SEMANTIC WEB
Unit-III

Syllabus

Resource Description Framework: Features, Capturing Knowledge with RDF. **XML Technologies:** XPath, The Style Sheet Family: XSL, XSLT, and XSL FO, XQuery, XLink, XPointer, XInclude, XMLBase, XHTML, XForms, SVG.

Introduction to RDF Language.

The XML tags can often add meaning to data, however, actually understanding the tags is meaningful only to humans. For example, given the following segment of XML markup tags:

<book>
<title>Godel, Escher, Bach: An Eternal Golden Braid
<title>
</book>

A human might infer that: **"The book has the title G¨odel, Escher, Bach: An Eternal Golden Braid."** This simple grammatical sentence is understood to contain three basic parts: a subject [The book], a predicate [has title], and an object [G¨odel, Escher, Bach: An Eternal Golden Braid]. A machine, however, could not make this inference based upon the XML alone.

For machines to do more automatically, it is necessary to go beyond the notion of the HTML display model, or XML data model, toward a "meaning" model. This is where RDF and metadata can provide new machine-processing capabilities built upon XML technology. What is metadata? It is information about other data. Building upon XML, the W3C developed the RDF metadata standard. The goal was to add semantics defined on top of XML.

RDF Triple

The RDF model is based on statements made about resources that can be anything with an associated URI (Universal Resource Identifier). The basic RDF model produces a triple, where a resource (the subject) is linked through an arc labeled with a property (the predicate) to a value (the object).

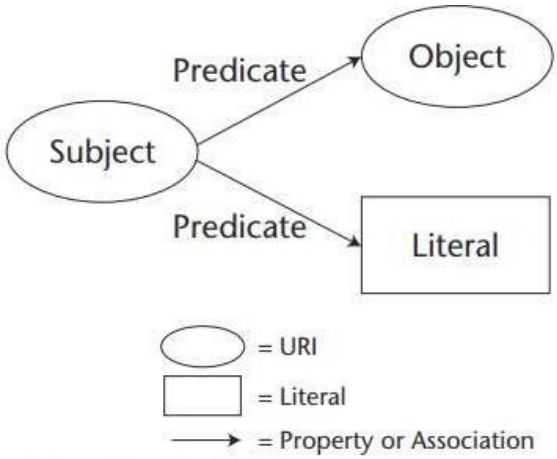

Figure 5.2 The RDF triple.

The key elements of an RDF triple are as follows:

Subject:- In grammar, this is the noun or noun phrase that is the doer of the action. In the sentence "The company sells batteries," the subject is "the company." The subject of the sentence tells us what the sentence is about. In logic, this is the term about which something is asserted. In RDF, this is the resource that is being described by the ensuing predicate and object. Therefore, in RDF, we want a URI to stand for the unique concept "company" like "http://www.business.org/ontology/#company" to denote that we mean a form of business ownership and not friends coming for a visit.

Predicate:- In grammar, this is the part of a sentence that modifies the subject and includes the verb phrase. Returning to our sentence "**The company sells batteries**" the predicate is the phrase "**sells batteries**" In other words, the predicate tells us something about the subject. In logic, a predicate is a function from individuals (a particular type of subject) to truth-values with an arity based on the number of arguments it has. In RDF, a predicate is a relation between the subject and the object. Thus, in RDF, we would define a unique URI for the concept "**sells**" like "http://www.business.org/ ontology/#sells".

Object:-In grammar this is a noun that is acted upon by the verb. Returning to our sentence "**The company sells batteries**" the object is the noun "**batteries**" In logic, an object is acted upon by the predicate. In RDF, an object is either a resource referred to by the predicate or a literal value. In our example, we would define a unique URI for "**batteries**" like "http://www .business.org/ontology/#batteries".

Statement

In RDF, the combination of the preceding three elements, subject, predicate, and object, as a single unit. Figure 5.3 displays a graph representation of two RDF statements. These two statements illustrate the concepts in Figure 5.2. Note that the object can be represented by a resource or by a literal value.

We should stress that resources in RDF must be identified by resource IDs, which are URIs with optional anchor IDs. This is important so that a unique concept can be unambiguously identified via a globally unique ID. This is a key difference between

relying on semantics over syntax. The syntactic meaning of words is often ambiguous. For example, the word "bark" in the sentences "The bark felt rough" and "The bark was loud" has two different meanings; however, by giving a unique URI to the concept of tree bark like "www.business.org/ontology/plant/#bark", we can always refer to a single definition of bark.

Capturing Knowledge with RDF

There is wide consensus that the triple-based model of RDF is simpler than the RDF/XML format, which is called the "serialization format." Because of this, a variety of simpler formats have been created to quickly capture knowledge expressed as a list of triples. Let's walk through a simple scenario where we express concepts in four different ways: as natural language sentences, in a simple triple notation called N3, in RDF/XML serialization format, and, finally, as a graph of the triples.

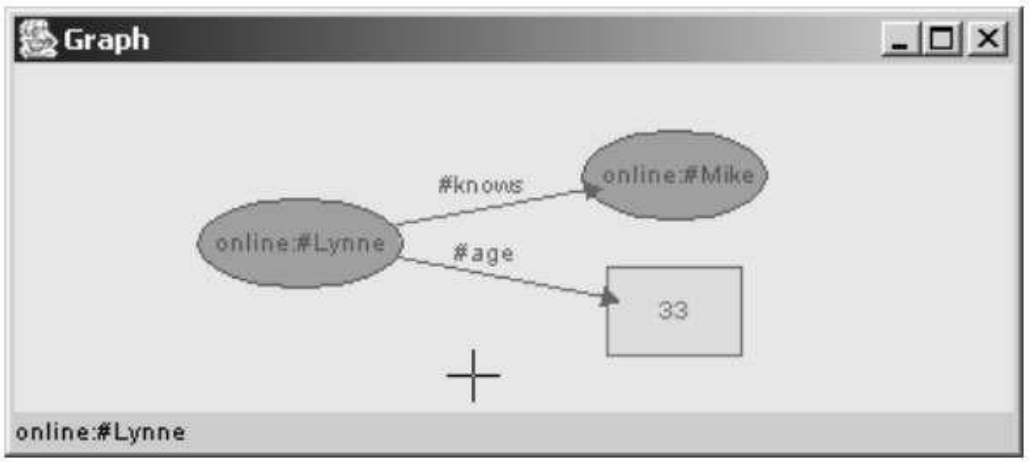

Figure 5.3 A graph of two RDF statements.

Following the linguistic model of subject, predicate, and object, we start with three English statements:

```
Buddy Belden owns a business.
The business has a Web site accessible at http://www.c2i2.com/~budstv.
Buddy is the father of Lynne.
```

In your business, you could imagine extracting sentences like these from daily routines and processes in your business. There are even products that can scan email and documents for common nouns and verbs. In other words, capturing statements in a formal way allows the slow aggregation of a corporate knowledge base in which you capture processes and best practices, as well as spot trends. This is knowledge management via a bottom-up approach instead of a top-down approach. Now let's examine how we capture the preceding sentences in N3 notation:

```
<#Buddy> <#owns> <#business>.
<#business> <#has-website> <http://www.c2i2.com/~budstv>.
<#Buddy> <#father-of> <#Lynne>.
```

From each sentence we have extracted the relevant subject, predicate, and object. The # sign means the URI of the concepts would be the current document. This is a shortcut done for brevity; it is more accurate to replace the # sign with an absolute URI like "http://www.c2i2.com/buddy/ontology" as a formal namespace. In N3 you can do that with a prefix tag like this:

```
@prefix bt:  <http://www.c2i2.com/buddy/ontology/>.
```

Using the prefix, our resources would be as follows:


```
<bt:Buddy> <bt:owns> <bt:business>.
```

Of course, we could also add other prefixes from other vocabularies like the Dublin Core:

```
@prefix dc:  <http://purl.org/dc/elements/1.1/>.
```

This would allow us to add a statement like "The business title is Buddy's TV and VCR Service" in this way:

```
<bt:business> <dc:title> "Buddy's TV and VCR Service".
```

Tools are available to automatically convert the N3 notation into RDF/XML format. One popular tool is the Jena Semantic Web toolkit from Hewlett-Packard, available at http://www.hpl.hp.com/semweb/. Listing 5.2 is the generated RDF/XML syntax.

```
<rdf:RDF
    xmlns:RDFNsId1='#'
    xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <rdf:Description rdf:about='#Buddy'>
        <RDFNsId1:owns>
            <rdf:Description rdf:about='#business'>
                <RDFNsId1:has-website
rdf:resource='http://www.c2i2.com/~budstv' />
            </rdf:Description>
        </RDFNsId1:owns>
        <RDFNsId1:father-of rdf:resource='#Lynne'/>
    </rdf:Description>
</rdf:RDF>
```

Listing 5.2 RDF/XML generated from N3.

