The Florida State University
College of Arts and Sciences

An Investigation of XML and XML Tools in Java

BY

Ashley S. Board

TECH REPORT TR-010402

April 2001

A project submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Project Committee Members:
Dr. Greg Riccardi (Major Professor)
Dr. Dan Schwartz
Dr. Robert Van Engelen

TABLE OF CONTENTS

ABSTRACT

This paper describes a project that investigates XML (Extensible Markup Language) and XML tools in Java. The project investigates the Document Object Model (DOM), the Simple API for XML (SAX), Document Type Definitions (DTDs), the Extensible Stylesheet Language (XSL), the Extensible Query Language (XQL) and the Project X XML processor from Sun Microsystems. DOM is an interface that specifies how XML documents can be manipulated in a standard way. SAX is an interface that specifies the processing of XML data through parser callbacks. XSL filters and transforms XML documents to produce new XML documents that may be in various formats. XQL simulates database queries by retrieving elements with certain properties from an XML document. Validating parsers use DTDs to ensure that XML documents meet certain structural and tag name constraints. This strategy allows applications to process XML data more effectively.

I decided to build a model of stock market data that is the focus of this investigation. I created the Stock Exchange Markup Language (SEML) to describe XML stock market data. This language describes four different types of SEML documents. These document types are stock price, stock report, industry and portfolio.

I developed software to create and manipulate SEML stock data and to present a web site that displays and publishes this data. One software component creates and manipulates stock price, stock report and industry documents. This software is used in command-line applications. A second software component uses Java Servlets and HTML forms as part of a web site. These servlets and forms are processed when a web client uses the web site. The servlets invoke methods to create, view and modify SEML portfolio documents. The web site software uses XSL transformations to display SEML stock data. The servlets and XSL are used to display SEML stock price, stock report and industry data.

I used several strategies to process XML stock data effectively. These strategies include

- XML to SEML transformations
- DOM tree manipulations
- customized element tag name to element class mappings
- XSL transformations (XSLT)

The XML to SEML transformations convert XML stock price data into SEML stock price data. DOM tree manipulations provide a convenient way for SEML applications to process SEML documents. Customized element tag name to element class name mappings are provided to an element factory that is used when constructing a DOM tree. The customized element classes allow application-specific behavior to be applied to certain element types. XSLT transforms and filters XML documents to produce dynamic HTML pages for the web site.

CHAPTER 1 - Introduction

The objective of this project is to investigate the use of XML and XML tools in Java. The realization of this goal is the creation of a web site that displays Extensible Markup Language (XML) stock market data. This data was chosen as the application to drive this investigation. XML [12] is used to describe data and provide logical structure to documents. It is an emerging standard for network data communication between organizations. Further details of XML will be introduced later in this document.

I created the Stock Exchange Markup Language (SEML) to describe XML stock data. SEML is an XML language with a vocabulary of tag names that describe data content and a grammar that describes the logical structure of SEML documents. The tag names and logical structure of SEML documents are defined by Document Type Definitions (DTDs).

SEML stock market data represents commonly available information in the stock exchange industry. Informational categories include daily stock prices, historical stock prices, periodic stock reports, industry comparisons and user portfolio data. Four classes of SEML documents were created to store statistical data in these categories. These classes are stock price, industry, stock report and portfolio. Each of these classes has its own DTD that defines the grammar for that document type.

Many Java packages and other XML tools were used in this project. I created a set of Java packages to produce XML documents, transform XML data into SEML, to manipulate SEML documents and to present web sites. Also, I created a collection of DTDs to constrain the structure of the four classes of SEML documents. In addition, I used the following packages that were created by some of the leaders in XML and Java-related technologies:

- Sun's Java Project X Technology Release 2 for XML document processing (Project X)
- GMD-IPSI Extensible Query Language (XQL) engine for XML document queries by Gerald Huck and Igno Macherius
- XT engine for Extensible Stylesheet Language (XSL) transformations by James Clark
- Java Servlet Development Kit (JSDK 2.0)
- Apache Web Server 1.3 configured with Apache JServ 1.0
- Java 2 Platform

The software for processing SEML data in this project has two primary components. The first component is software that I created to transform XML documents into SEML, and to create SEML stock report documents from text files of non-XML data. This software takes XML data and loads it as Java XML objects, transforms it into Java SEML objects, and outputs these objects as SEML. SEML industry documents then can be created from existing SEML stock price documents. The XML to SEML transformation makes the processing of stock data more effective.

The second component of the project is software that implements the web site. The software is used to transform and format XML data so that it can be presented effectively over the web. I created Java Servlet classes to produce a web site of dynamically displayed SEML stock price, stock report, portfolio, and industry data. Some of these classes enable users to create, view and modify their own SEML portfolio documents. These portfolio documents contain SEML data that is derived from preexisting SEML stock price documents.

The web site registers users, publishes SEML data, and captures SEML portfolio data. The web site contains a page that enables users to register for membership or log into an existing account. Web site members may view, create and update SEML portfolios using HTML forms. Both members and non-members can view stock report, stock price and industry data for stocks that have SEML data.

Fig. 1 shows the basic components and information flow that enable XML stock market data to be displayed on a web browser. The first step of the information flow is the transformation of XML stock data into SEML. After this step, the SEML data is available to some of the Java Servlets that are executed. This data becomes available to a web user after the user submits a request that is processed by one or more servlets. Some of the servlets display SEML data that has been processed by XSL transformations. In these servlets, an XSL processor transforms the SEML (XML) data into HTML. Then a servlet sends the HTML to the web browser.

Begin Figure 1

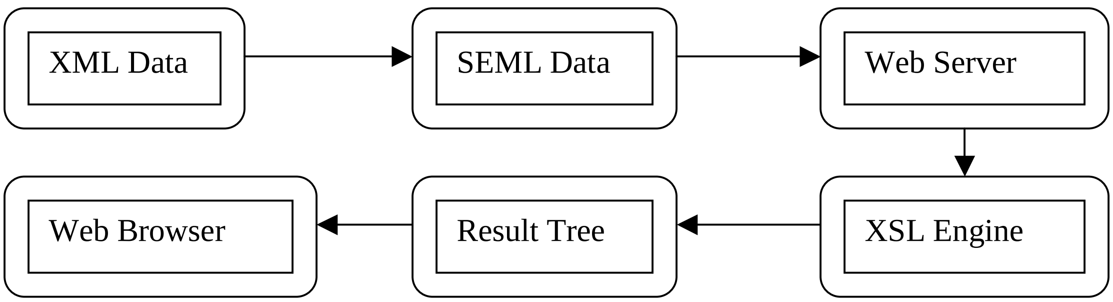


Figure 1. Basic Components and Information Flow to Display XML Stock Data to a Web Browser

CHAPTER 2 - XML Overview

The World Wide Web Consortium (W3C) has described XML as a framework for structured data on the Internet [12]. In 1996, the W3C created the XML Working Group to develop the XML specification [12]. The most recent version of the XML 1.0 W3C Recommendation is located at [12]. Since XML is an evolving technology, many of its components continue to change.

XML is a meta-language that provides a framework for the creation of other markup languages. A markup language is an XML language with its own pre-defined grammar. The grammar consists of the legal tag names and logical structure of the class of XML documents.
Although XML is a meta-language, it partially describes programs that process XML documents. Also, it is regarded as a set of document objects [12].

Some components for XML document processing include XSL, XQL, XPath and DTDs. These components are described as they occur in this paper.

XML provides a framework for data exchange with industry-specific markup languages [6]. These markup languages are defined by various organizations [1]. Some existing markup languages are the Weather Observation Markup Format (WOMF), the Chemical Markup Language (CML) and the Mathematical Markup Language (MML) that describe meteorological, chemical and mathematical data, respectively. Information about other markup languages is available at [5].

The restricted structure and nearly unlimited vocabulary of XML makes it a viable alternative to the Standard Generalized Markup Language (SGML) and the HyperText Markup Language (HTML). SGML is an XML superset that describes the markup of various forms of structured documents. Although SGML has been used extensively by organizations to exchange data, its flexibility results in computationally intensive processing that is unsuitable for data exchange over the web. That combined with the rigid presentation constraints of HTML provided the motivation for the XML specification [4].

XML will have an increasingly important role in data exchange over the web. Some of its primary advantages include:

- flexibility and robustness in providing data representations
- ability for documents to be dynamically generated without permanent storage
- extensibility
- ability to have existing XML documents reused in the production of new documents
- alternative use to proprietary data formats that restrict portability and prevent scalability [1]
- portability that makes it an ideal counterpart to the Java programming language [3]

CHAPTER 3 - Structure of an XML Document

3.1 XML Document Overview

Fig. 2 is an example of an XML document.

Begin Figure 2.

```
1   <?xml version=”1.0” encoding=”ISO-8859-1”?>
2   <?xml-stylesheet type=’text/xsl’ href=’stylesheet.xsl’?>
3   <!DOCTYPE stocks SYSTEM stocks “stockprice.dtd”>
4   <stocks>
5       <stock company=”CanoeRentals” ticker=”CRENT”>
6           <industry industryname=”Recreation”/>
7           <price>
8               <date>
9                   2000.08.24
10              </date>
11              <close>
12                  87.25
13              </close>
14              <open>
15                  87.00
16              </open>
17              <volume>
18                  300,000
19              </volume>
20              <high>
21                  87.75
22              </high>
23              <low>
24                  86.75
25              </low>
26          </price>
27      </stock>
28  </stocks>
```

Figure 2. XML document

Like all XML documents, this document is composed of *tags* and *text*. A tag consists of text enclosed in a pair of angle brackets (‘<’ and ‘>’). A tag is also called *markup*. Lines 1–8 are examples of tags.

Free text in an XML document is text that is not part of a tag. Lines 9, 12, and 15 of Fig. 2 are examples of free text.

XML tags come in several forms. Lines 1 and 2 of Fig. 2 shows a tag that begins with “<?”. This is an example of a tag for a processing instruction. Line 3 of Fig.2 begins with “<!”. This

denotes the beginning of a declaration. Line 4 contains a *begin tag.* It begins with “<” followed by a *tag name*. Each begin tag has a matching *end tag*, which is a tag that begins with “</” and has the same tag name as its corresponding end tag. Line 28 shows the end tag that is matched with the begin tag of Line 4. Similarly, lines 7 and 26 are a pair of begin-end tags. Line 6 is an empty element tag. Similar to a begin tag, it begins with “<” followed by a tag name. However, it ends with a “/>”.

The tags of an XML document allow it to have a logical structure and hierarchical representation. The tags or markup may be one of several forms including elements, entity references, declarations, and processing instructions. The free text that occurs outside of any matching pair of angle bracket markup delimiters is character data or content.

An XML document may be described in terms of its physical structure [4]. Its physical structure contains one or more constructs known as entities [12]. The XML document itself is an entity that is described as the document entity or root. Entities may be elements, declarations, character references, processing instructions, comments, or entity references [12]. Most of these types of entities are described later in this chapter.

The first section of an XML document is comprised of a header with processing instructions and possible declaration(s). A processing instruction conveys information to an application or a parser prior to XML document parsing. A declaration references a DTD that contains the grammar of a class of valid XML documents.

The header should begin with a processing instruction that indicates the XML version of the document. Line 1 of Fig. 2 specifies that version 1.0 of XML is used. The encoding attribute specifies a character representation that supports most of the western European characters. This is one of several encoding schemes that are available to represent characters that cannot be typed from an input device.

A second processing instruction refers to a stylesheet that transforms an XML document. Line 2 is an example of this instruction. Like all processing instructions, the outermost characters inside the angle brackets are ‘?’ characters. The first item after the ‘?’ character is an xml-stylesheet processing instruction target (PI target). It is conventional for instructions that reference a style sheet to have an "xml-stylesheet" PI target. The second item is a *type* attribute that specifies XSL as the type of stylesheet to be used. The *href* attribute specifies the location of the stylesheet that is applied to the XML document. Finally, a stylesheet instruction may have a *media* attribute that indicates the type of client for which a stylesheet is applied to an XML document.

The XML document often has a document type declaration (DOCTYPE declaration) that includes or points to a DTD. Every document that is validated against a DTD (discussed later) must have a document type declaration. The DTD grammar may be included in the declaration or outside of the document (e.g., in a text file). Line 3 of Fig. 2 is a document type declaration that externally references a DTD that is located in the "stockprice.dtd" file.

