Basic Document Design

The subject of document design is certainly not one that can be covered in its entirety in just one chapter. As such, we will stress the fundamental principles of document design, and focus on just three specific areas:

- **Narrative and Data-oriented document structures.** These two document styles comprise the vast majority of XML documents. In this section, we discuss broadly the defining characteristics of each style and look at examples of when to use each one. The majority of the rest of the chapter will deal with data-oriented document structures, but you can find good information on building narrative document structures in other books such as *XML in a Nutshell* from *O'Reilly and Associates (ISBN: 0-596-00292-0).*
- **Building Blocks: Attributes, elements, and character data.** While there are many other features of XML documents, few of them have the kind of impact on design, readability, and parsing performance as these three can and do have. In this section, we offer advice on the appropriate use of these basic building blocks, including performance implications.
- **General data modeling pitfalls.** Data modeling is a huge subject – one that could easily fill a book of its own. So, rather than trying to cover everything, we restrict our discussion to the common pitfalls that can have very serious side effects when it comes time to use your documents.

This chapter is meant to serve as a foundation for the material that will be covered in later chapters, which strive to help you design better documents for specific purposes, such as doing XSL transforms or using XML for long-term storage.

2.1 Narrative and Data-oriented Document Structures

XML documents are normally used to model data in one of two ways. A **narrative document** structure uses the XML content to augment some existing text-based data, in a similar way to how HTML tags are used to augment the text in web pages. In a **data-oriented document** structure, the XML content is itself the important data. In this section, we'll examine these two styles and walk through some examples of when you would want to use each. This section discusses, explicitly and briefly, ideas that are largely "common sense" but that are central to the rest of the chapter.

2.1.1 Use Data-oriented Document Structures to Model Well-structured Data

As discussed earlier, a data-oriented document structure is one in which the XML content directly describes the data in the document; in other words, the XML markup *is* the interesting data in the document. A small example illustrates this point well:

```
<shape>
  <type>octagon</type>
  <numberOfSides>8</numberOfSides>
  <numberOfDimensions>2</numberOfDimensions>
</shape>
```

This document uses XML to describe some properties of a shape. Here, the textual content (alternatively called parsed character data, or just character data; we'll use these terms interchangeably) in the document (`octagon`, `8`, and `2`) is meaningless without the XML tags. This example could also be written as follows:

```
<shape>
  <type name="octagon"/>
  <numberOfSides count="8"/>
  <numberOfDimensions count="2"/>
</shape>
```

This document encodes the same data as the first one, but instead of using character data to describe the values, it uses attributes. In this case, there is no character data at all, and it's clear that the XML content is, itself, the data in the document; if you were to remove the XML content, the document would be empty aside from white space. Note that the document style above isn't one that we'd recommend for use in practice, as we'll see in detail a bit later in the section. It's only presented here for the sake of comparison.

To continue the process, we could take this example one step further by building our XML as follows:

```
<shape type="octagon" numberOfSides="8" numberOfDimensions="2"/>
```

This document encodes the same data as the first two, but this time the type, side count, and dimension count are all data attributes of the `<shape>` tag. In this case, it's obvious that the XML markup is the only useful data in this document. It would easily be possible to encode this same data in many other ways, but the point is clear.

All three of these documents are simple examples of data-oriented documents in that they use XML markup to describe some well-structured data. With the data represented in these particular documents, you can imagine that the data has a one-to-one mapping with the properties of an object in our application, effectively making the XML document a serialized version of our object.

When you are encoding objects whose basic job is to communicate with software (as opposed to communicating with people), you are probably going to want to use a data-oriented document structure. The hierarchical nature of XML allows it to easily represent complicated data structures that include members that are also complicated data structures. Any data that can be encoded as text (including binary, which can be encoded using a base-64 encoding scheme or something similar) can be represented in XML.

This kind of serialization is a very common use of XML. In fact, Microsoft's .NET Framework Common Library includes functionality that can be used to automatically serialize any class to XML, and deserialize an XML stream into an object, via the `XmlSerializer` class in the `System.Xml.Serialization` namespace. At the time of writing, there was no corresponding class in the standard Java libraries, but there is an API currently under development, the functionality of which will be very similar. This API is a part of the Java XML Pack and is called the **Java API for XML Binding** (**JAXB**), and provides a way of mapping from Java classes to and from XML documents, as well as validating in-memory Java objects against a DTD or an XML Schema.

> *You can find out more information about JAXB from Sun's JavaSoft website at* http://java.sun.com/xml/jaxb/index.html*. We cover the .Net* `XmlSerializer` *class briefly at the end of Chapter 4.*

Obviously, data-oriented documents are not limited in usefulness to holding serialized data structures. For JavaBeans and other data structures, the mapping from member variables to XML content is usually very straightforward, but XML can be used to represent other kinds of data that don't map directly to a data structure. For instance, the Ant build system from Apache uses an XML file to describe a *process*, in this case the process of building a piece of software. (In fact, Ant can be used to do more than just build software, just like makefiles. You can find out more about Ant at the Apache website: http://jakarta.apache.org/ant/index.html.) A well-defined process can be thought of as well-structured data, so an encoding of a well-defined process can be done using a data-oriented document.

An Ant build file is functionally similar to a `makefile` in that it contains build targets that in turn contain the commands (Ant calls them "tasks") necessary to build something. In a manner similar to makefiles, targets can also have interdependencies. These targets and their dependencies together define the process of building a piece of software. Ant also uses the extensibility of XML to provide some interesting functionality that makefiles cannot directly support, such as allowing for writing custom tasks in Java, that can then be used in your build files just like any of the predefined tasks.

By now, it should be clear that data-oriented documents are the best choice for representing well-structured data of virtually any flavor. Let's move on and look at narrative documents, which are better suited for less structured data.

2.1.2 Use Narrative-style Document Structures to Augment Plain-text Data

The essential difference between narrative document structures and data-oriented document structures is that narrative document structures are designed to be consumed by people, whereas data-oriented document structures are generally designed to be consumed by applications. As opposed to data-oriented structures, narrative documents are usually human-readable text that is augmented in some way with XML markup.

Specifically, the two basic characteristics that distinguish a narrative document from a data-oriented document are as follows:

- **The meaningful content is not defined by the markup –** the XML is generally not an integral part of the information contained in the document, but is used to enhance or augment the text data in some way instead (we'll provide examples of this throughout this section).
- **The markup data is highly unstructured –** whereas data-oriented documents are used to describe a set of data, and are rigidly structured, narrative-style documents comprise meaningful content in the form of free-flowing text, much like the content of any book or magazine article (hence the term "narrative"). The data in these kinds of documents can be thought of as unstructured in the sense that the markup in the document doesn't tend to follow any kind of rigid or sequential pattern. For instance, the number of tags and their order within the `<body>` tag of an HTML document is extremely flexible, and may differ from document to document.

Probably the most obvious examples of narrative-style (though not strictly XML) documents are HTML web pages, which use markup in the form of HTML tags to augment the text of the web page with presentation information. Although this presentation data is very important, and definitely affects the reader's experience of the page (for example, it may describe the size, color, and location of images and text on the page), it is fair to say that meaningful data exists separately from the HTML markup.

Note that this is somewhat of an oversimplification. With the advances in modern browsers and scripting languages, it is certainly possible to have pages that contain "markup-only" content.