The first thing you should notice is that in the RDF/XML syntax, one RDF statement is nested within the other. It is this sometimes nonintuitive translation of a list of statements into a hierarchical XML syntax that makes the direct authoring of RDF/XML syntax difficult; however, since there are tools to generate correct syntax for you, you can just focus on the knowledge engineering and not author the RDF/XML syntax. Second, note how predicates are represented by custom elements (like RDFNsId1:owns or RDFNsId1:father-of). The objects are represented by either the rdf:resource attribute or a literal value.

Figure 5.4 displays an IsaViz graph of the three RDF statements. While the triple is the centerpiece of RDF, other elements of RDF offer additional facilities in composing these knowledge graphs.

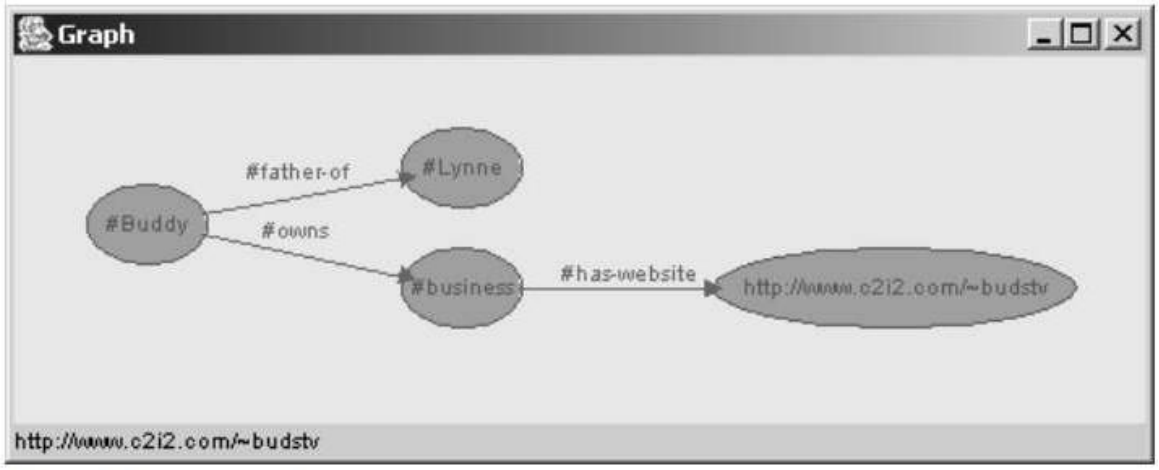

Figure 5.4 Graph of N3 notation.

Other RDF Features

The rest of the features of RDF assist in increasing the composeability of statements. Two main categories of features do this: a simple container model and reification (making statements about statements). The container model allows groups of resources or values. Reification allows higher-level statements to capture knowledge about other statements. Both of these features add some complexity to RDF, so we will demonstrate them with basic examples.

We need a container to model the sentence "The people at the meeting were Joe, Bob, Susan, and Ralph." To do this in RDF, we create a container, called a bag, for the objects in the statement, as shown in Listing 5.3.

```
<rdf:RDF
    xmlns:ex='http://www.example.org/sample#'
    xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <rdf:Description rdf:about='ex:meeting'>
        <ex:attendees>
        <rdf:Bag rdf:ID="people">
           <rdf:li rdf:resource='ex:Joe'/>
           <rdf:li rdf:resource='ex:Bob'/>
           <rdf:li rdf:resource='ex:Susan'/>
           <rdf:li rdf:resource='ex:Ralph'/>
        </rdf:Bag>
     </ex:attendees>
    </rdf:Description>
</rdf:RDF>
```

Listing 5.3 An RDF bag container.

In Listing 5.3 we see one rdf:Description element (one subject), one predicate (attendees), and an object, which is a bag (or collection) of resources. A bag is an unordered collection where each element of the bag is referred to by an rdf:li or "list item" element. Figure 5.5 graphs the RDF in Listing 5.3.

Figure 5.5 nicely demonstrates that the "meeting" is "attended" by "people" and that people is a type of bag. The members of the bag are specially labeled as a member with an rdf:_# predicate. RDF containers are different than XML containers in that they are explicit. This is the same case as relations between elements, which are also implicit in XML, whereas such relations (synonymous with predicates) are explicit in RDF. This explicit modeling of containers and relations is an effort to remove ambiguity from our models so that computers can act reliably in our behalf. On the downside, such explicit modeling is harder than the implicit modeling in XML documents. This has had an effect on adoption, as discussed in the next section.

Three types of RDF containers are available to group resources or literals:

Bag. An rdf:bag element is used to denote an unordered collection. Duplicates are allowed in the collection. An example of when to use a bag would be when all members of the collection are processed the same without concern for order.

Sequence. An rdf:seq element is used to denote an ordered collection (a "sequence" of elements). Duplicates are allowed in the collection. One reason to use a sequence would be to preserve the alphabetical order of elements. Another example would be to process items in the order in which items were added to the document.

Alternate. An rdf:alt element is used to denote a choice of multiple values or resources. This is referred to as a choice in XML. Some examples would be a choice of image formats (JPEG, GIF, BMP) or a choice of makes and models, or any time you wish to constrain a value to a limited set of legal values.

Now that we have added the idea of collections to our statements, we need a way to make statements either about the collection or about individual members of the collection. You can make statements about the collection by attaching an rdf:ID attribute to the container. Making statements about the individual members is the same as making any other statement by simply referring to the resource in the collection as the object of your statement.

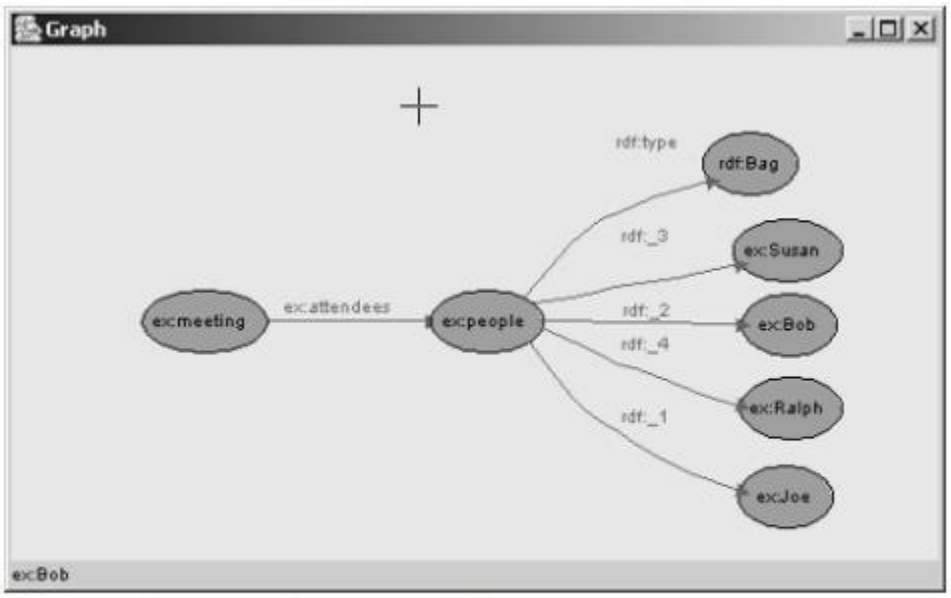

Figure 5.5 Graph of an RDF bag.

