Tapio Laitinen

XML PROCESSING IN JAVA

Information Systems Science

Bachelor's Thesis

6.6.2005

University of Jyväskylä

Faculty of Information Technology

Jyväskylä, Finland

ABSTRACT

Laitinen, Tapio Markus.

XML Processing in Java/Tapio Laitinen.

Jyväskylä: University of Jyväskylä, 2005.

40 pages.

Bachelor's Thesis.

XML and Java have both gained considerable attention in last few years. In this thesis, different mechanisms for processing XML in general, and in Java in particular, are examined. These mechanisms are the three XML processing models, SAX, DOM and JDOM, and XSL Transformations (XSLT). The purpose of this thesis is to examine how XML processing – or, more accurately, parsing, interpreting, and transforming – is done in Java programming language. In addition to that, some evaluation of suitability of Java for this task is carried out.

Complete code examples of different XML processing techniques are provided in appendices, but they are kept relatively short because extensive example applications are not appropriate for this thesis.

Every processing model has its strengths and weaknesses which makes some of the models more suitable for a particular task than the others. This thesis helps in making a choice between the processing models by listing some of their benefits and drawbacks. All in all, Java seems to suit adequately for most XML processing applications.

KEYWORDS: XML, Java, SAX, DOM, JDOM, XSLT.

TABLE OF CONTENTS

1 INTRODUCTION

XML and Java are two of today's "buzzwords". XML is a markup language that is used for facilitating the sharing of structured text and information across the Internet and applications; Java is a programming language that is characterized by the words "portable" and "object-oriented". Both of these technologies have lately gained considerable attention, some of which is hype, some of which is true.

In 1999, the first version of Java API for XML processing (JAXP) was included in Java. From then on, every Java programmer could have used this programming language to process data in XML format. JAXP enables the use of two different XML processing models, SAX and DOM, both of which have their strengths and weaknesses. Apart from SAX and DOM, and JAXP, is one very Java-centric XML processing model, JDOM, which is also examined in this thesis.

JAXP also makes performing XSL Transformations (XSLT) possible. XSLT is a technique for transforming XML documents into other XML documents or into other documents that are recognized by a browser, such as HTML documents.

The purpose of this thesis is to examine how XML processing – or, more accurately, parsing, interpreting, and transforming – is done in Java. In addition to examining different techniques and mechanisms for processing XML, this thesis presents some benefits and drawbacks of using Java for this task.

This thesis is based on information collected from different literal sources, of which the most important is the Internet.

2 XML IN A NUTSHELL

Extensible Markup Language, or XML for short, is a text-based markup language, a mechanism for representing a document and identifying structures in that document. It is used for facilitating the sharing of structured text and information across the Internet and applications [7]. It is also used for information storing.

The development of XML started in 1996 and was based on Standard Generalized Markup Language, SGML, which was published as a standard by International Organization for Standardization (ISO) in 1986. XML became a restricted form of SGML because the goal of its development was to create a generic markup language that does not have a large collection of rules – like SGML does – which would hinder its utilization. [13]

In 1998, when the first W3C Recommendation for XML 1.0 was published, HTML (Hypertext Markup Language) already had become a de-facto standard for publishing information on the World Wide Web [13]. XML resembles HTML in many ways: Firstly, XML, too, is a markup language. Secondly, also XML markup is done by using tags, pieces of data that delimit the start and end of elements. Thirdly, both XML and HTML are text-based languages, which means that documents created in them can be edited using a text editor.

Despite those similarities, XML also differs in many ways from HTML: First of all, XML is a meta-language, which means that it can be used to create other languages [3]. Secondly, XML was designed to describe data and to focus on what data is, while HTML was designed to display data and to focus on how data looks [21]. Thirdly, unlike HTML, XML is not limited to text only, but can also be used to define the structure of, for example, mathematical data (MathML), vector graphics (SVG), animation (SMIL Animation), and voice data (VoiceXML) [21].

Fourthly, XML tags are not predefined like HTML tags – it is possible to use any tags, preferably those that are appropriate for the given application domain [24]. Fifthly, XML elements must be marked precisely by using both an opening and a closing tag whereas in HTML it is possible to leave out the closing tag.

It should be kept in mind that XML is not meant to be a replacement for HTML, they were designed with quite different goals: XML to describe information, HTML to display information. [21, 24]

XML information is presented and shared as XML documents. All XML documents consist of the following items [20]:

- *Elements* are the main "building blocks" of an XML document. They can be described as grammatical fragments of a document.
- *Tags* are used to mark up the beginning and end of elements. In other words, data is identified by using tags. Tags are enclosed in angle brackets.
- *Attributes* provide additional information about elements. They are always placed inside the starting tag of an element and come in name/value pairs.
- *Entities* are variables that are used to define common text. One example of an entity is *"* which is expanded into a quotation mark when a document is parsed by an XML parser. There are five predefined entities in XML.
- *PCDATA* (parsed character data) is text, which is found between the start tag and the end tag of an XML element, and which will be parsed by a parser. In practice this means that all tags inside the text that is marked as PCDATA will be treated as markup and entities will be expanded.

- *CDATA* (character data) is text that will not be parsed by a parser: Tags inside the text that is marked as CDATA will not be treated as markup and entities will not be expanded.

Before an XML document can be correct, it must be well-formed and valid. A well-formed document conforms to all syntax rules of XML; a valid document is a well-formed document that also complies with a particular schema, a description of a type of a document [21]. XML schemas are expressed with languages like DTD and XML Schema.

