2 XML Foundations

2.0 INTRODUCTION

The essence of Document Engineering is the analysis and design methods that yield precise models describing the information required by business processes and the rules by which related processes are coordinated and combined. Neither the methods nor the models have anything inherently to do with XML or any other syntax. Nevertheless, XML has rapidly become the preferred format for representing the physical models used to exchange information, so some familiarity with XML is essential.

Document Engineering has nothing inherently to do with XML

If you have a web publishing or programming background you undoubtedly have some experience with XML. But if your expertise is in systems analysis or business, you are probably new to this material. Furthermore, even though XML is an essential technology for Document Engineering, just knowing XML doesn't make you a document engineer because of the interdisciplinary nature of this new field.

The web publishing perspective on XML is incomplete in some respects compared to the perspective we take in Document Engineering. If you work in web publishing, you might view XML as an improvement on HTML that enables greater automation and consistency in formatting. This is true, but just thinking of XML as a smarter HTML misses its central ideas of document types and validation.[1] If you came to web publishing from working in technical documentation with the Standard Generalized Markup Language (SGML), of which XML is a subset, you certainly understand these key ideas. But your experience is likely to be with text-oriented or narrative types of documents, not with the transactional varieties used in applications that exchange documents.

Many programmers see XML as an Internet-friendly, easy-to-parse, and nonproprietary data format to use instead of ad hoc syntaxes for application configuration and inter-process communication. So if you have come to XML as a programmer, you appreciate the need for structured information and strong data typing and validation. But unless you've worked with applications that exchange documents, you probably build software designed for tight coupling with fine-grained APIs. You need to learn to use XML as a format for describing document models that represent entire business events, not just tiny messages. Using coarse-grained documents as interfaces is the key idea behind web services and service oriented architectures.

The syntax isn't what is most important about XML

So this chapter will introduce XML from the perspective of realizing document models and model-based applications. We will emphasize the big ideas of XML and not dwell on XML syntax and schema languages, because there are plenty of excellent books about them that cover them in more depth than this book allows.[2] If you have a business strategy interest in Document Engineering, this chapter will introduce all the XML you need to know. If you want to learn more about XML, this chapter will make it much easier for you to learn it. If you already know XML, this chapter will help you apply that knowledge in new ways.

2.1 FROM HTML TO XML

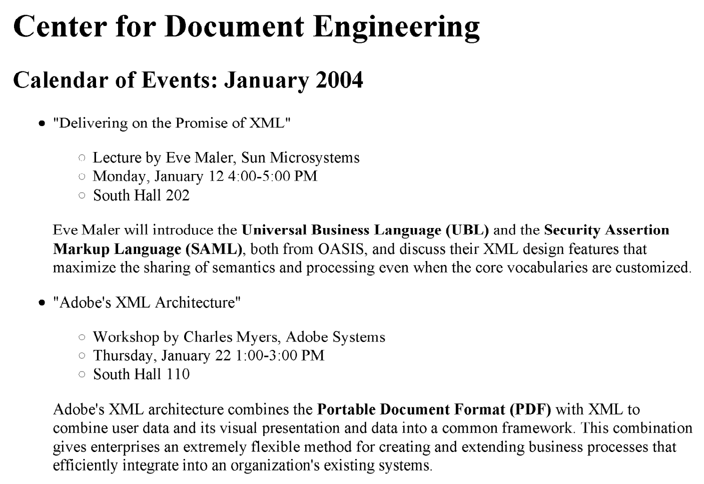
HTML, the language for publishing web pages, will go down in history as one of the most important inventions of our time. It is surely as significant to the creation and dissemination of information as the printing press. HTML and the web browser transformed the Internet, which had been around for two decades but was used primarily by scientists and engineers, into a ubiquitous publishing platform used by everyone from grade-school children to their grandmothers.

HTML took off because it was nonproprietary and because of the conceptual and technical simplicity of publishing with it. Authors used an ordinary text editor to "mark up" a document by surrounding bits of text with "pointy brackets" and tags whose name suggested their structural role or formatting, and the browser did the rest. These two ideas—using tags to enclose or surround content with labels, and relating the labels to the desired presentation of the content—are easy to understand, even for schoolchildren (see SIDEBAR).

A very simple example of an HTML document and how it appears in a browser is shown in Figure 2-1a and 2-1b.

```
<html>
<body>
 <h1>Center for Document Engineering</h1>
 <h2>Calendar of Events: January 2004</h2>
 <ul>
 <li><p>“Delivering on the Promise of XML”</p>
 <ul>
 <li>Lecture by Eve Maler, Sun Microsystems
       <li>Monday, January 12 4:00-5:00 PM
       <li>South Hall 202
       </ul>
       <p>Eve Maler will introduce the <strong>Universal Business Language
(UBL)</strong> and the <strong>Security Assertion Markup Language (SAML)</strong>
and discuss their XML design features that maximize the sharing of semantics and pro-
cessing even when the core vocabularies are customized.</p>
 </li>
 <li><p>“Adobe’s XML Architecture”</p>
 <ul>
 <li>Workshop by Charles Myers, Adobe Systems
        <li>Thursday, January 22 1:00-3:00 PM
        <li>South Hall 110
              </ul>
       <p>Adobe’s XML architecture combines the <strong>Portable Document Format
(PDF)</strong> with XML to combine user data and its visual presentation and data into
a common framework.
 </p>
 </li>
 </ul>
 </body>
</html>
```

Figure 2-1a. A Calendar Event in HTML



Center for Document Engineering

Calendar of Events: January 2004

- "Delivering on the Promise of XML"
    - Lecture by Eve Maler, Sun Microsystems
    - Monday, January 12 4:00-5:00 PM
    - South Hall 202

  Eve Maler will introduce the **Universal Business Language (UBL)** and the **Security Assertion Markup Language (SAML)**, both from OASIS, and discuss their XML design features that maximize the sharing of semantics and processing even when the core vocabularies are customized.

- "Adobe's XML Architecture"
    - Workshop by Charles Myers, Adobe Systems
    - Thursday, January 22 1:00-3:00 PM
    - South Hall 110

  Adobe's XML architecture combines the **Portable Document Format (PDF)** with XML to combine user data and its visual presentation and data into a common framework. This combination gives enterprises an extremely flexible method for creating and extending business processes that efficiently integrate into an organization's existing systems.

