Powering Pipelines with JAXP

Nigel Whitaker, DeltaXML.com [http://www.deltaxml.com] (Monsell EDM Ltd) <nigel.whitaker@deltaxml.com>

Thomas Nichols

The JAXP API allows Java programmers easy access to the power and flexibility of XML parsing and filtering and XSLT transformation. However, while many programmers utilize JAXP for simple XML parsing or single-shot XSLT transformation, going further to construct processing pipelines often proves difficult.

Using JAXP to construct pipelines of processing elements is a good idea; it allows complex problems to be decomposed into a number of simpler steps or components and also, in theory, provides the ability to benefit from concurrent processing. With careful attention to issues such as avoiding disk IO and, through the use of SAX event streaming to avoid re-parsing XML data, pipelines can provide good runtime performance.

However, in practice the construction of pipelines is often a difficult process for Java programmers. For example, programmers are often unsure whether to construct a pipeline using XMLFilters, TransformerHandlers or both; they start off by adopting some existing example code and then run into problems.

Using experience gained assisting and supporting programmers when constructing JAXP pipelines, this paper presents classification schemes, diagrams and tables which try to explain the pipeline construction process. Examples will show the construction of both simple and advanced processing pipelines. We will also describe some commonly encountered issues and problems, such as: preserving the DOCTYPE declaration or entity references, preserving whitespace and controlling output indentation and then present solutions or workarounds to overcome limitations in the current pipeline architecture.

We also briefly explain some of the challenges we encountered creating new XML-centric software components which are designed to integrate with the existing JAXP pipeline components and describe the rationale for our subsequent design decisions. Finally we will review future developments and proposals for XML processing pipelines from the Java Community Process (JCP) and W3C.

Table of Contents

1. Introduction

The Java API for XML Processing (JAXP) [JAXP], provided as part of the J2SE platform allows programmers easy access to XML capabilities and tools. In this paper the benefits of XML processing pipelines, using SAX event communication, are addressed. While this API is powerful, it is also difficult for beginners to start using effectively. In order to aid understanding, we present a general model of SAX event pipelines to clarify some design issues.

1.1. Why Pipelines?

The pipeline metaphor is one solution to managing the complexity of software development in XML. There are a number of similarities to the pipeline model that was established in UNIX and other operating systems and supported by many programming and scripting languages, including C and Java. The basis of the metaphor is to create simple software components that do fairly simple tasks well and efficiently. These components could be used independently but are also designed to communicate with other similar components in order to construct more complex and sophisticated systems.

The UNIX pipeline metaphor works by linking the `stdout` and `stdin` (to use C terminology; it is `System.out` and `System.in` in Java) of successive components together. In an XML pipeline a more optimal form of communication is in terms of SAX events. One component generates the SAX events, whilst another reads them by being a `ContentHandler`.

Constructing these types of pipeline provide a number of benefits to developers and software users:

- Reusable off-the-shelf pipeline components may already exist for a number of sub-tasks, saving development time.
- Smaller, simpler components tend to have simpler interfaces easing testing and reuse.
- Components may be easier to develop in multi-programmer teams as the pipeline enforces clean and simple interfaces between components.
- Code can run faster on multi-processor systems using concurrency.

1.2. Alternatives to Pipelines

While pipelines are often suitable for a number of tasks, there are occasions when developers or managers may find alternative approaches more attractive. There are costs associated with the pipeline metaphor; the intercommunication between components may require more CPU time and/or memory than designing a single special-purpose, dedicated component. There may also be architectural reasons for preferring a non-pipelined style of development. Two possibilities include:

- XSLT has internal mechanisms for reuse such as `xsl:import` and `xsl:include`. It is also possible to perform multi-pass XSLT transformation, perhaps using modes and temporary result trees. For certain kinds of processing the use of a single XSLT transformation rather than several pipelined steps may be preferred.
- It is also possible to perform a number of processing steps on a mutable in-memory tree representation such as W3C DOM or a similar representation with Java or database underpinnings.

1.3. What is JAXP?

JAXP is designed to facilitate the use of XML on the Java platform. It allows applications to parse and transform XML in a manner which is independent of particular parser or transformer implementations and also allows easy run-time switching between implementations. The JAXP API incorporates a number of existing APIs and technologies, which can be used independently and possibly with other languages and platforms, these include:

- The Transformation API for XML (TrAX) [TrAX]
- The Simple API for XML (SAX) [SAX]
- The W3C Document Object Model (DOM) [DOM]

Various JAXP versions have been standardized using the Java Community Process (JCP). Java Specification Request (JSR) 63 developed JAXP version 1.2 which is included in J2SE version 1.4. It is also possible to use JAXP with earlier versions of Java using a number of *add-on* jar files. JSR 206 is being used to develop JAXP version 1.3 which is planned for inclusion in the next major J2SE release. Some aspects of JAXP version 1.3 are discussed in Section 5.1, “JAXP 1.3”.

There are a number of systems that use the pipeline metaphor to link XML components which are not specifically JAXP. These include Apache Cocoon [COCOON], Apache Jelly [JELLY] and STnG [STnG]. These systems sometimes use pipelines for specific purposes (delivering web content in a servlet context in the case of Cocoon) or as parts of larger systems. In this paper we will concentrate on JAXP as it is a general purpose, yet fairly simple API, available to most Java users.

1.4. Pure JAXP and JAXP Compatibility

The JAXP API defines a number of standard pipeline components including Parsers and XSLT transformers/filters. There is also an extensible filter class (`XMLFilterImpl`) which allows Java programmers to create custom filters easily.