The first argument of the DOCTYPE declaration identifies the DOCTYPE that is the root element of the document. The root element of XML documents is described later in this paper. The second argument in the declaration is a SYSTEM keyword that indicates the location of a DTD. A PUBLIC keyword may be used instead of the SYSTEM keyword to denote a DTD that is publicly available.

Line 4 of Fig. 2 is a *stocks* element that is the root element of the document. The root element is the outermost element of an XML document. (By convention, element and attribute tag names are italicized in this paper). The *stocks* element comprises the logical structure of the XML document. The *stocks* element has one child *stock* element and no attributes. The *stock* element has *company* and *ticker* attributes (Line 5). The attribute values have `CanoeRentals` and `CRENT` values, respectively. The *stock* element also has *industry* and *price* elements. The *price* element has six child elements. Each of these child elements contains content between a matching set of angle bracket markup delimiters.

A document also is referred to as a combination of parsed and unparsed data. Parsed data include most of the constructs of a typical XML document. Unparsed data include CDATA sections that are passed directly to an application. A CDATA section generally contains a text sequence that either has one or more markup characters that must be represented literally and/or has whitespace that must be preserved. Whitespace includes newline, tab, carriage return, and blank space characters. Markup characters include angle brackets, quotation marks and ampersands. Everything inside of the ‘<!CDATA[‘ and ‘]]>’ character sequences represents the data of a CDATA section. One restriction on a CDATA section is that the ‘]]>’ character sequence cannot appear as the CDATA section’s data.

Fig. 3 is a CDATA section that escapes angle bracket characters so that they are not interpreted by a parser as markup delimiters. This section escapes these characters to avoid a parse-time error that would prevent an application from using the document that contains the section.

Begin Figure 3

```
<![CDATA [____:
        <item1> some text
        <item2> some more text
]]>
```

Figure 3. A CDATA section that escapes ‘<’ and ‘>’ characters

3.2 Logical Structure

The logical structure of an XML document refers to its elements. Elements are structured containers that provide logical meaning to their data components. An element allows its components to be depicted in a hierarchical manner. The root element is the outermost element of a document and it represents the logical structure of the document.

Every element has one or two tags that are enclosed in a set of matching angle brackets. The three types of element tags are *begin*, *end* and *empty*. Elements that have content and/or other elements (subelements) are required to have *begin* and *end* tags. Each element that has a *begin* tag must have a matching *end* tag. This means that the *begin* and *end* tags must have the same tag names. Empty tags may be used for elements that contain neither subelements nor content. An empty element has an *empty* tag and no other tags.

Each tag contains a tag name or element type identifier that describes the element. An element tag name must (1) begin with a letter or underscore, (2) not contain any blank spaces, and (3) not begin with the "xml" or "XML" character sequences. All tag names are case sensitive and tag name schemes should be consistent for similarly structured XML documents.

Any element may contain zero or more attributes. An attribute is a name-value pair that describes a single property of an element. All attributes of a non-empty element must occur in the *begin* tag of the element. All attributes of an empty element must occur in the *empty* tag. Attribute names have the same requirements as those of element tag names. An attribute value must occur inside single or double quotation marks. An element tag must not contain two attributes with the same name.

The following is an XML *stock* element with *price* and *date* subelements. The *stock* element and its subelements each have identical *begin* and *end* tag names. Each *begin* tag consists of the element tag name and zero or more attributes. The *stock* element *begin* tag has a *stock* tag name. This tag has *company* and *ticker* attributes with `XYZ` and `XYZZ` values, respectively. The *price* and *date* elements are children of the *stock* and *price* elements, respectively. The *price* element has a *price* tag name and a *close* attribute with a `75.25` value. The *date* element contains text with a `12-11-99` value. This element does not have subelements or attributes.

```
<stock company="XYZ" ticker="XYZZ">
        <price close="75.25">
                <date> 12-11-1999 </date>
        </price>
</stock>
```

The following is an empty *stock* element that has a *stock* tag name, two attributes and a ‘/’ character. Like all empty elements, the ‘/’ character is the last non-whitespace character inside the markup delimiters.

```
<stock company="XYZ" ticker="XYZZ"/>
```

Although attributes and subelements store data in an element, there are no strict rules that describe whether an attribute or subelement is more appropriate for representing a given component of element data. However, some suggested guidelines exist and I recommend using attributes and subelements for atomic and non-atomic data, respectively. For example, an identification number has no components. Therefore, the identification number may be more appropriately stored as an attribute. However, an element tag should contain at most a few attributes since XML document readability becomes compromised when many attributes appear in a tag. On the other hand, it generally is favorable to include attributes whenever possible since they typically require less physical space than subelements that have content.

Entity references may be used to represent the literal meaning of text characters that normally cannot be parsed. These characters include angle brackets, quotation marks and ampersands.

Pre-defined entity declarations allow entity references to escape these characters. An entity reference contains an ampersand, an entity name and a semicolon. An ‘&’ character is escaped by replacing it with an ampersand, an amp entity name and a semi-colon. Likewise, the ‘<’ character sequence denotes the literal meaning of the ‘<’ character. Other special characters and their corresponding entity references are given below.

```
>    &gt;
'    &apos;
"    &quot;
```

Entity references also may be used to replace frequently occurring blocks of text. The text may be included in the entity declaration or it may be located in an outside source that is referenced by the declaration. With the exception of the pre-defined entities, an entity declaration must occur before the first reference to it in a document.

CHAPTER 4 - Document Types and DTDs

A data object must conform to a set of well-formed constraints to be recognized as an XML document [12]. The XML 1.0 specification defines the constraints to which all XML documents must conform. A few of these constraints are that (1) an XML document must have one root element that contains all other elements in a document, (2) every non-empty element must have a properly nested set of *begin* and *end* tags, and (3) all attribute values must be delimited by quotation marks. Information about additional constraints is available at [12].

To improve XML data exchange between organizations, it may be necessary for applications to use valid XML documents that conform to a given DTD. A valid XML document is a well-formed XML document that conforms to the DTD referenced by the document's document type declaration. As mentioned previously, a DTD defines the logical structure and tag name constraints on a class of XML documents. These constraints enable organizations to more easily process XML data that meet their expectations.

An application must obtain access to a validating parser before the application can use a DTD to test a document for validity. A validating parser parses a data object and uses a DTD to determine if that data object is a valid XML document. To configure a validating parser using Project X, a `ValidatingParser` instance is created and then configured with an `XmlDocumentBuilder`, as shown below.

```
XmlDocumentBuilder builder = new XmlDocumentBuilder();
ValidatingParser parser = new ValidatingParser(true);
builder.setParser (parser);
```

The second statement of the following Java code creates a document type declaration and appends it to an XML document. The declaration enables a validating parser to test the document for validity against a DTD during subsequent parses of the document.

```
XmlDocument industryXmlDoc = …
industryXmlDoc.setDoctype ("industries", "industry.dtd", null);
```

A DTD is an XML document that defines the legal elements, attributes, external entities and element structures of a class of XML documents. The four types of markup declarations are element type, attribute list, notation and entity. An element type declaration has a content model that indicates allowable child elements and parsed character data [12]. The tag names, sequence lists and/or choice lists of an element type declaration are content particles [12]. An attribute list declaration may (1) require the presence of an attribute, (2) optionally allow an attribute in an element, (3) specify a default element attribute value, (4) fix an attribute value, or (5) specify a range of possible attribute values. Some element type and attribute list declarations are provided in the following portion of the SEML stock price DTD.

Begin Figure 4

```
<!ELEMENT stocks (stock)>
<!ELEMENT stock (industry, price+, latestinfo?)>
<!ATTLIST stock company CDATA #REQUIRED
```

```
                                   ticker  CDATA #REQUIRED>
<!ELEMENT industry (#PCDATA)>
<!ATTLIST industry industryname CDATA #IMPLIED>
<!ELEMENT price (#PCDATA | close | date | high | low | open | volume)*>
<!ELEMENT close (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT high (#PCDATA)>
<!ELEMENT low (#PCDATA)>
<!ELEMENT open (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
```

Figure 4. Portion of the SEML stock price DTD

The first element type declaration (indicated by the <!ELEMENT character sequence) defines a *stocks* root element that must have exactly one *stock* child element. This requirement is denoted by the absence of a special character at the end of the *stock* content particle.

The second element type declaration defines a *stock* element that has one *industry* element, at least one *price* element and at most one *latestinfo* element, in that order. The '+' character at the end of the *price* content particle constrains a *stock* element to having at least one *price* child element. The '?' character at the end of the *latestinfo* content particle constrains the *stock* element to having at most one *latestinfo* child element. The ',' characters require the content particles to occur in the sequence indicated.

The *stock* element also has an attribute list declaration (indicated by the <!ATTLIST character sequence). This declaration requires a *stock* element to have *company* and *ticker* attributes, as indicated by the #REQUIRED keywords. The CDATA keywords are unlike that of the CDATA section described in chapter 3. These keywords indicate that any legal strings may represent the *company* and *ticker* attribute values.

The third element type declaration defines an *industry* element with parsed character data and no subelements. The #PCDATA keyword specifies that parsed character data occurs as a child node of the *industry* element. The *industry* element has an optional *industryname* attribute in its attribute list declaration. The #IMPLIED keyword denotes that the presence of this attribute is optional. In this case, a missing attribute value is treated as if it does not exist in the *industry* element. The CDATA keyword is unlike that of the CDATA section described in chapter 3. This keyword indicates that any legal string may represent the *industryname* attribute value.

Finally, the price element type declaration allows *close*, *date*, *high*, *low*, *open* and *volume* subelements and parsed character data to occur in any sequence within the price element. The *price* element allows both subelements and parsed character data to occur within it. The '|' characters between the content particles indicate that the elements and character data may occur in any sequence. The '*' character allows any number of occurrences of the content particle sequence in the *price* element. The last six element type declarations define elements that have parsed character data but no child elements.

CHAPTER 5 - XML Processor Tools

I investigated two application-programming interfaces (APIs) that are available to provide XML processor standards. Conformance to these APIs enables application code to be reused with different processor implementations. The APIs are the Simple API for XML (SAX) and DOM. They are described in the following paragraphs.

5.1 SAX

SAX [7] is an event-based API that treats document parsing as a stream of events or parser callbacks. These callbacks are triggered by the detection of various tokens or character sequences during document parsing. The SAX 1.0 `org.xml.sax.DocumentHandler` interface defines the methods that are implemented by a SAX-conformant parser to use these callbacks. These callback methods may be customized to achieve application-specific functionality. For example, Project X allows an `XmlDocumentBuilder` to be configured with a SAX-conformant parser to construct a DOM tree during document parsing.

SAX was released in 1998 as a collaboration effort by a group of people from the XML-DEV mailing list. All major providers of XML processor software have implemented version 1.0 of SAX [1]. More recently, version 2.0 was created to improve version 1.0. In spite of its widespread use, SAX is not an official XML industry standard API as it is not supported by the W3C.

5.2 DOM

To manipulate an XML document and repeatedly to access portions of it, a DOM [8] tree must be constructed in memory. A DOM tree is composed of a set of nodes in which access to each node is relatively fast. DOM specifies how nodes may be traversed and manipulated.

The XML DOM specification is an official W3C standard that is an ongoing development. While DOM provides a common interface for manipulating an XML document, it does not specify how an application obtains a document. Therefore, parser implementations must dictate how XML documents are obtained. In addition, DOM defines very general solutions for applications; thus programmers may need to add functionality on top of DOM to achieve optimal DOM processor performance. However, these optimizations exist outside of the W3C standard, thereby making code less portable.

Several DOM recommendations either have been accepted or are works in progress. The DOM Level 1 specification is an accepted standard for XML processor implementations. The W3C director endorsed DOM as a recommendation in October 1998 [15]. The DOM Level 1 specification describes the navigation and manipulation of XML documents. The DOM Level 2 proposal was accepted as a recommendation in November 2000. It adds support for features such as XML queries and stylesheets. The DOM Level 3 specification is under development and had not been approved as a W3C standard at the time of this writing. Additional information about the DOM specifications and recommendations are available at [9].