Figure 2-1b A Calendar Event in HTML viewed with a Browser

A Primer On Markup Syntax

Markup is the repertoire of characters that takes a flat or undifferentiated stream of text and turns it into a set of elements, which consist of paired text labels and the content they surround or contain. The paired text labels, called the open (or start) tag and close (or end) tag, are distinguished from the text being marked up because they are enclosed by delimiters, the most common of which are the "pointy brackets."

In the open tag, the "<" bracket is immediately followed by the element's name, perhaps one or more element properties or attributes, and the ">" bracket, which indicates the end of the tag. In attribute-value pairs the value must be surrounded by quotes. The order of attributes is not significant.

After the open tag, the element can contain ordinary text content or other elements in an order that is significant, so if the order in which information appears must be preserved, that must be conveyed by using elements.

For example, consider the element: <Event type="Lecture"> in Figure 2-2.

<Event> is the open tag, type is an attribute, and Lecture is the attribute value.
The corresponding close tag </Event> follows after the element's content, which consists of elements for <Title>, <Description>, <Speaker>, <DateTime>, and <Location>. These are called the element's children.

An element can also contain other paired delimiters that mark up some enclosed text to be treated in some special way. The special delimiter sequences can:

- allow for embedded comments (<!– this example is Figure 2-2 in the Document Engineering book –>),
- suppress the interpretation of markup characters (<![CDATA <Event type="Lecture">]]>) so that delimiters can be treated as text content, or
- pass processing instructions to an application (<?xml-stylesheet type="text/xsl" href="calendar.xsl" ?>).

The close tag that follows all the element's children consists of the "<" bracket and a slash (/) followed by the element's name and the ">" bracket. If an element has no children, it is known as empty and the close tag can be omitted if a special syntax is used for the open tag (e.g. <Title/>).

The top-level element in a document is called the document element or root element; it contains or encloses all the other elements, which can be nested as deep as necessary to represent a semantic or structural hierarchy.

The earliest versions of HTML had about a dozen tags, mostly structural ones for describing parts of a document, and most of the earliest browsers had fixed or hard-wired display rules that determined the arrangement of the text, font, size, and everything else.

2.1.1 THE BROWSER WAR

Unfortunately HTML didn't stay this simple for very long. After the Mosaic browser introduced the Web to the masses in 1993, people wanted more control over the appearance and behavior of web pages. This led to the browser wars of the mid-1990s as Netscape and Microsoft added proprietary tags and scripting languages to HTML that worked only in their browsers.[3] The elegant and easily understood idea of fixed mapping between a limited markup vocabulary and display couldn't survive this transformation of the Web into a competitive battlefield.

The idea of a standard and simple HTML vocabulary didn't survive the browser wars

Simple browser displays with default formatting wouldn't enable businesses to create websites whose appearance could differentiate themselves and their products. But until the creation of the World Wide Web Consortium (W3C) in 1995, there was no control over the evolution of HTML and other technical standards for the Web. Browser vendors complied with customer demand and devised tags that enabled rich graphical sites with precise control of text display, blinking text, and spinning corporate logos.[4]

In 1997, the first W3C version of the Cascading Style Sheet[5] (CSS) recommendation emerged, which deprecated the formatting excesses of the proprietary HTML dialects and encouraged more systematic and reusable formatting by using rules that assigned sets of formatting properties to HTML element types.

2.1.2 FROM THE WEB FOR EYES TO THE WEB FOR COMPUTERS

A more fundamental problem with HTML emerged as the Web was transformed into a platform for commerce. Doing business on the Web requires more than just a highly branded website with attractive product catalogs. Businesses need to have both the "Web for eyes" that draws customers to their sites and a "Web for computers" that can encode product information, orders, invoices, payments, and other business documents in ways that can be processed by business applications. For this latter task HTML was fundamentally inappropriate.

Some of HTML's limitations for business applications were inevitable given a tag set heavy on headings, lists, and links. There were no tags for marking up information as product names, item numbers, prices, quantities, and so on to give it a business meaning.

HTML has no tags for marking up business meaning

Clever programmers tried to work around this limited markup vocabulary with code that used whatever markup was available to extract the business information from web pages. For example, a program might rely on the fact that in some web catalog the first item in a list was a product name, the second its item number, and the third the retail price. But programs like these are tedious to write and difficult to maintain; if the layout of the catalog changed, for example, what the program thought was the price of a pair of shoes might actually be the item number.

But the problem for business posed by HTML isn't just how to work around an inadequate set of element types. Using the Web as a business platform radically changes the problem to be solved by the markup language from presentational formatting to semantic modeling, that is, describing business entities and processes in ways that can be understood by business applications. No single vocabulary—HTML or otherwise—can ever be complete enough to describe information with enough semantic precision for all such applications.

No single vocabulary can have enough semantic precision for all applications

2.2 XML'S BIG IDEAS

What the world needed was a new approach to using tags to mark up documents. Instead of a fixed set of element types, we needed way to define whatever set of element types was required for the business application that would use them. We needed an extensible markup language.

There are five big ideas relating to XML that we'll introduce in the following sections:

- XML is extensible: it enables the creation of new sets of tags for domain-specific content.
- XML encodes content as well as presentation formatting; content and its presentation are kept separate.
- XML schemas define models of document types.
- XML schemas enable XML document instances to be validated.

• XML is often produced by converting non-XML information; and XML documents are often transformed to meet the requirements of specific implementations.