A number of third party components are also *JAXP compatible* and have been designed to operate as pipeline components. These components ensure compatibility by using JAXP classes, such as `SAXSource` or `StreamResult` as method arguments, or by extending JAXP classes such as `XMLFilter` or `ContentHandler`. Some of these components include:

- The DeltaXML Comparator [DXCORE] is a component which takes two XML trees (JAXP `Source` objects) as input and produces a *delta file*, which describes their differences, as a JAXP `Result` object .
- The DeltaXML Synchronizer [DXSYNC] takes three JAXP `Source` objects, the initial version of the XML tree and two edit trees. The `Result` object which is produced merges any independent changes made in the two edits and also highlights any conflicts.
- STX [STX] is an XSLT-like transformation language which avoids the storage of an in memory input tree used in many XSLT implementations. Like XSLT Transformers, it uses the JAXP Source and Result objects.
- The Apache XML Serializer [APSER] is a component which reads a SAX event stream and pro-

duces a textual XML file. It is typically used as the last element in a pipeline. It is JAXP compatible in that it implements the `XMLReader` and `ContentHandler` interfaces.

- The XInclude [XINC] Engine from Elliotte Rusty Harold includes an `XIncludeFilter` class which can be used as a pipeline component as it implements the JAXP `XMLFilter` class.

1.5. About this Paper

We have aimed the content of this paper towards programmers who are familiar with both Java and XML. The constraints imposed by print reproduction such as limiting column widths and the overall length have meant that we have had to shorten many of the code examples. We have neglected import statements and exception handling to save space and the indentation style is not ideal. In several places we have used ellipses "..." to indicate where code has been removed.

Complete code examples for a number of pipeline configurations together with sample test data and Ant scripts for compilation and running are available as part of the DeltaXML Pipeline Tutorial, available from http://www.deltaxml.com/pipelines/.

2. A Case Study: The SVG Comparator Pipeline

To demonstrate and introduce pipeline architectures we will use a system that we have constructed in the past. Our goal was to find a better way of displaying changes to SVG files. The use of animation was proposed and a pipeline was needed to generate this animated SVG.

The DeltaXML Comparator is a pipeline component which compares two XML trees and produces a third XML tree, the delta file, which describes the differences between the two inputs[1]. The core comparison engine is designed to be simple, and rather than add complexity to the core with new features and options we have chosen to provide these in terms of pipeline components. The following lists the requirements for an SVG comparison system that are added to the comparison engine using pipeline components:

1. The comparison engine is not restricted to either document-orientated or data-oriented XML and as such will only disregard whitespace which, through the presence of a DTD or schema, is classified by a SAX parser as *ignoreableWhitespace*. Some SVG instance files lack a suitable DOCTYPE to provide a DTD or perhaps the DTD is unavailable at the specified URL. When comparing two SVG files, indentation whitespace is not significant for identifying "graphical differences". One solution to this problem is to remove any unnecessary indentation whitespace, using a filter prior to comparing the data. This filter could be as simple as an XSLT identity transform, plus the use of `<xsl:strip-space elements="*"/>` or it could be coded in Java (see Section 2.5, "Optimized SAX Event Pipelines").

   This space-stripping, or space-normalizing, transform is not specific to SVG and can be used for different forms of data-oriented XML. Indeed for the SVG pipeline it was an existing off-the-shelf component that was simply reused.

2. The comparison engine by default treats all XML elements as ordered lists of child PCDATA and sub-elements. The engine also has the capability to treat the children of an element as an unordered *bag*. The selection of this orderless algorithm could have been implemented in a number of ways, such as having the comparator understand augmented schemas, to identify orderless elements. Another approach we could have taken would have been to use command line arguments to the comparator, specifying XPaths of elements which are to be considered orderless. Instead we have settled on using the component-oriented approach where the presence of a simple attribute, `deltaxml:ordered="false"`, in the input tree makes the comparison engine use the orderless algorithm for the children of that element. The addition of these attributes is achieved using a simple XSLT filter on the SVG data which is written with an understanding of the ordered or orderless nature of the various SVG elements.

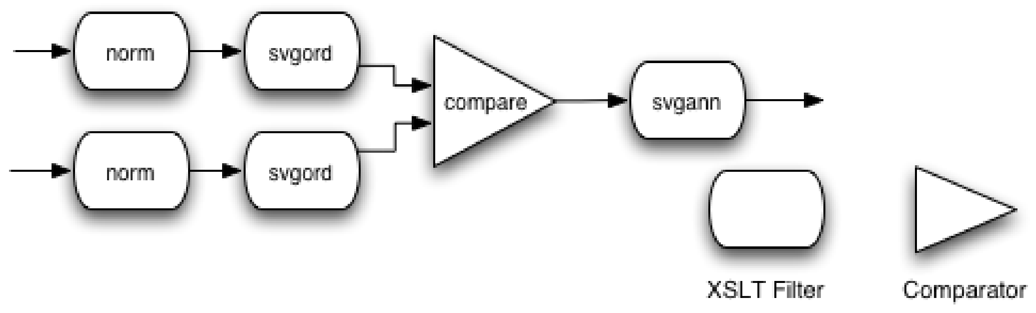
[1]Please see The DeltaXML website [http://www.deltaxml.com/core] for further details

This use of data specific filters means that the comparison engine does not need any knowledge of SVG specifically.

3. The final stage of the pipeline is one which processes the delta file and produces animated SVG which shows the differences between the two inputs. A "full context" delta file is essentially an SVG file containing the common parts of both inputs and where differences are present they are represented with DeltaXML delta elements and attributes. The SVG animation filter turns the DeltaXML delta elements and attributes back into plain SVG, but with the differences animated. The use of filters converting DeltaXML comparator-output back into pre-comparison data formats is a technique which we have applied to a number of different XML vocabularies.

Each of the input filters needs to be applied to both input files whilst the output filter is applied once to the delta file, as depicted in Figure 1, “Case-study pipeline configuration”.

Figure 1. Case-study pipeline configuration



We now look at various techniques which can be used to implement the requirements described above.