In the same way that we use HTML markup in HTML documents to define presentation characteristics, we use XML tags in our narrative documents to do things such as annotate the text with semantic information, such as definitions of terms. Let's look at a concrete example of how a narrative document structure might be used in practice. If you imagined that a hospital used XML to represent the minutes that were taken during the course of an operation, you might end up with a document structure that looks something like this:

```
<operation>
  <preamble>
    At <time type="begin">10:04 PM</time> on <date>10/24/2002</date>,
    the procedure was begun.  The surgeons were <surgeon>Dr. Alan Law
    </surgeon> and <surgeon>Dr. Christy Smith</surgeon>, assisted by
    medical student <student>Ari Light</student>.
  </preamble>

  <minutes>
    Dr. Law used a <tool>scalpel</tool> to make an incision just below
    the <incisionPoint>left kneecap</incisionPoint>.

    ...
  </minutes>
</operation>
```

You can see from this example document that the actual useful content contained in the document is just the plain-text description of the operation. The XML markup serves to add some semantic information to the text, such as the time and date that the procedure was begun, who the operating surgeons were, and other noteworthy pieces of information.

Narrative documents are useful for a wide variety of applications, including but not limited to:

- **Presentation**

  XHTML is a good example of how you can use XML markup in a narrative-style document to affect the presentation of text-based content (XHTML is a fully XML-compliant version of HTML. You can find out more about it at the W3C's website: http://www.w3.org/TR/xhtml1).

- **Indexing**

  Applications can do efficient indexing of text-based documents by using XML markup in the document to identify key sections, and then index the documents based on that content by using a relational database or full-text indexing software.

- **Annotation**

  An application can use XML to add annotations to existing documents, allowing users to add comments to an existing piece of text without modifying the text directly, much like when a reviewer adds comments to a word document (I saw plenty of those while writing my chapters for this book!).

Let's now look at another example of how we might use XML markup in a narrative-style document to index some content. Let's say that you are a software engineer at a company that provides a news content service. Your company receives written news stories in a plain-text format from several sources, and collects them for storage as text files on a central server. The application you're responsible for building needs to provide quick access to those documents, based on queries for the key pieces of information (such as people's names, places, companies, and so on) in the documents. You realize that in order to meet the query performance requirements of the application, you need to index the documents to identify the key information. To do this you decide to encode the news articles as narrative-style XML documents and use markup to enclose the information that will be queried.

For instance, you might get a news story that looks like this:

```
Wrox Press announces new book, "XML Design Handbook"

Today, Birmingham, England-based Wrox Press announced that sales of
its new book, "XML Design Handbook," topped 1,000,000 copies.
Commissioning editor Tony Davis was quoted as saying, "We're rather
disappointed. We'd assumed it would sell at least 1.5 million copies
by the end of the first week." Nonetheless, the collaboration of
United States- and United Kingdom-based authors has become the all-
time best-selling technical book, making its authors, including
Shrewsbury, Massachusetts' own Scott Bonneau, all very rich.
```

If you had to search through this entire document (and many others like it) for the key information every time a search was executed, the performance of the application would be terrible. However, by making use of XML to markup the documents as they enter the system, you could extract the keywords and put them into a well-indexed relational database, or you could insert the XML-encoded documents themselves into a full-text indexed database, both of which would improve your application's performance.

The sample document as a narrative XML document with the appropriate content enclosed in tags, would look something like this:

```
<story id="12345">
  <organization>Wrox Press</organization> announces new book,
  "XML Design Handbook"

  Today, <city>Birmingham</city>, <country>England</country>-based
  <organization>Wrox Press</organization> announced that sales of
  its new book, "XML Design Handbook" topped 1,000,000 copies.
```

```
    Commissioning editor <person>Tony Davis</person> was quoted as
    saying, "We're rather disappointed.  We'd assumed it would sell at
    least 1.5 million copies by the end of the first week."
    Nonetheless, the collaboration of <country>United States</country>
    and <country>United Kingdom</country>-based authors has become the
    all-time best-selling technical book, making its authors, including
    <city>Shrewsbury</city>, <state>Massachusetts</state>' own
    <person>Scott Bonneau</person>, all very rich.
  </story>
```

As you can see in this version of the document, all the key pieces of information (the semantics of the document) are indicated by XML tags that identify them by their category. This example actually has its roots in a real application; a company called NetOwl has a product called the NetOwl Extractor, which automatically analyzes text documents and adds precisely this kind of markup to them. You can find out more about it at http://www.netowl.com.

Let's now look at the building blocks of XML document design.

2.2 Building Blocks: Attributes, Elements, and Character Data

XML documents have many features, but the three that affect document design at the most fundamental level are attributes, elements, and character data. As such, it's reasonable to think of these three as the fundamental building blocks of XML, and the key to designing good documents is learning when to use which block.

One of the most common decisions that XML document designers encounter is whether to use attributes or elements to encode certain pieces of data. There is no universally correct solution and, in many cases, it boils down to a stylistic issue. However, under certain circumstances (particularly with very large documents), making the correct decision is critical to ensure good performance when processing the documents.

It would be impossible to try to envision every possible type of data that one might want to encode in XML. Therefore, it's equally impossible to provide a fixed set of rules dictating when to use attributes and when to use elements to encode that data. However, to help you make these decisions, we will, in this section, compare and contrast the fundamental characteristics of each of these building blocks and then use this as a basis to discuss the appropriate use of each. Understanding these differences will help you make intelligent decisions for your own documents, based on the requirements and constraints of your application. The majority of the discussion will center on the distinctions between attributes and elements, but as character data plays an important role in XML design, we'll discuss that in more detail towards the end of the section.

2.2.1 For Narrative-Style Documents, Understand "The Rule"

As opposed to data-oriented document structures, there is a very simple rule when it comes to deciding when to use attributes and when to use elements: **narrative text should be text content, and all information *about* the text should be elements and attributes for those elements**. This is the concept that led to the development of XML and the rise of these concepts.

This makes sense if you think about the purpose of the markup in a narrative-style document, which is simply to add some semantic data to the text. Anything that adds semantic information to the text should appear in the form of an element (like the `<time>` element from our hospital example in *Section 2.1.2: Use Narrative-style Documents to Augment Plain-Text Data*); anything that describes *that* semantic value (like the `type` attribute on that `<time>` element) should appear in the form of an attribute on that element. Finally, the content whose semantic is modified by the element should then appear as text content in the context of the element.

> **The narrative text goes in element content; information about the text goes in attributes.**

2.2.2 Understand the Differentiating Characteristics of Elements and Attributes

In some cases, the decision to use an element or attribute to encode a certain piece of data is not critical. However, they have very definite differentiating characteristics and, under certain circumstances, use of one or the other can have a major impact on the performance of your application. In the following sub-sections, we'll discuss those characteristics.

2.2.2.1 Elements are More Expensive than Attributes, Both in Space and in Time

Elements tend to take more space and more time to parse than attributes do when the two are used to represent the same data. The difference in cost between using a single attribute and a single element is not great and, for most documents, the total difference in cost is negligible. However, when you're dealing with extremely large documents, or you need to keep the document size to a minimum (if, for instance, you're transmitting XML documents over a low-bandwidth pipe), this issue can come in to play. Let's look at the space and time issues separately.