2.1 Document Type Definition

A Document Type Definition (DTD) defines the structure of an XML document by declaring a list of legal elements of that document and the order of those elements. When an XML document follows the restrictions declared in a DTD, the document is considered to be valid. Here is an example of a simple DTD for an XML document:

```
<!ELEMENT contact_info (tel,url?) >
<!ATTLIST contact_info xml:lang (fi | en) "fi" >
<!ELEMENT tel (#PCDATA) >
<!ELEMENT url (#PCDATA) >
<!ATTLIST url href CDATA #REQUIRED >
```

The DTD above declares the elements and their attributes for representing contact information. The element *contact_info* consists of a telephone number (*tel*) and an optional Web address (*url*). The element has an attribute *xml:lang* that defines whether the contact information is in Finnish or in English. The default value for the attribute is Finnish (*fi*). The element *tel* contains parsed character data, as does the element *url*. *url* also has a mandatory attribute *href* that contains the actual Web address.

A DTD can be built into an XML document (so called inline DTD) or it can be an external reference [20]. The example above is an external DTD, because it does not contain the DOCTYPE declaration that is required for inline DTDs.

DTDs enable independent groups of people to agree on the rules for interchanging data. Also verifying data becomes possible with the use of a DTD. [20] However, Document Type Definitions are not the only alternative for defining the structure of an XML document.

2.2 XML Schema

XML Schema is an XML-based alternative to DTD. The XML Schema language is also referred to as XML Schema Definition [23]. Below is an example of an XML Schema that declares the elements and their attributes for representing contact information, as did the DTD above:

```
<?xml version="1.0" encoding="UTF-8">
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="contact_info">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="tel" type="xs:string" />
        <xs:element ref="url" />
      </xs:sequence>
      <xs:attribute xml:lang type="xs:string" default="fi" />
    </xs:complexType>
  </xs:element>

  <xs:element name="url">
    <xs:complexType>
      <xs:attribute name="href" type="xs:string"
        use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In addition to DTDs and XML Schemas there are several other mechanisms for constraining an XML document, such as RELAX NG and DSDL. However, these other schema languages are not discussed in this thesis.

2.3 The Extensible Stylesheet Language

By nature, XML does not focus on formatting. The Extensible Stylesheet Language (XSL) is used for defining how input will be formatted so that it takes on a new output form. In other words, with XSL display information can be added to an XML document, so XSL is a stylesheet language for XML like CSS is for HTML. [22]

In fact, XSL is a family of recommendations, which consists of three parts: XSL Transformations (XSLT), the XML Path Language (XPath), and XSL Formatting Objects (XSL-FO). [12] Only the first one of these recommendations is examined in this thesis.

Extensible Stylesheet Language Transformations (XSLT) is a language for transforming XML documents into other XML documents or into other documents that are recognized by a browser, such as HTML documents. Transformation is done by applying XSL templates to XML documents in order to create new documents in desired formats. XSLT processor performs the formatting task according to the syntax and semantics provided by XSL [11].

With XSLT, it is possible to add elements to the output file, as well as remove elements from it. It also enables, for example, rearranging and sorting of elements, performing tests, and making decisions about which elements to hide and which to display. [22] XSLT is covered from the point of view of Java in chapter 3.4.

3 JAVA API FOR XML PROCESSING

Java programming language was developed by Sun Microsystems during early 1990's; its first official version was released in 1995 [15]. Since then, Java technology has evolved greatly and spread rapidly nowadays being right behind C in popularity [18].

There are five XML-related application programming interfaces, APIs, in Java [16]:

- Java Architecture for XML Binding (JAXB)
- Java API for XML Processing (JAXP)
- Java API for XML Registries (JAXR)
- Java API for XML-based RPC (JAX-RPC)
- SOAP with Attachments API for Java (SAAJ)

In this thesis, only the Java API for XML Processing (JAXP for short) is covered. The first version of this API was released by Sun Microsystems in late 1999 [11]. It is a platform-independent API, which, according to Sun, "enables applications to parse and transform XML documents independent of a particular XML processing implementation" [16].

Parsing means reading input (usually characters) and transforming it into meaningful tokens; in other words parsing means interpreting the structure. In case of XML parsing, "meaningful tokens" mean XML elements. XML parsing has been separated into two different processing models: Simple API for XML (SAX) and Document Object Model (DOM). In addition to these two APIs, JAXP supports the Extensible Stylesheet Language Transformations (XSLT) language.

It should be noted that neither SAX nor DOM are Java-specific APIs but there are also versions for several other programming environments [5, 9]. This means that these processing models are not solely related to JAXP, it only wraps them into a single XML API. Another important note is that JAXP itself does not provide parsing functionality, but needs other APIs to do this task [11].

There is also a Java-specific XML processing model called JDOM. It is discussed in chapter 3.3, which is a subchapter of this chapter, even though it strictly speaking is not a part of JAXP but instead only integrates well with this API.

3.1 Simple API for XML

Simple API for XML (SAX) is a mechanism for processing XML. While it originally was a "Java-only API", it since has been implemented in several other programming languages such as C++, Perl, Pascal, and Python. As of this writing, SAX's newest version is 2.0.2. [5]

SAX is based on an event-driven processing model where the data elements are interpreted on a sequential basis. In other words this means that the SAX parser reads XML data element by element thus creating a stream of events from this data (see Illustration 1). This is also called serial access. The fact that SAX does not load any XML documents into memory makes it appropriate for high-speed processing of XML.

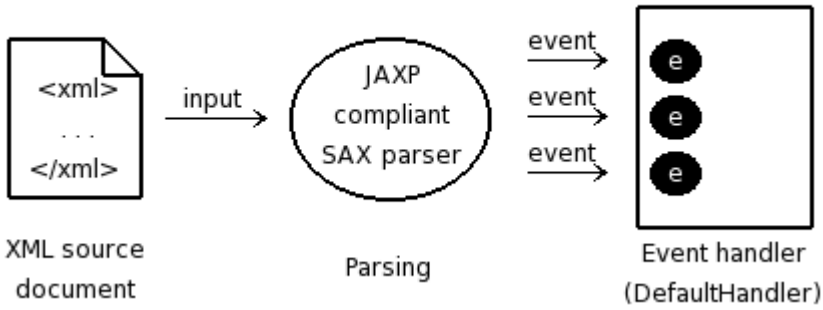

Illustration 1 SAX processing model

From the point of view of Java and JAXP, the basic processing model of SAX consists of the following steps [11]:

1. Implement a handler class. Create callback methods[1] for every type of XML construct that you need in your application.
2. Instantiate a new SAX parser class. The parser reads the XML source file and triggers a callback implemented in the handler class.
3. Read the XML source sequentially. The rest depends upon the implementation residing in the handler class.

Next, these three steps are gone through in more detail.

3.1.1 Implementing a Handler Class

A handler class is responsible for handling the events that the SAX parser generates. Since JAXP 1.1, it is required that the handler used with an instance of a SAXParser class extend the DefaultHandler class. The DefaultHandler class provides default implementations for several interfaces and the programmer can override the methods he or she is interested in.

1 The term *callback method* here means the mechanism where a method in one object asks a method in another object to "call me back" when an interesting event occurs. What results from this is that these methods are never directly invoked by a programmer but instead by another object.

The main interface that the DefaultHandler implements is ContentHandler. It contains all the callback methods that basic parsing requires, such as *startDocument*, *endDocument*, *startElement*, *endElement*, and *characters*. The *startDocument* method is invoked by the SAX parser when the parser encounters the beginning of an XML document, and *startElement* method when the parser encounters a start tag of an element. *endDocument* and *endElement* methods are self-explanatory. The *characters* method is invoked whenever the parser reports pieces of character data from inside the elements.

A complete sample implementation of an extended SAX handler class is provided in appendix A. It should be noted here that the handler does not have to be in a different compilation unit than the parser, like in the example in appendix A.

3.1.2 Instantiating a SAX Parser Class

After the event handler has been implemented, i.e. the appropriate methods have been overridden, the next step is to instantiate it and the parser class. Instantiating the handler is quite straightforward:

```
DefaultHandler handler = new SAXHandler();
```

An instance of a SAXParser class can be obtained by using one of the factory APIs of Java, SAXParserFactory. This type of instantiation enables parser vendors to be swapped without much effort, providing that they comply with the JAXP specification. Two common parsers and parser factories are Sun Microsystems' Crimson and the Apache Software Foundation's Xerces2. Below are the two lines of code that instantiate a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
```

The parser factory can additionally be configured to instantiate validating and/or namespace-aware parsers. If the instantiated parser is not validating, it cannot tell if the document is valid. To configure a parser factory to instantiate validating parsers, a programmer should write this before calling the *newSAXParser* method:

```
factory.setValidating(true);
```

A namespace-aware parser means a parser that is able to handle naming collisions, i.e. if there are for example two different DTDs that both contain an element named *title*, a namespace-aware parser is able to distinguish the two elements from each other. This is important when using multiple documents with the same application [11]. To set a parser factory to instantiate namespace-aware parsers, a programmer should invoke its *setNamespaceAware* method like this:

```
factory.setNamespaceAware(true);
```

After this step, the SAX parser can be used to read the XML source input and parse it.

3.1.3 Reading the XML Source

The final step is the actual reading and parsing. It can be accomplished by invoking SAXParser class's *parse* method with two parameters: XML data to be parsed and a handler that handles the events that the SAX parser generates. Here the first parameter is a file to be processed and its name is received as a command line argument:

```
saxParser.parse(new File(args[0]), handler);
```

A complete example of how to use a SAX parser is provided in appendix B.

3.1.4 Drawbacks of Using SAX

Several drawbacks are related to using SAX processing model [2, 11, 1]:

- SAX is not a W3C standard.
- With the sequential reading, it is not possible to randomly access elements in the structure. This makes modifying XML documents by using SAX difficult. Also complex searches can be difficult to implement because of this.
- The event-driven model of SAX is harder to visualize than the tree structure of DOM.
- Using SAX requires more programming than using DOM. DOM is also generally easier to use. This processing model is examined next.