2.1. Using the Command-Line

We can use shell scripts or batch files to record a sequence of commands which would typically be typed at the shell prompt. For example[2]:

```
#! /bin/sh
SAXON="java -jar /usr/local/saxon6_5_3/saxon.jar"
COMPARE="java -jar /usr/local/DeltaXMLAPI-2_8_2/command.jar"
$SAXON norm.xsl f1.svg > f1n.svg
$SAXON norm.xsl f2.svg > f2n.svg
$SAXON svgord.xsl f1n.svg > f1o.svg
$SAXON svgord.xsl f2n.svg > f2o.svg
$COMPARE compare-raw f1o.svg f2o.svg f12.svg
$SAXON svgann.xsl f12.svg > f12ann.svg
```

Producing such a script requires minimal programming skills and is easy to develop and prototype. Also, the intermediate files can be examined on disk, aiding debugging. However it is hard to work in an implementation-independent manner[3]. For example should the XSLT processor be changed to Xalan-J, command line argument orders and flags would need to be changed.

2.2. File to File with JAXP

When using JDK 1.4 we can make use of the built-in JAXP implementations and not have to worry

[2]These examples have been simplified, more realistic code would provide error/exception handling and would also used parameterized, rather than hardwired, filenames
[3]Apache Ant provides processor independence and a number of other advantages over shell/batch scripts or Makefiles

about which processor is used, nor where its jar file is located in our file system, nor how to specify its parameters. For example[2]:

```
TransformerFactory tf= TransformerFactory.newInstance();
Transformer norm= tf.newTransformer(new StreamSource("norm.xsl"));
norm.transform(new StreamSource("f1.svg"),
               new StreamResult("f1n.svg"));
norm.transform(new StreamSource("f2.svg"),
               new StreamResult("f2n.svg"));
...
XMLComparator c;
c= XMLComparatorFactory.newInstance().newXMLComparator();
c.compare(new StreamSource("f1o.svg"),
          new StreamSource("f2o.xvg"),
          new StreamResult("f12.svg"));
...
```

The above code could be compiled and run on a J2SE 1.4 system and use the platform default XSLT processor.

Although not shown in the above example code, it is also possible to have finer-grain control over any errors or exceptions thrown. For example, it is possible to differentiate between errors present in the XSLT script versus those in the transformer input. Nested exceptions can be also be unrolled to report circumstances such as file access control problems to the user.

2.3. JAXP In-Memory

Instead of using disk files to store the intermediate results, it is also possible to store them in an in-memory buffer. The following code shows how two XSLT transformation stages can communicate in this manner:

```
Transformer norm= stf.newTransformer("norm.xsl");
Transformer svgord= transFact.newTransformer("svgord.xsl");
ByteArrayOutputStream f1pass1= new ByteArrayOutputStream();
norm.transform(new StreamSource("f1.svg"),
               new StreamResult(f1buf));
ByteArrayInputStream f1pass2=
    new ByteArrayInputStream(f1pass1.toByteArray());
svgord.transform(new StreamSource(f1pass2),
                 ...)
```

Reading and writing in-memory buffers is usually faster and more reliable than disk IO. This solution should be faster, as reading and writing in-memory buffers is usually faster than to and from disk. It should also be more reliable as events such as disks becoming full, or failing, cannot occur. However its main drawback is that the intermediate results need to be stored in Java heap memory which increases the overall memory requirements of the system.

2.4. JAXP with SAX Events

Linking components which produce and consume XML using SAX events is more efficient than using raw XML files as serialization and parsing can be avoided. The following code demonstrates how the components, filters for SVG animation and ordering and XML normalization used in our case study can be linked together[42]:

```
SAXTransformerFactory stf= ...;
Templates norm, svgord;  // precompile xsl for use in 2 filters
norm= stf.newTemplates(new StreamSource("norm.xsl"));
svgord= stf.newTemplates(new StreamSource("svgord.xsl"));
XMLFilter norm1, norm2, svgord1, svgord2;
TransformerHandler svgann;
norm1= stf.newXMLFilter(norm);
norm2= stf.newXMLFilter(norm);
```

[4]The details of the JAXP code used here will be explained in more detail later

```
svgord1= stf.newXMLFilter(svgord);
svgord2= stf.newXMLFilter(svgord);
svgann= stf.newTransformerHandler(new StreamSource("svgann.xsl"));
svgord1.setParent(norm1); // link filters
svgord2.setParent(norm2); // link filters for 2nd input
svgann.setResult(new StreamResult("f12ann.svg"));
c.compare(new SAXSource(svgord1, new InputSource("f1.svg")),
          new SAXSource(svgord2, new InputSource("f2.svg")),
          new SAXResult(svgann));
```

This approach uses SAX events to communicate between the subcomponents. Whilst this code may be more complicated it should be faster (there is no CPU time needed for unnecessary serialization or reparsing of data) and use less memory than the techniques used in the previous section.

The precise memory requirements are determined by the implementations used for the various stages. While the above code may avoid storing the intermediate results as XML files in memory buffers, it is possible that intermediate result trees are still required in memory. Various components such as Xalan-J, Saxon and the DeltaXML Comparator currently use in-memory input tree representations. These are typically read-only data structures optimized for the operations performed by the subcomponent. Some alternative technologies may not need to store the entire input data in memory, for example STX [STX]. STX is an XSLT like transformation language which is designed for streaming.