2.2.2.1.1 Spatial Issues

Spatially, elements will always take up more space than attributes because the element requires at least some XML markup. Consider this example:

```
<address>
  <street>123 Main Street</street>
  <city>AnyTown</city>
  <state>TX</state>
  <country>USA</country>
</address>
```

This is a simple example of an address, which is something you might commonly see encoded in XML. If we were to define a C# class called `address`, with string members called `street`, `city`, `state`, and `country`, this is the XML that would be generated if you serialized the class using the `System.Xml.Serialization.XmlSerializer` class mentioned earlier. This particular encoding takes up 108 characters, ignoring white space. We could encode the same data using a mixture of elements and attributes, which would save us from having to have end element tags:

```
<address>
  <street v="123 Main Street"/>
  <city v="AnyTown"/>
  <state v="TX"/>
  <country v="USA"/>
</address>
```

This brings us down to 94 characters, at the cost of some readability (*I* know that that `v` stands for "value," but would anyone else?), and saves us some parsing complexity since we don't have to deal with any free text content (character data). (Once again, as noted earlier, this "mixed" style is not recommended for the reasons we'll cover shortly.)

We can represent this same data by exclusively using attributes:

```
<address street="123 Main Street" city="AnyTown" state="TX"
  country="USA"/>
```

This is minimal encoding, with 69 characters, ignoring white space (a 36% saving in terms of space). This is just a trivial example; for very large documents containing more data per record as well as more records, this savings can be critical, both in terms of the space the document takes up, and in terms of the sheer time it takes to read the extra text from the stream and parse it.

To illustrate this point, I generated a series of documents, all representing precisely the same data, using three different styles.

The first style uses attributes to represent all the data. A sample file looks like this:

```
<document>
  <element attribute0="attribute text" attribute1="attribute text"/>
  <element attribute0="attribute text" attribute1="attribute text"/>
  <element attribute0="attribute text" attribute1="attribute text"/>
</document>
```

The second style uses elements for all the data, and uses character data for the attribute values:

```
<document>
  <element>
    <attribute0>attribute text</attribute0>
    <attribute1>attribute text</attribute1>
  </element>
  <element>
    <attribute0>attribute text</attribute0>
    <attribute1>attribute text</attribute1>
  </element>
  <element>
    <attribute0>attribute text</attribute0>
    <attribute1>attribute text</attribute1>
  </element>
</document>
```

The third style is mixed; it uses elements for all the data, but instead of character data, it uses attributes *on* those elements to store the attribute values:

```
<document>
  <element>
    <attribute0 value="attribute text"/>
    <attribute1 value="attribute text"/>
  </element>
  <element>
    <attribute0 value="attribute text"/>
    <attribute1 value="attribute text"/>
  </element>
  <element>
    <attribute0 value="attribute text"/>
    <attribute1 value="attribute text"/>
  </element>
</document>
```

Using these styles, I generated two sets of three documents. The first set had 10,000 "elements" (top-level pieces of data), each of which had 10 "attributes" (the actual attribute values), and the second set had 50,000 elements each with 10 attributes. The first striking thing about these documents is their size. The following table demonstrates the size penalty of using elements:

Style	# "Elements"	# "Attributes" per "Element"	File Size
Attributes	10,000	10	2,940,025 bytes
Elements	10,000	10	4,770,025 bytes
Mixed	10,000	10	4,470,025 bytes
Attributes	50,000	10	14,700,025 bytes
Elements	50,000	10	23,850,025 bytes
Mixed	50,000	10	22,350,025 bytes

The attribute-based documents are almost 40% smaller than the element-based documents, and about 35% smaller than the mixed-style documents. In this case, the mixed-style documents only end up being about 6% smaller than element-only documents. Based on this data, it's clear that in situations where size is important you'd want to use attributes over elements wherever possible. Keep in mind that the documents don't have to approach this size all by themselves for this problem to crop up; if your application generates or consumes many smaller documents, you're going to run into precisely this same problem.

The space overhead related to elements can also extend to memory, depending on what parsing technology you use. For instance, DOM has to construct a new `Node` instance in its tree for every element that it encounters in your document. This in turn means constructing all of the members of the `Node` instance, all of which occupy memory (and take non-zero time to construct).

The overhead of creating and storing attributes in a DOM tree is much less, and furthermore, some DOM implementations optimize for attributes by not actually parsing out an element's attributes until they're accessed. This leads to savings in parsing time, and cuts down on memory overhead since the text representation of the attributes that the DOM implementation caches can take up considerably less space than that same data would in object form.

2.2.2.1.2 Parsing Time

Elements are also more expensive than attributes in terms of time because of the way that major low-level parsing technologies DOM and SAX work (we'll cover parsing and the details of DOM and SAX in *Chapter 4*). Attributes can exhibit a substantially smaller cost in optimized DOM implementations since they may not be processed until they're accessed. Moreover, you save all the overhead of the unnecessary object creation. All these things add up very quickly, and as we'll see in *Chapter 4*, they can make DOM unusable quickly as the size of the document grows to hundreds of thousands of elements. However, unless your favorite DOM implementation is extremely well documented, then short of looking at its source, there is no way to tell how optimally it handles attributes or elements unless you do some empirical testing.

With SAX, the costs of elements are a little more clear-cut. Every element in a document translates into at least two method calls: `startElement()` and `endElement()`, on your `ContentHandler` instance. In addition, if your document uses character content to represent its values (like the first example from above), there will be at least one additional call to the `characters()` method on your `ContentHandler` (for text areas that are large, or ones that contain entity references [like `<`], or even sometimes just at line breaks, SAX can break up the character data, causing multiple calls to `characters()`). Again, if your document has too many elements, the cost of making two or more extra method calls can be very substantial and can degrade the performance of your parsing code significantly. For attributes, SAX merely groups them into a data structure, aptly named `Attributes`, that is passed as an argument to a single call to `startElement()`. Thus, even to account for the creation and initialization of the `Attributes` structure itself, there is much less overhead involved.

To investigate this further, I wrote two simple programs, one using SAX and one using DOM, that simply parse any XML document, and count the number of elements and attributes that they encounter.

For the sake of completeness, let's quickly look at the code used in these tests. Here is the `main` method for the DOM-based code:

```
public static void main(String[] args) throws Exception
{
  if (args.length < 1)
  {
    System.err.println("Must specify filename.");
    return;
  }

  DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();

  long start = System.currentTimeMillis();

  Document document = builder.parse(new File(args[0]));
  int[] counts =
    countElementsAndAttributes(document.getDocumentElement());

  long end = System.currentTimeMillis();

  double totalTimeSeconds = (double)(end - start)/1000.0;
  System.out.println("Found " + counts[0] + " elements and " +
    counts[1] + " attributes in " + totalTimeSeconds + " seconds");
}
```

It's all very straightforward. It uses a DOM `DocumentBuilder` to parse the document, and then calls into another method to count the elements and attributes. At the end it catalogs the time the entire process took and outputs the count and time information. Let's look at the `countElementsAndAttributes()` method to see what it's doing:

```
// Returns an int array, with the first int being the element
// count (including the passed element itself), and the second
// being the attribute count.
private static int[] countElementsAndAttributes(Element element)
{
  int[] counts = new int[2];

  // Element count is at least 1, including this element
  counts[0] = 1;

  // Get attribute count from the element
  counts[1] = element.getAttributes().getLength();

  NodeList children = element.getChildNodes();
  for (int x = 0, xSize = children.getLength(); x < xSize; ++x)
  {
    Node child = children.item(x);
    if (child.getNodeType() == Node.ELEMENT_NODE)
    {
      int[] childCounts =
        countElementsAndAttributes((Element)child);
      counts[0] += childCounts[0];
      counts[1] += childCounts[1];
    }
  }

  return counts;
}
```

This simple recursive method counts the number of elements and attributes, for itself and all of its children. It initializes the `int` array that it's going to return with an element count of `1` (for itself) and the count of all of its attributes. It then iterates over all of its children, and looks for element nodes. For each element node that it finds, it calls itself recursively, and adds the counts returned by the recursive call to its own counts, which it finally returns at the end of the method.