3.2 Document Object Model

According to the W3C working group, the definer and maintainer of the Document Object Model (DOM), DOM is "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents." DOM is the official W3C standard. [9]

The greatest difference between SAX and DOM processing models is that in DOM the entire XML document is read into memory and a tree representation of the structured data is built (see Illustration 2). This process is naturally resource intensive especially when the XML document is large, but by having the data in memory, DOM provides random access to any node in the tree, unlike SAX. DOM also enables the manipulation of data, for example inserting, editing, rearranging, and deleting tree elements. [1]

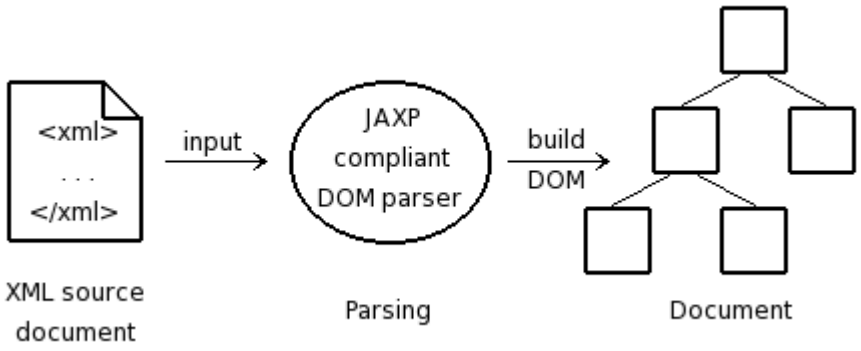

Illustration 2 DOM processing model

From the point of view of Java and JAXP, the basic steps of the DOM processing model are the following [11]:

1. Instantiate a new Builder class that reads the XML data and transforms it into a tree representation.
2. Create the Document object once the data is transformed. The Document object represents the XML source data.
3. Use the Document object to access nodes representing elements from the XML document.

Next, these three steps are gone through in more detail.

3.2.1 Instantiating a Builder Class

Similarly to instantiating a SAX parser class, also a DOM parser, or to be exact a document builder, can be obtained by using a factory API. In this case the factory is called DocumentBuilderFactory, which is instantiated as follows:

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

After that, a document builder can be obtained from the factory:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

Like the SAX factory, also the DOM factory can be configured to instantiate validating and/or namespace-aware parsers by invoking *setValidating* and/or *setNamespaceAware* methods, respectively.

3.2.2 Building the Document Object

Once the document builder is obtained, it can be used to create an in-memory representation of the XML data – so called document, an instance of the org.w3c.dom.Document class. In DOM, a Document instance is a tree structure that contains nodes created from the XML source.

In most implementations, DOM relies on the SAX model for reading the XML data into memory. This is not mandated by the JAXP specification, which means that the DOM parser does not have to use a SAX parser internally, but it is reasonable because their internal functionality is quite similar for example in error handling. [11, 16]

The Document object can be built by invoking DocumentBuilder's *parse* method and passing this method the XML data to parse as a parameter. Here the data is in a file whose name is received as a command line argument:

```
Document document = builder.parse(new File(args[0]));
```

A complete sample implementation of reading data into DOM is provided in appendix C. After the parsing, the Document object can be used to access nodes of the tree representing the XML data.

3.2.3 Accessing Nodes

DOM represents XML data in a tree structure, which consists of nodes. The tree is stored in the Document object. In a DOM tree, each node contains one of the components from an XML structure – elements, attributes, and so on – and

implements the Node interface. Also the Document object itself implements this interface [15].

There are many different types of nodes in a DOM tree[2] but the two most common ones are element nodes and text nodes. An element node contains information about a particular element such as its name, its attributes etc. Text nodes represent the text found between the start and end tags of an element. Text nodes always exist "under" element nodes and the actual data is stored in text nodes, not in element nodes. In DOM, this element content is considered a child node of an element node itself. [10]

Every node contains information about an XML element it represents. For example, the name of the element can be obtained like this:

```
String name = node.getNodeName();
```

If the element in question is a comment, variable *name* will contain a string "*#comment*" after the execution of the line; if the element is text, the variable will contain a string "*#text*", and so on.

A complete sample implementation of traversing a DOM tree and printing out its contents is provided in appendix D.

3.2.4 Drawbacks of Using DOM

There are couple of drawbacks related to using DOM processing model [2]:

- DOM is memory intensive because entire document must be instantiated into memory before it can be used.

2 Refer to the javadoc documentation of the interface org.w3c.dom.Node for a complete list of node types.

- DOM does not enable serialization of objects into an XML document except through vendor-specific methods, which conflicts with JAXP ideology.
- DOM is slow especially when it is used for processing large XML documents.

3.3 JDOM

Despite its name[3], JDOM has little to do with DOM. JDOM is a Java technology -centric processing model, which means that it uses mechanisms typical of Java such as method overloading, reflection, collections, and exceptions. JDOM is an open source API and one of the Apache Projects. It was publicly announced in April 2000 and its newest version is 1.0. [2, 6]

Similarly to DOM, JDOM is a document object model that provides a way to represent an XML document in a tree-based structure. Also JDOM enables random access to the document's contents and manipulation of it. Unlike DOM, JDOM is optimized for Java so it is platform-neutral like the programming language itself. On the other hand, DOM has been implemented in several programming languages while JDOM has been implemented in only one, so using JDOM requires that programmers also use Java.

The processing model of JDOM is quite similar to DOM. First, a parser reads the source and processes it. Then a tree representation of an XML document is built based on the information received from a parser (see Illustration 3). The default parser that is included in JDOM is Apache's Xerces, but it is possible to use other XML parsers as well [10].