2.5. Optimized SAX Event Pipelines

While the above approaches provide fairly good performance there are some implementation details of XSLT processors, that give problems with memory consumption. We have suggested STX as a possible solution, but another is to go *"under the hood"* and code filters in Java. One technique which can be used for *pull-mode filters*[5] is to code them directly in Java for the best CPU performance and smallest memory footprint. For example consider the `norm.xsl` filter, this could simply be an XSLT identity transform and a `strip-spaces` statement. This could however be replaced by the following Java class:

```
class WhitespaceFilter extends XMLFilterImpl {
  public void characters(char[] ch, int start, int length)
        throws SAXException {
      // remove intra element whitespace-only nodes
      if (!new String(ch, start, length).trim().equals(""))
        super.characters(ch, start, length);
    }
  }
  public void ignorableWhitespace(char[] ch, int start, int length)
        throws SAXException {
    // ignore if no DTD/schema is specified and not parser stripped
  }
}
```

Similarly the other filters used in this case study could be replaced by Java code for best performance. The cost/benefit of doing so depends on the nature of the filtering being performed (certain uses of XSLT such as: `//element` are difficult to perform without a complete in-memory tree), the expected input data sizes, the number of times a filter is used, the Java vs XSLT abilities of the development team.

3. A General Model of JAXP Pipelines

In this section we introduce our general model for JAXP based SAX event pipelines. We are providing this because many of the existing examples have a single focus or theme and thus users do not learn the intricacies of the various techniques which can be used to construct a general pipeline. Existing pipeline examples include:

[5]This term will be defined in Section 3, “A General Model of JAXP Pipelines”

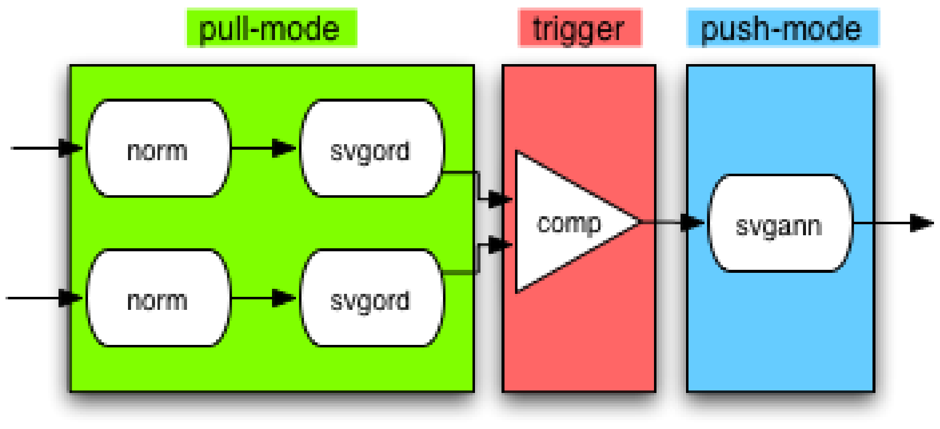
- Pipe.java from the Xalan-J samples - uses TransformerHandler
- UseXMLFilters.java from the Xalan-J samples - uses XMLFilter
- exampleXMLFilterChain from Saxon's TraxExamples.java - uses XMLFilter
- Concatenating Transformations with a Filter Chain from the J2EE Tutorial [http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXPXSLT8.html]

A general SAX event pipeline consists of a number of processing stages which communicate with each other using SAX events, these stages can be classified into one of 3 types:

Triggered	A triggered stage initiates the actual processing usually by calling method such as `parse()`, `transform()` or `compare()`.
Pull-mode	A pull-mode stage is one which logically appears before a triggered stage, and from which the triggered stage, directly or indirectly, pulls SAX events.
Push-mode	A push-mode stage is one which logically appears after a triggered stage, and to which the triggered stage, directly or indirectly, pushes SAX events.

A SAX event pipeline always contains a triggered stage. It may contain zero, one or more pull-mode stages and zero, one or more push-mode stages. The SAX event implementation of the case study pipeline (see Section 2.4, “JAXP with SAX Events”) is depicted graphically in Figure 2, “Case study pipeline classification”.

Figure 2. Case study pipeline classification



A pipeline needs to be first configured so that all the stages are linked together and then invoked by triggering one of the interlinked stages. The following sections describe some appropriate code patterns which conform to this model.

3.1. Using Triggered Components

A triggered component is one which is invoked directly by calling an appropriate method. Some classes/methods include:

- `Transformer.transform(Source, Result);`
- `XMLReader.parse(InputSource);`
- `XMLComparator.compare(Source, Source, Result);`

Although `XMLReader.parse(InputSource)` at first glance appears to be associated with parsing, it is also callable in a number of circumstances where is it not the first component of a pipeline. This is because a number of JAXP classes such as `XMLFilter` extend the `XMLReader` class. The Xalan-J UseXMLFilters sample program uses a pipeline in this manner. Three of the components used in the pipeline are XMLFilters. The last filter in the chain is triggered by calling its parse method. The argument to the parse method is an `InputSource` which describes the input supplied to the start of the pipeline. The triggered component is linked to the previous pull-mode stages using the `setParent()` method. For example, consider the case where the third and final XMLFilter in a pipeline triggers the processing:

```
xmlFilter3.setParent(xmlFilter2);
...
xmlFilter3.parse(new InputSource("f1.xml"));
```

An alternative but similar pipeline could have been constructed using the `Transformer` class. In this case the invoked method is `transform()` and the pipeline construction is achieved using a SAXSource argument to the transform method. The SAXSource specifies both the pull-mode filter which sends the input SAX events and also the input supplied to the start of the pipeline, for example:

```
XMLFilter stage2= ...;
Transformer stage3= stf.newTransformer("foo3.xsl");
stage3.transform(new SAXSource(stage2, new InputSource("foo.xml")),
                 new StreamResult(System.out));
```

This style of configuring a three stage filter chain has the benefit of using standard JAXP `StreamResult` class, whereas the Xalan-J sample code, using a chain of XMLFilters, needs a `ContentHandler`. As such a serializer is not available as part of JAXP, the implementation-specific, Apache XML serializer is needed to convert the SAX events produced by the final stage into a file.