The SAX code for doing this is a little more straightforward. Here's what the `main` method from our SAX version looks like:

```
public static void main(String[] args) throws Exception
{
  if (args.length < 1)
  {
    System.err.println("Must specify a filename.");
    return;
  }

  GenericParserSAX myParser = new GenericParserSAX();
  XMLReader reader = XMLReaderFactory.createXMLReader();
  reader.setContentHandler(myParser);

  long start = System.currentTimeMillis();
  reader.parse(new InputSource(new FileReader(args[0])));
  long end = System.currentTimeMillis();

  double totalTimeSeconds = (double)(end - start)/1000.0;

  System.out.println("Found " + myParser.m_elementCount +
                     " elements and " + myParser.m_attributeCount +
```

```
                          " attributes in " + totalTimeSeconds +
                          " secoonds");
    }
```

This method instantiates an instance of our `GenericParserSAX` class, which extends SAX's `DefaultHandler` class, and then instantiates an `XMLReader` for the actual parsing. It sets our class's instance (`myParser`) as the content handler for the reader, and then tells the reader to parse. The parser code itself, as we'll see in a minute, actually does the counting. Finally, this method outputs the time the process took as well as the counts.

> *`DocumentHandler` implements `ContentHandler`, as well as `ErrorHandler`, `DTDHandler`, and `EntityResolver`. I used `DocumentHandler` here because it has no-op implementers of all of the `ContentHandler` methods (as well as those for the other interfaces as well). Since this example only uses a few of the `ContentHandler` methods, extending `DefaultHandler` rather than implementing `ContentHandler` directly means that we don't have to write our own implementers for the `ContentHandler` methods that the example doesn't use.*

Our class only implements three `ContentHandler` methods, `startElement()`, `endElement()`, and `characters()`, all of which are overridden from `DefaultHandler`. To simulate a real parsing application, this code maintains a stack of `StringBuffer` instances (one per element that it encounters). The `startElement()` method pushes a new one on the stack, `characters()` appends to it, and `endElement()` pops it off the stack and calls `toString()` on it.

Calling `toString()` on a `StringBuffer` is almost a no-op; no copying of the internal `char[]` is done. The `StringBuffer`'s internal `char[]` is used to construct a `String` instance, which is then returned to you. However, it forces any further manipulation of the `StringBuffer` to be done on a new copy of the `char[]`, since otherwise the immutability of the `String` instance would be violated.

Here is what the code looks like:

```
    private int m_elementCount = 0;
    private int m_attributeCount = 0;
    private Stack m_buffers = new Stack();

    public void startElement(String uri, String localName,
                             String qName, Attributes attributes)
      throws SAXException
    {
      m_elementCount++;
      m_attributeCount += attributes.getLength();
      m_buffers.push(new StringBuffer(32));
    }
    public void endElement(String uri, String localName, String qName)
      throws SAXException
```

```
    {
      StringBuffer buffer = (StringBuffer)m_buffers.pop();

      buffer.toString();
    }

    public void characters(char ch[], int start, int length)
      throws SAXException
    {
      if (!m_buffers.isEmpty())
      {
        StringBuffer buffer = (StringBuffer) m_buffers.peek();
        buffer.append(ch, start, length);
      }
    }
```

We ran each of the six documents listed in the previous table through both of these programs. All of the programs were run on an AMD 1.67GHz machine with 512MB of RAM, using Sun's JDK 1.4.0_01 for Windows and using a Xerces SAX2 driver and DOM `DocumentBuilderFactory` implementation. Here are the results:

Parser Type	# Elements / # Attributes	Attribute Style	Element Style	Mixed Style
SAX	10,000/10	0.828 seconds	0.906 seconds	1.375 seconds
DOM	10,000/10	1.063 seconds	2.297 seconds	2.516 seconds
SAX	50,000/10	2.468 seconds	3.062 seconds	3.859 seconds
DOM	50,000/10	6.703 seconds	9.765 seconds	13.844 seconds

Let's consider the first two rows in this table first. You can see that the attribute style ends up being slightly more efficient than the pure element style, which is what we'd expect given what we know about SAX. The mixed style performs the worst of the three, incurring the overhead of both the added elements and the attributes.

As for DOM, as we anticipated, using attributes ends up yielding a huge savings, by more than a factor of two. The mixed style ends up being the worst of both worlds here as well, and takes slightly more time than the pure element-based approach.

Moving on to the final two rows in the table, you'll notice a couple of interesting pieces of data. First, for SAX you'll notice that the ratios between the styles stay roughly the same, but it's clear that the mixed style is an under-performer, taking nearly twice as long as a pure attribute-based approach. By this point, it should be clear that the mixed style is not very appropriate if performance is an issue.

In the DOM row, you can see that the pure attribute-based approach yields a savings of nearly 45% over the pure element-based approach, and is more than twice as fast as the mixed approach. You'll also notice that the DOM solution is more then three times as expensive in terms of time than the corresponding SAX solution. This example illustrates the fact that DOM does not deal well with documents with exceedingly high element counts.

We have hard data indicating the relative size and time costs between elements and attributes. You might argue that the results of this test are dependent on the parser implementation used during the test, and to some extent, this is always going to be true, regardless of the context. However, the tests here are broadly applicable because they test the structure of the APIs much more than the characteristics of the implementation. Therefore, the behaviors elicited by the tests are fundamental and characteristic of the *interface* more than the underlying *implementation*. Clearly, a poorly implemented version of the API is going to behave differently than an efficient implementation. However, that doesn't change the fact that while, for instance, there is a need to make a call to both the `startElement()` and `endElement()` functions for each element in the document, this need does not exist for any attribute in the document.

To quell any potential conspiracy theories, I ran these tests with three different SAX drivers (the Apache Crimson driver that ships with JDK 1.4, the Xerces driver, and a SAX2 driver based on an XML pull parser, which we'll discuss more thoroughly in *Chapter 4*), all of which yielded similar results. The complete results are available in *Appendix A*, and you can run these tests yourself with your favorite SAX driver by downloading the example source from http://www.wrox.com.

2.2.2.2 Elements are more Flexible than Attributes

Attributes are extremely limited in terms of the data that they can represent; an attribute can only represent one string value. Attributes are not suited to holding any kind of structured data. They are intended to hold short strings. Elements, on the other hand, are obviously very well suited to representing structured data, since they can contain other nested elements as well as character data.

You may store structured data in an attribute, but then you will also be responsible for writing all of the code to parse that string value out. In some cases, this may be acceptable; for instance, it's reasonable to store a date as an attribute. Your parsing code is likely to parse out the string containing the date, into an in-memory date instance for later use. This is actually quite a prudent approach; you get the benefits of using a single attribute as opposed to several elements to represent your date value, plus the time it takes to parse out a date string is minimal, so you save the additional XML processing time as well. (Note that this approach works with dates because they have a very common set of formats when represented as a string and so they don't require a lot of *a priori* knowledge in order to parse. However, overusing this technique destroys the usefulness of the XML representation, so be careful not to overdo it.)