3 According to the founders of the JDOM project, Jason Hunter and Brett McLaughlin, the name JDOM is not an acronym [6].

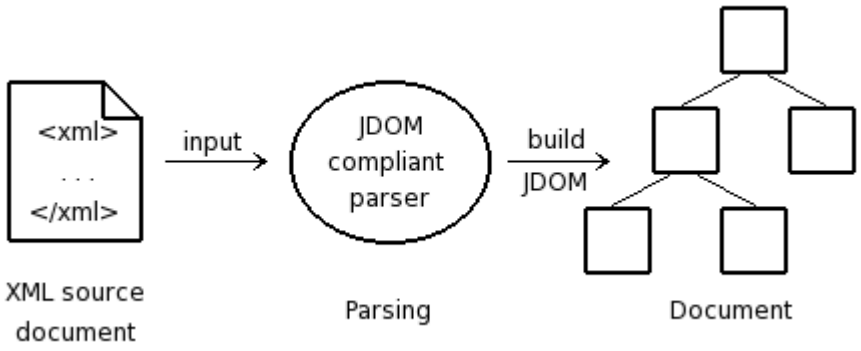

Illustration 3 JDOM processing model

Also here the tree structure resulting from the parsing is called a document. This time it is an instance of the org.jdom.Document class. In the above illustration, the document is built from an XML source file but JDOM documents can also be built for example from SAX events or DOM trees. JDOM documents can also be converted to any one of these alternatives. [6]

To build a JDOM document from an existing source, the programmer should do the following [10]:

1. Instantiate a builder class.
2. Build the Document object using the builder.

Next, these steps are discussed more thoroughly.

3.3.1 Instantiating a Builder Class

If the source is XML that is not in a DOM tree already, instantiating SAXBuilder class is the best solution [11]. This way the programmer gets the benefits of SAX, the most important of which is its speed. Instantiation is done like this:

```
SAXBuilder builder = new SAXBuilder();
```

A validating SAXBuilder can be instantiated by invoking the constructor that takes a boolean parameter. Namespaces are handled by using the org.jdom.Namespace class but this is not discussed in this thesis.

If the programmer already has his or her document in a DOM structure, DOMBuilder is a better alternative than SAXBuilder [11]. It is instantiated like this:

```
DOMBuilder builder = new DOMBuilder();
```

Note that no factory APIs are needed in either case, unlike with SAX and DOM. This is because every class in JDOM API is concrete. When a builder is instantiated, it can be used to produce the Document object.

3.3.2 Building the Document Object

Using the SAXBuilder, the Document object can be built like this. It is assumed here that the name of a source file is received as a command line argument:

```
Document document = builder.build(new File(args[0]));
```

Using the DOMBuilder, the document is built as follows. Here a reference to the existing DOM structure is stored in a variable named *myDOMDocumentObject*:

```
Document document = builder.build(myDOMDocumentObject);
```

After the Document object is created, it can be used to manipulate or otherwise process the XML data. A complete example of how to build a JDOM document from an existing XML file using a SAXBuilder, and output the file's contents on the display is provided in appendix E.

3.3.3 Drawbacks of Using JDOM

As mentioned before, it is possible to use both SAX and DOM with JDOM so refer to those chapters for information on their drawbacks – though JDOM eliminates some of those by for example creating an object-oriented model from SAX events, which enables random access to elements in the structure.

Naturally, because JDOM is very Java-centric, it has the same drawbacks as using the programming language itself. These drawbacks are glanced in chapter 4 but they are not the actual subject of this thesis.

3.4 Extensible Stylesheet Language Transformations

In Java, one possibility to do XSL Transformations is through Xalan-Java processor (from now on referred to as simply Xalan). It is an Apache Project and currently implements XSLT Version 1.0 as well as XPath Version 1.0. Xalan also utilizes the Transformation API for XML (TrAX), which provides a standard API for performing XML transformations. [17]

The key elements of the processing model of XSLT are rules and the processor. XSLT rules are defined by a combination of templates and patterns. The output is obtained by passing the XML source to the XSLT processor, which transforms it (see Illustration 4). The processor interprets the source tree node by node and tries to locate a template rule that contains a pattern that matches the particular node. When it locates this rule, it instantiates the template and creates the result. [11]

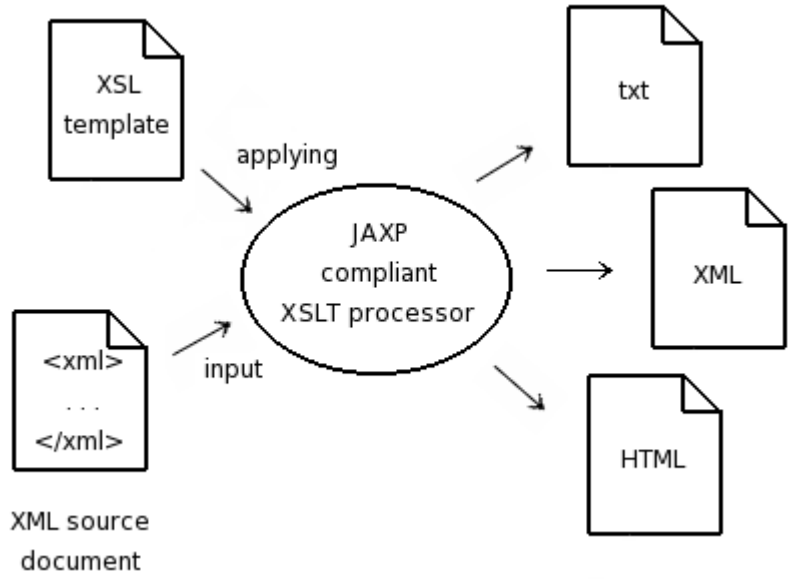

Illustration 4 XSLT processing model

To perform an XSL transformation, a Java programmer should do the following [11]:

1. Obtain a transformer class using a transformation factory class (cf. using SAX and DOM parser factories). Pass the transformer class the stylesheet for formatting the output.
2. Transform the data by using the transformer class.