2.3 CREATION OF NEW SETS OF TAGS FOR DOMAIN-SPECIFIC CONTENT

Figure 2-2 shows a simple XML document in which the text content is nearly identical to that of the HTML document in Figure 2-1.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="calendar.xsl" ?>
<Calendar>
    <Organization>Center for Document Engineering</Organization>
    <TimePeriod>January 2004</TimePeriod>
    <Events>
    <Event type="Lecture">
            <Title>Delivering on the Promise of XML</Title>
            <Description>Eve Maler will introduce the <Keyword>Universal Business
Language (UBL)</Keyword> and the <Keyword>Security Assertion Markup Language
(SAML)</Keyword> and discuss their XML design features that maximize the sharing of
semantics and processing even when the core vocabularies are
customized.</Description>
            <Speaker>
                <Name>Eve Maler</Name>
                <Affiliation>Sun Microsystems</Affiliation>
            </Speaker>
            <DateTime>Monday, January 12 4:00-5:00 PM</DateTime>
            <Location>South Hall 202</Location>
    </Event>
    <Event type="Workshop">
            <Title>Adobe's XML Architecture</Title>
            <Description>Adobe's XML architecture combines the <Keyword>Portable
            Document Format (PDF)</Keyword> with XML to combine user data and its
            visual presentation and data into a common framework.
            </Description>
            <Speaker>
                <Name>Charles Myers</Name>
                <Affiliation>Adobe Systems</Affiliation>
            </Speaker>
            <DateTime>Thursday, January 22 1:00-3:00 PM</DateTime>
            <Location>South Hall 110</Location>
    </Event>
    </Events>
</Calendar>
```

Figure 2-2 Simple XML Document

At first glance, this XML document doesn't look that different from the HTML one. Both XML and HTML use the same markup syntax, except for the declaration at the start of the XML document that announces that it should be treated as XML and the processing instruction that specifies a stylesheet.

The XML specification is more precise about syntax than HTML is, but most of the differences between HTML and XML enforce the best practices in HTML anyway, such as case-sensitive names and including close tags even when they can be inferred by the presence of the next open tag (HTML allows them to be omitted; see the <li> items in Figure 2-1a). So it isn't syntax that distinguishes HTML and XML.

What matters is that a document that starts with an <html> tag has a fixed set of tags that it might contain. In contrast, XML is extensible: there is essentially no limit to the element types an XML document can contain, and the elements are often named to suggest the meaning of the content. In Figure 2-2 the first open tag is <Calendar>, and in the container formed by this tag and its associated close tag of </Calendar> we can see elements for <Organization>, <TimePeriod>, and <Event>. Inside each event element we see the specific types of content that define an event. Software that displays calendars or searches for events can easily extract the information it needs.

But the difference between HTML and XML isn't just that the former has a fixed set of presentational structure and formatting tags while the latter allows an unlimited set of content-oriented ones. The difference is that HTML is a specific language, a fixed set of element types plus the grammar or rules that govern where in a document each type of element can occur.

XML defines the rules by which specific markup languages are created

XML, on the other hand, is a metalanguage. It defines the rules by which specific XML markup languages are created but says nothing about what element types they use. These specific XML languages are also called XML vocabularies or XML applications.[6] For example, XHTML is an XML vocabulary that recasts HTML in XML syntax to make it more modular and to more rigorously separate content and presentation. And UBL, the Universal Business Language, is an XML vocabulary for business documents.

So while the XML document in Figure 2-2 might be an instance of an XML-defined markup language for describing event calendars, other types of documents like a Shakespeare play or a purchase order would be encoded using completely different sets of elements. Some element types, of course, like <Title>, <Name>, and <Location> are useful in many different types of documents, not just in event calendars.

This last observation has two crucial implications. If common elements are reused, then XML documents can contain element types from more than one XML vocabulary. But a tag name like <Title> might be part of a vocabulary for books, a deed of ownership, or honorifics for a person, so we need some syntactic mechanism for distinguishing vocabularies from each other. We'll defer this problem until Section 2.5.4 when we discuss XML schemas.

2.4 SEPARATION OF CONTENT AND PRESENTATION

Every document—whether it is an event calendar, purchase order, Shakespearean play, chemistry text, or tax form—contains a variety of types of meaningful information. When we use XML tags to encode this meaning, we can label parts of the document to distinguish different types of content: <Speaker>, <Name>, <Address>, <Personae>, <Scene>, <Speech>, <Molecule>, <Income>, and so forth. These are purely conceptual distinctions, and these bits of content don't have any inherent formatting or presentation associated with them.

It is only when XML documents are printed, displayed, spoken, or otherwise rendered to communicate with people that formatting or presentational information, such as page numbers, type fonts and sizes, color, indentation, column organization, underlining, pitch, and intonation, needs to be added. These presentational devices can assist in understanding the content but generally don't carry much content-specific meaning.

Of course there are important conventions and correlations between presentation and meaning: large type implies more importance than small type, red may signal a warning, line breaks in poems support meter, and so on. We celebrate graphic designers, artists, and book designers when they exploit or violate these conventions in clever ways. But most presentational decisions are more arbitrary. For example, the typeface in which this book is printed has little or no effect on its meaning. We will discuss these issues in more detail in Chapter 12, "Analyzing Document Components."

Sometimes content and presentation are bound together or confounded, often implicitly, as with HTML or with word processors that use style sheets or formatting templates to apply formatting to otherwise unlabeled information components. Cascading style sheets have reduced the implicitness and ad hoc-ery of HTML formatting, but they weren't designed to separate content and presentation. They were just a way to regain some of the core simplicity of HTML by delegating more sophisticated format control to a separate style processor in the browser.[7]

In XML the separation of content and presentation is inherent and desirable

In XML the separation of content and presentation is inherent and desirable. If an XML document can contain any type of element it needs to describe its content, there is no way that a browser can know in advance what it means or how to display it. Most web browsers render an XML document with indentation that corresponds to the hierarchical structure created by its tags, but this display might not be optimal or even appropriate for the semantics embodied in the content. It is almost always necessary to apply to the XML document a transformation or stylesheet that creates HTML or some other presentation-oriented vocabulary to the XML information. Sometimes a stylesheet is then also applied to the transformed HTML to optimize its presentation.

The extra step needed to display an XML document isn't a bug, but a feature. It makes a requirement out of what should be a good habit to practice in any case, that of paying explicit attention to the relationship between content and presentation. It emphasizes the idea that XML elements should be used to encode conceptual distinctions in a presentation-independent manner to enable the reuse and repurposing of information for different contexts or implementations.

Even if an XML document contains elements with the same name as HTML ones that browsers readily display, like <h1> or <p> or <li>, they don't get displayed because no presentation is ever assigned by default. XML elements contain content, pure and simple. So it is misleading and pointless to use element names that assume otherwise.

XML's separation of content and presentation also reinforces and rewards specialization in skills between information modeling and user interface or graphic design. User interface and graphic design skills are useful in Document Engineering, but good information modeling skills are essential.

2.5 DEFINITION OF DOCUMENT TYPES

Documents are ubiquitous. All documents share the idea that they are purposeful representations and organizations of information, but they exhibit great variety. On any given day we encounter dozens of different types of documents.[8] We might start the day with a morning newspaper, go on to deal with reports, emails, catalogs, reference books, calendars, or lectures, and end up with a restaurant menu, murder mystery, TV program guide, or MP3 playlist.

It is easy to distinguish a dictionary from an invoice, a newspaper from a novel, or a restaurant menu from a collection of poems, because each document follows a characteristic structural pattern to arrange types of content unlikely to be found in the other. Because these types of document are so different, even a simple list of the varieties of content in each document would accurately classify any given instance of the document.

This intuitive notion of models of different types of documents is very useful. It explains why we have had standard business forms for centuries, style guides for authors, national and international standards for electronic business messages, templates in word processors and spreadsheets, and various other ways of describing expectations about content and its arrangement in documents.

2.5.1 DOCUMENTS AS IMPLEMENTATION MODELS

In the domain of Document Engineering, we need to define models of different types of documents in a rigorous and unambiguous way so that we can automate their process or exchange within or between applications. We also want to use their formal definitions to generate and drive some of the software needed to process the documents. Implementations or instances of these document models enable software to locate and extract the information needed to connect related document exchanges that combine to form supply chains, auctions, marketplaces, and other business patterns.