Note that the use of XML schemas, which we'll discuss in the next chapter, allows for some limited structuring of attribute data, such as representing sequences of decimal numbers. Beyond simple cases like this, however, attributes should be avoided when representing structured data.

It is generally unwise to try to get too creative with encoding structured data into a single string. The most extreme case I've ever seen of this had an entire additional XML-escaped (for instance "`>`" instead of ">", etc.) XML document as the text content of an attribute. At parsing time, that attribute was read into a string (the parser un-escaped the string during parsing), and then that string value, which was at that point a complete, well-formed XML document, was itself sent to another parser to be parsed.

This is not the best use of XML, nor is it a recommended practice, but it clearly demonstrates the lack of flexibility that attributes provide. If you are using attributes in this manner, you're most likely underutilizing the flexibility that XML provides. Attributes are great at storing relatively short and unstructured strings, and that is what they should be used for the majority of the time. If you find yourself still needing to store some relatively complicated structured text in your document, you should consider representing it as character data instead for a very good reason: performance.

In addition to the fact that very long strings in an attribute value are stylistically undesirable, they can also pose a performance problem. Since attribute values are just string instances, the parser must hold the entire value in memory at once. For very long strings, memory can become an issue if your attribute value is excessively large. In SAX, very large character data is broken up into several chunks that are passed individually to the `characters()` method of the `ContentHandler`. Thus, the entire string doesn't need to be in memory all at once (this point is debatable in DOM, since the entire document is represented in memory all the time).

2.2.2.3 Character Data versus Attributes

This is very much a stylistic issue, but there are some guidelines to consider when making a decision on this, so we'll cover them briefly here.

Consider using character data when:

- **The data is very long**

  Attributes were not meant to store very long values, as they require the entire string to be in memory at once. From an implementation standpoint, character data can be broken up into smaller chunks that can be processed independently.

- **There are a large number of XML-escaped characters in the data**

  If you encode the data using character data, you can use a CDATA section to avoid having to XML-escape the string. For instance, using an attribute to represent a boolean expression you might end up with something looking like this: `"a < b && b > c"`. However, if you used character data, you could encode the same string in a much more readable fashion: `<![CDATA[a < b && b > c]]>`.

- **The data is short but performance is an issue and you're using SAX**

  As we saw in *Section 2.2.2.1.2: Parsing Time* with the mixed-style documents, character data can be considerably less expensive than attributes in terms of parsing time when using SAX.

Consider using attributes when:

- **The data is short and unstructured**

  This is what attributes were designed for, but be aware that performance can suffer versus character data for extremely large documents (> 100,000 elements) when using SAX.

- **You're using SAX to parse it, and want to keep the parsing code simple**

  It's much simpler to parse out attribute data than character data, since attributes are available when you're parsing the start of an element's context. Parsing out character data is not too complicated, but it does require a few extra steps and some extra logic that can be error-prone, particularly for SAX beginners.

2.2.3 Favor Attributes for Data that Identifies the Element

In many cases, data objects have a property, or a set of properties, that is used to uniquely identify that piece of data from others of the same type. For instance, an ISBN number or the combination of the author and title may be used to uniquely identify a "book" object:

```
 public class Book
{
  private String m_isbn;
  private String m_title;
  private String m_author;
  private int m_numPages;
  private String m_description;
  private String m_genre;
  private String m_format;

  public Book(String isbn)
  {
    m_isbn = isbn;
  }

  public Book(String title, String author)
  {
    m_title = title;
    m_author = author;
  }

  // Getters & Setters omitted
  // ...
}
```

This is a simple class describing a book, including a few relative properties and a couple of constructors; all of the getter and setter methods have been omitted. There is no default constructor, which is not uncommon, particularly for data classes that have some immutable members. In this case, the constructors require that you either have an ISBN number or the title and author.

Let's say you want to encode the data in this class as XML. For classes or structures like this, it usually makes sense to model that data as an attribute of the element containing that object's data for a variety of reasons; most importantly, the impact that this can have on the parsing process.

The use of attributes rather than elements can simplify the parsing process under a few different circumstances. When there is no default constructor for the object that you're deserializing, having the key data as attributes makes the construction of that object less complicated. This is so because you can construct the object in the code that handles the start of the element that contains the data describing the object. If you were to write the parsing code to handle the construction of our book object by using SAX, it would look something like this:

```
// Used to hold the current book instance during parsing.
private Book m_currentBook = null;

public void startElement(String uri, String localName,
                         String qName, Attributes attributes)
  throws SAXException
{
  if ("book".equals(qName))
  {
    // Get the relevant key values from the attribute list
    String isbn = attributes.getValue("ISBN");
    String author = attributes.getValue("author");
    String title = attributes.getValue("title");

    // Switch to see what key data exists...
    if (isbn != null)
    {
      // Construct the book based on the ISBN.
      m_currentBook = new Book(isbn);
    }
    else
    {
      // Make sure that there are both an author and a title before
      // constructing the book.  If not, throw an exception.
      if (author != null && title != null)
        m_currentBook = new Book(title, author);
      else
        throw new RuntimeException("Book requires either ISBN " +
                                   "or both author and title.");
    }
  }
}
```

This is clean parsing code; it guarantees that whenever we are within the `<book>` tag's context, there will be a valid `Book` instance to be operated on. This example is slightly more complex than you might normally see, as there are two possible exclusive keys (ISBN or author and title), but it's definitely within the realm of situations that you may encounter.

If you decide to use elements instead of attributes here, writing clean, robust code gets more difficult:

```
    // Used to hold the current book instance during parsing.
    private Book m_currentBook = null;

    // Temporary variable used to store the name of the author.
    private String m_author = null;

    public void startElement(String uri, String localName,
                             String qName, Attributes attributes)
      throws SAXException
    {
      if ("book".equals(qName))
      {
        // Do nothing, since we can't construct a new Book without the
        // key information.
        return;
      }

      if ("isbn".equals(qName))
      {
        // We'll assume that an XML schema or DTD enforces that if there
        // is an ISBN element that it comes before the author and title.
        // We'll also assume that the value of the ISBN is stored in
        // an attribute called "value" to simplify parsing.
        m_currentBook = new Book(attributes.getValue("value"));
        return;
      }

      if ("author".equals(qName))
      {
        // We'll assume that the value for the author is in an attribute
        // called "value" to simplify parsing.
        String author = attributes.getValue("value");
        if (m_currentBook != null)
          m_currentBook.setAuthor(author);
        else
          m_author = author;
        return;
      }

      if ("title".equals(qName))
      {
        // We'll assume that the value for the title is in an attribute
        // called "value" to simplify parsing.
        String title = attributes.getValue("title");
        if (m_currentBook != null)
          m_currentBook = new Book(title, m_author);
        else
          m_currentBook.setTitle(title);
        return;
      }
    }
```

This parsing code is more complex as it constantly needs to react differently, based on whether or not the `m_currentBook` member variable has been initialized. In contrast to the previous example, the parsing state is not clear-cut when dealing with the elements within the `<book>` context. This can lead to parsing code that is dirtier and harder to maintain, which brings with it innocuous parsing bugs that can be difficult to track down.

