



A Framework for Multilingual Device Independent Web Sites

Marc J. Hadley
Chief Engineer
Chrystal Software
Key West
53-61 Windsor Road
Slough, Berkshire SL1 2EE
UK
email: marc_hadley@chrystal.-
co.uk
web: www.chrystal.com

Marc Hadley is the Chief Engineer for Chrystal Software in Europe. He has 10 years of experience in technical software development, mainly in the areas of communications and distributed systems and has spent the last 3 years working on the design and implementation of SGML and XML component management and delivery systems. He has been a speaker at a variety of international conferences and industry events and holds a doctorate in physics from the University of York in the UK.

Introduction

Today, many existing websites exhibit two common failings:

1. The content is targeted at a limited range of web access devices, typically visual web browsers running on traditional PCs. With the increasing proliferation of alternative web access devices, web site authors need to consider ways of presenting common content to a wide variety of devices and specialising that content to exploit the capabilities of the target device.
2. The content is available in only one language, usually English. According to a recent Forrester report: "63% of Fortune 100 web sites are available in English only" and "By 2004, \$3.3 trillion of goods and services (50% of all online sales) will occur outside the US... Firms that cling to English-only sites will cede their share of this business to more globally minded competitors..."

This paper describes a framework for creating multilingual, device independent web sites using XML and related standards as the enabling technologies. The key features of the framework are the separation of content and style and the use of parameterised dynamic document assembly to allow on-the fly construction of web pages in a target language styled for a particular device. The framework has been implemented as a Java Servlet.

Designing for Language and Device Independence

Typical HTML web pages consist of text and graphic elements comprising the substance of the page (we will call this content) convolved with decorative graphics and markup describing the layout and formatting of the page elements (we will call this style). Whilst human friendly, this convolution of content and style locks in the information and prevents it's re-use elsewhere. E.g. a set of sports scores in HTML format is difficult to use in creating a corresponding WML based sports score page.

Designing for device independence involves separating the content and style such that the same content can be used in pages formatted for any device. XML combined with XSLT provides this functionality when XML is used to store content in a device independent format and XSLT stylesheets are used to transform the XML based content into a device specific format.

Language independence takes this one step further. In the framework described herein, a web page becomes a template, devoid of all textual content. Prior to delivery the content is gathered in the desired language and passed on to the styling process described above.

Often, XSLT stylesheets are used to add textual and graphical elements to a page during the styling process, e.g. field names on a form or table column headings. If all styling is done in a single pass then there needs to be a stylesheet for each language and device combination, so if a web site needs to support five languages and four delivery formats then 20 stylesheets are required. An alternative described in this paper is to use a pipelined transformation consisting of two phases: language dependent styling followed by device specific styling. This approach requires one stylesheet per language and device supported, 9 instead of 20 in the preceding example, but is more processor and memory intensive.

The rest of this paper describes the design of a framework to implement a language and device independent web site and a sample web site built using the framework.

Dynamic Document Assembly Using SAX Streams

This section describes the design of a dynamic XML document assembly engine. The reader is assumed to be familiar with the simple API for XML SAX and the concept of SAX filters, see <http://www.megginson.com/SAX> for more information. Some knowledge of Java and HTTP is also assumed.

SAX Streams

A SAX stream consists of a SAX event producer, a chain of one or more SAX filters and a SAX event consumer. Typically, the SAX event producer will be an XML parser and the event consumer will be something that serialises the SAX events to a target device in some XML-like format (HTML or WML say). The intervening SAX filters may modify the SAX event stream in the following ways:

1. Remove events

If a filter removes events from the event stream then downstream filters and consumers never "see" the corresponding content or markup. E.g. if the first filter removes all **startElement** and **endElement** events when the element name is "title" then the rest of the filter chain behaves as if the source document contained no "title" elements at all.

2. Insert events

If a filter inserts events into the event stream the downstream filters "see" the new events as if they originated in the source document. E.g. if the first filter inserts a new **startElement** event at the start of the event stream and a new **endElement** event at the end of the event stream the rest of the filter chain behaves as if the source document were wrapped by a new top level element.

3. Modify events

If a filter modifies events the rest of the filter chain see's the modified events as if they were in the source document. E.g. a filter module could add a new dynamically generated ID attribute to each element - all subsequent modules would behave as if that attribute were in the source document.

Assembly Process

A SAX stream can be used to assemble a dynamic document in the following way:

- A source template is created as an XML document.
- The template contains instructions to SAX filters in the SAX stream encoded as XML elements.
- The source template is parsed and sent through the SAX filter stream.

Device-Independent Web Accessibility

- SAX filters remove their instruction events from the stream and insert the result of their processing as new events (i.e. the results are encoded in XML)
- The SAX event consumer serialises the SAX events back into a textual XML format.