XML was designed to give the intuitive idea of a document model a more physical, formal foundation.[9] XML gives us syntactic mechanisms that capture the semantic distinctions between documents in terms of the sets of elements and attributes used to encode their content and the rules that govern their occurrence and organization. Two semantically related document models like purchase order and invoice may share elements from a common library or subset, but they are distinguished by elements that occur only in one of them or that have different possible values in each. So we use different vocabularies to mark up the content of purchase orders and of invoices.

XML can realize document models suitable for implementation in applications

XML's ease of use, its expressive power, and its processability have made it attractive for Document Engineering because it can realize document models suitable for implementation in applications.[10] But what really matters is the quality of the analysis and design that gets represented in conceptual models before we encode them in XML vocabularies. XML is a convenient syntax for encoding the models, but XML per se doesn't help us create good models, and many people have found it a convenient syntax for creating poor ones. We'll return to this problem of the quality of document models in Chapter 6, and starting in Chapter 7 we'll introduce the methods and technologies of Document Engineering to explain how to create good ones.

2.5.2 XML SCHEMAS

The formal description of a document model in XML goes by various names, but it is most useful for our introduction here to call it the XML schema. Simply put, an XML schema defines the possible types of content in a document and the rules that govern the structure and values of that content.

Every XML schema contains definitions of element types. But as we've pointed out, because many types of elements occur in more than one type of document (<Title>, <Name>, <Date>, and so on), a list of legal element types is often not sufficient to distinguish different types of documents. Furthermore, even though the name of an element type can suggest what it means, it is not self-describing.[11] An XML schema also specifies the attributes that can be associated with elements, but they're not self-describing either. So if the full meaning of an element isn't conveyed by its name, where is it conveyed?

The meaning of elements is represented in an XML schema through the constraints or rules that govern the structural arrangement of elements and the values that elements and attributes can have. We call these constraints business rules.

The term, business rule, like model and pattern and other fundamental concepts of Document Engineering, has numerous incompatible or overloaded definitions.[12] Everyone agrees that a business rule expresses a constraint about some aspect of the data or processes used by a business. Furthermore, everyone agrees that it is desirable to represent rules independently from the generic aspects of applications instead of scattering them into multiple layers of application software. But there is little agreement about how to classify business rules and how to translate them from expressions of requirements into implemented systems. We'll present a classification scheme for business rules in Chapter 8, and we'll stress the roles they play in developing an adequate conceptual model of the documents and process in some specified business context. For now we'll focus on the kinds of business rules that can be represented in XML schemas.

The kinds of rules expressed in XML element definitions include containment relationships ("a dictionary entry consists of a word, a pronunciation, and a definition"); sequence and cardinality relationships ("the abstract must be followed by one or more chapters and possibly one or more appendixes"), choices ("the location must be a street address or a pair of latitude and longitude coordinates"), and recursion ("the bill of materials is a list of parts, each of which may consist of a list of parts"). Of course, these kinds of rules are not mutually exclusive; we can represent a containment rule that defines a legal sequence of elements, each of which consists of a choice, one option of which is recursive.

There is often a gap between the conceptual model and what can be described in XML

The document model of a purchase order might include business rules like "the quantity ordered must be an integer less than 1,000," "the unit price must be expressed as a number with two decimal digits," or the "the country code must be one of those contained in ISO 3166." It would be highly desirable to encode these rules in the XML schema that implements the model of a purchase order as constraints on the values of elements or attributes. But as we'll see in the next section, there is often a gap between the conceptual document model and what can be described in XML.

2.5.3 SCHEMA LANGUAGES

There are currently several XML schema languages that differ substantially in how completely they can express the business rules that underlie a document's model. Which schema language to use is influenced by where the document lies on the Document Type Spectrum (see Figure 1-3), because that determines what aspects of the model are most important to express (see SIDEBAR).

Understanding XML Schemas by Analogy

To explain XML schemas it may help to make the analogy to relational database schemas, which describe the database content in terms of possible field values, relationships between fields in tables, and constraints between tables. An XML schema could describe the semantics of a class of documents so that different types of content can be identified and extracted as if they were in a document database. An XML schema can ensure that information exported from a database or other application is assembled as a valid document.

Likewise, we can make an analogy between XML schemas and class definitions in a modern programming language. A class is a template that specifies the meaning of the variables used by an object in terms of their data types or possible values, and classes can be related to each other by association, specialization or generalization. An XML schema might specify the required data types for document content, and might also express relationships between types of document content. This equivalence enables XML schemas to be treated just like classes to guide the creation of objects, a process usually called data binding. This view of XML schemas is appropriate for transactional documents and also very useful when describing web forms and other information-intensive user interfaces.

Finally, we can say that an XML schema defines a vocabulary for a document model expressed with a formal grammar. A grammar for any language is a system consisting of a finite set of tokens and a finite set of rewrite rules that generate all the valid sequences or sentences of those tokens. For an XML schema the tokens are the elements and attributes and the sentences are the document instances. This linguistic perspective on XML schemas fits very well for narrative documents and less well for transactional ones.

The first XML schema language was Document Type Definition (DTD), a legacy of XML's SGML heritage. Because of SGML's origins in technical publishing, DTDs were designed to represent the structural properties of documents, but they treat most data as just text and can't represent meaningful information models.

DTDs have a very simple and compact syntax, but this syntax is not itself XML. DTDs are sufficient for describing models of narrative document types like newspapers, dictionaries, and reports, whose content is primarily text and intended for use by people. DTDs can also easily express mixed content models in which character data can contain "in-line" elements, a very common requirement in narrative documents. For example, a product description is text that can contain glossary terms, company names, or URLs, all of which would be tagged as elements mixed in with the text of the product description.

But as we move on the Document Type Spectrum toward the transactional or data-centric document types that are primarily used by business applications, structural description alone captures fewer of the most important aspects of the document's content. For example, constraints on data values are crucial.

For transactional document types the most useful schema language is the one recommended by the W3C called XSD or XML Schema (with a capital S). XML Schema was developed to meet a much broader and more computer-oriented set of requirements than DTDs were. XML Schema documents are encoded using XML syntax and overcome most of the limitations of DTDs. The XML Schema language includes all the basic data types common in programming languages and databases (string, Boolean, integer, floating point, and so on), as well as mechanisms for deriving new data types. For example, an XML Schema schema can define a Student as a specialization of a Person type with additional required elements, or an alphanumeric PartNumber as a string whose values are restricted using regular expressions.

An extremely important facility in XML Schema is its support for namespaces, a mechanism for distinguishing XML vocabularies so that a schema can reuse definitions while avoiding conflicts between elements with the same name that mean different things (as we suggested at the end of Section 2.3, <Title> might be part of a vocabulary for books, legal documents, or honorifics for a person). A prefix associated with each namespace can be attached to elements in document instances, so that <book:Title>, <legal:Title>, and <honorific:Title> aren't confused. Using a namespace to identify the additional elements needed to customize a standard vocabulary maintains the integrity of the base vocabulary.

Needless to say, the greater expressiveness and extensibility of XSD comes with substantially more complexity, as we can see in Figures 2-3a and 2-3b, which compare a DTD and XSD for the same document model, that of a simple calendar like the example in Figure 2-2.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for simple calendar -->
<!-- calendar metadata -->

<!ELEMENT Calendar (Organization, TimePeriod, Events)>
<!ELEMENT Organization (#PCDATA)>
<!ELEMENT TimePeriod (#PCDATA)>

<!-- a calendar is a list of events -->
<!ELEMENT Events (Event+)>

<!-- definition of each event, optional Event Type attribute -->
<!ELEMENT Event (Title, Description?, Speaker?, DateTime, Location)>
<!ATTLIST Event
    type (Lecture | Workshop) #IMPLIED>

<!ELEMENT Title (#PCDATA)>

<!-- mixed content definition to allow for keywords in Description -->
<!ELEMENT Description (#PCDATA | Keyword)*>

<!ELEMENT Keyword (#PCDATA)>
<!ELEMENT Speaker (Name, Affiliation)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Affiliation (#PCDATA)>
<!ELEMENT DateTime (#PCDATA)>
<!ELEMENT Location (#PCDATA)>
```