Another case where having the identifying information included with the enclosing element as attributes can simplify the parsing process is while validating the contents in a document. Occasionally, you might want to do a quick, cursory check over a document to validate its contents, before doing a more in-depth and extensive processing of the document. This is particularly useful for tiered or distributed systems; by doing some validation up front, you can eliminate unnecessary traffic to a lower tier or across a network boundary (such as an order entry system validating the format of a credit card number or an account number, before submitting an order to an order fulfillment system).

In these situations, it helps tremendously to have the identifying or key information as an attribute rather than an element. The parsing code is greatly simplified, as we saw above, and the performance is much better, since you can completely ignore the vast majority of the content in a document and often terminate the parsing process early.

Ultimately, the decision of attributes over elements in this situation is largely stylistic since you can achieve the same result using either approach. However, there *are* some situations where there are clear advantages of using attributes for identifying data; so, when designing your document structure, make sure to consider whether or not your document falls into those categories.

2.2.4 Avoid Attributes for Data where Order is Important

Unlike elements, there is no enforceable ordering for attributes; no matter how you specify attributes in a schema or DTD, there is no requirement that the attributes as they appear in a given document will actually conform to any specific order. As a quick example, look at the following DTD (we'll look at XML Schemas in *Chapter 3*, but we'll use this simple DTD for the sake of this example):

```
<!DOCTYPE doc [
  <!ELEMENT anElement (#PCDATA)>
    <!ATTLIST anElement anAttribute CDATA #REQUIRED>
    <!ATTLIST anElement anotherAttribute CDATA #REQUIRED>
]>
```

The DTD specifies a simple document with a single element, `anElement`, which has two required attributes, `anAttribute` and `anotherAttribute`. Consider the following two documents based on this DTD:

```
<!DOCTYPE doc [
  <!ELEMENT anElement (#PCDATA)>
    <!ATTLIST anElement anAttribute CDATA #REQUIRED>
    <!ATTLIST anElement anotherAttribute CDATA #REQUIRED>
]>

<doc>
  <anElement anAttribute="foo" anotherAttribute="bar"/>
</doc>
<!DOCTYPE doc [
  <!ELEMENT anElement (#PCDATA)>
    <!ATTLIST anElement anAttribute CDATA #REQUIRED>
    <!ATTLIST anElement anotherAttribute CDATA #REQUIRED>
]>

<doc>
  <anElement anotherAttribute="bar" anAttribute="foo"/>
</doc>
```

The only difference between these two attributes is the ordering of the attributes; in the first document, the `anAttribute` attribute is specified before `anotherAttribute`, and in the second document, the ordering is reversed. Since XML doesn't enforce any kind of attribute ordering, both of these documents are legal and conform to the DTD.

As a result, in situations where the ordering of the values is important, elements are better suited than attributes. In keeping with this example, one could easily specify a DTD where one element needs to appear before another:

```
<!DOCTYPE doc [
  <!ELEMENT anElement (aSubElement, anotherSubElement)>
  <!ELEMENT aSubElement (#PCDATA)>
  <!ELEMENT anotherSubElement (#PCDATA)>
]>
```

Using this DTD, we produced the following two documents:

```
<!DOCTYPE doc [
  <!ELEMENT anElement (aSubElement, anotherSubElement)>
  <!ELEMENT aSubElement (#PCDATA)>
  <!ELEMENT anotherSubElement (#PCDATA)>
]>

<doc>
  <anElement>
    <aSubElement>Sub Element #1</aSubElement>
    <anotherSubElement>Sub Element #2</anotherSubElement>
  </anElement>
</doc>
```

```
<!DOCTYPE doc [
  <!ELEMENT anElement (aSubElement, anotherSubElement)>
  <!ELEMENT aSubElement (#PCDATA)>
  <!ELEMENT anotherSubElement (#PCDATA)>
]>
```

```
<doc>
  <anElement>
    <anotherSubElement>Sub Element #2</anotherSubElement>
    <aSubElement>Sub Element #1</aSubElement>
  </anElement>
</doc>
```

As before, these two documents are identical except for the ordering of the sub-elements. In this case, however, the second document (where `anotherSubElement` comes before `aSubElement`) does not conform to the DTD. If you were to run these documents through a validating SAX parser, the first document would parse correctly, but the second one would cause the parser to throw an exception indicating that it doesn't conform to the specified DTD.

That wraps up our discussion of attributes, elements, and character data. In the next section, we'll cover some guidelines for modeling data in your XML documents.

2.3 General Data Modeling Pitfalls

The manner is which you model your data in XML dictates just about every aspect of how people and applications are going to interact with those documents. Thus, it is critical that the data be modeled to make that interaction as straightforward, efficient, and painless as possible. Thus far, we've covered modeling at a very low level, discussing which XML primitives (such as attributes and elements) are more appropriate in which situations. For the rest of the chapter, we'll discuss some guidelines that apply at a higher level.

2.3.1 Avoid Designing for a Specific Platform or Parser Implementation

A large part of the appeal of XML is that it is portable and platform-independent, which makes it a great tool for sharing data in a heterogeneous environment. When documents are used to share data and communicate externally to an application, the design of the document is extremely critical, and as a designer, you must take into account a whole host of issues relating to the fact that there will be third-party consumers of your documents. For one, the cost of making a change to the data structure once it's published and people have written code based on that structure is extremely high, potentially to the point of being prohibitive. In addition, once you expose a document structure to the outside world, you effectively lose all control over *how* documents conforming to that structure will be processed, which can be a major problem if you haven't considered that case at design time.

> *Note that XML certainly has found its way into places where sharing data isn't a concern, such as in situations where XML is used to represent some internal state of a closed system; (for instance, when it is used to store a configuration state that isn't accessible to end users). However, in situations like these, document design issues are usually far less critical because the cost of making a change to the document structure in the event that an issue is discovered is very low. The changes are localized to the given application and don't affect the outside world.*

I would go as far as saying that it is never a good idea to design a document structure to match a particular parser implementation. This is not like designing a document structure that's appropriate for a given parsing technology (like DOM, SAX, or pull-parsers) – it's more like suggesting that you write your Java code tailored to a specific VM; it rather defeats the purpose. Parser implementations will evolve and only get better, so it doesn't make too much sense to optimize for one particular implementation. Since the nature of XML is that it is platform-parser-technology-agnostic, it's impossible to control precisely the performance characteristics of parsing a given document when you're no longer in charge of deciding how it will be processed.

As a result, the best thing that you, as a document designer, can do is to tailor your document to suit the task it's trying to serve. A document structure is a contract, and nothing more – it's an interface, not an implementation. It is therefore extremely important to have an understanding of the usage of documents conforming to *that* structure. For example, if you were building an API in your favorite programming language, you wouldn't design an interface for that API without understanding how that interface was going to be used. If you did, the result would most likely be an unusable API that doesn't suit the needs of its would-be users. This fact is as true for XML document structures as it is for programming interfaces.

To crystallize this point, think about the different design you might have for a bulk- vs. single-row-oriented database interface. If your API will be affecting hundreds or thousands of rows in a database, you would likely design it to use arrays or collections of objects as parameters, because a single database query that affects a thousand rows is significantly faster than a thousand queries that each affect a single row. You don't need to know the innards of Oracle or SQL Server to make an intelligent design decision here. Similarly, when designing document structures, it's much more important to account for the broad truisms of the technologies today, than it is to worry about the specific issues of a given implementation. To tie this point to what we discussed earlier in the chapter, understanding, for instance, the general performance characteristics of attributes versus elements, is more important than knowing if a particular parser implementation has an optimized (or sub-optimal) engine for processing large numbers of attributes.