While containers affect the modeling of a single statement (for example, an object becoming a collection of values), reification allows you to treat a statement as the object of another statement. This is often referred to as "making statements about statements" and is called *reification*. Listing 5.4 shows a simple example of reification.

```
@prefix : <http://example.org/onto#>.
@prefix earl: <http://www.w3.org/2001/03/earl/0.95#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix dc: <http://purl.org/dc/elements/1.1/>.

:Jane earl:asserts
     [ rdf:subject :MyPage;
       rdf:predicate earl:passes;
       rdf:object "Accessibility Tests" ];
   earl:email <mailto:Jane@example.org>;
   earl:name "Jane Jones".

:MyPage
   a earl:WebContent;
   dc:creator <http://example.org/onto/person/Mary/>.
```

Listing 5.4 N3 example of reification.

Listing 5.4 demonstrates (in N3 notation) that Jane has tested Mary's Web page and asserts that it passes the accessibility tests. The key part relating to reification is the statement with explicit subject, predicate, and object parts that are the object of "asserts." Listing 5.5 shows the same example in RDF:

```
<rdf:RDF
    xmlns:dc='http://purl.org/dc/elements/1.1/'
    xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
    xmlns:earl='http://www.w3.org/2001/03/earl/0.95#'>
    <rdf:Description rdf:about='http://example.org/onto#Jane'>
        <earl:asserts rdf:parseType='Resource'>
            <rdf:subject>
                <earl:WebContent
                   rdf:about='http://example.org/onto#MyPage'>
                   <dc:creator
                   rdf:resource='http://example.org/onto/person/Mary/'/>
                </earl:WebContent>
            </rdf:subject>
            <rdf:predicate>
             rdf:resource='http://www.w3.org/2001/03/earl/0.95#passes'/>
            <rdf:object>Accessibility Tests</rdf:object>
        </earl:asserts>
        <earl:email rdf:resource='mailto:Jane@example.org'/>
        <earl:name>Jane Jones</earl:name>
    </rdf:Description>
</rdf:RDF>
```

Listing 5.5 Generated RDF example of reification.

The method for reifying statements in RDF is to model the statement as a resource via explicitly specifying the subject, predicate, object, and type of the statement. Once the statement is modeled, you can make statements about the modeled statement. The reification is akin to statements as argument instead of statements as fact, which is useful in cases where the trustworthiness of the source is carefully tracked (for example, human intelligence collection). This is important to understand, as reification is not applicable to all data modeling tasks. It is easier to treat statements as facts.

Figure 5.6 displays a graph of the reified statement. Note that the statement is treated as a single entity via an anonymous node. The anonymous node is akin to a Description element without an rdf:about attribute. The rdf:parseType attribute in Listing 5.5 means that the content of the element is parsed similar to a new Description element.

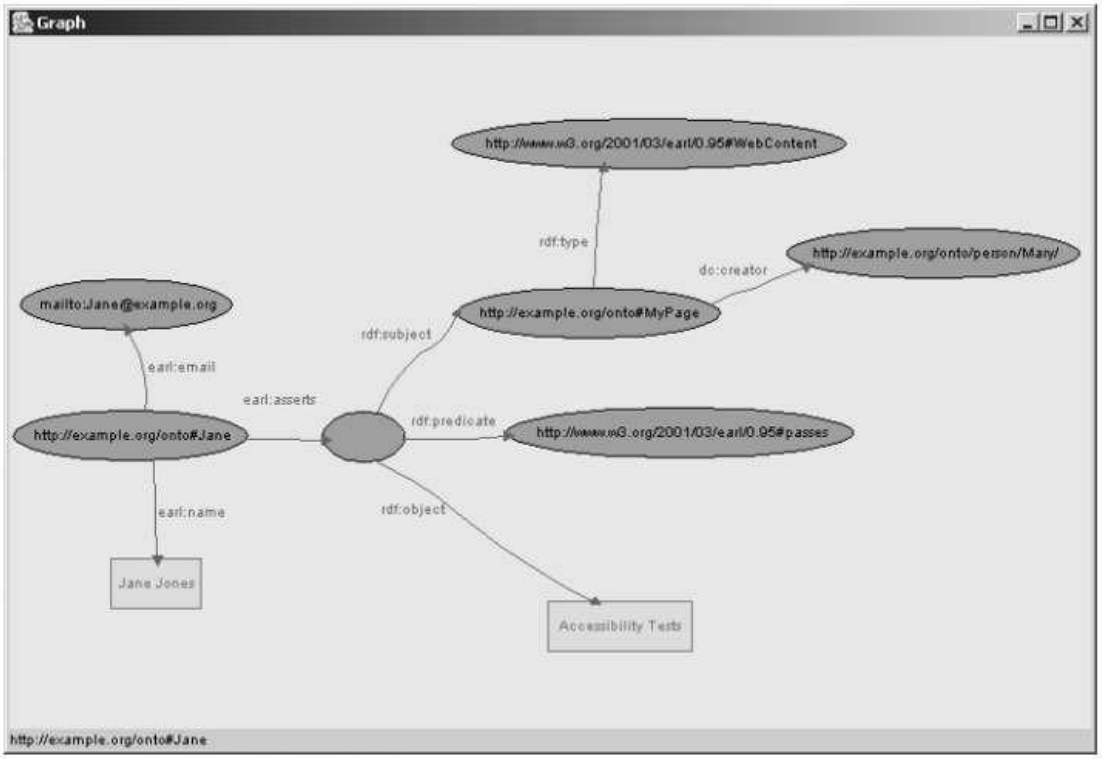

Figure 5.6 Graph of reification.

Admittedly, reification is not simple. Many people come to RDF understanding the basics of the triple but missing the utility of reification. Most databases treat data as facts, so it is a stretch to think about data as assertions. One commonsense application of reification is annotations of other people's work. Annotations, by nature, are statements about someone else's statements. So, clearly, reification fits there. At the same time, it will take training for developers and modelers to understand where to use reification and what rules apply to reified statements. In fact, some current Semantic Web applications explicitly eliminate reification from their knowledge bases to reduce the complexity. Complexity hurts adoption, and the adoption of RDF by mainstream developers has been significantly slower than other W3C technologies. The next section examines the reasons why.

XML Technologies

XPath

XPath is the XML Path Language, and it plays an important role in the XML family of standards. It provides an expression language for specifically addressing parts of an XML document. XPath is important because it provides key semantics, syntax, and functionality for a variety of standards, such as XSLT, XPointer, and XQuery. By using XPath expressions with certain software frameworks and APIs, you can easily reference and find the values of individual components of an XML document.

W3C Recommendation written in 1999, XPath 1.0 was the joint work of the W3C XSL Working Group and XML Linking Working Group, and it is part of the W3C Style Activity and W3C XML Activity. In addition to the functionality of addressing areas of an XML document, it provides basic facilities for manipulation of strings, numbers, and booleans. XPath uses a compact syntax to facilitate its use within URIs and XML attribute values. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document. Because we can use XPath to specifically address where components can be defined, it provides an important mechanism that is used by other XML standards and larger XML frameworks and APIs.

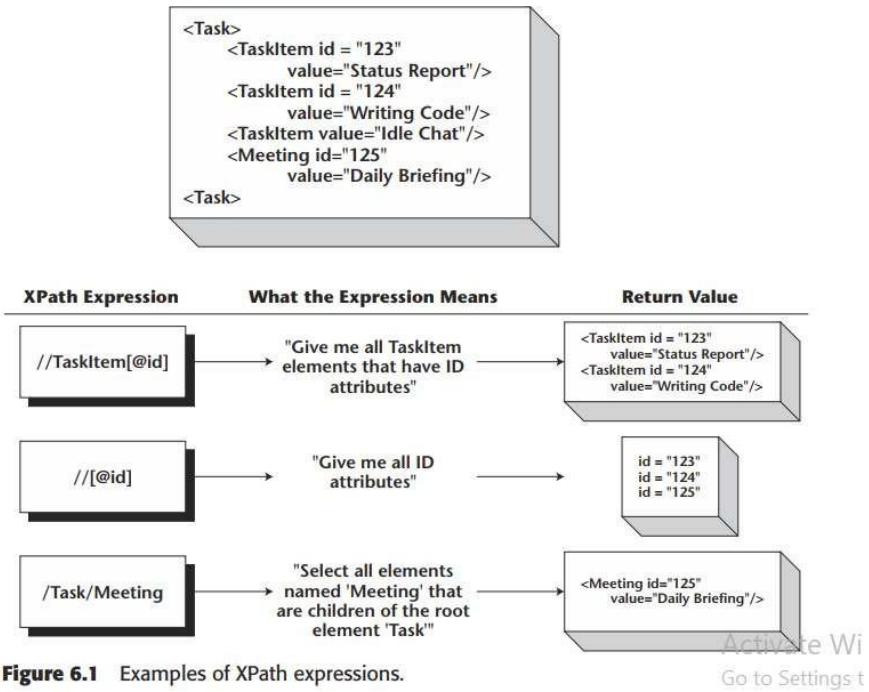

Figure 6.1 Examples of XPath expressions.