One of the most important decisions that can confront an application writer is the choice of using SAX with or without DOM. SAX is primarily intended for applications that only need to access data in an XML document. An application that does not need to modify an XML document generally should use SAX without DOM. Such an application may extract data during document parsing and store that data into Java objects. For applications that only use SAX, document parsing typically is very fast and memory use is minimized.

However, an application that needs to update or create an XML document may need to use a DOM implementation. Although a SAX and DOM configuration is slower, there may not be a reasonable alternative for some applications.

As a final option, an application may use SAX to parse a document, extract its data and store the data into Java objects that are modified. Then the data in these objects can be written to an output source as well-formed XML (without using DOM).

5.3 Parsers for XML

An application must obtain access to an XML parser before the application can use XML data. Fortunately, a number of vendors provide parser implementations that conform to industry standards. Some widely used XML parser (and processor) tools are Java Project X by Sun Microsystems, XJParser by DataChannel, AElfred XML Parser by Microstar, IBM XML for Java and Oracle XML for Java. While Java, with its portability and large supply of APIs (McLaughlin), may be the most used programming language for XML processing, processors have been created in other languages.

This project uses the Java Project X Technology Release 2 XML toolkit from Sun Microsystems. The toolkit provides (1) Java classes that implement the SAX and DOM APIs, (2) two high performance SAX parsers, including a fast validating and slower non-validating parser, (3) DOM tree support that conforms to the W3C XML 1.0 specification, and (4) convenience classes that optimize document processing. Although the convenience classes aid application writers, they exist outside of the DOM 1.0 standard, and therefore, they are not available to other XML processor implementations. Furthermore, Project X is an experimental technology that is subject to future changes.

5.4 Extensible Stylesheet Language (XSL)

XSL [13] is composed of two languages that transform and format XML documents into result trees tailored to specific clients. One language is an XML vocabulary of literal elements that describe the format properties of XML data. A second XSL language consists of transformation elements that describe how certain input XML document elements are transformed. This language defines the XSL Transformation (XSLT) elements that control document format conversions. An XSLT element is distinguished from other elements by an "xsl" prefix in its tag name. The prefix indicates an element that is used for XSL transformations.

The "xsl" portion of the element tag refers to a namespace that dictates the interaction between an XML document and an XSL stylesheet. The tag name prefix identifies the XSL namespace to an XML processor or parser. This project uses stylesheet specifications that are given by the following attribute name-value pair: xmlns:xsl= “http://www.w3.org/1999/XSL/Transform". This is the specification for stylesheets that should conform to the most recent W3C recommendations [8]. The attribute value is the URI of the namespace that is located at the W3C site. Several other namespaces are available to the XML community.

Namespaces are referenced by element tag name prefixes. The prefixes provide additional meaning to elements that use namespaces. The namespace references are designed to prevent name collisions between elements with similar tag names in different classes of XML documents. An element namespace reference appears at the beginning of an element tag name and is separated from the base tag name by a single colon character. The namespace specification requires that the namespace be associated with a unique Uniform Resource Indicator (URI). This URI distinguishes elements of a particular namespace from elements of other namespaces.

The XSLT element syntax may specify operations such as conditional evaluation, element copying, element sorting, and element access. XSLT element types indicate how zero or more input XML document elements are transformed from one document format to another. These transformation elements comprise a vocabulary that is not intended for general-purpose XML documents.

An XSLT element allows input XML document elements to be matched by an XML Path Language (XPath) expression. An XPath expression results in the evaluation of zero or more elements or attributes, depending on the content of the input XML document. The expression can specify an element path that is relative to either the current context or the root of the document.

XSL is a W3C Candidate Recommendation that has been proposed as an alternative to Cascading Style Sheets (CSS). This recommendation is based on the Document Style Semantics and Specification Language (DSSSL) that has been used previously with SGML. Although XSL is a candidate recommendation, it is considered to be stable. Therefore, the W3C has encouraged its implementation.

Several methods exist to transform and format XML data using XSL. In one method, a Java XSLT processor uses an input XML document and XSL stylesheet to produce an XML result tree. The result tree is an XML document that is a transformed and formatted version of the input XML document. I used the Java-based XT processor by James Clark to transform and format XML documents. A second procedure allows XSL processing to occur inside the web browser. In this approach, an XML document that references an XSL stylesheet is sent to a browser. The browser applies the XSLT to the document and displays the result tree. Microsoft's Internet Explorer 5.0 is a web browser that provides this capability. A third method uses a web server-based content publishing framework to transform XML data using XSL. Apache Cocoon is one example of a framework that uses this approach.

5.5 XSLT Constructs

The *xsl:stylesheet* element is the first element of an XSL stylesheet. All XML documents must begin with an *xsl:stylesheet* or *xsl:transform* element. An *xsl:stylesheet* element may indicate the use of formatting objects for binary representations of XML document transformations. The use of these objects is specified by a *result-ns="fo"* attribute pair.

An *xsl:template* element contains rules that are applied to zero or more elements of an input XML document. The XSLT processor has access to all elements within the matched element of the template. The rules are applied to input XML document elements that match the XPath expression in the *match* attribute of the *xsl:template* element. For XML to HTML document transformations, the template rules typically are composed of transformation and HTML presentation elements.

The rules of the *xsl:template* element of Fig. 5 are applied to all input XML document elements that match the specified XPath expression. The elements that match the XPath expression are *stockreport* elements that (1) are children of the currently processed node (2) and have a *quarter* attribute value of `none`. In the following element, entity references denote single quotation marks that surround the 'none' character sequence of the XPath expression.

Begin Figure 5

```
<xsl:template match="stockreport[@quarter=&apos;none&apos;]">
    <p> <xsl:value-of select=”@year”/> </p>
</xsl:template>
```

Figure 5. An xsl:template element

The *xsl:apply-templates* element processes the rules of its corresponding *xsl:template* element. The *xsl:apply-templates* element normally has a *select* attribute (an XPath expression) that specifies the input XML document elements to process. For this template to be applied, and for any input document elements to be processed, the path to the matched input document elements must refer to the same context as an *xsl:template* match attribute. When the processor locates an input document element whose path matches that of the *xsl:apply-templates* element *select* attribute XPath expression, the rules of the corresponding *xsl:template* element are applied to the matched elements.

It is possible for an *xsl:apply-templates* element not to have a *select* attribute. In this case, the rules of the matched template element are applied to the elements that match the current *xsl:apply-templates* element context.

The following *xsl:apply-templates* element processes the template defined in Fig. 6.

Begin Figure 6

```
<xsl:apply-templates select="reports/stockreport[@quarter=&apos;none&apos;]" />
```

Figure 6. An xsl:apply-templates element

The *select* attribute XPath expression of this element differs from the *xsl:template* element match attribute. However, the reports element is the parent element of *stockreport* elements (not shown). The *xsl:template* element is in the context of the *reports* element while the *xsl:apply-templates* element is in the document entity context. Therefore, the *xsl:apply-templates* element XPath expression refers to the same context as the *xsl:template* element XPath expression. Additional examples of XSL constructs and transformations are given in Chapter 10.

5.6 Extensible Query Language (XQL)

XQL [16] simulates database queries for large XML documents, similar to the Structured Query Language (SQL). An XQL query returns a node set of XML elements that have certain properties. The node set is a well-formed XML document fragment but generally is not a well-formed XML document. A document fragment obeys all well-formed XML constraints except for the presence of a root element. However, a root element may be appended as a parent of the outermost node set element(s) to produce a well-formed XML document.

XQL is not a W3C standard but it soon may become the "de facto" standard in the XML community [3]. Texcel, webMethods and Microsoft created a joint proposal for XQL. The XML Working Group received the proposal in September 1998 [16]. XQL specifies how large XML documents may effectively be queried. Stated differently, the proposal specifies how XML document formats may be suitable for database queries [3]. Additional information about XQL is available at [14].

Several XQL processing engines are available to the Internet community. These include Datachannel by Rio, webMethods by B2B, Microsoft Internet Explorer 5.0 (XQL in IE 5.0) and GMD-IPSI by Gerald Huck with help from Igno Macherius. I used the GMD-IPSI engine (version 1.0.2), a Java-based application for querying large XML documents [2]. The GMD-IPSI is based on a complete implementation of the XQL language and a "persistent implementation of W3C-DOM Document objects" [2]. One caveat to the GMD-IPSI implementation is that it is an experimental development that may encounter future changes.

The GMD-IPSI implements an `XQL.execute` method that executes an XQL query. This method uses a query string and XML document as parameters and returns a node set of elements that match specified properties in the query string. The method header is given below.

```
XQL.execute (String queryString, XmlDocument xmldoc)
```

When given as the first argument of the `XQL.execute` invocation, the following query string indicates that a node set of all *stockreport* elements that (1) are descendants of a *reports* element (2) have *ticker* and *year* attributes that equal `AAAA` and `2000`, respectively and (3) have a *quarter* attribute are returned.

```
"reports//(stockreport[@ticker='AAAA' $and$ @year='2000' $and$ @quarter])"
```

The @ character indicates to an XSL processor that an attribute name is referenced. If the above query string is modified to the following, only those *stockreport* elements that (1) are child elements of a *reports* element, (2) have *ticker* and *year* attributes that equal `AAA` and `2000`, respectively, and (3) have a *quarter* attribute are returned.

```
"reports/(stockreport[@ticker='AAAA' $and$ @year='2000' $and$ @quarter])"
```

The '/' character denotes that the *stockreport* element must be a child of the *reports* element.

The following string is used in an XQL query that returns all *price* elements of a *stock* with a `CompanyXYZ` *company* attribute.

```
"stocks//stock[company='CompanyXYZ']//price??"
```

Stated differently, all *price* elements with (1) a *stock* superelement containing a `CompanyXYZ` *company* attribute (2) and a *stocks* superelement of the *stock* element are inserted into the node set. The '//' character sequence denotes that the *stockreport* element must be a descendant of the *reports* element. This string does not reference any attributes since no @ characters prefix tag names.

CHAPTER 6 - The Stock Exchange Industry

The stock exchange industry contains a wealth of statistical information that is available to stock investors. This section gives a description of some statistical quantities that stock investors use to evaluate stock companies. This description focuses on statistical data that is integrated into SEML stock documents. When referring to the stock exchange industry, our discussion is based on the daily trading activity of the New York Stock Exchange (NYSE) and NASDAQ.

The NYSE and NASDAQ feature regular stock trading sessions between 9:30 AM and 4:00 PM on weekdays throughout the year except for selected holidays. A relatively new feature of the stock exchange industry is after-hours trading. This trading period enables investors to trade stocks after regular trading hours.

Daily stock company trading information is of frequent use to investors. Some stock session statistics include daily stock close, open, high, and low prices as well as daily volume. The stock price is denoted by its value as the number of dollars per stock share. The daily volume signifies the number of company stock shares that are traded during a particular trading session. Another daily stock statistic is the absolute and percent difference in a stock's closing prices between the two most recent trading sessions.

Quarterly stock reports contain statistical quantities that cover a three-month time period. These reports typically include fiscal quarter dividends and earnings per stock share (EPS) for a company. The EPS specifies the quarterly company earnings per stock share, for the number of shares owned by shareholders. Newly released EPS reports sometimes result in short-term stock price changes. The dividend per share is the monetary value per stock share that a company has returned to its investors during the most recent twelve-month period (in which reports have been issued).

Some other statistics are derived from existing stock price data. The price/earnings ratio is a derived quantity that may suggest whether a stock is undervalued or overvalued. This quantity is computed from the current stock price and the latest twelve-month EPS. The stock yield is a second derived statistic that indicates the percentage of the stock share value that a company has returned to its shareholders during the most recent twelve-month period (in which reports have been issued).

Finally, various stock industry publications often provide charts that graphically depict stock data to investors. One example is a stock company performance/price trend or moving average chart. A moving average describes the mean of a stock's closing price for a given number of stock sessions (usually 10 or 20 days). Some investors examine a moving average chart to predict significant changes in near-future stock price behavior.

CHAPTER 7 - XML-SEML Transformations

The transformation of an XML stock price document into SEML is described in this section. The objective of this transformation is to convert XML data into Java objects and to convert Java objects into SEML data. One should not assume that an input XML stock price document has a particular tag set or logical structure. For this reason, it is inappropriate for such a document to be validated against the SEML stock price DTD. However, for the purposes of this discussion, it is assumed that an XML stock price document contains at least some of the stock price tag names that are used in an SEML stock price document.