Figure 2-3a. DTD for a Simple Calendar

The DTD for a simple calendar is very compact because of the use of of +, ?, and * to represent occurrence constraints. Commas separate the members of a sequence, and the vertical bar distinguishes choices. Every element has a declared data type of "PCDATA" (parsed character data), which means a string of text in DTD.

In contrast, the XSD for the simple calendar in Figure 2-3b is much more verbose than the DTD. Occurrence constraints, sequences, and choices are all expressed explicitly. It is easy to get lost in embedded definitions.[14] But the syntax is XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<!-- XSD Schema for Calendar -->
<xs:element name="Calendar">
<xs:complexType>
  <xs:sequence>
    <xs:element name="Organization" type="xs:string"/>
    <xs:element name="TimePeriod" type="xs:string"/>
<!-- Definition of Event as Sequence of Other Elements -->
    <xs:element name="Events">
    <xs:complexType>
       <xs:sequence>
         <xs:element name="Event" maxOccurs="unbounded">
         <xs:complexType>
            <xs:sequence>
              <xs:element name="Title" type="xs:string"/>
              <xs:element name="Description" minOccurs="0">
                 <xs:complexType mixed="true">
                   <xs:choice minOccurs="0" <MaxOccurs="unbounded">
                <xs:element name="Keyword" type="xs:string"/>
                   </xs:choice>
                </xs:complexType>
                </xs:element>
               <xs:element name="Speaker" minOccurs="0">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="Name" type="xs:string"/>
               <xs:element name="Affiliation" type="xs:string"/>
             </xs:sequence>
                </xs:complexType>
                </xs:element>
               <xs:element name="DateTime" type="xs:string"/>
              <xs:element name="Location" type="xs:string"/>
          </xs:sequence>
      <xs:attribute name="type">
      <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="Lecture"/>
          <xs:enumeration value="Workshop"/>
          </xs:restriction>
      </xs:simpleType>
     </xs:attribute>
  </xs:complexType>
  </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Figure 2-3b. XML Schema for a Simple Calendar

2.5.4 RULES THAT SCHEMA LANGUAGES CAN'T REPRESENT

Every XML schema language makes tradeoffs that determine the range of document models it can realize, the ease with it defines them, and how readily it can reuse a model or parts of models in more than one schema. For example, even though XML Schema is a powerful schema language, it isn't capable of expressing dependency constraints on element content ("the start time for a calendar event must be earlier than the end time" or "if the total is greater than $1,000 the purchase order requires an authorization code"), even though these may be important rules for the context of use.

Every XML schema language makes tradeoffs

Rules that concern multiple values in a document are easy to express using XML constraint based languages such as Schematron.[15] This uses the XPath language for describing parts of XML documents to make Boolean assertions based on their content. But the tradeoff here is that this approach makes Schematron incapable of representing structural rules except in very tedious ways.

Another grammar based schema language called RELAX NG[16] is widely regarded by experts as more elegant and simpler than XML Schema, but with about the same expressive power. However, because it wasn't developed by the W3C, RELAX NG isn't as widely supported by vendors of XML software.

Obviously no schema language is perfect at encoding all models in XML. But that's probably a good thing, because it reinforces our message that analysis and modeling skills are more fundamental to Document Engineering than XML is.