Role played by XPath in other standards

1. With XSLT, you can define a template in advance using XPath expressions that allow you to specify how to style a document.
2. XQuery is a superset of XPath and uses XPath expressions to query XML native databases and multiple XML files.

3. XPointer uses XPath expressions to "point" to specific nodes in XML documents.
4. XML Signature, XML Encryption, and many other standards can use XPath expressions to reference certain areas of an XML document.
5. In a Semantic Web ontology, groups of XPath expressions can be used to specify how to find data and the relationships between data.
6. The DOM Level 3 XPath specification, a Working Draft from the W3C, also provides interfaces for accessing nodes of a DOM tree using XPath expressions.

The Style Sheet Family: XSL, XSLT, and XSLFO

Style sheets allow us to specify how an XML document can be transformed into new documents, and how that XML document could be presented in different media formats. The languages associated with style sheets are

1. XSL (Extensible Stylesheet Language),
2. XSLT (Extensible Stylesheet Language: Transformations), and
3. XSLFO (Extensible Stylesheet Language: Formatting Objects)

A style sheet processor takes an XML document and a style sheet and produces a result. XSL consists of two parts: It provides a mechanism for transforming XML documents into new XML documents (XSLT), and it provides a vocabulary for formatting objects (XSLFO). XSLT is a markup language that uses template rules to specify how a style sheet processor transforms a document and is a Recommendation by the W3C.

It is important to understand the importance of styling. Using style sheets adds presentation to XML data. In separating content (the XML data) from the presentation (the style sheet), you take advantage of the success of what is called the **Model-View-Controller (MVC)** paradigm. The act of separating the data (the model), how the data is displayed (the view), and the framework used between them (the controller) provides maximum reuse of your resources. When you use this technology, XML data can simply be data. Your style sheets can transform your data into different formats, eliminating the maintenance nightmare of trying to keep track of multiple presentation formats for the same data. Embracing this model allows you to separate your concerns about maintaining data and presentation. Because browsers such as Microsoft Internet Explorer have style sheet processors embedded in them, presentation can dynamically be added to XML data at download time.

Figure 6.2 shows a simple example of the transformation and formatting process. A style sheet engine (sometimes called an XSLT engine) takes an original XML document, loads it into a DOM source tree, and transforms that document with the instructions given in the style sheet. In specifying those instructions, style sheets use XPath expressions to reference portions of the source tree and capture information to place into the result tree. The result tree is then formatted, and the resulting XML document is returned. Although the original document must be a well-formed XML document, the resulting document may be any format. Many times, the resulting document may be postprocessed. In the

case of formatting an XML document with XSLFO styling, a post-processor is usually used to transform the result document into a different format (such as PDF or RTF etc).

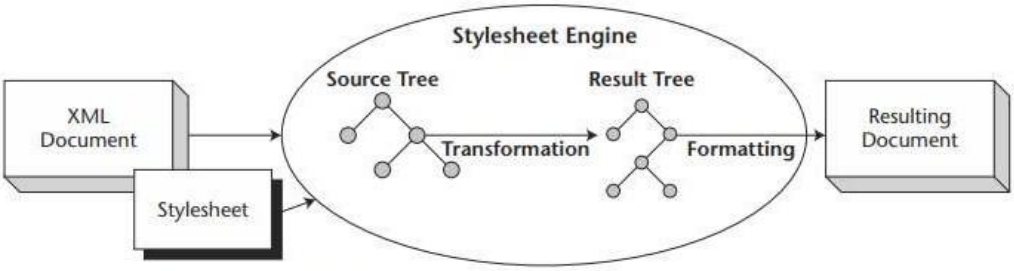

Figure 6.2 Styling a document.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="simplestyle.xsl" type="text/xsl"?>
<project name="Trumantruck.com">
   <description>Rebuilding a 1967 Chevy Pickup Truck</description>
   <schedule>
           <workday>
               <date>20000205</date>
               <description>Taking Truck Body Apart</description>
           </workday>
           <workday>
               <date>20000225</date>
               <description>Sandblasting, Dismantling Cab</description>
           </workday>
           <workday>
               <date>20000311</date>
               <description>Sanding, Priming Hood and Fender</
description>
           </workday>
  </schedule>
</project>
```

Listing 6.1 A simple XML file.

To create an HTML page with the information from our XML file, we need to write a style sheet. Listing 6.2 shows a simple style sheet that creates an HTML file with the workdays listed in an HTML table. Note that all pattern matching is done with XPath expressions. The <xsl:value-of> element returns the value of items selected from an XPath expression, and each template is called by the XSLT processor if the current node matches the XPath expression in the match attribute.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
    <xsl:template match="/">
        <html>
            <TITLE>Schedule For
            <xsl:value-of select="/project/@name"/>
- <xsl:value-of select="/project/description"/>
            </TITLE>
            <CENTER>
                <TABLE border="1">
                    <TR>
                        <TD><B>Date</B></TD>
                        <TD><B>Description</B></TD>
                    </TR>
                    <xsl:apply-templates/>
                </TABLE>
```

Listing 6.2 A simple style sheet. *(continued)*

```
            </CENTER>
        </html>
    </xsl:template>
    <xsl:template match="project">
        <H1>Project:
        <xsl:value-of select="@name"/>
        </H1>
        <HR/>
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="schedule">
        <H2>Work Schedule</H2>

        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="workday">
        <TR>
            <TD>
                <xsl:value-of select="date"/>
            </TD>
            <TD>
                <xsl:value-of select="description"/>
            </TD>
        </TR>
    </xsl:template>
</xsl:stylesheet>
```

Listing 6.2 *(continued)*

The resulting document is rendered dynamically in the Internet Explorer browser, shown in Figure 6.3.

Figure 6.3 A browser, using style sheets to render our example.

Why are style sheets important?

In an environment where interoperability is crucial, and where data is stored in different formats for different enterprises, styling is used to translate one enterprise format to another enterprise format. In a scenario where we must support different user interfaces for many devices, style sheets are used to add presentation to content, providing us with a rich mechanism for supporting the presentation of the same data on multiple platforms.

Figure 6.4 shows a diagram of many practical examples of style sheets in use. In this diagram, different style sheets are applied to an XML document to achieve different goals of presentation, interoperation, communication, and execution. At the top of the diagram, you see how different style sheets can be used to add presentation to the original XML content. In the case of XSLFO, sometimes a post-processor is used to transform the XSLFO vocabulary into another format, such as RTF and PDF. In the "interoperation" portion of Figure 6.4, a style sheet is used to transform the document into another format read by another application. In the "communication" portion of Figure 6.4, a style sheet is used to transform the XML document into a SOAP message, which is sent to a Web service. Finally, in the "execution" section, there are two examples of how XML documents can be transformed into code that can be executed at run time.

These examples should give you good ideas of the power of style sheets.