The first transformation step begins with the construction of a DOM tree during the parsing of an XML stock price document. Figure 7 is a document that does not satisfy the stock price DTD rules of Fig. 4.

Begin Figure 7

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<stocks>
    <stock company="CompanyZ" ticker="COZZE" ceo="CougarSmith">
        <price date="1993.09.06" close="40.25" low="40.00" open="40.375">
            <high> 40.50 </high>
            <volume> 48,100 </volume>
            <headquarters> Siberia </headquarters>
        </price>
        <price date="1993.08.30" close="50.50" low="50.125" open="50.25">
            <high> 50.75 </high>
            <volume> 66,100 </volume>
        </price>
    </stock>
    <stock>
        <company numemployees="85,123"> CompanyAEO </company>
        <industry> Telecommunications </industry>
        <ticker> A9 </ticker>
        <price date="1997.08.08" close="51.75" low="51.00" open="51.25">
            <high> 51.875 </high>
        </price>
    </stock>
    <stock ticker="C3" industry="Retail" company="CompanyAEO">
        <price date="1997.08.10" low="52.125" open="52.25">
            <close> 53.75 </close>
            <high> 52.50 </high>
            <volume> 214,350 </volume>
        </price>
        <price date="1998.08.12" close="52.75" low="52.625" open="54.25">
            <high> 54.25 </high>
            <volume> 400,300 </volume>
        </price>
```

```
        </stock>
</stocks>
```

Figure 7. XML stock price document

In the figure, the first *stock* element has a *ceo* attribute and a *headquarters* subelement that are not part of the SEML stock price document tag set. The second *stock* element has a *numemployees* subtag that also is not part of this tag set. However, these tag values are not extracted from the XML stock price document. The extracted tag values are only those whose tag names are defined in the stock price DTD.

The MergeDocs.processDocuments method (part of which is shown in Fig. 8) retrieves the *company*, *ticker* and *industryname* tag values from each *stock* element.

Begin Figure 8

```
for (int i = 0; i < numXmlFiles; i++) {
xmldoc = builder.buildDocument (path + files[i]);
stockElements = xmldoc.getElementsByTagName ("stock");
if (stockElements == null) continue;
numStockElements = stockElements.getLength();

// for all stock elements in a document
for (int m = 0; m < numStockElements; m++) {
        stockElement = (Element) stockElements.item(m);
        company = xmlUtil.getXmlValue (stockElement, "company");
        industryname = xmlUtil.getXmlValue (stockElement, "industryname");
        ticker = xmlUtil.getXmlValue (stockElement, "ticker");
        priceElements = stockElement.getElementsByTagName ("price");
        numPriceNodes = priceElements.getLength();
        for (int n = 0; n < numPriceNodes; n++) {
                priceNode = priceElements.item (n);
                StockData stockData = new StockData ();
                stockData.setCompany (company);
                stockData.setIndustryname (industryname);
                stockData.setTicker (ticker);
                stockData.setClosingPrice (xmlUtil.getXmlValue (priceNode, "close"));
                stockData.setDate (xmlUtil.getXmlValue (priceNode, "date"));
                stockData.setDailyHigh (xmlUtil.getXmlValue (priceNode, "high"));
                stockData.setDailyLow (xmlUtil.getXmlValue (priceNode, "low"));
                stockData.setOpeningPrice (xmlUtil.getXmlValue (priceNode, "open"));
                stockData.setVolume (xmlUtil.getXmlValue (priceNode, "volume"));

                if (! ((stockDataMap.containsKey ((String) ticker)))) {
                        stockDataVector = new Vector (1, 5);
                        stockDataVector.addElement (stockData);
                        stockDataMap.put (ticker, stockDataVector);
```

```
29                  } else {
30                        stockDataVector = (Vector) stockDataMap.get ((String) ticker);
31                        stockDataVector.addElement (stockData);
32                        stockDataMap.put (ticker, stockDataVector);
33                  } // end if-else
34            } // end inner-for loop
35      } // end middle-for price element loop
36 } // end outer-for XML document loop
```

Figure 8. Part of method that places XML stock price data into a HashMap

The *stock* element of Line 7 uses the Element.getElementsByTagName method invocation of Line 11 to generate a NodeList of *price* element descendants of the *stock* element. A NodeList is a data structure that comprises a list of DOM tree nodes. The getElementsByTagName method header is given below.

public NodeList getElementsByTagName (String tagName);

The *price* elements that are generated in Line 11 are descendants of the processed *stock* element. In the inner for loop that begins on Line 13, the values of the first existing *date*, *open*, *close*, *high*, *low* and *volume* tags are extracted from each node of the NodeList. The tag names in the previous sentence are the content particles of the *price* element declaration in the stock price DTD of figure 4. For each *price* element, a StockData object is instantiated (Line 15) and all extracted tag values of the *price* element (Lines 19-24) are inserted into that object. StockData is a value container class with fields that hold the extracted tag values. Null values are assigned to StockData object fields whose expected tag values are missing from a particular element of the XML stock price document. After all fields have been given a value, the StockData object is added to a Vector (Lines 27 and 31).

The MergeDocs.processDocuments method has getXmlValue method invocations in Lines 8-10 and 19-24. These method invocations extract tag values from the XML document before the values are inserted into a StockData object. The getXmlValue method header is given below.

public String getXmlValue (Node node, String tagName);

The getXmlValue method returns the first occurrence of a tag or subtag value in the node that is given by the first argument. The second argument is the tag name of the value that is returned.

In the second transformation step, the contents of the Vector are printed to a file as well-formed XML. The MergeDocs.printStocksToXml method (not shown) retrieves each StockData object from the Vector. The field names and values of each StockData object are appended to a StringBuffer. These names and values are embedded in XML markup characters within the StringBuffer. Then the StringBuffer is converted to a String that is written to a file as well-formed XML.

To conform to the stock price DTD, each stock must have its price data in its own document. The MergeDocs class ensures that each stock in an XML document has its own Vector of stock data. A HashMap, which is used in Lines 25–32, has a mapping of stock tickers to stock price Vector objects. Therefore, the printStocksToXml method produces an XML document in which each *stock* element has all of its price data in a single element that pertains to that company.

Fig. 9 is the XML document that is produced after the second transformation step has been applied to the document in Fig. 8. All *price* elements of CompanyZ and CompanyAEO are children of a single *stock* element for each of these companies. The document grammar conforms to all but one of the stock price DTD rules. The only stock price DTD violation is that more than one *stock* element appears in the document.

Begin Figure 9

```
<stocks>
	<stock company="CompanyZ" ticker="COZZE">
		<industry industryname="Banking" />
		<price>
			<close> 40.25</close>
			<date>1993.09.06</date>
			<high> 40.50 </high>
			<low>40.00</low>
			<open>40.375</open>
			<volume> 48,100 </volume>
		</price>
		<price>
			<close> 50.50</close>
			<date>1993.08.30</date>
			<high> 50.75 </high>
			<low>50.125</low>
			<open>50.25</open>
			<volume> 66,100 </volume>
		</price>
	</stock>
	<stock company=" CompanyAEO " ticker=" A9 ">
		<industry industryname=" Telecommunications " />
		<price>
			<close> 51.75</close>
			<date>1997.08.08</date>
			<high> 51.875 </high>
			<low>51.00</low>
			<open>51.25</open>
			<volume>null</volume>
		</price>
		<price>
			<close> 53.75 </close>
			<date>1997.08.10</date>
			<high> 52.50 </high>
			<low>52.125</low>
```

```
                <open>52.25</open>
                <volume> 214,350 </volume>
            </price>
            <price>
                <close> 52.75</close>
                <date>1998.08.12</date>
                <high> 54.25 </high>
                <low>52.625</low>
                <open>54.25</open>
                <volume> 400,300 </volume>
            </price>
        </stock>
</stocks>
```

Figure 9. XML document after the second transformation step

The UpdateStockPrices class uses a document similar to that of figure 9. This class uses the price data for each stock and either inserts the data into an existing SEML stock price document for a particular company or it creates a new SEML stock price document for the company.

As indicated by the stock price DTD, SEML stock price documents are restricted to having one *stock* element. This constraint avoids possible performance degradation for large documents with data for multiple companies. If the stock price DTD had been created to allow multiple *stock* elements in a document (e.g., for multiple companies), memory storage requirements might create or exacerbate application performance problems.

After the transformation steps produce an XML stock price document that almost conforms to the DTD, stock data for each company is added to its own SEML stock price document. For each StockElement, the company *price* elements are appended to the StockElement of the company's existing SEML stock price document. If no SEML company stock price document exists for a particular company, a new document is created, and the processed company StockElement is appended to the new document. Fig. 10 is the SEML stock price document for `CompanyZ` after the first StockElement of Fig. 9 has been appended to the document. In this example, `CompanyZ` did not have an SEML stock price document prior to this step. If `CompanyZ` did have a document prior to this step, the two `CompanyZ` *price* elements would have been appended to the existing document StockElement.

Begin Figure 10

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE stocks PUBLIC 'stocks' 'stockprice.dtd'>
<stocks>
        <stock company="CompanyZ" ticker="COZZE">
            <industry industryname="Banking" />
            <price>
                <close> 40.25</close>
                <date>1993.09.06</date>
                <high> 40.50 </high>
```

```
            <low>40.00</low>
            <open>40.375</open>
            <volume> 48,100 </volume>
        </price>
        <price>
            <close> 50.50</close>
            <date>1993.08.30</date>
            <high> 50.75 </high>
            <low>50.125</low>
            <open>50.25</open>
            <volume> 66,100 </volume>
        </price>
    </stock>
</stocks>
```

Figure 10. SEML company stock price document that conforms to SEML stock price DTD

CHAPTER 8 - Customized Elements

An implementation of the Project X ElementFactory interface supports the creation of customized element classes. These classes allow applications to perform XML document processing that is specific to element types. To use customized elements, a SAX parser and a DOM DocumentBuilder must use a properly configured ElementFactory implementation. A customized ElementFactory can create customized element class instances during the parsing of an XML document and construction of a DOM tree. To create these classes during parse time, an ElementFactory implementation is given a mapping of element types to implementation classes. A properly configured factory produces a customized element class instance for each element whose type is mapped to a customized element class. These element classes should extend a class that implements the DOM Element and Node interfaces.

Project X supports customized element creation with a SimpleElementFactory class that implements the ElementFactory interface. A SimpleElementFactory is given a Dictionary of element type to implementation class mappings. Then the SimpleElementFactory is given to an XmlDocumentBuilder. The XmlDocumentBuilder then is configured with a SAX parser.

Fig. 11 (a) shows SEML stock price document element type to class name mappings that are given to a SimpleElementFactory. Typically, these mappings are specified in a configuration file that can be loaded by a Properties object. Fig. 11 (b) is Java code that creates a SimpleElementFactory (Line 2), retrieves a Properties object of mappings (Line 3), adds the Properties object to the SimpleElementFactory (Line 4) and configures an XmlDocumentBuilder to use the SimpleElementFactory (Line 6).

Begin Figure 11

```
1    stock=javaxml.elements.StockElement
2    price=javaxml.elements.PriceElement
3    stocks=javaxml.elements.StocksElement
4    industry=javaxml.elements.IndustryElement
5    *Element=javaxml.elements.CustomElement
```

(a)

```
1    Utility util = new Utility();
2    SimpleElementFactory factory = new SimpleElementFactory();
3    Properties elementClassTable = util.getProperties (configFile);
4    factory.addMapping (elementClassTable, null);
5    XmlDocumentBuilder builder = new XmlDocumentBuilder();
6    builder.setElementFactory (factory);
```

(b)

Figure 11. Customized element mappings and Java code to configure a customized element factory

When the XmlDocumentBuilder uses the mappings of Fig. 11 (a), each parse of a *stock* element *begin* (or *empty*) tag results in the instantiation of a StockElement. Line 1 specifies this mapping. Each parse of a *price* element *begin* (or *empty*) tag results in the instantiation of a PriceElement. Line 2 specifies this mapping. Mappings for StocksElement and IndustryElement classes (Lines 3 and 4) are handled similarly. In addition, Line 5 of Fig. 11 (a) has an *Element string to CustomElement class mapping. This mapping enables CustomElement class instances to be created for those elements whose types do not have explicit mappings. The CustomElement class is a superclass of the other customized element classes. Finally, if no *Element string exists in a Dictionary of mappings, Project X ElementNode instances are created for elements whose types do not have mappings.