2.6 VALIDATION

An XML schema communicates the model of a document type to people or applications that need to create or receive document instances. In this sense the XML schema is a contract that defines the rules that any documents must follow. Validation is the process of testing whether an XML document follows the rules defined in an associated schema. A document that follows or satisfies the schema is said to be valid.

For XML documents described by simple DTDs the schema can be carried along with the document content in its prolog, but it is far more common for an XML document to refer to an external schema. This indirect binding is more efficient and flexible than including the schema in the document, because it allows a single schema definition to be reused by all documents of the same type. And of course, if two parties in an ongoing business relationship are exchanging documents with each other, they've already come to terms about the schemas that define what they send and receive. Once the business process is established, there is no need to send schemas with the documents.

A person who understands XML schemas and syntax can examine an XML instance document and validate by eye. But validation is most often carried out by a validating parser embedded in an XML-aware text editor, application server, integration tool, or other software that processes XML. In an XML-aware text editor, an XML schema can speed the creation of documents by inserting required tags and by dynamically controlling the structure of menus or selectors to ensure that only valid documents are created.

A much weaker criterion of quality checking for XML documents is called well-formedness and requires that the XML document meets some minimal syntactic constraints, such as having exactly one root element and having matching start and end tags that don't overlap. An XML document that isn't even well-formed will be rejected by an XML parser and not passed on for further processing.

Even an XML document that is well-formed but that fails some constraint defined in its associated schema might still be acceptable. For example, it would be a good business practice to try to process a purchase order from a potential customer even if it omitted the required postal code in the shipping address.

A document without a schema is just a bag of tags

On the other hand, a well-formed but schema-less document is little more than a bag of tags whose meanings are undefined. It makes little business sense to invent a set of tags and not bother to formally define them with a schema, and it would be risky to attempt to process such documents. Suppose a document from a potential customer begins with a <PurchaseOrder> tag, but other tags inside it contain instructions to empty out a firm's bank account or crash its systems. If that document claimed to conform to the firm's schema for purchase orders we'd be able to tell that it didn't.

Nevertheless, because of the unavoidable limitations in every XML schema language, it is impossible to capture every rule and requirement of a conceptual document model. So even a strong claim that a document is valid should always be understood to mean "with respect to the class of constraints that the schema language being used is capable of encoding." Knowing that a piece of data is in its expected location and of the required data type doesn't mean that it is correct.

Ultimately how much validation is necessary in any situation is a separate question from how much validation power is inherent in the schema language. What matters the most is having a common intention between the producer and consumer of an XML document. Imagine an XML document used by a single software program for the sole purpose of saving its private data. If the file is saved correctly, the information will be valid when that software next uses it. Validation is hardly necessary. At the other extreme, suppose an XML document arrives from a company halfway around the world with which a firm has no prior business relationship. Validation against its assigned schema is irrelevant. The firm would be wise to validate the document against expected data requirements before letting it enter their business application. This is especially important in situations where accepting the document creates a legally-binding commitment between the sender and recipient of the document.

2.7 CONVERSION AND TRANSFORMATION

XML is often produced by converting non-XML information, and XML documents are often transformed to meet the requirements of other contexts or implementations. Conversion to XML and transformation from XML might seem like two views of the same activity, but while related they differ in many respects so we'll discuss them separately. The issues and problems that arise in conversion and transformation are also shaped by where the source and target documents lie on the Document Type Spectrum (see Figure 1-3); the greater semantic precision in transactional documents makes them easier to convert to or transform, regardless of the source or target syntax.

2.7.1 CONVERSION TO XML

A common reason for converting information to XML is to facilitate a single-source publishing strategy in which content is created once and then reused many times. Reuse can involve the same content included in all the instances of a document, as might be the case for a copyright notice, standard terms and conditions, or similar boilerplate text. A variant of single-source publishing is syndication, in which a single source of content is simultaneously published or distributed for reuse in other contexts. Many websites and web publishers convert syndicated news, blogs, or other time-sensitive content to an XML vocabulary called the RDF Site Summary (RSS).

A different perspective on reuse involves extracting or formatting the same piece of content in many different ways to create different documents. This form of reuse is often called repurposing. An example would be using some of the same information in a system's product documentation, a troubleshooting guide, and training materials.

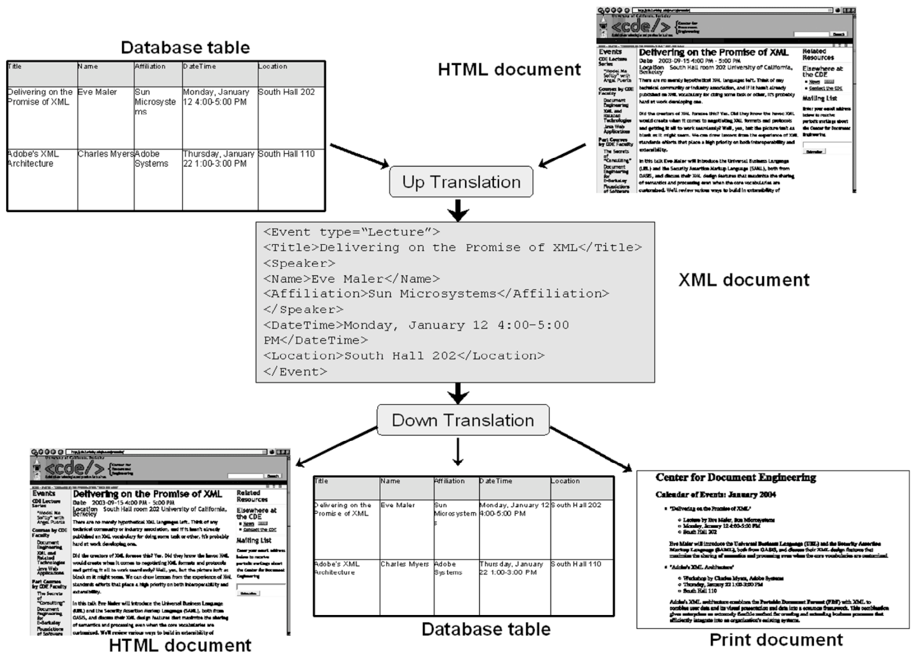
Another important reason for converting information to XML is to extract information from a database, ERP system, or legacy application primarily used inside an enterprise to enable Internet-based transactions with customers or business partners. A similar type of conversion takes place in many EDI implementations, where business-to-business document exchanges in supply chains move to XML to make the content easier to process.[17]

The conversion of information to XML can be completely automated if the information source is well structured with explicit semantics and the structure and semantics are rigorously described with a schema. This description fits databases and some of the file formats used by ERP systems and other enterprise applications. This doesn't mean that mapping between the non-XML format and the target XML document type is automatic. Only that once it is in place, we can create software that converts one into the other.