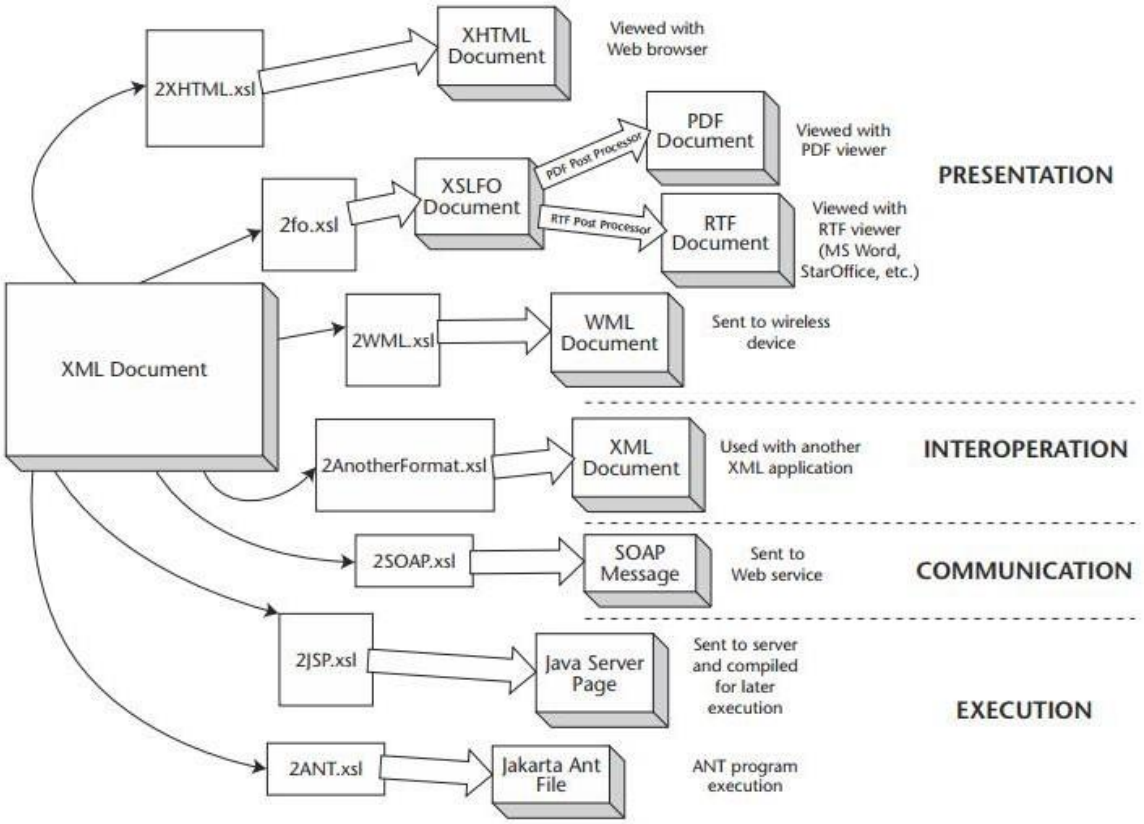

Figure 6.4 Practical examples of style sheets in use.

XQuery

XQuery is a language designed for processing XML data, and it is intended to make querying XML-based data sources as easy as querying databases. Although there have been attempts at XML query languages before—such as XQL, XML-QL, and Quilt—XQuery, a product of the W3C, is the first to receive industry-wide attention. The Query Working Group of the W3C has made it clear that there is a need for a human-readable query syntax, and there is also a requirement for an XML-based query syntax. XQuery was designed to meet the human-readable syntax requirement. It is an expression language that is an extension of XPath. With few exceptions, every XPath expression is also an XQuery expression.

Similar to using style sheet transformations, the XQuery process uses XPath expressions and its own functions to query and manipulate XML data. It is a strongly typed language, and its syntax is very similar to a high-level programming language, such as Perl. To demonstrate how XQuery works, we will use the example XML file shown in Listing 6.1. The following is a valid

XQuery expression that returns the workdays from the file, sorted alphabetically by the description of each workday:

```
let $project := document("trumanproject.xml")/project
let $day := $project/schedule/workday
return $day sortby (description)
```

The first line loads the XML document and assigns a variable to the <project> element. The second line assigns the collection of <workday> elements to the $day variable by using an XPath expression. Finally, the last line sorts the elements alphabetically based on the value of the <description> element. The result of that XQuery expression is as follows:

```
<workday>
  <date>20000225</date>
  <description>Sandblasting, Dismantling Cab</description>
</workday>
<workday>
  <date>20000311</date>
  <description>Sanding, Priming Hood and Fender</description>
</workday>
<workday>
  <date>20000205</date>
  <description>Taking Truck Body Apart</description>
</workday>
```

As you can see by the result of the XQuery expression, the <workday> elements have now been sorted alphabetically based on their <description> child element.

As you can see by the result of the XQuery expression, the <workday> elements have now been sorted alphabetically based on their <description> child element. XQuery is important because it will provide a standard query interface to XML databases and data stores. XQuery is currently in Working Draft status at the W3C (http://www.w3.org/TR/xquery/), and it is continuing to evolve.

XLink

XLink is the XML Linking Language from the W3C. It allows elements to be inserted into XML documents in order to create and describe links between resources, associate meta data, and link external documents. Part of the specification is similar to hyperlinks in HTML, but XLink goes far beyond the linking capabilities in HTML in offering advanced behavior features that make hyperlinking more flexible. XLink allows you to link multiple targets, roles, resources, and responses to elements.

A *link* is defined by the XLink W3C recommendation as "an explicit relationship between resources or portions of resources."[1] It is made explicit by an XLink-conforming XML element. With XLink, not only can XML authors link external documents, but also elements can be linked to other elements, and the relationships between them can be linked. XLink-compliant links can be simple and complex. Listing 6.3 shows a simple example of XLink in use. As you can see, the example is quite similar to the <A> linking element in HTML.

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
 <name xlink:type='simple'
 xlink:href='instructors/busdriver.xml'>Clay Richardson</name>
 is an employee at
   <company xlink:type='simple' xlink:href='#mcbrad'>
        McDonald Bradley, Inc.
   </company>, and teaches
   <course xlink:type='simple' xlink:href='#CS593'>
         Computer Science 593
   </course>.
  <employers>
    <company id='mcbrad'>McDonald Bradley</company>
    <company id='btg'>BTG</company>
    <company id='grumman'>Northrup Grumman</company>
    <company id='orionsci'>Orion Scientific</company>
  </employers>
  <courselist>
    <course id='CS141'>CS 141 - Intro to Comp Sci</course>
    <course id='CS444'>CS 444 - Operating Systems</course>
    <course id='CS593'>CS 593 - Object-Oriented Design</course>
    <course id='CS669'>CS 669 - Keeping it Real</course>
  </courselist>
</doc>
```

Listing 6.3 Simple XLink example.

In Listing 6.3, we have an example that discusses employers and courses in a document. Throughout the XML file, the xlink:type attribute of the element is set to 'simple', and the xlink:href attribute is set to link to external documents and to elements within the document. This is the simplest example of linking.

XLink can also connect relationships together. If, for example, we wanted to link descriptions with our courses and employers, we could use the XLink 'extended' link type to connect our descriptions with our elements. A bit more complicated than the simple links shown in Listing 6.3, extended links allow arcs to describe relationships between elements and remote resources. Listing 6.4 shows an example of our simple XLink example from Listing 6.3 modified to incorporate relationships between some of the elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<doc>
 <extendedlink xlink:type='extended'>
    <loc xlink:type='locator' xlink:label='mcbrad' xlink:href='#mcbrad'>
    </loc>
    <loc xlink:type='locator' xlink:label='cs593'xlink:href='#CS593'>
    </loc>
    <loc xlink:type='locator' xlink:label='cs593description'
         xlink:href='courses/cs593.xml'>
    </loc>
    <loc xlink:type='locator' xlink:label='mcbraddescription'
         xlink:href='employers/mcbrad.xml'>
    </loc>
    <arc xlink:type='arc' xlink:from='cs593' xlink:to='cs593description'>
    </arc>
    <arc xlink:type='arc' xlink:from='mcbrad' xlink:to='
mcbraddescription'>
    </arc>
 </extendedlink>
 <name xlink:type='simple'
       xlink:href='instructors/busdriver.xml'>Clay Richardson</name>
                    is an employee at
 <company xlink:type='simple' xlink:href='#mcbrad'>
        McDonald Bradley, Inc.
 </company>, and teaches
 <course xlink:type='simple' xlink:href='#CS593'>
         Computer Science 593
 </course>
 <employers>
     <company id='mcbrad'>McDonald Bradley</company>
     <company id='btg'>BTG</company>
 </employers>
 <courselist>
     <course id='CS141'>CS 141 - Intro to Comp Sci</course>
     <course id='CS444'>CS 444 - Operating Systems</course>
     <course id='CS593'>CS 593 - Object-Oriented Design</course>
     <course id='CS669'>CS 669 - Keeping it Real</course>
 </courselist>
</doc>
```

Listing 6.4 An XLink example with element relationships.