3.2. Pull-Mode Components

A pull-mode component is one which provides SAX events to a triggered component. It does this by implementing or extending the `XMLReader` interface and providing suitable implementation methods. An upstream pull-mode or a triggered pipeline stage would then use perhaps a `setParent()` method or `SAXSource` constructor on an object which implements or encapsulates the `XMLReader` class.

A pull-mode component may be an take a number of forms depending upon how it produces the SAX events which it supplies on its output, for example:

- A `SAXParser` reads a textual XML input stream parses, and optionally validates and augments, the input and supplies SAX events.
- An `XMLFilter`, or rather an implementation of this interface, receives an input SAX event stream and filters/modifies this stream to produce the output SAX event stream. The upstream source of SAX events is specified using the `XMLFilter.setParent(XMLReader)` method. The use of the `XMLReader` argument type as a source of SAX events is consistent and completes the orthogonality of the filter.
- The DeltaXML `XMLSynchronizerFilter` is similar to the JAXP standard `XMLFilter` in that it produces a stream of SAX events as output. However rather than take a single stream of SAX events as input, it takes 3 streams of SAX events and synchronizes them together to produce the single output. Instead of providing a `setParent(XMLReader)` method, it provides the following methods:
    - `setBaseParent(XMLReader)`
    - `setEdit1Parent(XMLReader)`
    - `setEdit2Parent(XMLReader)`

3.3. Push-Mode Components

These components implement the `ContentHandler` interface in order to be integrated in SAX event pipelines. In order to integrate a push-mode component with a triggered pipeline stage the ContentHandler is passed directly, or indirectly, to a suitable method or class. In the following example, the triggered stage (`reader`) is linked to a push-mode `TransformerHandler` (`push1`) through the `setContentHandler()` method prior to triggering the pipeline with the `parse()` method.

```
XMLreader reader= ...;
SAXTransformerFactory stf= ...;
TransformerHandler push1=
   stf.newTransformerHandler(new StreamSource("push1.xsl"));
reader.setContentHandler(push1);
reader.parse(new InputSource("f1.xml"));
```

Another technique to link a push-mode pipeline is through the use of the `SAXResult` constructor which takes a `ContentHandler` argument. The following example demonstrates this integration:

```
SAXTransformerFactory stf= ...;
Transformer stage2; // triggered stage
TransformerHandler stage3;  // push stage
stage2= stf.newTransformer(new StreamSource("stage2.xsl"));
stage3= stf.newTransformerHandler(new StreamSource("stage3.xsl"));
stage3.setResult(new StreamResult("f2.xml"));
stage2.transform(new SAXSource(...),
                 new SAXResult(stage3));
```

The example above also shows how the output push-mode can generate its output. The `setResult()` method is used with the JAXP `TransformerHandler`. Note that some "end-mode" components such as the `XMLSerializer` do not support this method.

Note

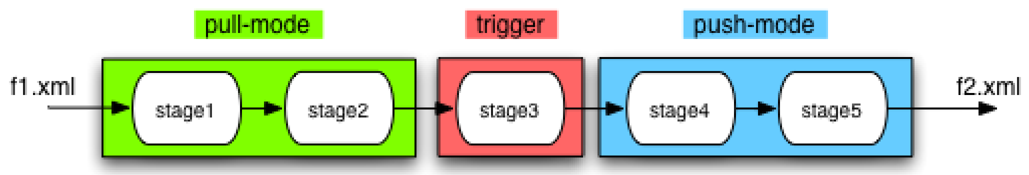
We believe the `Transformer.transform()` method, with its use of the abstract `Source` and `Result` argument classes, to be more flexible than `XMLFilter` and its associated methods, as well as being easier to understand.

3.4. A Complete Example

The following code demonstrates how some of the above techniques can be used to construct a general five stage pipeline, as depicted in Figure 3, “A five stage pull-trigger-push pipeline”. This pipe consists of two pull-mode filters, a triggered filter and then two push-mode filters:

```
SAXTransformerFactory stf= ...;
XMLFilter stage1, stage2;
Transformer stage3;
TransformerHandler stage4, stage5;
stage1= stf.newXMLFilter(new StreamSource("stage1.xsl"));
stage2= stf.newXMLFilter(new StreamSource("stage2.xsl"));
stage3= stf.newTransformer(new StreamSource("stage3.xsl"));
stage4= stf.newTransformerHandler(new StreamSource("stage4.xsl"));
stage5= stf.newTransformerHandler(new StreamSource("stage5.xsl"));
stage2.setParent(stage1);
stage4.setResult(new SAXResult(stage5));
stage5.setResult(new StreamResult("f2.xml"));
stage3.transform(new SAXSource(stage2, new InputSource("f1.xml")),
                 new SAXResult(stage4));
```

Figure 3. A five stage pull-trigger-push pipeline



3.5. Implementation Summary

A number of JAXP and JAXP-compatible components have been classified according to our model and are summarized in Table 1, “Pipeline component classification”.