As with anything in engineering, the designer must use some common sense to achieve a balance between the technical ideal and the current state of reality. This is why we spent so much time in the chapter discussing issues like "elements are more expensive than attributes," because it's true regardless of technology, and it can affect the document design in the context of the intended usage of the document. To continue the example from above, if you're designing a document structure that you know is going to be used to hold thousands of records, you would be well served to understand that the use of elements can cause a significant performance penalty in terms of disk space and in terms of time when those documents are parsed.

With the advent of **Simple Object Access Protocol (SOAP)**, XML-based web services, and other XML-based communication mechanisms, the need to design document structures in a platform-agnostic manner is clearer than ever. So, when your documents are not exclusively used internally to your application, be sure to design your document structures in such a way that they don't depend heavily on any kind of platform-specific or parser implementation-specific features or performance characteristics.

2.3.2 The Underlying Data Model often isn't the Best Choice as an XML Encoding

XML is often used in situations when it's not the persistent representation for the encoded data; for instance, XML-based RPC is a cornerstone of all of the new Web Service technology that's being hyped so heavily by Microsoft and the Java community. In the case of XML-based RPC, the parameters to the remote procedure, as well as its return value (if any), are encoded in XML inside a SOAP-compliant document. The XML used to consummate the remote procedure call is clearly not the primary representation of that data; it is simply a secondary representation used for marshalling data across the socket.

In situations like this where XML is used as a transient encoding of the data, it is often a good idea to rethink the structure of your data rather than just reusing the on-disk structure as your XML-based structure. Often, the underlying data model isn't the most optimal one for XML.

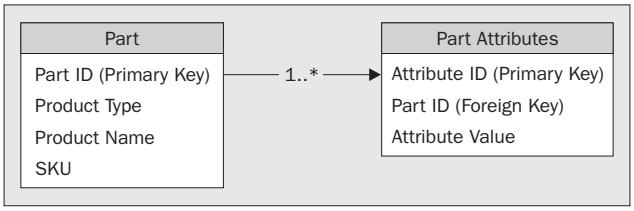
The rationale behind this is straightforward; most persistent storage mechanisms, such as **Relational Database Management Systems (RDBMS)**, flat files, object databases, registries, etc., were not designed with XML and its structure in mind. Each was designed with specific purposes in mind, and has strengths and weaknesses directly in line with those purposes. Many XML-based databases were designed with this problem in mind. SQL Server 2000 from Microsoft also has been designed to support XML internally.

One of the more common uses of XML is to encode data whose persistent representation is as relational data in an RDBMS. Relational databases have a flat structure that consists of discrete tables whose only relationship to one another is a logical one; namely, the relationship described by keys between tables. Using relational databases to describe hierarchical data usually means having a table of data that represents the "parent" in the hierarchical relationship, and a separate table (or tables) that represent(s) the "children" in the relationship, with the child tables having a key that links to their parent. (This is a simplification, but it doesn't omit any details pertinent to this discussion.)

If you were to mimic this table structure in an XML document, the result would be a highly inefficient document structure that adds unnecessary complications and overhead to the parsing process because it ignores one of XML's more powerful features: the ability to represent hierarchical data. We'll cover this situation in-depth in *Section 2.3.4: Avoid Pointer-heavy Documents*, but let's look at a quick example to illustrate the point.

Let's say that you were building an application that implemented an online catalog of parts, and you used a relational database to store the catalog data. Parts in the catalog have a variable number of attributes, so you build a table structure that looks like this:

Figure 1



This is a sensible table structure for this data, as its separation of part data from part attribute data allows the flexibility that is required in terms of having a variable number of attributes for each part in the catalog.

If you were to take this table structure and map it directly to XML, you'd end up with a document structure that looks something like this:

```
<catalog>
  <part id="12345" sku="AZDF1023" productType="Widget" name="Gizmo"/>
  <part id="54321" sku="LKJS7892" productType="Widget" name="Foobar"/>
  <part id="98765" sku="RJSOI22" productType="Widget" name="Wizbang"/>

  <partAttribute partID="12345" attribute="Color" value="Blue"/>
  <partAttribute partID="12345" attribute="RAM" value="128MB"/>
  <partAttribute partID="12345" attribute="HDD" value="2GB"/>

  <partAttribute partID="54321" attributeID="Color" value="Red"/>
  <partAttribute partID="54321" attributeID="RAM" value="64MB"/>
```

```
    <partAttribute partID="98765" attributeID="Color" value="White"/>
    <partAttribute partID="98765" attributeID="RAM" value="512MB"/>
    <partAttribute partID="98765" attributeID="OS" value="Linux"/>
    <partAttribute partID="98765" attributeID="Office" value="Star"/>
  </catalog>
```

In this document, the catalog consists of a list of parts (mimicking the part rows in the table definition), followed by a list of part attributes (mimicking the part attribute rows). While this is a perfectly valid document structure, it's probably not the best structure for this data, mostly because it completely ignores the hierarchical nature of the data and XML's ability to represent that data hierarchically.

In this example, the hierarchy is enforced logically, using references (or pointers); each `partAttribute` element has a `partID` attribute that acts as a pointer into a `part` element somewhere else in the document (we'll discuss pointer-heavy document structures in-depth in *Section 2.3.4: Avoid Pointer-heavy Documents*). This certainly mimics the way that relational databases represent this data, but it doesn't take advantage of XML's strengths.

A much more natural XML structure for this type of document would be to nest the part attribute data as children of the part. It would look like this:

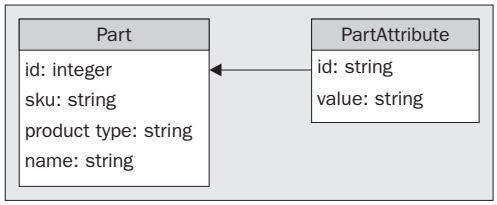
```
  <catalog>
    <part id="12345" sku="AZDF1023" productType="Widget" name="Gizmo">
      <partAttribute attributeID="Color" value="Blue"/>
      <partAttribute attributeID="RAM" value="128MB"/>
      <partAttribute attributeID="HDD" value="2GB"/>
    </part>

    <part id="54321" sku="LKJS7892" productType="Widget" name="Foobar">
      <partAttribute attributeID="Color" value="Red"/>
      <partAttribute attributeID="RAM" value="64MB"/>
    </part>

    <part id="98765" sku="RJSOI22" productType="Widget" name="Wizbang">
      <partAttribute attributeID="Color" value="White"/>
      <partAttribute attributeID="RAM" value="512MB"/>
      <partAttribute attributeID="OS" value="Linux"/>
      <partAttribute attributeID="Office" value="Star"/>
    </part>
  </catalog>
```

This document reads much more easily (both to humans and parsers) than the previous one because this one uses XML's inherent hierarchical nature to represent the data in a more natural way. Each `partAttribute` element is contained within the context of its owning part, which makes it obvious which part owns which attributes, and also eliminates the need for the `partID` attribute (which you can see has been omitted in this version of the document) on the `partAttribute` elements. Structuring the document in this way also makes the parsing code quite a bit simpler and more efficient for larger documents, the reasons for which we'll discuss in *Section 2.3.4: Avoid Pointer-heavy Documents*.