Next, these steps are gone through in more detail.

3.4.1 Obtaining a Transformer Class

The class that performs the transformation is an instance of the javax.xml.transform.Transformer class. It can only be obtained through a factory class similarly to obtaining SAX and DOM parsers [11]. The factory class is called TransformerFactory and is instantiated like this:

```
TransformerFactory factory =
  TransformerFactory.newInstance();
```

After that, the transformer can be obtained. The factory method *newTransformer*, which is used for this purpose, is given a stylesheet wrapped in an instance of a StreamSource class as a parameter. The parameter could also be a SAX event reader (class SAXSource) or a DOM tree (class DOMSource) [11].

```
Transformer transformer =
  factory.newTransformer(new StreamSource("sheet.xsl"));
```

The instantiated transformer applies the stylesheet to the input XML data during the transformation process.

3.4.2 Performing the Transformation

Once the transformer is obtained, it can be used to perform the transformation by invoking its *transform* method. The parameters required for this are the XML source (a class that implements the Source interface) and the output target (a class that implements the Result interface):

```
transformer.transform(new StreamSource("source.xml"),
  new StreamResult("target.html"));
```

In the line above, the result is an HTML document produced by an instance of a StreamResult class, but as with the source, the result could also be written using ContentHandler callback methods (class SAXResult) or directly to a DOM Document object (class DOMResult) [11].

A complete sample implementation of a Java class that performs an XSL transformation to produce an HTML document from an XML document is provided in appendix F.

4 EVALUATION OF XML PROCESSING WITH JAVA

In the previous chapters, several mechanisms for processing XML data particularly from the point of view of Java were covered. In this chapter, the benefits and drawbacks of using Java for XML processing are discussed. The drawbacks of different processing models were listed in chapters 3.1.4, 3.2.4, and 3.3.3. This chapter is not meant to be a comparison between Java and other programming languages but rather a brief, objective survey of the field.

4.1 The Benefits

Java and XML fit together especially when they are used to build portable business applications. This is because Java is able to provide the portability of code that can be executed on various platforms and XML is able to provide the portability of data that can be processed on multiple platforms. [11] Platform-independence also makes changes in technology less critical [19].

Both Java and XML support Unicode [15, 4], which minimizes the risks of problems with character sets. Unicode also allows almost any information in any human language to be communicated.

By using Java system properties, JAXP allows any conformant parser or processor to be used for XML processing – not just Xerces or Xalan that are the default implementations for the newest version of JAXP [16]. It even allows this implementation to be changed at runtime. This improves portability and flexibility.

Many XML implementations for Java are open source, which allows programmers all over the world to develop, debug, and test the source code. It also enables anyone who needs a deeper insight into the behavior of these implementations

inspect the code and get the insight this way. In addition to those benefits, open source code, at least potentially, improves the security because anyone who wants to can check the source to see that no harmful code is hidden in it.

Big corporations, such as Sun Microsystems, IBM, and Microsoft, support XML, which drives its development. Partly with the help of considerable resources from these corporations completely free XML libraries for processing are developed or available already, some of which are for Java.

Java is nowadays very familiar to programmers [18], which makes extensive experience, support, and software available. All in all, Java seems to suit adequately for most XML processing applications.

4.2 The Drawbacks

Despite the many benefits of using Java to process XML data, every coin has two sides. XML itself has some disadvantages that could hinder its processing, not just with Java but with any other programming language too. First one of these concerns data types available: Even though XML Schema can be used to constrain XML data to a particular data type [23], basic parsing usually handles only strings. The second disadvantage is that XML does not suit well for modeling overlapping, non-hierarchical data [14].

Kirkegaard, Møller, and Schwartzbach argue that existing general-purpose programming languages, one of which is Java, do not provide any special support for XML transformations. According to them, dynamically generated XML documents are typically modeled as text strings (e.g. XHTML) or DOM trees, both of which are error-prone and tedious to use. [8]

It is generally known that performance of Java is not as good as for example performance of C. This issue is improving with every new version of Java virtual machine but if very high performance XML processing is needed, Java might not be the right choice.

To perform multiple concurrent operations safely, i.e. to achieve thread safety, XML transformations must be performed by using Templates class instead of TransformerFactory class [17]. This requires some extra programming. Using Templates class is not discussed in this thesis.

5 SUMMARY

In this thesis, different mechanisms for processing XML data in Java programming language were presented. Before this, an overview of XML – its history, its similarities with and differences to HTML, and the structure of XML documents – was given. Also Document Type Definitions (DTDs), XML Schemas, and the Extensible Stylesheet Language (XSL) were briefly discussed.

The main chapter of this thesis concentrated on covering the Java API for XML Processing (JAXP). The mechanisms discussed in this context were the following three processing models: Simple API for XML (SAX), Document Object Model (DOM), and JDOM. Every model has its strengths and weaknesses that were examined. Especially the drawbacks of using a certain model were listed. XSL Transformations (XSLT) were also discussed.