Table 1. Pipeline component classification

component	triggering methods	pull-mode class	push-mode class
SAX/Java	XMLReader.parse()	XMLFilter[a]	not available
XSLT	Trans-former.transform()	XMLFilter[b]	TransformerHandler
DeltaXML Core	XMLComparat-or.compare()	XMLComparatorFilter[c]	not available
DeltaXML Sync	Synchronizer.sync()	XMLSynchronizerFil-ter[d]	not available
Apache XML Serializer	not available	not available	ContentHandler[e]
STX	Trans-former.transform()	XMLFilter	TransformerHandler
XInclude	XIncludeFilter.parse()	XIncludeFilter	not available

[a]easily implemented in Java by extending `XMLFilterImpl`
[b]created with: `SAXTransformerFactory.newXMLFilter()`
[c]planned for release in the DeltaXML Core API v3.0
[d]created with: `SynchronizerFactory.newSynchronizerFilter()`
[e]created with: `Serializer.asContentHandler()`

4. Issues in Pipeline Construction

4.1. Controlling Parsing

It is possible to create pipelines without creating a parser component. For example, the following two code fragments do not explicitly create a parser:

```
Transformer t= ...;
t.transform(new StreamSource("f.xml"), ...);
```

```
SAXTransformerFactory stf= ...;
Transformer t= ...;
XMLFilter filter= stf.newXMLFilter(new StreamSource("filter.xsl"));
t.transform(new SAXSource(filter, new InputSource("f.xml")),
                ...);
```

In the case of a `StreamSource` input, the transformer is told to process the input stream, it typically does this using a SAX parser. Similarly if a *"parent-less filter"* (one which has not called `set-Parent()`) is being used then that filter will use an implementation default SAX Parser. However, in certain circumstances you may wish to have more control, for example:

- You may wish to use features only available in a specific parser, for example schema validation in Xerces-J.
- You may wish to control certain Features or Properties of the parser, for example options to control InfoSet augmentation.

In these cases it is simplest to explicitly create a parser object and use it at the start of the pipeline. For example:

```
System.setProperty("javax.xml.parsers.SAXParserFactory",
                   "org.apache.xerces.jaxp.SAXParserFactoryImpl");
SAXParserFactory spf= SAXParserFactory.newInstance();
Transformer t= ...;
spf.setValidating(true);
spf.setFeature("http://apache.org/xml/.../schema/augment-psvi",
               false);
XMLReader reader= spf.newSAXParser().getXMLReader();
XMLFilter filter= stf.newXMLFilter(new StreamSource("filter.xsl"));
filter.setParent(reader);
t.transform(new SAXSource(filter, new InputSource("f.xml")),
            new StreamResult(...));
```

4.2. Programmatic Output Control

Some pipeline configurations make it difficult to control aspects of the output including formatting issues such as indentation and also any desired DOCTYPEs. In pipelines such as Saxon's exampleXMLFilterChain which consist entirely of XMLFilter components it can be difficult to control formatting programmatically. It is possible to *hardwire* formatting information in the last filter of such pipelines by editing the `<xsl:output .../>` statement in the associated script. However, to control formatting programmatically requires access to the underlying transformer of the last pipeline component. This is the case if the last component is a Transformer or TransformerHandler, for example:

```
TransformerHandler h= ...;
h.getTransformer().setOutputProperty(OutputKeys.INDENT, "yes");
h.getTransformer().setOutputProperty(OutputKeys.DOCTYPE_PUBLIC,
                    "-//W3C/DTD XHTML 1.0 Strict//EN");
```

Note that a similar mechanism is not available if an XMLFilter is being used.

Some suggestions/guidelines for programmatic output control follow:

- Avoid using XMLFilters as the last pipeline element, as they do not provide access to the underlying Transformer.
- If a component does not provide control over output formatting consider adding an identity transform (a newTransformer call without specifying a stylesheet: `TransformerFactory.newTransformer()`) which can then have its output properties set programmatically.
- If using an `XMLFilter` with Saxon 7 or later it can be cast to the `net.sf.saxon.Filter` class which provides access to the underlying `Transformer`.

4.3. DOCTYPE Preservation

One common query when using XSLT based processing pipelines is, "where has my DOCTYPE gone?". The usual answer to this question is that the XSLT processing model has no knowledge of the input DOCTYPE or DTD, but that the output DOCTYPE information can be configured using `<xsl:output doctype-system="..."` in the final stage of any pipeline. Alternatively it

could be controlled programmatically as demonstrated in the previous section.

A more complex requirement is to determine and preserve the input DOCTYPE. We have met two situations where the simple "specification in the output approach" is not powerful enough:

- On the input data the DOCTYPE's system id is a URI is to a relative file, for example: `"../format.dtd"`, in the output file the URI cannot be resolved into an absolute URI and needs to be preserved as a valid relative URI.
- The pipeline is designed for a number of related vocabularies. For example a generic XHTML pipeline could process *Strict*, *Transitional* and *FrameSet* data as specified in the input DOCTYPE declaration. The output DOCTYPE should contain the correct DOCTYPE, as per the input.

One potential solution is to combine the techniques addressed in the previous sections. Using an explicit parser component with a SAX `EntityResolver`, or if available, `LexicalHandler`, the input DOCTYPE can be determined and stored in a Java field. This could then be used to programmatically configure the output DOCTYPE using the `setOutputProperty()` method. There is one problem here: the input DOCTYPE is only determined through the `EntityResolver` or `LexicalHandler` when the `transform()` or `parse()` triggering method is invoked for the pipeline; while the output DOCTYPE needs to be configured prior to calling this method. We can suggest two alternative approaches, neither of them ideal, to solving this issue:

- For the input file use the SAX parser to parse (or "preparse") the input solely in order to determine the input DOCTYPE information. Then use a more complete pipeline to do the normal processing and reinstate the DOCTYPE information at the output stage.
- On the input stage of the pipeline convert the DOCTYPE information into some suitable form of markup which can then flow through the pipeline and be converted back into a DOCTYPE at the end. Unfortunately it is not possible to do this operation with XSLT as the DOCTYPE information is not available in the XSLT/XPath input tree. Using Java code for SAX Filters and Serializers may provide an alternative.

Given these problems with DOCTYPEs and pipelines; using one of the new schema languages may provide a more attractive solution since they allow us to avoid the use of DOCTYPEs completely.

4.4. Parameterized XSLT Filters

Parameterized XSLT filters allow a user to control various aspects of the operation of the filter, generally aiding modularization and reusability. Parameters are set through the `Transformer` object and its `setParameter()` method. As discussed in Section 4.2, “Programmatic Output Control” access to the `Transformer` object is not directly available in an `XMLFilter`. This limitation is more of a concern for parameterization than it is for output property control as substituting an `XMLFilter` with a `Transformer` or `TransformerHandler` may be less convenient in the general case, than it is for output control on the final stage of a pipeline.