In Listing 6.4, adding the <extendedlink> element with the xlink:type attribute of 'extended' allows us to make relationships between the elements. In the example, an 'arc' xlink:type attributes allow us to connect a course with its description, an employer, and a description.

What is XLink's importance?

It is the W3C's recommendation for generic linking, and it is intended for use in hypertext systems. Languages such as Scalable Vector Graphics (SVG) and MathML use XLink for hypertext references. At this point, the Resource Directory Description Language (RDDL) uses XLink syntax to specify the relationship between URLs and

other resources. It is the intention of the W3C that XLink be used for generic linking mechanisms in XML languages.

XPointer

XPointer is the XML Pointer Language from the W3C, which is used as a fragment identifier for any URI reference that locates an XML-based resource. A powerful language, XPointer can "point" to fragments of remote documents using XPath expressions and can point to ranges of data and points of data in documents.

An example of using XPointer is shown in Listing 6.5, where we can rewrite the data contained in the <extendedlink> element of Listing 6.4. As you can see, XPointer expressions are used in the xlink:href attributes of the <loc> element in order to point directly to the elements. The links address the <company> element with an ID "mcbrad" and a <course> element with the ID attribute of "CS593".

```
<extendedlink xlink:type='extended'>
  <loc xlink:type='locator' xlink:label='mcbrad'
      xlink:href='#xpointer(//company[@id='mcbrad'])'></loc>
  <loc xlink:type='locator' xlink:label='cs593'
      xlink:href='#xpointer(//course[@id='CS593'])'></loc>
  <loc xlink:type='locator' xlink:label='cs593description'
      xlink:href='courses/cs593.xml'></loc>
  <loc xlink:type='locator' xlink:label='mcbraddescription'
      xlink:href='employers/mcbrad.xml'></loc>
  <arc xlink:type='arc' xlink:from='cs593'
       xlink:to='cs593description'></arc>
  <arc xlink:type='arc' xlink:from='mcbrad'
       xlink:to='mcbraddescription'></arc>
</extendedlink>
```

Listing 6.5 A simple XPointer example.

As is seen in Listing 6.5, XPointer relies on XPath expressions to point to resources. In addition, XPointer includes functions that make it able to do some things that XPath cannot do, such as specifying ranges of elements and specifying points within a document. For example, the following XPointer expression points to objects within our document that was shown in Listing 6.1:

```
xpointer(string-range(//*,'Truck'))
```

That expression points to every 'Truck' string in the document. Looking at our earlier Listing 6.1, that expression would point to the 'Truck' string in the text node 'Rebuilding a 1967 Pickup Truck' in the <description> element, as well as the 'Truck' string in the 'Taking Truck Body Apart' node, shown later in the document. XPointer is a powerful specification, and it offers a rich mechanism for addressing pieces of XML documents.

In the summer of 2002, the XML Linking Group of the W3C broke the XPointer specification into four Working Draft pieces:

1. An XPointer core framework.
2. The XPointer element() scheme for addressing XML elements.
3. The XPointer xmlns() scheme to allow correct interpretation of namespace prefixes in pointers, and
4. The XPointer xpointer() scheme for full XPointer addressing flexibility.

Both XLink and XInclude can use XPointer expressions, and it is likely that there will be more adoption into products and other specifications.

XInclude

XML Inclusions (XInclude) is a W3C Candidate Recommendation used for the purpose of including documents, elements, or other information in an XML document. With it, you can build large XML documents from multiple fragmentary XML documents, well-formed XML documents, or non-XML text documents. It also has a "fallback" feature in case the external document is not available or the server is down. The simplest example of this is shown in the following, where the beginning of this chapter could be separated in multiple files:

```
<?xml version="1.0"?>
<chapter xmlns:xi="http://www.w3.org/2001/XInclude">
  <title>Understanding the Rest of the Alphabet Soup</title>
  <xi:include href="http://www.wiley.com/SemWeb/ch6/xpath.xml"/>
  <xi:include href="http://www.wiley.com/SemWeb/ch6/stylesheets.xml"/>
  <xi:include href="http://www.wiley.com/SemWeb/ch6/xquery.xml"/>
  <xi:include href="http://www.wiley.com/SemWeb/ch6/xlink.xml"/>
</chapter>
```

This code is very basic. External XML files for each chapter section are included in one document with XInclude. The rendered version would look like a large document. XInclude is a bit more powerful than that, however. Listing 6.6 gives a more complex example where non-XML content and portions of documents can be brought into an XML document, and where the XInclude "fallback" mechanism is used

```
<?xml version='1.0'?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <p>The relevant excerpt is:</p>
  <quotation>
    <xi:include
        href="source.xml#xpointer(string-range(chapter/p[1],
'Sentence 2')/
            range-to(string-range(chapter/p[2]/i,'3.',1,2)))">
       <xi:fallback>
          <xi:include href="error.xml"/>
       </xi:fallback>
    </xi:include>
  </quotation>
  <p>Code that was used to write the cool program was:</p>
  <code>
    <xi:include parse="text" href="code/Datamover.java">
      <xi:fallback>
        Sorry - The code is unavailable at this time!
```

Listing 6.6 A more complex example of XInclude.

```
      </xi:fallback>
    </xi:include>
  </code>
</document>
```

Listing 6.6 *(continued)*

Support for XInclude is limited, but it is growing. Many in the XML community are looking at security implications of browser-based XInclude, because there could be potential misuses.

XML Base

XML Base is a W3C Recommendation that allows authors to explicitly specify a document's base URI for the purpose of resolving relative URIs. Very similar to HTML's base element, it makes resolving relative paths in links to external images, applets, form-processing programs, style sheets, and other resources. Using XML Base, an earlier example in the last section could be written the following way:

```
<?xml version="1.0"?>
<chapter xmlns:xi="http://www.w3.org/2001/XInclude"
         xml:base="http://www.wiley.com/SemWeb/ch6">
  <title>Understanding the Rest of the Alphabet Soup</title>
  <xi:include href="xpath.xml"/>
  <xi:include href="stylesheets.xml"/>
  <xi:include href="xquery.xml"/>
  <xi:include href="xlink.xml"/>
</chapter>
```

The xml:base attribute in the <chapter> element makes all the referenced documents that follow relative to the URL "http://www.wiley.com/SemWeb/ch6." In the href attributes in the <include> elements in the preceding example, the following documents are referenced:

- http://www.wiley.com/SemWeb/ch6/xpath.xml
- http://www.wiley.com/SemWeb/ch6/stylesheets.xml
- http://www.wiley.com/SemWeb/ch6/xlink.xml

Using XML Base makes it easier to resolve relative paths. Developed by a part of the W3C XML Linking Working Group, it is a simple recommendation that makes XML development easier.

XHTML

XHTML, the Extensible Hypertext Markup Language, is the reformulation of HTML into XML. The specification was created for the purpose of enhancing our current Web to provide more structure for machine processing. Why is this important? Although HTML is easy for people to write, its loose structure has become a stumbling block on our way to a Semantic Web. It is well suited for presentation for browsers; however, it is difficult for machines to understand the meaning of documents formatted in HTML. Because HTML is not well formed and is only a presentation language, it is not a good language for describing data, and it is not extremely useful for information gathering in a Semantic Web environment. Because XHTML is XML, it provides structure and extensibility by allowing the inclusion of other XML-based languages with namespaces. By augmenting our current Web infrastructure with a few changes, XHTML can make intermachine exchanges of information easier.

XHTML 1.0, a W3C Recommendation released in January 2000, was a reformulation of HTML 4.0 into XML. The transition from HTML to XHTML is quite simple. Some of the highlights include the following:

1. An XHTML 1.0 document should be declared as an XML document using an XML declaration.
2. An XHTML 1.0 document is both valid and well formed. It must contain a DOCTYPE that denotes that it is an XHTML 1.0 document, and that also denotes the

DTD being used by that document. Every tag must have an end tag.

3. The root element of an XHTML 1.0 document is <html> and should contain a namespace identifying it as XHTML.
4. Because XML is case-sensitive, elements and attributes in XHTML must be lowercase.