The suitability of Java for XML processing was evaluated. A complete list of strengths and weaknesses of Java and XML was not given, but rather a brief survey of the field. All in all, XML processing capabilities of Java are adequate for most applications.

The appendices provide concrete examples of how to use SAX, DOM, JDOM, and XSLT.

BIBLIOGRAPHY

[1] Armstrong E., Ball J., Bodoff S., Bode Carson D., Evans I., Green D., Haase K., Jendrock E. 2004. The J2EE 1.4 Tutorial For Sun Java System Application Server Platform Edition 8.1 2005Q1. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/. 16.5.2005.

[2] Bain S.L. 2002. XML Parsing Strategies for the Java™ Platform. JavaONE: Sun's 2002 Worldwide Java Developer Conference. San Fransisco, USA, March 2002.

[3] Bedunah J.B. 1999. XML: The Future Of The Web. Crossroads archive 6 (2). New York: ACM Press, 5-10.

[4] Bray T., Paoli J., Sperberg-McQueen C.M., Maler E., Yergeau F. (ed.) Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/REC-xml/. 16.5.2005.

[5] Brownell D., Megginson D. 2004. SAX. http://www.saxproject.org/. 16.5.2005.

[6] Hunter J., McLaughlin B. 2004. JDOM. http://www.jdom.org/. 16.5.2005.

[7] Kay M.H. 2003. XML five years on: a review of the achievements so far and the challenges ahead. Proceedings of the 2003 ACM symposium on Document engineering. Grenoble, France, November 20-22, 2003. 29-31.

[8] Kirkegaard C., Møller A., Schwartzbach M.I. 2004. Static Analysis of XML Transformations in Java. IEEE Transactions on Software Engineering 30(3), 181-192.

[9] Le Hégaret P., Whitmer R., Wood L. 2005. W3C Document Object Model. http://www.w3.org/DOM/. 16.5.2005.

[10] McLaughlin B. 2001. Java & XML, 2nd Edition: Solutions to Real-World Problems. New York: O'Reilly.

[11] Nagappan R., Skoczylas R., Sriganesh R.P. 2003. Developing Java Web Services: Architecting and Developing Secure Web Services Using Java. Indianapolis: Wiley Publishing Inc.

[12] Quin L. 2005. The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL/. 16.5.2005.

[13] Salminen A. 2004. XML Family of Languages. Overview and Classification of W3C Specifications. http://www.cs.jyu.fi/~airi/xmlfamily.html. 16.5.2005.

[14] Shaw K. 2004. XML and JavaScript – Lecture 10. http://www.stanford.edu/class/cs193i/slidesSum2004/XML-Javascript.pdf. 16.5.2005.

[15] Sun Microsystems. 2005. Java Technology. http://java.sun.com/. 16.5.2005.

[16] Sun Microsystems. 2005. Java Technology: XML http://java.sun.com/xml/. 16.5.2005.

[17] The Apache Software Foundation. 2004. Xalan-Java. http://xml.apache.org/xalan-j/. 16.5.2005.

[18] TIOBE Software BV. 2005. TIOBE Programming Community Index for May 2005. http://www.tiobe.com/tpci.htm. 16.5.2005.

[19] Tyma P. 1998. Why Are We Using Java Again? Communications of the ACM 41(6). New York: ACM Press, 38-42.

[20] W3 Schools. 2005. DTD Tutorial. http://www.w3schools.com/dtd/default.asp. 16.5.2005.

[21] W3 Schools. 2005. XML Tutorial. http://www.w3schools.com/xml/default.asp. 16.5.2005.

[22] W3 Schools. 2005. XSLT Tutorial. http://www.w3schools.com/xsl/default.asp. 16.5.2005.

[23] W3 Schools. 2005. XML Schema Tutorial. http://www.w3schools.com/schema/default.asp. 16.5.2005

[24] Weiss A. 1999. XML gets down to business. netWorker 3(3). New York: ACM Press, 36-43.

APPENDIX A: A SAMPLE HANDLER CLASS – SAX

```
import java.io.*;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class SAXHandler extends DefaultHandler {
  private static Writer out;

  public SAXHandler() throws UnsupportedEncodingException {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");
  }

  // Invoked when the parser encounters the beginning of the
  // document
  public void startDocument() throws SAXException {
    // Write an XML declaration on the display
    write("<?xml version='1.0' encoding='UTF-8'?>");
    nl();
  }

  // Invoked when the parser encounters the end of the document
  public void endDocument() throws SAXException {
    // Write the final newline and flush the output stream.
    // Possible I/O exceptions are wrapped in SAX exceptions.
    try {
      nl();
      out.flush();
    } catch (IOException ex) {
      throw new SAXException("I/O error", ex);
    }
  }

  // Invoked when the parser encounters a start tag
  public void startElement(String namespaceURI, String localName,
  String qualifiedName, Attributes attrs) throws SAXException {
    String elementName = localName;
    // If namespace processing is not enabled, use element's
    // qualified name instead of its local name
    if (elementName.equals("")) elementName = qualifiedName;
    write("<" + elementName);

    if (attrs != null) {
      for (int i = 0; i < attrs.getLength(); i++) {
        String attrName = attrs.getLocalName(i);
```

```
        // Same as before, use qualified name if local is not
        // available
        if (attrName.equals("")) attrName = attrs.getQName(i);
        write(" " + attrName + "=\"" + attrs.getValue(i) + "\"");
      }
    }

    write(">");
  }

  // Invoked when the parser encounters an end tag
  public void endElement(String namespaceURI, String localName,
  String qualifiedName) throws SAXException {
    write("</" + localName + ">");
  }

  // Invoked when the parser encounters character data
  // (e.g. elements)
  public void characters(char buf[], int offset, int len)
  throws SAXException {
    write(new String(buf, offset, len));
  }

  // Writes the string output to stream. I/O exceptions can occur
  // while writing but ContentHandler methods throw
  // SAXExceptions. This method wraps I/O exceptions in SAX
  // exceptions, which are thrown back to the SAX parser.
  private void write(String str) throws SAXException {
    try {
      out.write(str);
      out.flush();
    } catch (IOException ex) {
      throw new SAXException("I/O error", ex);
    }
  }

  // Write a new line to output stream
  private void nl() throws SAXException {
    String lineEnd = System.getProperty("line.separator");
    try {
      out.write(lineEnd);
    } catch (IOException ex) {
      throw new SAXException("I/O error", ex);
    }
  }
}
```