CHAPTER 9 - Manipulation of SEML Documents

A significant component of this project is the Java classes that manipulate and create SEML stock documents. This section describes some of the Java methods that I created. The customized element classes described in the previous chapter contain many of these methods. Fig. 12 lists the element types and customized element classes that are used in each type of SEML document.

Begin Figure 12

SEML stock price documents:
stock=javaxml.elements.StockElement
price=javaxml.elements.PriceElement
stocks=javaxml.elements.StocksElement
industry=javaxml.elements.IndustryElement
*Element=javaxml.elements.CustomElement

SEML portfolio documents
stock=javaxml.elements.PortfolioStockElement
price=javaxml.elements.PriceElement
portfolio=javaxml.elements.PortfolioElement
*Element=javaxml.elements.CustomElement

SEML reports documents
reports=javaxml.elements.ReportsElement
stockreport=javaxml.elements.StockreportElement
*Element=javaxml.elements.CustomElement

SEML industry documents
industry=javaxml.elements.IndustryElement
stock=javaxml.elements.StockElement
*Element=javaxml.elements.CustomElement

Figure 12. Custom element classes available to the four classes of SEML documents

9.1 SEML Stock Price Documents

SEML stock price documents frequently should be processed to include computational data based on daily stock price updates. A PriceElement that has the latest stock price information for a company should be added to that company’s stock price document before any computations are performed. Some of the methods of the javaxml.elements package (which contains the customized elements classes) compute updated data. The data then are appended to the company’s SEML stock price document. Although the package has too many methods to provide a discussion of each, two methods are described in the following two paragraphs.

The 365-day stock company price range is one statistic that generally should be updated daily. The StockElement.get365DayPriceRange method (not shown) returns a company's 365-day price range. This method computes the company price range by using a javaxml.collections.PriceList of company PriceElement nodes. I created a PriceList data type that stores only PriceElement objects. The StockElement.getPriceElements method returns this PriceList to the get365DayPriceRange method. The StockElement.getPriceElements method is shown below.

```
1    public PriceList getPriceElements () {
2        NodeList priceList = getElementsByTagName ("price");
3        if (priceList == null) return null;
4        int numPriceElements = priceList.getLength();
5        PriceList priceElements = new PriceElementList (numPriceElements);
6        for (int j = 0; j < numPriceElements; j++) {
7            if (priceList.item(j) instanceof PriceElement)
8                priceElements.add ((PriceElement) priceList.item(j));
9    }
10   return priceElements;
} // end method
```

Line 2 generates a NodeList of price elements that are descendants of this StockElement. Line 5 instantiates a PriceList to hold these elements. Line 8 inserts an element into the PriceList if it is a PriceElement. *Price* elements that are not instances of the PriceElement class are not included in the PriceList. This constraint prevents the use of non-customized *price* elements that may have been added inadvertently to the document. In addition, this constraint may reduce or eliminate the casting that an application using the PriceList has to perform. For each PriceElement in the PriceList, the get365DayPriceRange method uses a Calendar object to determine if a PriceElement pertains to a stock session that occurred within 365 days of the current date. If the age of the PriceElement data is not greater than one year, the daily low and high stock prices are retrieved from the stock price document. These prices are compared to the minimum and maximum prices of a set of PriceElement nodes. This set is the nodes that have been processed in the execution of a for-loop that iterates the PriceElement PriceList (not shown).

9.2 SEML Stock Report Documents

As previously mentioned, one of the four types of SEML documents is a stock report document. This type of document contains periodic stock report data that typically is issued by a company every three months. Stock report data may represent statistics such as quarterly dividends and earnings. From these statistics, the latest stock company twelve-month earnings and dividends may be computed (if sufficient data is available).

The ReportsElement.getFourLatestQuarterNodes method returns the four *stockreport* elements that have the latest stock data for a particular company. The ticker argument of the following method header determines the company to which the *stockreport* elements pertain.

```
public XQLResult getFourLatestQuarterNodes (String ticker);
```

The getFourLatestQuarterNodes method executes a query that returns those *stockreport* elements that have a *quarter* attribute, the given *ticker* and the given *year*. An XQL query generates the node set of elements that is returned from this method as an XQLResult object. The following string is the first argument of the XQL.execute invocation in this method.

"reports//[stockreport[@ticker=' + ticker.trim() + ' and @year=' + Integer.toString(year) + ' and @quarter]]"

The *quarter* and *year* attribute values of the *stockreport* elements are used to determine the four *stockreport* elements that have the latest quarterly data. These four *stockreport* elements are inserted into a List that is returned to an application. The query can be executed to return elements from the previous year in addition to the given year. One can include additional terms in the query string to allow two different *year* values to be selected.

Finally, the getLast4QuarterQuantity method (not shown) returns the sum of the values of a specified tag in the four *stockreport* elements returned by the getFourLatestQuarterNodes method. The sum of these values may be used to compute a latest twelve-month statistic for a company. Two examples of statistics are the price/earnings ratio and yield for a stock company. These statistics can be computed from the earnings and dividend data of the *stockreport* elements. This is possible since the stock report DTD requires the existence of one *earnings* and one *dividend* attribute for each *stockreport* element.

9.3 SEML Industry Documents

The javaxml.industries package creates SEML industry documents from existing SEML stock price documents. Fig. 13 is an example of an SEML industry document.

Begin Figure 13

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE industry PUBLIC 'industry' 'industry.dtd'>
<?xml-stylesheet type='text/xsl'?>
<industry industryname="HomeProducts">
	<stock company="HomeProductsInc" ticker="hpinc">
		<latestinfo>
			<current currdate="2000.07.16" earnings="---" dividend="---" stochastic="1
2.20">
				<currclose>51.000</currclose>
				<tendaypricerange>60.00-49.75</tendaypricerange>
				<pricerange>52.875-49.750</pricerange>
			</current>
			<onedayprevdate="2000.07.15" pricechange="-2.50" pctpricechange="-
		0.05">
				<prevclose>53.500</prevclose>
			</oneday>
			<oneyearpricerange="70.00-44.12" pricechange="9.25"
			percentchange="0.22" close="41.750" date="1999.08.18" />
		</latestinfo>
	</stock>
```

</industry>

Figure 13. SEML industry document

The IndustryXmlDocuments class creates an SEML industry document for each industry that has at least one stock with SEML stock price data. For each stock in an industry, the stock's SEML stock price document is parsed and a DOM tree is constructed. The *industry*, *ticker* and *company* tag values are retrieved from a company's stock price document and placed into the appropriate StockElement of the industry document.

The stock data should be based on the latest updates to the SEML stock price document so that industry documents reflect the latest stock price data. The creation and modification of industry documents should be repeated for each company in an industry and for each industry in a document.

9.4 Industry Elements

The IndustryElement class has methods that compute price change data for all stocks in an industry. These stocks must have SEML stock price data. The IndustryElement. getIndustryPriceChangeAverage method returns a HashMap of average daily stock price change versus trading day mappings for the stocks in a particular industry. This method uses a List of StockElement objects that are retrieved from an SEML industry document.

The StockElement.getStockClosingPricesAndDates method is invoked for each StockElement in the List. This method returns a HashMap of closing prices and price ages for one particular stock. After a HashMap is generated for each stock in an industry, the getIndustryPriceChangeAverage method invokes the IndustryElement.getAverageDailyPriceChanges method.

The getAverageDailyPriceChanges method returns a HashMap of average percent daily stock price change versus trading day mappings for the stocks in a given industry. For each trading day of the time period, the following steps are repeated for each stock. The closing prices of a stock session and a previous stock session are retrieved from the stock's HashMap. The percent change between the two prices is computed. The total percent change is determined for the stocks in an industry. The industry average percent change is computed from this total. An average percent change versus trading day age mapping is inserted into a HashMap. After these steps have been performed for all trading days in a time period, a HashMap of the data is returned to the application.

9.5 Creation of SEML Portfolio Documents

The PortfolioElement, PortfolioXml and several Java Servlet classes enable SEML portfolio documents to be created and manipulated. The PortfolioElement class is an SEML element class that processes customized PortfolioElement objects. The PortfolioXml class creates portfolio documents and provides some other operations. The servlet classes allow users to create, view and update portfolios on the project web site.

The NewPortfolioForm Servlet displays a web form that enables a user to enter stock ticker symbols of stocks to be included in the user's portfolio. Only those stock tickers that have SEML stock price data are included. After the stock tickers have been submitted through a web form, a servlet-generated web page allows the user to enter the number of shares of each stock that are owned.

A PortfolioXml.createXmlPortfolio method creates the header processing instructions, document type declaration and root PortfolioElement that are necessary in an SEML portfolio document. A second createXmlPortfolio method uses a HashMap of stock ticker versus *numsharesowned* mappings to create PortfolioStockElement objects. This method also performs the same tasks as the method described in the first sentence of this paragraph. The PortfolioXml class also allows users to delete stocks from an existing portfolio.

9.6 Portfolio Elements

The PortfolioElement class processes customized portfolio elements. The PortfolioElement.appendStockElements method creates a PortfolioStockElement for each company whose ticker is in a Collection of stock tickers. Elements are created only for the companies that have SEML stock price documents. A configuration file of stock ticker to company name mappings provides the name of the company that corresponds to a given stock ticker.

To add stock data to a user portfolio, the PortfolioElement class allows stock data to be included for each stock in a portfolio. Line 2 of the following Java code creates a customized PortfolioStockElement. Line 3 changes its owner to this document. The fourth and fifth lines add *ticker* and *company* attributes to the PortfolioStockElement. Line 6 appends the new PortfolioStockElement to this PortfolioElement.