For example, consider the following XML template:

```
<doc xmlns:datemod="datemoduri <date><datemod:insertdate/> </date><time><datemod:inserttime/> </time></doc>
```

This template embeds instructions to a fictional SAX filter that knows about the date and time. After processing by the filter the resulting serialised event stream would look as follows:

```
<doc xmlns:datemod="datemoduri <date><day>25</day><month>7</month><year>2000</year></date><time><hour>18</hour><min>15</min></time></doc>
```

Note that the embedded instructions have been replaced by the results of executing the instructions, in this case to insert the date and time in the locations indicated.

Using this in-place replacement scheme, one filter can produce instructions or input for another filter. Consider the following template:

```
<prefs><prefs:get xmlns:prefs="prefsuri <param:get xmlns:param="paramuriuserid</param:get> <prefs:get> </prefs>
```

After the parameter module the event stream might look like:

```
<prefs><prefs:get xmlns:prefs="prefsuri marc <prefs:get> </prefs>
```

After the preferences module this might be reduced to

```
<prefs><lang>en</lang><topic>xml</topic></prefs>
```

Clearly, the order of SAX filters is important in such a case.

Assembly Modules

For the purposes of the framework, an assembly module is a SAX filter that behaves in the way described above: it extracts instructions from the SAX event stream and replaces them with results encoded as SAX events.

The following assembly modules have been implemented or are envisioned:

- **Parameter**

Retrieves HTTP request parameters or headers.

- **Profile**

Retrieves user profile information.

- **RDBMS**

Reads data from a relational database.

- **XML Repository**

Inserts data from an XML repository.



- **XSLT**

Transforms event stream based on XSLT stylesheet.

- **Switched XSLT**

As above, but switches stylesheet based on some parameter, e.g. HTTP User-Agent.

- **Invoke**

Runs user defined script (in Java or JPython) on document subtree. A catch all assembly module for embedding any desired processing with a SAX stream.

Assembly Templates

The initial input to the assembly process is a template in the form of an XML document. This contains the following:

- Static boilerplate text and markup. For language independent web sites then only static markup will be present since the inclusion of text would tie the template to a single language.
- "Active" markup. This markup is subject to processing by assembly modules and is likely to be replaced with the results of executing the encoded instructions.
- A list of assembly modules and the order in which they should be placed. This list makes the template self describing and ensures that the correct modules in the correct order will be used for processing the template.

Consider the following simple template:

```
<?module com.chrysal.eclipse.Parameter?><html xmlns:paramns="paramuri <head><-title>Simple Template</title></head><body><p>Your browser claims to be <paramns:parameter key="User-agent"/>. </p></body></html>
```

This template requires one assembly module implemented by the Java class **com.chrysal.eclipse.Parameter** and includes one "active" element: **<paramns:parameter>** whose **key** attribute defines the HTTP header value to retrieve. Following assembly (and assuming the user is using IE5.01) this template would be transformed to read as follows:

```
<module com.chrysal.eclipse.Parameter?><html xmlns:paramns="paramuri" <head><-title>Simple Template</title></head><body><p>Your browser claims to be Mozilla/4.0 (compatible; MSIE 5.01; Windows NT) . </p></body></html>
```

A slightly more complex template including an XSLT transformation might be:

```
<?module com.chrysal.eclipse.Parameter?><?module com.chrysal.eclipse.XSLT href="/stylesheets/doc.xsl"?><doc xmlns:paramns="paramuri <title>Simple Template</title><p>Your browser claims to be <paramns:parameter key="User-agent"/> </para></doc>
```

Using the switching XSLT module, AgentSniff, we could make a device independent template to perform the same function:

```
<?module com.chrysal.eclipse.Parameter?><?module com.chrysal.eclipse.AgentSniff family="device"?><doc xmlns:paramns="paramuri <title>Simple Template</title><p>Your browser claims to be <paramns:parameter key="User-agent"/> </para></doc>
```

Device-Independent Web Accessibility

The switching XSLT module uses a configuration file to determine the parameter to switch upon and the stylesheet to use based on the value of that parameter. An example configuration file might look like:

```
<es:sniffdoc xmlns:es="sniffuri" <es:family name="device" fieldName="User-Agent
<es:member id="UPG" stylesheet="/stylesheets/doc_wap.xsl" mimetype="text/
vnd.wap.wml" /> <es:member id="Mozilla" stylesheet="/stylesheets/doc.xsl" mime-
type="text/html" /> <es:member id="default" stylesheet="/stylesheets/doc.xsl" mime-
type="text/html" /> </es:family> <es:family .../> </es:sniffdoc>
```

This configuration specifies that the **doc** family of stylesheets is switched on the **User-Agent** HTTP header. If the value of that parameter begins with "UPG" then the "doc_wap.xsl" stylesheet should be used and the MIME type set to **text/vnd.wap.wml**. If the value of the parameter starts with "Mozilla" then we use a different stylesheet and set the MIME type to HTML. Finally, if the value of the parameter doesn't match any of the other options then assume HTML is required.

As a final example we could make the template both language and device independent:

```
<?module com.chrystal.eclipse.Parameter?><?module com.chrystal.eclipse.Agen-
tSniff family="lang"?><?module com.chrystal.eclipse.AgentSniff family="devi-
ce"?><doc xmlns:paramns="paramuri" <device><paramns:parameter key="User-agent"/
></device></doc>
```

The first switched style sheet would insert the **<title>** element and transform the **<device>** element into a **<para>** element with the text "Your browser claims to be" in the desired language. The second style sheet would then transform this intermediate format into the desired device specific format as above.