4.5. Java Push-Mode

The XMLFilterImpl class which is available only in pull-mode, allows easy development of Java-based filters. As shown in Section 2.5, “Optimized SAX Event Pipelines” this provides a default class whose default action is to copy events from input to output. Java extension is then used to customize this behaviour. Such an extensible class is currently missing for creating the corresponding push-mode Java components, although this can be tackled in a number of ways.

4.6. Error Handling

There is always a temptation in programming to "leave the error handling until later". When using JAXP this can provide a few surprises. Although the JAXP API does provide user controllable error handling[6], the default settings can catch programmers unawares. Many Java programmers have developed an "if it doesn't throw an exception it must be working" mind-set. Unfortunately, JAXP will report errors to `System.err` if the user does not register their own custom `ErrorListener`. In some development environments such error messages are difficult to distinguish from other messages. In some runtime environments such as servlet containers or application servers `System.err` messages may go unnoticed in log files for long periods. While we appreciate that the ErrorListener architecture is powerful and allows application programmers to implement systems which report multiple, recoverable error messages in a similar manner to that of a compiler, we are dubious about the default of reporting errors to `System.err`. We would encourage developers to "program defensively" and perhaps instantiate a *rethrowing* `ErrorListener` until it is decided to handle errors differently. A rethrowing `ErrorListener` follows the following code pattern:

```
class RethrowingListener implements ErrorListener {
  public void error(TransformerException e)
    throws TransformerException {
      throw e;
  }
  public void warning ...
  public void fatalError ...
}
...
Transformer t;
t.setErrorListener(new RethrowingListener());
```

A similar pattern can be applied to the SAX `ErrorHandler` interface.

4.7. JAXP Implementation Selection

The JAXP API allows pluggability with different implementations of Parsers and Transformers being controlled at runtime. The precise mechanism used is described in the factory `newInstance()` methods, for example: `TransformerFactory.newInstance()` and `SAXParserFactory.newInstance()`. This ordered lookup mechanism often goes unnoticed or unappreciated by Java programmers. "I've added saxon.jar to my classpath and now I'm using SAXON for my XSLT transforms, ... must be a classpath thing", not realizing that adding `saxon.jar` *also* effectively sets the factory configuration property using the services API mechanism. In order to completely and effectively control the use of the implementation class programmers need to:

- Ensure that the implementation class is available to the appropriate classloader through the classpath or other mechanisms such as a servlets `WEB_INF/lib` subdirectory.
- Use a suitable configuration mechanism to set the Factory class property as documented by `newInstance()`
- If trying to override an implementation class used in the JDK (for example the old/buggy Xalan-J implementations used in J2SE 1.4.0 and 1.4.1), use the Java Endorsed Standards Override mechanism (as documented in J2SE).

While it is generally possible to change the runtime implementations, there are some circumstances which test the pluggability mechanisms to the extreme. For example, when exploiting the reusable component model suggested in this paper a user may find that different XSLT-based off-the-shelf filters use different processor-specific extensions. Controlling the appropriate JAXP property it is possible to instantiate two such components in the same pipeline, for example:

```
System.setProperty("javax.xml.transform.TransformerFactory",
                   "net.sf.saxon.TransformerFactoryImpl");
SAXTransformerFactory saxonSTF=
  (SAXTransformerFactory) TransformerFactory.newInstance();
TransformerHandler stage2= saxonSTF.newTransformerHandler(...);
...
System.setProperty("javax.xml.transform.TransformerFactory",
          "org.apache.xalan.processor.TransformerFactoryImpl");
SAXTransformerFactory xalanSTF=
  (SAXTransformerFactory) TransformerFactory.newInstance();
```

[6]We have some criticisms of JAXP error handling which we may address in the paper presentation

```
TransformerHandler stage3= xalanSTF.newTransformerHandler(...);
...
```

The use of `System.setProperty()` in the example above is the only way different filter Transformer implementations may be used in the same pipeline. However, since it is the highest priority method in the factory implementation lookup order, some caution needs to be exercised. In particular, consideration needs to be given to the ClassLoader environment in which the change is made. For example, when running in a Servlet container, changing the XSLT processor used in one pipeline of a servlet may have unforeseen consequences on any other servlets in that WebApp.

5. Future Developments

5.1. JAXP 1.3

JAXP 1.3 is scheduled for inclusion in the next major J2SE release. Some small changes have been made to the existing JAXP 1.2 API. More significant developments include changes to the underlying implementation technologies in the J2SE implementations supplied by Sun and the incorporation of new technologies such as DOM Level 3 and Validators.

The changes to the implementation technologies include switching the SAX parser from Crimson to Xerces and and changing the XSLT processor from the interpreted Xalan-J to XSLTC. Some initial testing has shown that this change is likely to affect the error detection and reporting behaviour.

A new feature which can be used in SAX event pipelines is the `Validator` class. This consumes a Source object and optionally produces a Result, with possible InfoSet augmentation. In previous versions of JAXP it was only possible to validate using a parser, typically at the start of a pipeline. This new facility will allow validators to be used at various stages of a pipeline, for example, after a filter as a check of its output validity.

Other new features, not directly related to SAX event pipelining, include:

- Support for XInclude
- XPath 1.0 expression evaluation
- Support for DOM level 3

5.2. SXPIPE

SXPIPE [SXPIPE] is a proposal for a simple declarative pipeline language where a sequence of operations can be applied. It shares some similarity with an earlier W3C pipeline proposal [W3CPIPE] [PIPE2], but with a simplified syntax/model it appears easier to use, if not as flexible/general.