```
1	XmlDocument owner = (XmlDocument) getOwnerDocument();
2	stockEl = (PortfolioStockElement) owner.createElementEx ("stock");
3	owner.changeNodeOwner (stockEl);
4	stockEl.setAttribute ("ticker", ticker);
5	stockEl.setAttribute ("company", company);
6	appendChild (stockEl);
```

The PortfolioElement.getNetWorth method (not shown) returns the total stock share value of a user’s portfolio. The share value of each stock is computed from the number of shares that an investor owns and the latest stock closing price of a company. The latest closing price must be added to the document to reflect the latest stock data. The closing price is included as a descendant of a PriceElement. The total net worth of an investor’s stocks then is computed as the sum of the total stock share values of each company in the user portfolio.

CHAPTER 10 - Web Site Presentation Using Java Servlets and XSL

The software that produces the web site allows users to view historical and current stock price data, periodic stock reports and industry data. The software also allows users to create, edit and view SEML portfolio documents.

10.1 Historical Stock Price Documents

The web site allows users to view historical stock price data for any company that has an SEML stock price document. The following web page (Fig. 14) has a drop-down list of these companies.

Begin Figure 14

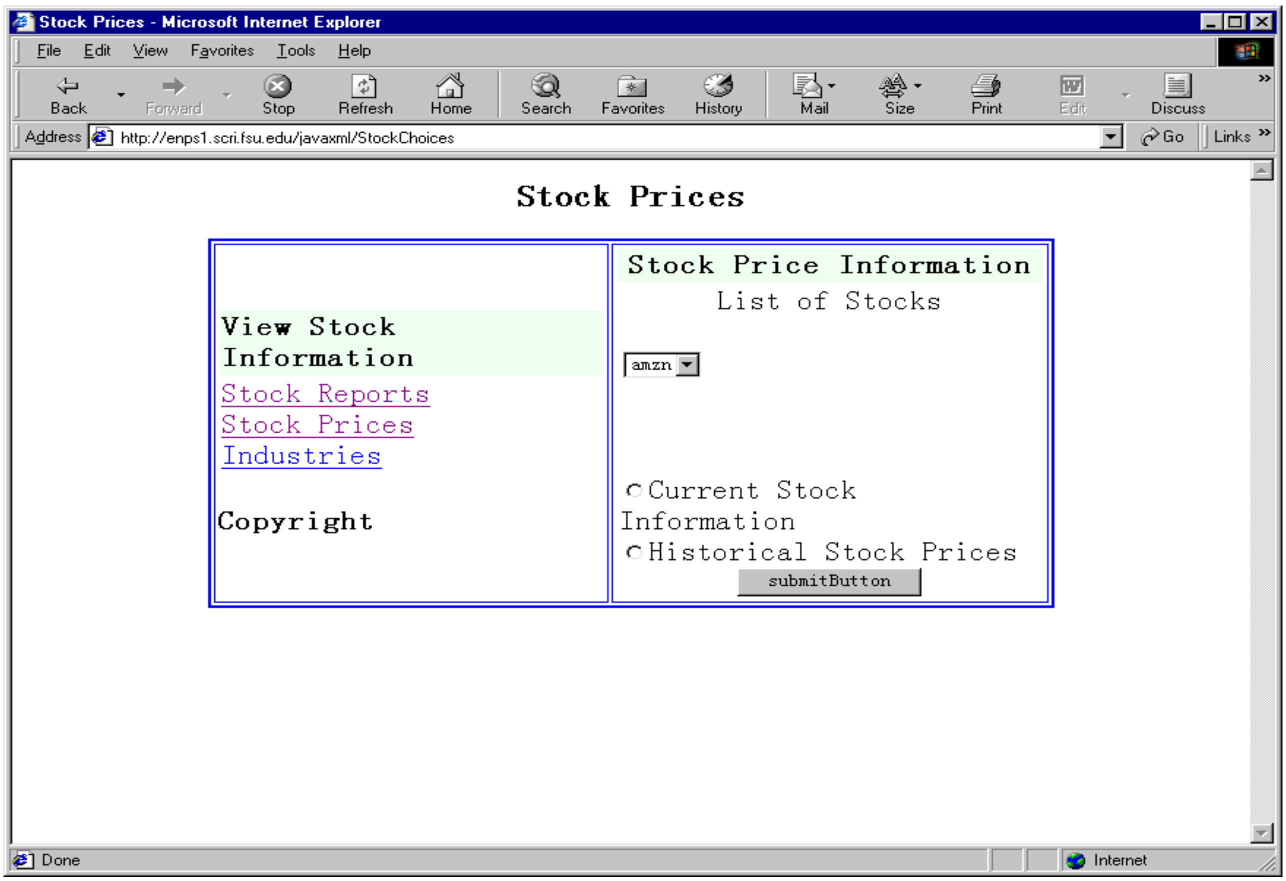


Figure 14. Web page that allows users to view stock price information for a company

After the user selects a company from the drop-down list and submits the web form, a servlet invokes methods that construct an SEML stock price document and XSL stylesheet. The document, stylesheet and an empty XML document are given to the XT processor. The processor produces a filtered and transformed version of the original stock price document. The new document contains HTML markup that can be displayed on the client's web browser. After the transformation process finishes, the servlet gives the result tree to a `PrintWriter` object. The `PrintWriter` outputs the result tree to the web browser.

To illustrate the XSL transformation process, the following example shows the documents that are used when a web user chooses to view historical stock price data for `CompanyABZ`. Fig. 15 (a) shows the SEML stock price document for this company.

Begin Figure 15 (a)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE stocks PUBLIC ’stocks’ ’file:/home/aboard/javaxml/xml/stockprice.dtd’>
<stocks>
	<stock company="CompanyABZ" ticker="CABZO">
		<industry industryname="Apparel" />
		<price>
			<close> 22.937</close>
			<date>2000.09.05</date>
			<high>24.000</high>
			<low>22.875</low>
			<open>23.437</open>
			<volume>4,649,800</volume>
		</price>
		<price>
			<close> 23.312</close>
			<date>2000.09.06</date>
			<high>23.750</high>
			<low>22.937</low>
			<open>22.937</open>
			<volume>3,171,100</volume>
		</price>
		<price>
			<close> 22.937</close>
			<date>2000.09.07</date>
			<high>23.750</high>
			<low>22.812</low>
			<open>23.375</open>
			<volume>5,393,700</volume>
		</price>
		<price>
			<close> 23.875</close>
			<date>2000.09.08</date>
			<high>23.937</high>
			<low>23.125</low>
			<open>23.250</open>
			<volume>3,972,900</volume>
		</price>
		<latestinfo>
```

```
                <current currdate="2000.09.08" earnings="2.26" dividend="1.17">
                        <currclose> 23.875</currclose>
                        <tendaypricerange>26.94-21.12</tendaypricerange>
                        <pricerange>23.937-23.125</pricerange>
                </current>
                <oneday prevdate="2000.09.07" pricechange="0.94" pctpricechange=”.04”>
                        <prevclose> 22.937</prevclose>
                </oneday>
        </latestinfo>
    </stock>
</stocks>
```

Figure 15 (a). SEML stock price document

The document has *company*, *ticker* and *industryname* attributes. It also has two *price* elements that have stock price data for a given date.

Fig. 15 (b) is the XSL stylesheet that transforms and formats the XML document of Fig. 15 (a).

Begin Figure 15 (b)

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0" xmlns:xt="http://www.jclark.com/xt" extension-element-prefixes="xt">
<xsl:output method="text" />
<xsl:strip-space elements="*" />
<xsl:template match="/">
        <html><head><title>Historical Stock Price Data</title>
        </head><body style="background-color:white" />
        <h1 align="center">Historical Stock Price Data</h1>
        <h2 align="center">
        <xsl:for-each select="/stocks/stock">
                <tr><p><td><span>
                        <xsl:value-of select="@company" />
                </span></td></p></tr>
        </xsl:for-each>
        </h2>
        <table align="left" border="1">
                <col width="175" />
                <col width="175" />
                <th align="left" width="50.0" bgcolor="fbd349">Date</th>
                <th align="left" width="50.0" bgcolor="fbd349">Close</th>
                <xsl:for-each select="/stocks/stock/price">
                        <tr><p><td><span>
                                <xsl:value-of select="date" />
```

```
24                  </span></td><td><span>
25                      <xsl:value-of select="close" />
26                  </span></td></p></tr>
27              </xsl:for-each>
28          </table>
29      </html>
30      </xsl:template>
31      </xsl:stylesheet>
```

Figure 15 (b). XSL stylesheet for XML document of figure 14 (a)

The *xsl:template* element of Line 5 contains the rules that are applied to the document of Fig. 15 (a). The rules of this template are applied to the logical section of the XML document since the element's *match* attribute denotes the document context. The *xsl:template* element contains HTML tags and XSL transformation elements.

Line 10 is the begin tag of an *xsl:for-each* element. The rules of this element are applied to all *stock* elements that are children of the root *stocks* element. The *select* attribute of this element is an XPath expression that determines a node set over which to iterate. The *company* attribute of each matched *stock* element is inserted into the result tree.

The XSLT processor uses the *xsl:value-of* element (Line 12) to retrieve the *company* attribute value of each *stock* element in the node set. This attribute value then is inserted into the result tree. If a *company* attribute is not present in the *stock* element, an empty string is embedded in HTML markup that is placed in the result tree. Moreover, each *company* attribute value in the result tree is embedded in HTML tags. The *company* attribute value replaces its corresponding *xsl:value-of* element in the stylesheet. The rules of the *xsl:for-each* element in lines 21-27 are applied to all *price* elements that are children of a *stock* element and grandchildren of the *stocks* root element. The XSLT processor uses the *xsl:for-each* element to obtain a node set of *price* elements over which to iterate. The *date* and *close* subelement values of each matched *price* element are inserted into a result tree. The result tree values replace the *xsl:value-of* elements in the stylesheet. The *xsl:value-of* element of Line 23 matches the *date* child element value of a *price* element and inserts that value into the result tree. Likewise, the *xsl:value-of* element of Line 25 matches the *close* child element value of a *price* element and inserts that value into the result tree. The result tree values are embedded by the HTML markup in the xsl:for-each element of Lines 21-27.

Fig. 15 (c) is the XML result tree that is the transformed and formatted version of the document in Fig. 15 (a). This document consists of an XML processing instruction followed by HTML elements.

The *span* elements of the result tree enclose the XML data that was retrieved from the input XML document. Line 7 is the *span* element that encloses the *company* attribute value that was retrieved from Line 12 of Fig. 15 (a). Lines 16 and 21 contain the *date* element values of the two *price* elements of Fig. 15 (a). Finally, lines 18 and 23 of this figure contain the *close* element values that were retrieved from the input XML document.

Begin Figure 15 (c)

```
1       <?xml version="1.0"?>
```

```
<html><head><title>Historical Stock Price Data</title>
        </head><body style="background-color:white" />
        <h1 align="center">Historical Stock Price Data</h1>
        <h2 align="center">
                <tr><p><td>
                        <span>Company55</span>
                </td></p></tr>
        </h2>
        <table align="left" border="1">
                <col width="175" />
                <col width="175" />
                <th align="left" width="50.0" bgcolor="fbd349">Date</th>
                <th align="left" width="50.0" bgcolor="fbd349">Close</th>
                <tr><p><td>
                        <span>1998.08.19</span>
                </td><tr>
                        <span>49.125</span>
                </td></p></tr>
                <tr><p><td>
                        <span>1998.08.20</span>
                </td><td>
                        <span>48.687</span>
                </td></p></tr>
        </table>
</html>
```

Figure 15 (c). XML result tree of XML document and XSL stylesheet of figures 15 (a) and (b)

Fig. 15 (d) is the portion of a Java Servlet that invokes a method to transform an XML document and send its result tree to a web browser. Line 3 creates an XSLTransformEngine object. The XSLTransformEngine.createTransform method in Line 3 accepts an input XML document, an XSL stylesheet and an XML result tree as arguments. The result tree should be an empty XML document when this method is invoked. The createTransform method produces the XML result tree and stores it in the result tree document. The second to last statement of the figure give the result tree to a PrintWriter. The PrintWriter sends the tree to the web browser.

Begin Figure 15 (d)

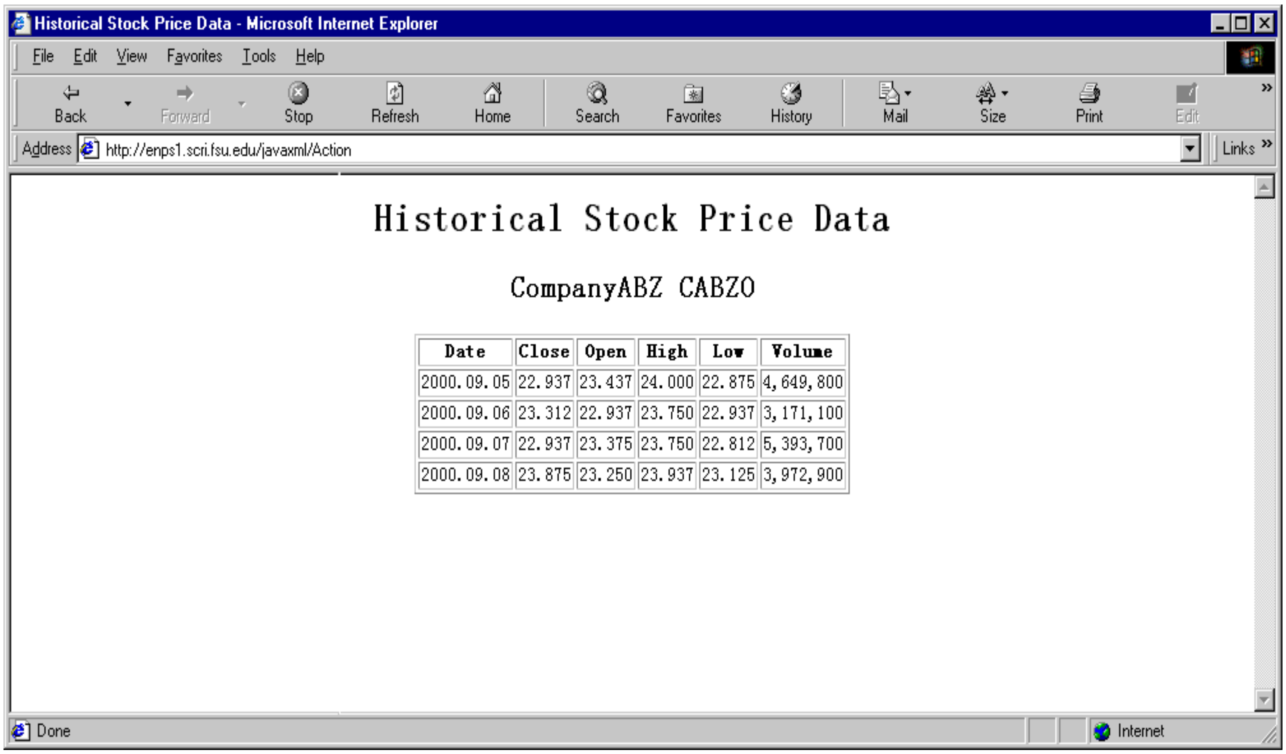
```
1       PrintWriter out = res.getWriter();
2       try {
3               newXSLTransformEngine().createTransform(xslStylesheet).transform(xmldoc,
                resultTree);
4       } catch (TransformException te) {
5               System.err.println (te.getMessage());
6       }
7       resultTree.write (out);
```