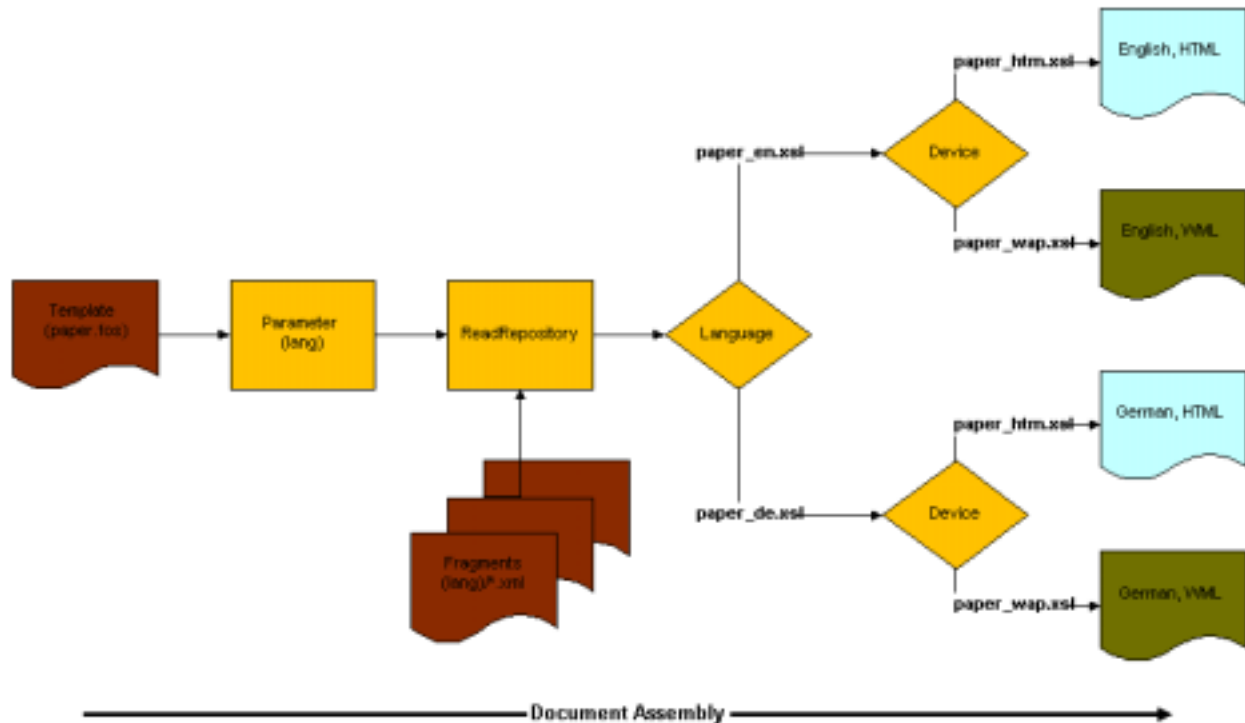
Sample Web Page

This section describes a sample web page illustrating the techniques described above. The page shows a list of abstracts of recent conference papers and is viewable using a standard HTML browser or a WML browser in either English or German.

For this site, content is brought in from an XML repository - in this case the file system. Each of the paper abstracts is stored in a separate XML instance with the language of the instance specified by its file path. So, the file /fragments/en/xml2000.xml contains the abstract for this paper in English and the file /fragments/de/xml2000.xml contains the abstract for this paper translated into German.

The desired language is specified using a HTTP parameter "lang" and the delivery format is determined by examining the HTTP header "User-Agent" as described above. This is illustrated in the diagram below:

Document Construction Schematic



The template for the page is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?><?module com.chrystal.eclipse.assembly.Parameter?><?module com.chrystal.eclipse.assembly.ReadRepository?><?module com.chrystal.eclipse.assembly.AgentSniff family="Language"?><?module com.chrystal.eclipse.assembly.AgentSniff family="Device"?><papers xmlns:rr="http://www.chrystal.com/eclipse/assembly/ReadRepository" xmlns:p="http://www.chrystal.com/eclipse/assembly/Parameter <rr:getObject> <rr:objectID> file:///fragments/<p:parameter key="lang" default="en"/> </rr:objectID> </rr:getObject> </papers>
```

The first assembly module extracts the value of the **lang** parameter such that the URL will become either **file:///fragments/en** or **file:///fragments/de**. The **getObject** function of the **ReadRepository** module traverses the specified directory and inserts the content of each file found. After this step the template will consist of a top level **<papers>** element with a number of children, one per file found in the specified directory. The text will all be in the desired language.

The next module executes an XSLT transform that adds in any language specific text such as user interface prompts for the WML version of the page.

The final transform takes the language specific XML and converts it into either HTML or WML depending on the users browser.