In its simplest form a sequence of operations can be performed and the output InfoSet of one stage is fed into the next one listed. However stages can also be labeled which can then be used as inputs for stages which do not follow in strict textual order. The specification currently describes six stages which must be supported by all implementations:

- Identity - copies input to output
- Read - loads a XML file/url and produces an InfoSet
- Transform - XSLT transformation of an InfoSet
- Validate - InfoSet validation and augmentation
- Write - serializes its input
- XInclude - performs XInclude processing/expansion

The following example shows a two stage transformation, plus auxiliary processing, pipeline:

```
<pipeline xmlns="http://sxpipe.dev.java.net/xmlns/sxpipe/">
  <stage process="http://sxpipe.dev.java.net/stages/XInclude"/>
  <stage process="http://sxpipe.dev.java.net/stages/Validate"
         schema="schema.xsd"/>
  <stage process="http://sxpipe.dev.java.net/stages/Transform"
         stylesheet="stage1.xsl"/>
  <stage process="http://sxpipe.dev.java.net/stages/Transform"
         stylesheet="stage2.xsl"/>
  <stage process="http://sxpipe.dev.java.net/stages/Write"/>
</pipeline>
```

In this case the pipeline does not specify where its inputs are obtained or where the result, from the write stage, will go. The specification leaves these as being implementation defined. This, we feel, makes the language ideal for embedding in pipelined XML applications which are designed for easy extensibility/adaptability by their end users.

Note that the specification does not define how the underlying pipeline should be implemented, only how it is declared. The reference Javadoc which accompanies the specification does suggest the use of W3C DOM used as the underlying implementation.

6. Conclusions

The model presented in this paper presents an overview view of how to create SAX event based pipelines using JAXP. When viewed in conjunction with the more complete examples in the DeltaXML pipelining tutorial [http://www.deltaxml.com/pipelines/], and also the paper presentation, we hope to have addressed some aspects we feel are missing from existing pipeline examples. These include the benefits of using TransformerHandlers over XMLFilters and the advantages of using methods which use the overloaded `Source` and `Result` classes as arguments.

We hope that the case study presented in this paper, demonstrates the benefits of the pipeline metaphor: developing or reusing simple communicating components whenever possible. We are persuaded of the benefits of this approach and have designed our software components to be easily integrated into JAXP based SAX event pipelines. Rather than add new features into our comparison and synchronization engines we would prefer to keep them both architecturally simple and easy of end-users to use. When we are asked to provide customized or enhanced capabilities, whenever practical, our first consideration is to see if we can do this with new pipeline components.

While simple, declarative styles of pipeline construction, such as SXPIPE, will meet the needs of a number of user communities, the full capabilities of JAXP should be available to "power users". However we also see the need for a more declarative style of pipeline construction and intend to follow the development of languages such as SXPIPE.

The authors would like to thank Martin Bryan, for complaining (with justification) that our existing JAXP sample code was hard to understand. This criticism lead to the development of a richer pipeline tutorial and this paper. DeltaXML customers, by asking support questions and posing problems, have also prompted the development of this paper; we hope that it is a useful resource both for them and the wider XML community.

Bibliography

[APSER] Apache XML Serializer, Available as part of Xalan-J, at: http://xml.apache.org/xalan-j/apidocs/org/apache/xml/serializer/package-summary.html

[COCOON] *Cocoon: XML and web application development* Steven Noels, XML Europe 2004. Available at: http://www.idealliance.org/papers/dx_xmle04/papers/02-06-01/02-06-01.html

[DOM] *Document Object Model (DOM) Level 2 Core Specification* W3C Recommendation, 13 November 2000. Available at: http://www.w3.org/TR/DOM-Level-2-Core/

[DXCORE] *Change Control for XML: Do It Right* Robin La Fontaine. XML Europe 2003. Available at: http://www.deltaxml.com/pdf/deltaxml-xmleurope2003.pdf

[DXSYNC] *Merging XML Files: A New Approach Providing Intelligent Merge of XML Data Sets* Robin La Fontaine. XML Europe 2002. Available at: ht-tp://www.deltaxml.com/pdf/merging-xml-files.pdf

[JAXP] Java API for XML Processing. Available at: http://java.sun.com/xml/jaxp/index.jsp

[JELLY] Available at: http://jakarta.apache.org/commons/jelly/pipeline.html

[PIPE2] *Re-Interpreting the XML Pipeline Note: Adding Streaming and On-Demand Invocation* Henry S. Thompson. XML 2003. Available at: ht-tp://www.markuptechnology.com/XML2003.html

[SAX] Simple API for XML Available at: http://www.saxproject.org

[SXPIPE] *SXPipe: Simple XML Pipelines* Norman Walsh, Sun Microsystems, Inc. Working Draft 28 July 2004 Available at: https://sxpipe.dev.java.net/nonav/specs/sxpipe.html

[STnG] *STnG - a Streaming Transformations and Glue framework* K. Ari Krupnikov. Extreme Markup Languages 2003. Available at: ht-tp://www.mulberrytech.com/Extreme/Proceedings/html/2003/Krupnikov01/EML2003Krupnikov01.html

[STX] *Transforming XML on the fly: How STX Enables the Processing of Large Documents* Oliver Becker. XML Europe 2003. Available at: ht-tp://www.idealliance.org/papers/dx_xmle03/papers/04-02-02/04-02-02.html

[TrAX] Transformation API for XML Available at: http://xml.apache.org/xalan-j/trax.html

[W3CPIPE] *XML Pipeline Definition Language Version 1.0*Simple XML Pipelines Norman Walsh, Eve Maler. W3C Note 28 February 2002 Available at: ht-tp://www.w3.org/TR/2002/NOTE-xml-pipeline-20020228/

[XINC] XInclude Engine, Available at: http://xincluder.sourceforge.net