```
out.flush();
```

Figure 15 (d). Java code to produce a result tree from an XML document and XSL stylesheet

Figure 15 (e) is the HTML page that is generated from the transformation steps given in this section.

Begin Figure 15 (e)



Historical Stock Price Data

CompanyABZ CABZO

Date	Close	Open	High	Low	Volume
2000.09.05	22.937	23.437	24.000	22.875	4,649,800
2000.09.06	23.312	22.937	23.750	22.937	3,171,100
2000.09.07	22.937	23.375	23.750	22.812	5,393,700
2000.09.08	23.875	23.250	23.937	23.125	3,972,900

Figure 15 (e). Web page that displays historical stock price data for a company

10.2 Stock Report Documents

The web site allows users to view stock report data (e.g., earnings and dividends) for certain companies. The following web page has a drop-down list of companies that have SEML stock report data. Similar to the situation described in section 10.1, a Java Servlet uses XSL transformations to present SEML stock report data. The transformations generate an HTML result tree that is displayed to a user through a web browser.

To begin to illustrate the transformation of an SEML stock report document, Fig. 16 (a) is an SEML stock report document that has quarterly *stockreport* elements for two companies.

Begin Figure 16 (a)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE reports SYSTEM "stockreport.dtd">
<reports>
        <stockreport ticker="cozze" quarter="2" year="2000">
                <data earnings=".24" dividend=".16" />
        </stockreport>
        <stockreport ticker="cozze" quarter="1" year="2000">
                <data earnings=".46" dividend=".14" />
        </stockreport>
        <stockreport ticker="cozze" quarter="4" year="1999">
                <data earnings=".51" dividend=".14" />
        </stockreport>
        <stockreport ticker="cozze" quarter="3" year="1999">
                <data earnings=".15" dividend=".12" />
        </stockreport>
        <stockreport ticker="cozze" quarter="2" year="1999">
                <data earnings=".28" dividend=".12" />
        </stockreport>
        <stockreport ticker="cozze" year="1999">
                <data dividend=".57" earnings="2.18"/>
        </stockreport>
        <stockreport ticker="cozze" year="1998">
                <data dividend=".30" earnings="2.18"/>
        </stockreport>
        <stockreport ticker="cozze" year="1997">
                <data dividend=".45" earnings="2.18"/>
        </stockreport>
        <stockreport ticker="cozze" year="1996">
                <data dividend=".41" earnings="2.18"/>
        </stockreport>
        <stockreport ticker="cozze" year="1995">
                <data dividend=".17" earnings="2.08"/>
        </stockreport>
</reports>
```

Figure 16 (a). SEML stock report document

The *ticker* attributes are the symbols for these companies. Each *stockreport* element has *quarter* and *ticker* attributes. The *quarter* attributes pertain to a three-month time period. Each element has a *data* subelement. Each *data* element has EPS and dividend per share values that are stored as *earnings* and *dividend* attributes, respectively.

Fig. 16 (b) is an XSL stylesheet that specifies the XSL transformation and HTML format elements that are applied to the document of Fig. 16 (a).

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
        xmlns:xt="http://www.jclark.com/xt" extension-element-prefixes="xt">
<xsl:output method="text" />
<xsl:strip-space elements="*" />
<xsl:variable name="color">ead649</xsl:variable>
<xsl:variable name="hostname">enps1.scri.fsu.edu</xsl:variable>
<xsl:variable name="alignment">center</xsl:variable>

<xsl:template match="reports">
<html><head><title>Stock Report Data</title></head>
<body style="background-color:white">
<h1 align="center">Stock Report Data</h1>
<h3 align="center">MotorWorld(cozze)</h3>
<table align="center">
        <tr><td>
        <table align="center" border="1">
                <h4 align="center">Annual Data</h4>
                <th align="{alignment}" bgcolor="{$color}"> Year</th>
                <th align="{$alignment}" bgcolor="{$color}" >Dividend</th>
                <th align="{$alignment}" bgcolor="{$color}" >Earnings</th>
                <xsl:apply-templates select="stockreport[@quarter=&apos;none&apos;]" />
        </table></td>
        <td width="200">
        <table width="200"><tr align="right"><td align="right">
                <a href="http://{$hostname}/javaxml/StockReports">Stock Reports</a>
                </td></tr>
                <tr align="right"><td align="right">
                <a href="http://{$hostname}/javaxml/StockChoices">Stock Prices</a>
                </td></tr>
                <tr align="right"><td align="right">
                <a href="http://{$hostname}/javaxml/IndustryServlet">Industries</a>
                </td></tr>
        </table></td></tr>
        <tr><td><hr />
        <table align="left" border="1">
                <h4 align="center">Quarter Data</h4>
                <th align="{$alignment}" bgcolor="{$color}">Year</th>
                <th align="{$alignment}" bgcolor="{$color}">Quarter</th>
                <th align="{$alignment}" bgcolor="{$color}">Dividend</th>
                <th align="{$alignment}" bgcolor="{$color}">Earnings</th>
                <xsl:apply-templates select="stockreport[@quarter and not (@quar
      ter=&apos;none&apos;)]" />
        </table></td></tr></table></body></html>
</xsl:template>
```

```
43   <xsl:template match="stockreport[@quarter=&apos;none&apos;]">
44       <xsl:if test="self::stockreport[@ticker=&apos;cozze&apos;]">
45           <td><tr height="3"><p>
46           <td> <xsl:value-of select="@year" /> </td>
47           <td> <xsl:value-of select="data/@dividend" /> </td>
48           <td> <xsl:value-of select="data/@earnings" /> </td>
49           </p></tr></td>
50       </xsl:if>
51   </xsl:template>


52   <xsl:template match="stockreport[@quarter and not (@quarter=&apos;none&apos;)]">
53       <xsl:if test="self::stockreport[@ticker=&apos;cozze&apos;]">
54           <td><tr height="3"><p><td><xsl:value-of select="@year" />
55           </td><td><xsl:value-of select="@quarter" />
56           </td><td><xsl:value-of select="data/@dividend" />
57           </td><td><xsl:value-of select="data/@earnings" />
58           </td></p></tr></td>
59       </xsl:if>
60   </xsl:template>
61   </xsl:stylesheet>
```

Figure 16 (b). XSL stylesheet applied to XML document of figure 16 (a)

The XSLT processor applies the first *xsl:template* element (Lines 8-42) of Fig. 16 (b) to the entire logical section of the stock report document. This element matches the root *reports* element of the document. The element has HTML tags and two *xsl:apply-templates* elements.

The *xsl:apply-templates* element of Line 20 processes the rules of the *xsl:template* element of Lines 43-51. The *xsl:apply-templates* element has an XPath expression that references all *stockreport* elements that have a *quarter* attribute value of `none`. This attribute value denotes *stockreport* elements that have annual stock report data. The *xsl:template* element of Lines 43-51 has an *xsl:if* element (Lines 44-50). The *xsl:if* element further constrains the elements that are processed by the rules of the *xsl:template* element. The *xsl:if* element only processes *stockreport* elements that have *ticker* and *quarter* attribute values of `cozze` and `none`, respectively. The *xsl:value-of* elements of Lines 46-48 obtain the *year*, *earnings* and *dividend* attribute values of the *stockreport* elements that satisfy the *xsl:if* element condition. These attribute values are inserted inside HTML markup in a result tree.

The *xsl:apply-templates* element of Line 40 processes the rules of the *xsl:template* element of Lines 52-60. Although the XPath expressions of these two elements may appear to differ, their context is the same. The XPath expression of the *xsl:apply-templates* element forces the XSLT engine to process *stockreport* elements that have quarterly stock report data. Quarterly stock report data is indicated by a *quarter* attribute value that is not equal to `none`. The *xsl:template* element has an *xsl:if* construct. This construct only allows *stockreport* elements that have a *ticker* attribute value of `cozze` to be processed. The *quarter*, *year*, *dividend* and, *earnings* attribute values of the processed elements are inserted into a result tree. These values are surrounded by HTML markup that also is inserted into the result tree.

Fig. 16 (c) is the document that results from the application of the stylesheet in Fig. 16 (b) to the XML document in Fig. 16 (a).

```
<?xml version="1.0"?>
<html><head><title>Stock Report Data</title></head>
<body style="background-color:white">
<h1 align="center">Stock Report Data</h1>
<h3 align="center"> (cozze)</h3>
<table align="left">
<tr><td><table align="center" border="1">
<h4 align="center">Annual Data</h4>
        <th align="left" width="100.0" bgcolor="fbd349">Year</th>
        <th align="left" width="100.0" bgcolor="fbd349">Dividend</th>
        <th align="left" width="100.0" bgcolor="fbd349">Earnings</th>
        <td><tr height="3"><p>
             <td width="75">1999</td>
             <td width="75">.57</td>
             <td width="75">2.18</td>
            </p>
         </tr>
        </td>
        <td>
         <tr height="3">
            <p><td width="75">1998</td>
             <td width="75">.30</td>
             <td width="75">2.18</td>
            </p>
         </tr>
        </td>
        <td>
         <tr height="3">
            <p>
             <td width="75">1997</td>
             <td width="75">.45</td>
             <td width="75">2.18</td>
            </p>
         </tr>
        </td>
        <td>
         <tr height="3">
            <p>
             <td width="75">1996</td>
             <td width="75">.41</td>
```

```
              <td width="75">2.18</td>
            </p>
      </tr>
    </td>
    <td>
      <tr height="3">
            <p>
              <td width="75">1995</td>
              <td width="75">.17</td>
              <td width="75">2.08</td>
            </p>
      </tr>
    </td>
  </table>
</td>
<td width="200">
  <table width="200">
    <tr align="right">
      <td align="right">
            <a />
      </td>
    </tr>
    <tr align="right">
      <td align="right">
            <a />
      </td>
    </tr>
    <tr align="right">
      <td align="right">
            <a />
      </td>
    </tr>
  </table>
</td>
</tr>
<tr>
<td>
  <hr />
  <table align="center" border="1">
    <h4 align="center">Quarter Data</h4>
    <th align="left" width="75.0" bgcolor="fbd349">Year</th>
    <th align="left" width="75.0" bgcolor="fbd349">Quarter</th>
    <th align="left" width="75.0" bgcolor="fbd349">Dividend</th>
    <th align="left" width="75.0" bgcolor="fbd349">Earnings</th>
    <td>
      <tr height="3">
```