The benefits of converting to XML are more compelling when information is encoded in less structured or semantically expressive formats such as ASCII, RTF, UN/EDIFACT, ANSI ASC X12, or HTML that don't embody XML's big ideas. But it's a lot of work to design an appropriate XML vocabulary and then apply markup correctly to the content.

If authors follow structure and style standards when they create office documents or web pages, some of the conversion effort can be automated by exploiting the implicit relationships between formatting styles or HTML tags and the target XML vocabulary. But few authors have this much discipline, so conversion usually requires expensive and tedious work by people who understand the content to supply the missing meaning.

The process of adding value to information by converting it to XML is often called up translation to express the work it takes to give XML the informational equivalent of potential energy. Once information is in XML syntax its greater potential energy makes it easy and straightforward to create any other format, so naturally the transformation from XML to a non-XML format is often called down translation. These relationships between XML and other formats are illustrated in Figure 2-4.



Title	Name	Affiliation	DateTime	Location
Delivering on the Promise of XML	Eve Maler	Sun Microsystems	Monday, January 12 4:00-5:00 PM	South Hall 202
Adobe's XML Architecture	Charles Myers	Adobe Systems	Thursday, January 22 1:00-3:00 PM	South Hall 110

```
<Event type="Lecture">
<Title>Delivering on the Promise of XML</Title>
<Speaker>
<Name>Eve Maler</Name>
<Affiliation>Sun Microsystems</Affiliation>
</Speaker>
<DateTime>Monday, January 12 4:00-5:00
PM</DateTime>
<Location>South Hall 202</Location>
</Event>
```

Title	Name	Affiliation	DateTime	Location
Delivering on the Promise of XML	Eve Maler	Sun Microsystem s	Monday, January 12 4:00-5:00 PM	South Hall 202
Adobe's XML Architecture	Charles Myers	Adobe Systems	Thursday, January 22 1:00-3:00 PM	South Hall 110

Center for Document Engineering

Calendar of Events: January 2004

- "Delivering on the Promise of XML"
  - Lecture by Eve Maler, Sun Microsystems
  - Monday, January 12 4:00-5:00 PM
  - South Hall 202

  Eve Maler will introduce the Universal Business Language (UBL) and the Security Assertion Markup Language (SAML), both from OASIS, and discuss their XML design features that maximize the sharing of semantics and processing even when the core vocabularies are customized.
- "Adobe's XML Architecture"
  - Workshop by Charles Myers, Adobe Systems
  - Thursday, January 22 1:00-3:00 PM
  - South Hall 110

  Adobe's XML architecture combines the Portable Document Format (PDF) with XML to combine user data and its visual presentation and data into a common framework. This combination gives enterprises an extremely flexible method for creating and extending business processes that efficiently integrate into an organization's existing systems.

Figure 2-4. Up- and Down-Translation with XML

A corollary here is that if we anticipate that information we are about to create will someday need to be represented in XML, it is more cost-effective to create and manage it as XML and then down-translate to whatever other formats we need in the short term.

2.7.2 TRANSFORMATION FROM XML

XML schemas and documents are often transformed to meet the requirements of other contexts or implementations. Transforming an XML document involves selecting, reordering, or restructuring its content. Transforming an XML document so that it conforms to a different XML schema is often followed by down-translation to a non-XML format, for example by EDI gateway applications.

Transformation reuses or amortizes the investment made to encode information in XML in the first place. Put another way, when we create XML schemas and documents, we should design them expecting to transform them to preserve and extend their value.

As we said earlier, with XML the separation of content and presentation is both inherent and desirable. Thus it is often necessary to transform or down-translate XML to HTML so that it can be viewed in a web browser. The process of applying a presentation to an XML document is sometimes called styling but it is more useful to conceive of applying a style as two separate processes of transformation and formatting. This way of thinking lines up conceptually with two complementary W3C Recommendations: XSLT, the Extensible Stylesheet Language for Transformation, and XSL FO, the Extensible Stylesheet Language Formatting Objects.

XSLT is an XML-aware functional programming language that operates on logical "node sets" derived from the element and attribute structure of XML documents. XSLT has the usual constructs for logical flow of control like conditional, loops, and switches. What makes it most useful for transforming XML are its XPath facilities for expressing and matching patterns in the logical XML structures so that arbitrary trees or subtrees can be selected and rearranged. This is the approach used by the Schematron schema language.

XSL FO, often a target vocabulary of an XSLT transform, is designed for typesetting-quality control of printed XML output. An XSLT transform from XML to HTML can be as simple as a set of rules that assign an HTML tag to each XML element type, defaulting all presentation control to the browser. An XSLT transformation like this can be used to enforce presentation standards for all instances of a document. Figure 2-5 shows a simple XSLT program that transforms the XML calendar instance in Figure 2-2 to HTML to reproduce the appearance of the HTML calendar shown in Figure 2-1b. The processing instruction in the second line of the Figure 2-2 instance associates the XSLT program (calling it "calendar.xsl") with the instance.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Calendar">
    <html>
    <head>
        <title><xsl:value-of select="/Calendar/Organization"/><xsl:text>
</xsl:text><xsl:value-of select="/Calendar/TimePeriod"/><xsl:text>
</xsl:text>Calendar</title>
    </head>
    <body>
        <h1><xsl:value-of select="/Calendar/Organization"/></h1>
        <h2>Calendar of Events: <xsl:value-of select="/Calendar/TimePeriod"/></h2>
        <xsl:apply-templates select="Events"/>
    </body>
    </html>
</xsl:template>
<xsl:template match="Events">
    <xsl:for-each select="Event">
        <ul>
            <li>"<xsl:value-of select="Title"/>"</li>
            <br/><br/>
            <ul>
                <li><xsl:value-of select="@type"/> by <xsl:value-of
select="Speaker/Name"/>, <xsl:value-of select="Speaker/Affiliation"/></li>
                <li><xsl:value-of select="DateTime"/></li>
                <li><xsl:value-of select="Location"/></li>
            </ul>
            <br/><xsl:apply-templates select="Description"/>
        </ul>
    </xsl:for-each>
</xsl:template>
```