APPENDIX B: PARSING – SAX

```
import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.helpers.DefaultHandler;

public class SAXParserExample {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("Usage: java SAXParserExample" +
        " filename");
      System.exit(1);
    }

    try {
      // Instantiate the SAX event handler
      DefaultHandler handler = new SAXHandler();

      // Use the default (non-validating, non-namespace-aware)
      // parser
      SAXParserFactory factory = SAXParserFactory.newInstance();

      // Instantiate the SAX parser
      SAXParser parser = factory.newSAXParser();

      // Parse the input
      parser.parse(new File(args[0]), handler);
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

APPENDIX C: PARSING – DOM

```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;

public class DOMParserExample {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("Usage: java DOMParserExample" +
        " filename");
      System.exit(1);
    }

    try {
      // Use the default (non-validating, non-namespace-aware)
      // parser
      DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();

      // Instantiate the DOM parser (a document builder)
      DocumentBuilder builder = factory.newDocumentBuilder();

      // Parse the input and create the document object
      Document document = builder.parse(new File(args[0]));

      // Instantiate the object for traversing the DOM tree
      DOMTreeHandler treeHandler = new DOMTreeHandler();
      // Print out the contents of the Document object
      treeHandler.printTree(document);
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

APPENDIX D: AN EXAMPLE OF TRAVERSING A DOM TREE

```
import org.w3c.dom.DocumentType;
import org.w3c.dom.Entity;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;

public class DOMTreeHandler {
  // Prints some information about a single node. Note that
  // also Document objects are nodes.
  private void printNode(Node node) {
    String val = null;

    // Node's name (e.g. "#comment", "#text", or element's
    // tag name)
    val = node.getNodeName();
    if (val != null) {
      System.out.print("node=\"" + val + "\"");
    }

    // Node's value (null for most of the nodes)
    val = node.getNodeValue();
    if (val != null) {
      if (!val.trim().equals("")) {
        System.out.print(" nodeValue=\"" + val + "\"");
      }
    }
    System.out.println();
  }

  // Prints a complete tree on the display node by node
  public void printTree(Node node) {
    int type = node.getNodeType();
    switch (type) {
      case Node.DOCUMENT_TYPE_NODE:
        printNode(node);
        // If the node is a document type, also print every
        // entity declared in the DTD, both internal and external
        NamedNodeMap nodeMap =
          ((DocumentType)node).getEntities();
        for (int i = 0; i < nodeMap.getLength(); i++) {
          Entity entity = (Entity)nodeMap.item(i);
          printTree(entity);
        }
        break;
      case Node.ELEMENT_NODE:
```

```
        printNode(node);
        // Elements often have attributes, print them too
        NamedNodeMap attrs = node.getAttributes();
        for (int i = 0; i < attrs.getLength(); i++) {
          Node attr = attrs.item(i);
          printNode(attr);
        }
        break;
      default:
        printNode(node);
        break;
    }

    // If the node has children, print them
    for (Node child = node.getFirstChild(); child != null;
    child = child.getNextSibling()) {
      printTree(child);
    }
  }
}
```

APPENDIX E: PARSING – JDOM

```
import java.io.File;
import java.util.Iterator;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

public class JDOMParserExample {
  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("Usage: java JDOMParserExample" +
        " filename");
      System.exit(1);
    }

    try {
      // Use the default (non-validating, non-namespace-aware)
      // parser
      SAXBuilder builder = new SAXBuilder();
      Document document = builder.build(new File(args[0]));

      // Create the outputter and output the document on the
      // display
      XMLOutputter outputter = new XMLOutputter();
      outputter.output(document, System.out);

      // Also accessing elements individually is possible
      Iterator iter = document.getDescendants();
      while (iter.hasNext()) {
        System.out.println(iter.next());
      }
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

APPENDIX F: AN EXAMPLE OF XSLT

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class XSLTExample {
  public static void main(String[] args) {
    try {
      // Obtain a transformer factory
      TransformerFactory factory =
        TransformerFactory.newInstance();

      // Instantiate the transformer
      Transformer transformer =
        factory.newTransformer(new StreamSource("sheet.xsl"));

      // Perform the transformation
      transformer.transform(new StreamSource("source.xml"),
        new StreamResult("target.html"));
    } catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```