```
        <p>
         <td width="75">2000</td>
         <td width="75">2</td>
         <td width="75">.16</td>
         <td width="75">.24</td>
        </p>
  </tr>
</td>
<td>
  <tr height="3">
        <p>
         <td width="75">2000</td>
         <td width="75">1</td>
         <td width="75">.14</td>
         <td width="75">.46</td>
        </p>
  </tr>
</td>
<td>
  <tr height="3">
        <p>
         <td width="75">1999</td>
         <td width="75">4</td>
         <td width="75">.14</td>
         <td width="75">.51</td>
        </p>
  </tr>
</td>
<td>
  <tr height="3">
        <p>
         <td width="75">1999</td>
         <td width="75">3</td>
         <td width="75">.12</td>
         <td width="75">.15</td>
        </p>
  </tr>
</td>
<td>
  <tr height="3">
        <p>
         <td width="75">1999</td>
         <td width="75">2</td>
         <td width="75">.12</td>
         <td width="75">.28</td>
        </p>
```

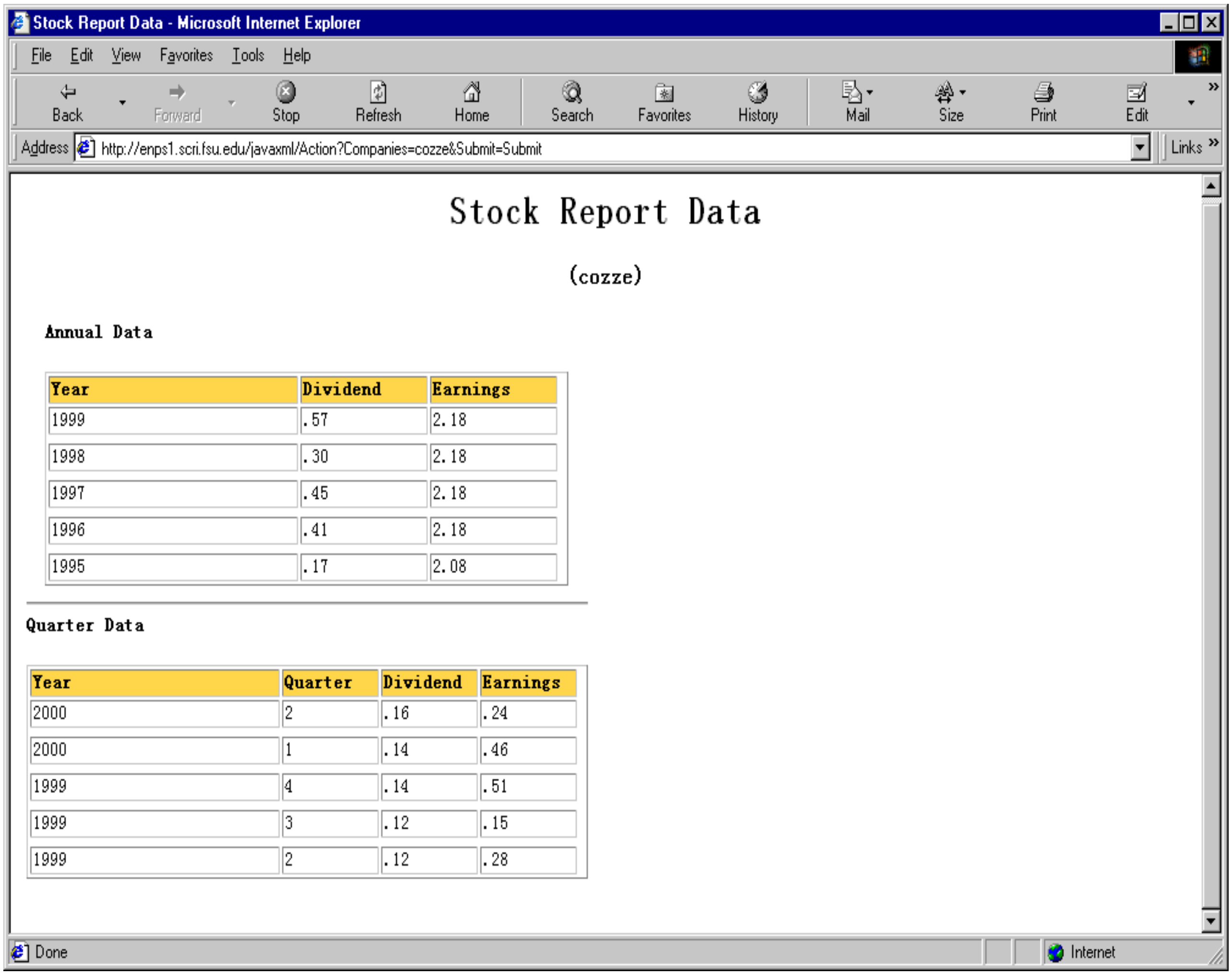
```
          </tr>
         </td>
        </table>
       </td>
    </tr>
   </table>
  </body>
</html>
```

Figure 16 (c). XML result tree of the XML document and XSL stylesheet of Fig. 16 (a) and (b)

This result tree is composed of an XML processing instruction and HTML tags. Many of the HTML elements have values that are extracted from the input XML document of Fig. 16 (a). The lines with *td* element names and *width* attributes are HTML elements that contain some of these values. These values replace their corresponding *xsl:value-of* elements in the stylesheet of Fig. 16 (b). Stated differently, the values replace lines 46-48 and 55-57 of Fig. 16 (b).

Fig. 16 (d) is an HTML page that results from the transformation process that uses the documents of Fig. 16 (a) and (b).

Begin Figure 16 (d)



Stock Report Data

(cozze)

Annual Data

Year	Dividend	Earnings
1999	.57	2.18
1998	.30	2.18
1997	.45	2.18
1996	.41	2.18
1995	.17	2.08

Quarter Data

Year	Quarter	Dividend	Earnings
2000	2	.16	.24
2000	1	.14	.46
1999	4	.14	.51
1999	3	.12	.15
1999	2	.12	.28

Figure 16 (d). Tables of annual and quarterly stock report data for a stock company

10.3 Sorting XML Elements Using XSL

This section describes an XSL stylesheet that sorts an SEML stock price document on the basis of *date* element values. This transformation step can make XML document processing more effective for some applications. For example, an XML-RPC client/server application may be configured so that XML documents always are sorted on the server before a client application accesses XML data stored on the server. This may benefit a client application that only needs to retrieve a small subset of the most recent XML stock price data. In this case, it may be unsuitable for an application to traverse an entire document to ensure that it has the most recent price data.

The stylesheet of Fig. 17 sorts the price elements of an SEML stock price document.

Begin Figure 17

```
<?xml version="1.0"?>
```

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0" xmlns:xt="http://www.jclark.com/xt" extension-element-prefixes="xt">
<xsl:output media-type="text/xml" method="xml" indent="yes" encoding="ISO8859-1"
      doctype-system="javaxml/xml/stockprice.dtd"/>
<xsl:strip-space elements="*" />
<xsl:template match="stocks">
        <xsl:element name="stocks">
                <xsl:apply-templates select="stock"/>
        </xsl:element>
</xsl:template>

<xsl:template match="stocks/stock">
        <xsl:copy>
                <xsl:for-each select="@*">
                        <xsl:copy/>
                </xsl:for-each>
                <xsl:apply-templates select="industry"/>
                <xsl:apply-templates select="price">
                        <xsl:sort select="date" order="descending"/>
                </xsl:apply-templates>
                <xsl:apply-templates select="latestinfo"/>
        </xsl:copy>
</xsl:template>

<xsl:template match="stocks/stock/industry">
        <xsl:copy>
                <xsl:for-each select="@*">
                        <xsl:copy/>
                </xsl:for-each>
        </xsl:copy>
</xsl:template>

<xsl:template match="stocks/stock/price">
        <xsl:element name="price">
                <xsl:copy-of select="*"/>
        </xsl:element>
</xsl:template>

<xsl:template match="stocks/stock/latestinfo">
        <xsl:element name="latestinfo">
                <xsl:copy-of select="*"/>
        </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

Figure 17. XSL stylesheet that sorts an XML stock price document

This stylesheet is divided into several templates that provide easier stylesheet maintenance and readability.

The *xsl:template* element of Lines 5-9 has an *xsl:element* element with a *name* attribute value of *stocks*. The latter element copies the *stocks* tag name to the result tree. This tag name is embedded in markup delimiters when placed in the result tree. All other result tree data will be contained within the skeleton that is produced by the *xsl:element* element. The skeleton is shown below.

```
<stocks>

</stocks>
```

The *xsl:apply-templates* element of Line 7 processes the *xsl:templates* element of Lines 10-21. Lines 11 and 20 are the tags of an *xsl:copy* element that copies the *stock* element name to a result tree and inserts the name in markup delimiters. After this step, the logical structure of the result tree is the following.

```
<stocks>
  <stock>
  </stock>
</stocks>
```

Lines 12-14 contain an *xsl:for-each* element that inserts each *stock* element attribute into the result tree. The *xsl:for-each* element has a *select* attribute with a "@*" character sequence. The second character is a wildcard that forces all attributes of the *stock* element to be copied into the result tree. Line 15 applies a template that matches all *industry* elements that are children of a *stock* element and grandchildren of the root *stocks* element. The *xsl:template* element of Lines 22-28 copies all *industry* elements and their attributes into the result tree.

Lines 16-18 apply the template of lines 29-33 to all *price* elements that are children of a *stock* element and grandchildren of the *stocks* element. Line 30 copies the *price* element tag name and markup delimiters into begin and end tags in the result tree. Line 31 copies the subelements and their text into the result tree. Inside the *xsl:apply-templates* element of Lines 16-18, an *xsl:sort* element appears. This element sorts the *price* elements based on the *date* child element value. The *order* attribute of the *xsl:sort* element indicates that the order is descending. The *xsl:sort* element may only appear as a child of an *xsl:apply-templates* element or an *xsl:for-each* element.

CHAPTER 11 - Conclusions

The objective of this project is to investigate XML and XML tools in Java. An underlying goal of this project is to use these tools to process XML stock market data effectively. The realization of these objectives is a web site that presents XML stock market data to web users.

I employ several strategies to process XML stock market data effectively. One strategy is the creation of SEML to describe stock market data. Several of my Java applications use SEML to process XML data more effectively. The SEML DTDs constrain these and other applications to using documents whose structures conform to a particular DTD. As a result of the DTD constraints, these applications require fewer instructions and less error checking.

The customized element factories enable SEML documents to be processed with application-specific behavior. As a result, many applications use elements that are instances of specific classes. This makes SEML applications easier to use and maintain. The XSL transformations (XSLT) that I use make SEML stock data processing more effective. These transformations produce dynamic web pages by separating the XML data content from the HTML format properties.

There are other XML issues and tools that I would like to have investigated if this project was of a longer duration. One issue is the creation of XSL stylesheets to transform XML documents into multiple formats for various clients. The Apache Cocoon publishing framework has this capability. Second, I would like to have closely investigated XML software for distributed computing. Two examples are the Simple Object Access Protocol (SOAP) and XML Remote Procedure Call software (XML-RPC). SOAP is a protocol that allows remote procedure calls to be executed on other machines using XML and the HyperText Transfer Protocol (HTTP). XML-RPC is a server-client protocol that allows a client to request a program to be executed on a machine that is an XML-RPC server. Finally, I would like to have investigated the evolving XML Schema technology. However, the XML Schema technology had not been standardized at the time of this writing.

REFERENCES

1. “Frequently Asked Questions about the Extensible Markup Language”, web site, http://www.ucc.ie/xml

2. “GMD-IPSI XQL Engine Version 1.0.2”, web site, http://xml.darmstadt.gmd.de/xql/

3. McLaughlin, B., *Java and XML*, Cambridge, MA, O’Reilly, 2000

4. Navarro, White and Burman, *Mastering XML*, Alameda, CA, Sybex, 1999

5. “Oasis, The XML Cover Pages, Extensible Markup Language”, web site http://xml.coverpages.org/xml.html

6. Many authors, *Professional Java Server Programming*, Birmingham, England, Wrox Press, 1999.

7. “SAX 2.0: The Simple API for XML”, web site http://www.megginson.com/SAX/

8. S. St. Laurent and E. Cerami, *Building XML Applications*, New York, McGraw-Hill, 1999

9. “W3C Architecture Domain, Document Object Model Activity Statement”, web site, http://www.w3.org/DOM/Activity

10. “W3C Architecture Domain, Document Object Model FAQ”, web site http://www.w3.org/DOM/faq.html

11. “W3C Architecture Domain, Extensible Markup Language”, web site, http://www.w3.org/XML

12. “W3C, Extensible Markup Language (XML) 1.0 (Second Edition)”, web site, http://www.w3.org/TR/REC-xml

13. “W3C Extensible Stylesheet Language Version 1.0, web site http://www.w3.org/TR/xsl/

14. “XML Query Language – Frequently Asked Questions”, web Site, http://www.ibiblio.org/xql/

15. “W3C Architecture Domain, Document Object Model”, web site http://www.w3.org/DOM

16. “XML Query Language (XQL)”, web site, http://www.w3.org/TandS/QL/QL98/pp/xql.html