Let's look at a simple example of making the transition from HTML to XHTML 1.0. The HTML in Listing 6.7 shows a Web document with a morning to-do list.

```
<HTML>
  <HEAD>
     <TITLE>Morning to-do list</TITLE>
  </HEAD>
  <BODY>
     <LI>Wake up
     <LI>Make bed
     <LI>Drink coffee
     <LI>Go to work
  </BODY>
</HTML>
```

Listing 6.7 An HTML example.

Going from the HTML in Listing 6.7 to XHTML 1.0 is quite easy. Listing 6.8 shows how we can do it. The first change is the XML declaration on the first line. The second change is the DOCTYPE declaration using a DTD, and the root tag <html> now uses the XHTML namespace. All elements and attributes have also been changed to lowercase. Finally, we make it a well-formed document by adding end tags to the <li> tags. Otherwise, nothing has changed.

```
<?xml version="1.0"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Morning to-do list</title>
  </head>
  <body>
     <li>Wake up</li>
     <li>Make bed</li>
     <li>Drink coffee</li>
     <li>Go to work</li>
   </body>
</html>
```

Listing 6.8 Simple XHTML 1.0 file.

The difference between Listings 6.7 and 6.8 shows that this transition between HTML and XHTML is quite smooth. Because XHTML 1.0 is well formed and valid, it can be processed easier by user agents, can incorporate stronger markup, and can reap the benefits of being an XML-based technology.

In XML, it is easy to introduce new elements or add to a schema. XHTML is designed to accommodate these extensions through the use of XHTML modules. XHTML 2.0, a W3C Working Draft released in August 2002, is made up of a set of these modules that describe the elements and attributes of the language. XHTML 2.0 is an evolution of XHTML 1.0, as it is not intended to be backward-compatible.

SVG

Scalable Vector Graphics (SVG) is a language for describing two-dimensional graphics in XML. A W3C Recommendation since September 2001, there are many tools and applications that take advantage of this exciting technology. With SVG, vector graphics, images, and text can be grouped, styled, and transformed. Features such as alpha masks, filter effects, and nested transformations are in this XML-based language, and animations can be defined and triggered. Many authors use scripting languages that access the SVG's Document Object Model to perform advanced animations and dynamic graphics.

The potential for SVG is quite exciting. Because it is an XML language, data content can be transformed into SVG to create graphically intense programs and animations. Online maps can easily convey the plotting of data, roads, and buildings with SVG.

What does an SVG file look like? Listing 6.11 gives a brief example.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
    <svg width="5cm" height="3cm" viewBox="0 0 5 3"
        xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <desc>Example link01 - a link on an ellipse</desc>
  <rect x=".01" y=".01" width="4.98" height="2.98"
        fill="none" stroke="blue"  stroke-width=".03"/>
  <a xlink:href="http://www.w3.org">
    <ellipse cx="2.5" cy="1.5" rx="2" ry="1" fill="red" />
</a>
</svg>
```

Listing 6.11 Simple SVG example.

Listing 6.11, an example taken from the SVG Recommendation of the W3C, creates an image of a red ellipse, shown in Figure 6.7. When a user clicks on the ellipse, the user is

taken to the W3C Web site. Of course, this is one of the simplest examples. SVG takes advantage of XLink for linking.

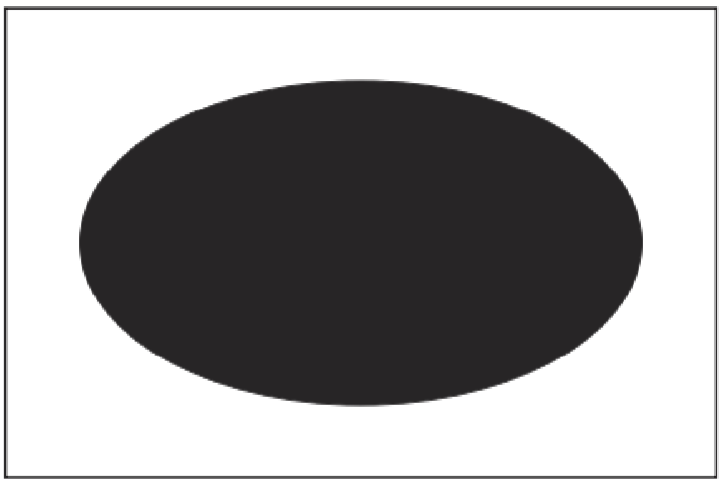

Figure 6.7 Rendering a simple SVG file.

If product adoption is any indicator, the SVG specification is quite successful. In a very short time, vendors have jumped on the SVG bandwagon. The Adobe SVG Viewer, the Apache Batik project, the SVG-enabled Mozilla browser, the W3C's Amaya editor/browser, and Jasc's WebDraw applicationsupport SVG, to name a few.

XForms

XForms is a W3C Candidate Recommendation that adds new functionality, flexibility, and scalability to what we expect to existing Web-based forms. Dubbed "the next generation of forms for the Web," XForms separates presentation from content, allows reuse, and reduces the number of round-trips to the server, offers device independence, and reduces the need for scripting in Web-based forms.6 It separates the model, the instance data, and the user interface into three parts, separating presentation from content.

Web forms are everywhere. They are commonplace in search engines and e-commerce Web sites, and they exist in essentially every Web application. HTML has made forms successful, but they have limited features. They mix purpose and presentation, they run only on Web browsers, and even the simplest form-based tasks are dependent on scripting. XForms was designed to fix these shortcomings and shows much promise.

Separating the purpose, presentation, and data is key to understanding the importance of XForms. Every form has a purpose, which is usually to collect data. The purpose is realized by creating a user interface (presentation) that allows the user to provide the required information. The data is the result of completing the form. With XForms, forms are separated into two separate components: the XForms model, which describes the purpose, and the XForms user interface, which describes how the form is presented. A conceptual view of an XForms interaction is shown in Figure 6.5, where the model and the presentation are stored separately. In an XForms scenario, the model and presentation are parsed into memory as XML "instance data." The instance data is kept in memory during user interaction. Because XML Forms uses XML Events, a general-purpose event framework described in XML, many triggered events can be script-free during this user interaction. Using an XML-based syntax, XForms developers can display messages to users, perform calculations and screen refreshes, or submit a portion (or all) of the instance data. After the user interaction is finished, the instance data is serialized as XML and sent to the server. Separating the data, the model, and the presentation allows you to maximize reusability and can help you build powerful user interfaces quickly.

The simplest example of XForms in XHTML 2.0 is in Listing 6.9. As you can see, the XForms model (with element <model>) belongs in the <head> section of the XHTML document. Form controls and user interface components belong in the <body> of the XHTML document. Every form control element has a required <label> child element, which contains the associated label. Each input has a ref attribute, which uniquely identifies that as an XForms input.

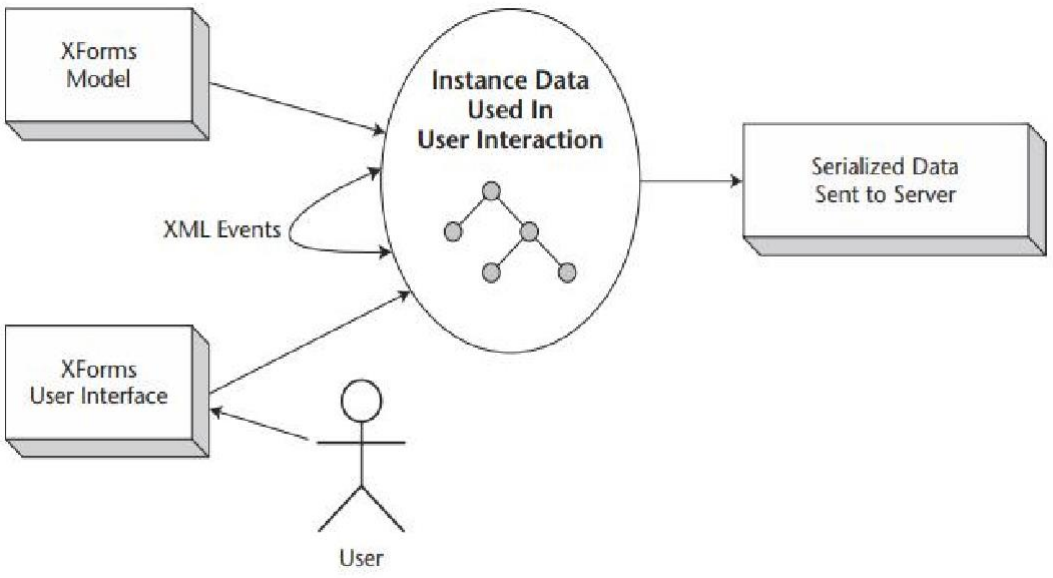

Figure 6.5 Conceptual view of XForms interaction.

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 2.0//EN"
   "http://www.w3.org/TR/xhtml2/DTD/xhtml2.dtd">
<html xmlns="http://www.w3.org/2002/06/xhtml2"
  xmlns:xforms="http://www.w3.org/2002/08/xforms/cr">
    <head>
        <title>Simple example</title>
        <xforms:model id="simpleform">
            <xforms:submission
                action="https://www.wiley.com/simpleXFormsExample/
submit"/>
        </xforms:model>
    </head>
    <body>
        <p>Enter your credit card number below</p>
        <xforms:input ref="username">
            <xforms:label>Name:</xforms:label>
         </xforms:input>
        <xforms:input ref="creditcard">
            <xforms:label>Credit Card:</xforms:label>
        </xforms:input>
        <xforms:input ref="expires">
            <xforms:label>Expires:</xforms:label>
        </xforms:input>
        <xforms:submit>
            <xforms:label>Submit</xforms:label>
        </xforms:submit>
    </body>
</html>
```

