Servlet
EasyPub O-O Framework (runtime engine)
Access I/F
Object
Object
Object
Object
Application Server

Figure 1: A runtime model of the IStudio environment

2.1. Runtime engine

The runtime engine is the EasyPub system, Telepresence Systems Inc., Toronto, Canada. It includes a Java Virtual Machine, an object naming service, a component model in the style of Enterprise Java Beans, and database modules for Oracle and mySQL.

2.2. Specification Languages

Typically, a service specification includes the definition of presentation, business logic, data, configuration, and documentation. In IStudio, the specification entities are components, fragments, and a configuration file. Components primarily contain the business logic and data definitions, while fragments primarily define the presentation. Documentation may appear in both components and fragments, and the configuration is specified in the configuration file.

2.2.1. Components. A component specification defines a set of query and/or update actions for its instantiated objects. For each action, it defines the output result and presentation element(s), including any required computation. It may also override the actions and methods that it inherits from its superclasses. The specification is an XML document that maps actions to output presentation elements, designating any Java methods that are required for computation. The document also defines local state variables and other data entities, such as database tables, as well as annotations for component documentation. The Java methods can be defined within the XML document using CDATA elements, or as a separate class file created, for example, with a Java editor.

A component definition is partially shown in Figure 2, namely a component named **`TRWEB`** that manages an XHTML interface for an application. It defines the allowable actions that objects of this component can process (i.e., the `<actionList>` elements) together with mapping rules that define how actions cause processing, such as database queries, and how the results are to be returned as output (i.e., the combination of the `<formatList>` and the `<outputList>` elements).

2.2.2. Fragments. A fragment represents a display template for the results of an action invocation. Fragments consist of elements from a native XML presentation language (XHTML, WML, or VoiceXML) interwoven with elements from the IStudio namespace, which specify how the native elements are to be dynamically altered for display. The engine processes a fragment as follows: it interprets the IStudio namespace elements to invoke methods on other objects, to embed other fragments or information from database queries, to bind attribute values in native presentation elements, etc., while

```
<componentDef name="TRWEB" class="transferWeb" superClass="iText">
   <actionList>
      <action actionList="login addUser newUser"/>
      <action actionList="validatedUser" process="validatedUser"/>
   </actionList>
   <formatList>
      <format actionList="login addUser newUser addedUser"/>
   </formatList>
   <outputList>
      <defaultOutput process="generateLayout"/>
   </outputList>
 </componentDef>
```

Figure 2: A partial definition for component "TRWEB"

```
<is:fragment name="body">
  <form method="post" action="validateUser">
    <is:attr name="action">
      <is:link objAlias="TransferTable" clearParams="true">
        <is:param name="action"><is:content/></is:param>
      </is:link>
    </is:attr>
    <table>
      <tr><td>Cellular # (10 digits):</td>
        <td><input type="text" name="userID" size="10" value="">
            <is:attr name="value"><is:temp name="userID"/></is:attr>
        </input></td></tr>
    </table>
    <p><input type="submit" name="submit" value="SUBMIT"/></p>
  </form>
 </is:fragment>
```

Figure 3: A fragment named "body" that defines an XHTML form

```
<serviceDef name=".com.att.TransferPlus">
   <componentList>
      <component name="TRWEB" source="transferWeb"/>
      <component name="OPTLIST" source="optionList"/>
   </componentList>
   <objectList>
      <object name="TransferWeb" component="TRWEB">
         <setGroup group="common" format="shared"/>
         <setGroup group="login" format="login"/>
      </object>
      <object name="PhoneList" component="OPTLIST">
         <setGroup group=".IStudio.OptionList.optionList" format="shared"/>
         <setGroup group=".IStudio.OptionList.multipleSelected" list="singleSelected"/>
      </object>
   </objectList>
</serviceDef>
```

Figure 4: The configuration specification for a service named “TransferPlus”

it simply passes the native presentation elements (post attribute-binding) through for display.

Figure 3 shows a fragment named **`body`** that defines an XHTML form used in a document that collects login information. The `“is:”` elements are defined by the `IStudio` namespace; e.g., the `<is:attr name= “action”>` subelement of `<form>` binds the value of the form’s `“action”` attribute to a computed URL.

2.2.3. Configuration. A configuration specifies a service instantiation. It is an XML document that designates the service name, the set of components, the set of objects, and for each object, the associated component class, fragments, data initialization, and any aliases within the naming service.