```
<xsl:template match="Description">
    <xsl:apply-templates/>
</xsl:template>
<xsl:template match="Keyword">
    <b><xsl:apply-templates/></b>
</xsl:template>
</xsl:stylesheet>
```

Figure 2-5. XSLT Transformation Program

Transforming XML to HTML can be a highly sophisticated process. For example, a set of XML transforms can create a website of highly interlinked HTML files and by making multiple passes through the input documents can extract titles and headings to create tables of contents and navigation aids. These ancillary structures can be regenerated automatically whenever the XML content changes, and cascading style sheets can be switched in and out for precise control of site appearance.

Transforming XML to HTML is often just a small part of a single-source publishing strategy in which XML content is transformed for a variety of output devices or channels such as PDAs, wireless phones, text-to-speech synthesizers, Braille devices, and of course, printers. This form of transformation for reuse in different devices or media is often called repackaging. In this case a given XML document instance may have different XSLT transforms applied to it in different implementations.

XML may also be transformed to send information back into a database, ERP system, legacy application, or EDI exchange. Chapter 6 discusses how transforming XML documents from one schema to another, or extracting and combining information from one or more documents to create an instance that conforms to another schema, are essential techniques for making information interoperable.

2.7.3 WHERE TO TRANSFORM

XML is now used everywhere in distributed computing architectures. It can be the native format in an XML database or created by conversion from a non-XML database, ERP application, legacy system, or EDI data source. XML can be sent any-where inside or outside the enterprise to expose information or functionality or to create an extended enterprise like the virtual drop shipment bookstore hypothesized in Chapter 1. Since many web browsers contain XML parsers and support XSLT, XML can go all the way to the end user's client. Figure 2-6 illustrates this "XML everywhere" phenomenon.

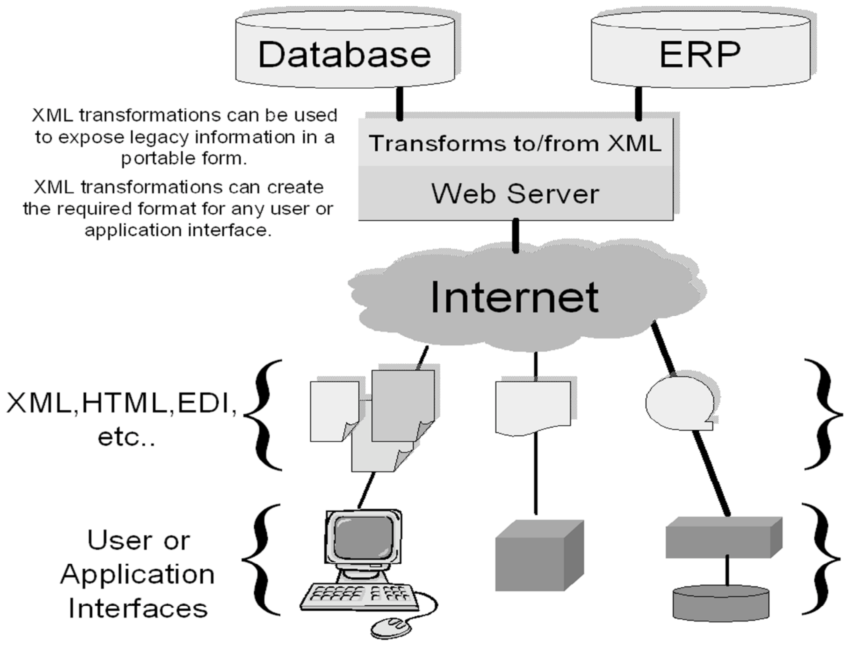

Database
ERP
XML transformations can be used to expose legacy information in a portable form.
XML transformations can create the required format for any user or application interface.
Transforms to/from XML
Web Server
Internet
XML,HTML,EDI, etc..
User or Application Interfaces

Figure 2-6. XML Everywhere in a Generic System Architecture

XML is now everywhere in distributed computing architectures

But XML is often not sent all the way through a distributed application. Instead it is sometimes transformed to HTML or to non-XML formats before it gets to the browser or the legacy application. But, given XML's flexibility, portability, and processability, why would anyone down-translate to a less expressive and computable format? Sometimes decisions about where to transform are based on technical capabilities. It might be easier, in terms of the tools and people available to do it, to transform XML to another format on one side of a document exchange rather than another.

Or the decision might be based on efficiency considerations. A business may implement all the transformations needed to support its supply chain or trading partners at a single gateway or hub. This consolidates all of the know-how, required technology, and support personnel in one place and allows all the external enterprises to continue using their legacy technology to produce and consume the documents they exchange with the hub enterprise. Documents are also likely to be smaller when they are optimized for a specific device or application.

The decision about where to transform is a business one

Ultimately the decision about where to transform is a business one. Exchanging an XML document and the schema that governs it reveals a great deal of information about how an enterprise organizes its information and conducts its business processes. The information model in a schema might include principles of product classification, manufacturing tolerances, schedule flexibility, pricing algorithms, capacity allocation, and other valuable proprietary information.

We may want to exchange this information with a trusted business partner for mutual benefit, or we may choose to send a substantially down-translated instance that conveys a much simpler view of the business. We might even create customized transformations of our information whose richness depends on how much someone is willing to pay for it.

2.8 KEY POINTS IN CHAPTER TWO

- Using the Web as a business platform changes the problem from presentational formatting to semantic modeling.
- HTML has limited use for business applications because it has no tags for marking up information to give it business meaning.
- XML has rapidly become the preferred format for representing physical models of documents and business processes.
- XML is a metalanguage for markup, and markup languages can be created for very specific document models.
- With XML, the separation of content and presentation is inherent and desirable.
- XML schemas define the rules that govern the arrangement and values of a document's content.
- An XML document without a schema is little more than a bag of tags whose meanings are undefined.
- There is often a gap between the conceptual model of a document and what can be described in an XML schema.
- XML is now everywhere in distributed computing architectures.
- The decision about where to transform documents is a business one.