Listing 6.9 A simple XHTML 2.0 XForms example.

If the code from Listing 6.9 were submitted, the instance data similar to the following would be produced:

```
<instanceData>
   <username>Kenneth Kyle Stockman</username>
   <creditcard>55555555555555</creditcard>
   <expires>5/92</expires>
</instanceData>
```

Of course, this was a simple example. XForms also can take advantage of model item constraints by placing declarative validation information in forms from XML Schemas and XForms-specific constraints. In the preceding example, we could bind the <creditcard> and <expires> values to be valid to match certain schema types. We could also describe our data in our <model>, like the example shown in Listing 6.10, with validation constraints. In that example, you see that the instance is defined in the model. The <xforms:bind> element

uses the isValid attribute to validate the form. In this case, if someone attempts to submit the information without typing in anything, it will throw an invalid XForm event. Also notice that in the body of the document, individual components of the model are referenced by XPath expressions (in the ref attribute of the input elements)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">-->
<html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:ev="http://www.w3.org/2001/xml-events"
         xmlns:testcase="testcase"
         xmlns:xforms="http://www.w3.org/2002/01/xforms">
    <head>
        <link href="controls.css" rel="stylesheet" type="text/css"/>
        <xforms:model id="form1">
            <xforms:submitInfo id="submit1"
                    localfile="temp2.xml" method2="postxml"
                    target2="http://www.trumantruck.com/"/>
            <xforms:instance id="instance1" xmlns="">
                <testcase>
                    <username/>
                    <secret/>
                 </testcase>
            </xforms:instance>
            <xforms:bind
                isValid="string-length(.)&gt;0" ref="testcase/secret"/>
            <xforms:bind
                isValid="string-length(.)&gt;0" ref="testcase/username"/>
        </xforms:model>
    </head>
    <body>
        <b>User Name:</b>
        <xforms:input ref="testcase/testcase:input" xmlns:my="test">
            <xforms:caption>Enter your name</xforms:caption>
        </xforms:input>
        <b>Password:</b>
        <xforms:secret ref="testcase/secret">
            <xforms:caption>Password</xforms:caption>
        </xforms:secret>
        <b>submit</b>
        <xforms:submit>
            <xforms:caption>Submit Me</xforms:caption>
        </xforms:submit>
    </body>
</html>
```

Listing 6.10 An XForms example with validation.

Figure 6.6 shows the result rendered in the XSmiles browser, a Java-based XForms-capable browser available at http://www.xsmiles.org/. In this example, the username was entered, but the password was not. Because our XForm specified that it would not be valid, an error was thrown.

XForms is one of the most exciting tools that will be included in the XHTML 2.0 specification. It is still a Working Draft, which means that it is continuing to evolve. Because of its power and simplicity, and because instance data is serialized as XML, XForms has the potential to be a critical link between user interfaces and Web services. Commercial support for XForms continues to grow.

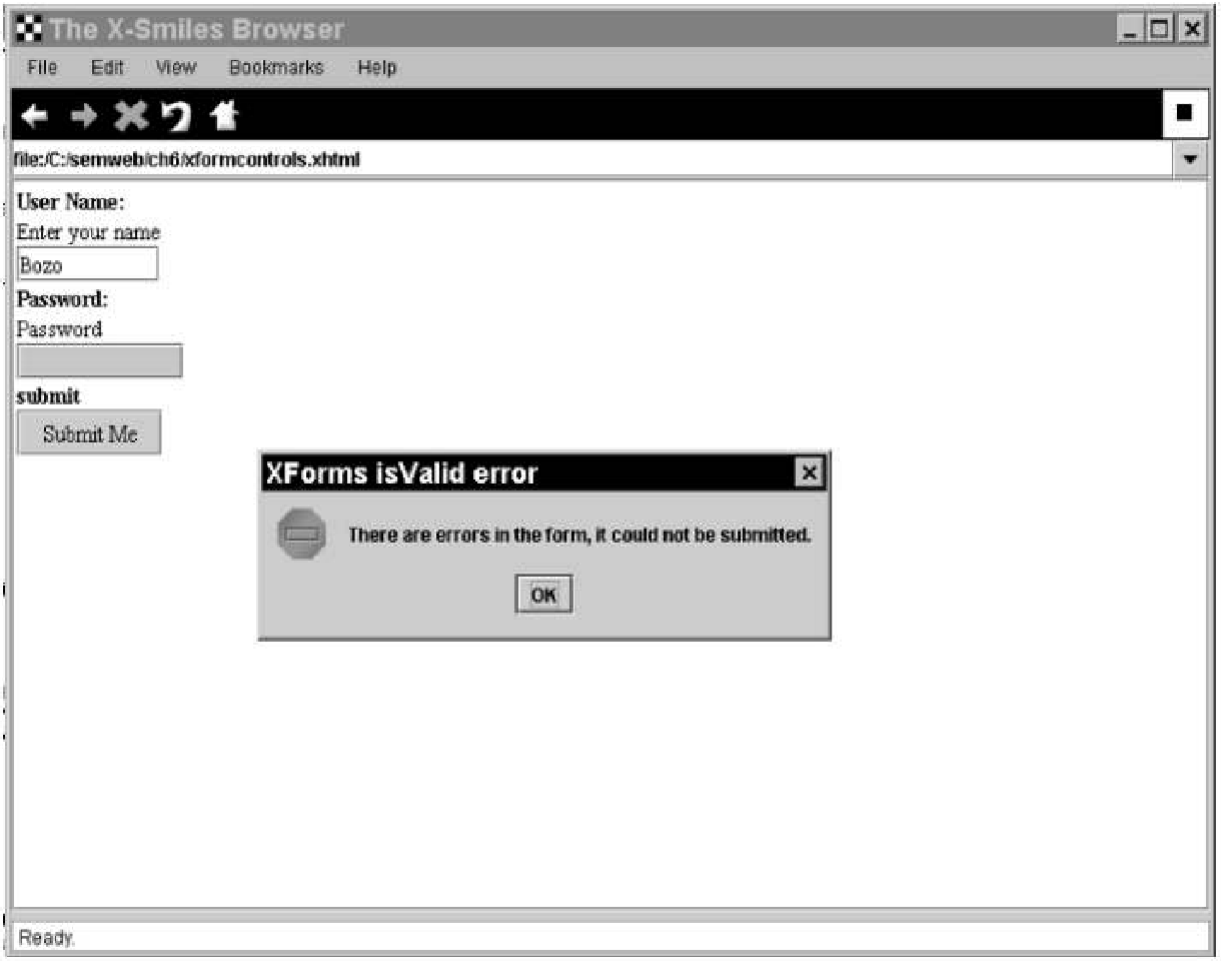

Figure 6.6 Example rendering of an XForm-based program.