For example, Figure 4 shows portions of a service configuration that uses the **`TRWEB`** component and the **`body`** fragment. It specifies two objects: the first is an XHTML presentation manager and the second manages a dynamically constructed list of options, like an XHTML `<select>` element. Each `<object>` element causes the named object to be instantiated and configured according to `<object>`’s sub-elements. The `<setGroup group="login" .../>` element associates the first object with the group of fragments that contains **body**. The element `<setGroup group=".Istudio.Option-List.optionList" .../>` associates the second object with fragments defined by a different application (**OptionList**) and shows reuse. We use a reverse domain name scheme for applications and their parts; the leading "." denotes an absolute path.

2.3. Utilities

The IStudio utilities are implemented as XSL, Java, and Perl scripts that are invoked directly from the command line or from within a makefile. Figure 5 illustrates the various utilities.

The utilities perform the following functions: parsing and transformation of the XML and Java in component specifications into Java classes; translation of the fragments to interpreted EasyPub elements or compilation of fragments into Java classes; generation of service documentation through extraction of structure, contents and explicit annotations from the service specifications; initialization of each service directory with the required files; initialization and maintenance of each service makefile; compilation of Java and other forms of source code included in the makefile (e.g., C-code); and configuration and installation of the service into the engine.

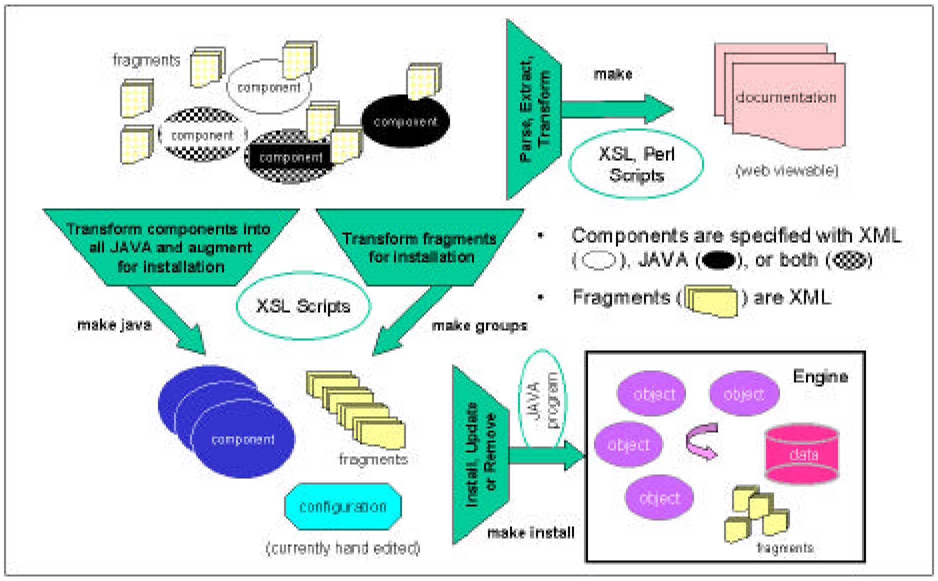

Figure 3: The suite of IStudio utilities

3. Conclusions

XML and its associated structure have helped enormously in building our development environment. The use of XML for specifications, XML namespaces for annotations of XML presentation languages, XSL for compilation, and XML and XSL for documentation generation, have enabled us to be more productive and to simplify the concepts that users of the environment need to learn.

We have chosen not to exploit the often-touted use of XSL to transform application data into presentation (e.g., to take a database result and produce XHTML, WML, or VoiceXML through a stylesheet). There are two key reasons for this. First, when used for presentation devices where the transformation is performed at a gateway (as in WML and VoiceXML) the cost of doing the transformation, even if compiled, puts a significant load on the gateway, thus substantially increasing the cost. Run-time server-side transformation is the same. Only when the transformation is done on a private desktop client are performance scalability less of an issue. Second, XSL approaches do not account for the variability among user interface scripting in different presentation mediums (e.g., visual vs. audio, PCs vs. hand-held devices). In these cases, the same data content may need to be presented over a different collection of pages and in different orders.

4. References

[1] G. Pour, "Enterprise Java Beans, Java Beans & XML, Expanding the Possibilities for Web-Based Enterprise Application Development", *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS)*, 31, 1999, pp. 282-291.

[2] M. Sweeney, "BUS: a Browser Based User Interface Service for Web Based Applications", *Proceedings of First Australasian User Interface Conference, 2000*, pp. 102-109.