The key concept to keep in mind as you're designing your document structure is that it should be based on the abstract object model that best describes what you are trying to encode, not on the implementation of that model in some other technology, like the Java (or C++, etc.) class that implements the model in your application. I'm not saying that there aren't situations in which your software implementation of the model and your XML implementation won't be very similar, because for many cases, such as simple object serialization, they will be. The important thing to realize is that your XML should be a model of your data, not a model of the Java model of your data.

When designing the catalog example that we just walked through, the first thing that we did was look at the table structure that was being used in the database to store the data. The result of using this relational description of the data as a basis for the document was not a particularly good one, as we just saw. However, if we were to consider a more abstract description of the model (rather than the implementation – in this case the relational database structure), we would have arrived at the better solution immediately. To see an example of this, let's look at a UML diagram representing the catalog structure:

Figure 2



This simple UML diagram provides a basic abstract description of the relationship between parts and their attributes: namely, a part consists of an identifier, a SKU, a product type, and a name, and is associated with some number of attributes, which consist of an identifier and a value. The diagram implies that part attributes are associated with their parts in a container-containee relationship, which is precisely the structure modeled in the second document from the example.

By considering the data you're trying to model outside the context of any particular implementation, you can often arrive at a better document structure for your data than you would otherwise. This kind of analysis will usually help you model your data in a more XML-friendly structure, which will lead to more robust documents that will lend themselves to simpler parsing code, and will perform better when it comes time to parse them.

2.3.3 Avoid Over-stuffing your Documents

One of the side effects of using XML to encode data is the overhead that goes hand-in-hand with that encoding. To put it bluntly, XML is not the most efficient encoding mechanism from a signal-to-noise ratio standpoint. For every piece of data in your document, some corresponding markup takes up some space for both itself and the data it represents.

In general, it makes sense to try to keep the size of your documents as small as possible, for several reasons:

- Even though disk space is cheap, it never makes sense to waste it
- Every single bit in that file will run through an XML parser at some point; the more the data in the document, the longer it will take to parse
- If the document is going to be used as a communications mechanism, like a SOAP document for an XML-based Web Service, it's going to all have to be sent on the wire

Now, I'm not suggesting that you should take this to the extreme and end up producing a document that looks like this:

```
<a>
  <b c="Some variable">
    <d>Some text</d>
  </b>
  <e f="Some other variable/>
</a>
```

While long element and attribute names can play a part in very large documents (100K+ elements), it's also usually very valuable to make sure your documents are easily readable. As a result, while you don't necessarily want excessively long element and attribute names, it's typically not the place to cut back when you're trying to keep your documents to a reasonable size. (Keep in mind that these names will compress easily if you choose to compress your documents as well.)

It usually makes more sense to analyze the actual data that you're putting into your documents, for any unnecessary bits. There is a tendency when encoding data in XML to just "throw in the kitchen sink," so to speak.

In general, try to omit redundant data when building your document structure. A specific situation when this can crop up is when serializing classes that have attributes that are functions of other attributes. An extreme example of this is the `TimeSpan` class in C#. This class has a slew of public properties that allow you to access the underlying time span value, which itself is probably just an integral number representing the number of nanoseconds (or just about any other unit of time that you can think of). It has properties to represent the span in *total* years, days, etc., as well as properties that represent the *relative* number of years, days, and so forth, covered by the span. For example, for a time span of 62 total seconds, there would be just 2 relative seconds, because that is the remainder after calculating the number of minutes.

If you were to serialize this class, it would be better to just serialize the single underlying integral value rather than all the properties, since each of those other properties can be derived from *that* single underlying value. This is an extreme example, but it's not uncommon to see, say, a class that does event scheduling and calculates a next scheduled execution time, based on a previous execution time and the current time. There is no reason to serialize that value, since it can easily be recreated when the object is reconstructed.

Any field that you would mark as `transient` for regular Java serialization should probably not be serialized to XML either (for those of you who aren't familiar, fields marked with the keyword `transient` are ignored by the Java serialization code). Fields that are only used to store temporary state, such as caches or intermediate state used during long-running calculations, are good candidates for transient variables.

There will also be situations where it makes sense *not* to encode even non-transient state. For instance, suppose you're writing some code that keeps track of the differences between distributed data stores to keep those stores synchronized, and you're using XML to pass the data between the stores. If the records in your data store have many fields, of which only a few are likely to have changed since the last time the stores were synchronized, it would make a lot of sense to only encode into your XML the fields that *have* changed. For records with hundreds of fields and stores with thousands of records, this can add up to substantial savings very quickly.

It's important not to overdo it when trying to trim the amount of data in your documents; make sure that you always have the right set of data to perform the necessary function. If you look at the data in your document and realize that some of it isn't used right now, but it may be in the near future, you may consider keeping it in because the cost of making the change later on might be much greater than the cost of having some extra data floating around in the meantime. It's always a balancing act between the performance *now* and flexibility in the *future*, and you have to make decisions that make sense in the context of your own applications and its requirements. By paying a little more attention to what data is actually in your documents and comparing it against what is needed, you can end up keeping your documents smaller, making them less expensive and easier to parse.

2.3.4 Avoid Pointer-heavy Document Structures

One of the biggest pains when you write XML parsing code is realizing that the document structure that you're parsing makes heavy use of references (also referred to as pointers). For example, look at this small document:

```
<document>
  <anElement name="aParent"/>
  <anElement name="aChild" parent="aParent"/>
</document>
```

This document demonstrates a very simple use of pointers in XML. The second `anElement` element, `aChild`, has an attribute, `parent`, which references another element (in this case, the first `anElement` element, `aParent`). (Note that the pointers are simply text-based, and there are only very limited language features to support them; the resolution and management of references is normally left up to the application code.)

Pointers in XML documents can be useful in certain circumstances. For example, the Ant build system uses XML to describe the build process, and in an Ant build file, you use pointers to specify the dependencies for your target. Here's a quick example:

```
<project name="myproject" default="all" basedir=".">
  <target name="all" depends="clean,build,package"/>

  <target name="clean">
    <!-- Do the cleaning -->
  </target>

  <target name="build">
    <!-- Do the building -->
  </target>

  <target name="package">
    <!-- Do the packaging -->
  </target>
</project>
```

Here, pointers have been used in a couple of places. First, the `project` element has an attribute called `default`, which points to a target to be used when no target is specified. Second, in the target "all," the attribute `depends` points to a list of all of the other targets that must be processed before this target can be evaluated.

Pointer-heavy documents can make the parsing process quite complicated, and depending on the size of the document and how it's structured, dealing with pointers can erode the performance of the parsing code, and make it very difficult for people to follow what the document is trying to represent to a degree that would make them untenable.

The problem with parsing pointer-heavy documents is that you have to do the pointer resolution yourself. (This is at least true when parsing documents using DOM or SAX. XSLT, which sits at a higher level than DOM and SAX, has more robust mechanisms for dealing with pointer resolution.) We'll cover the topic of building efficient pointer-resolving parsing code in *Chapter 4*. Strictly speaking, pointer resolution isn't an enormously difficult technical problem; it's generally rather easy to write the code, particularly when the relationship is one-to-one or one-to-many.

To do resolution, you need to keep a certain amount of state in memory during the process; something you ordinarily wouldn't have to do. This is no big deal if the document is small, but as the size of the document grows, the amount of state that you need to keep in memory usually grows as well, and at some point that can become a problem. Note that if you're using DOM to parse the document, then this is rather a moot point, since DOM requires that the entire document be in memory anyway.

Let's look at an example. Say you're parsing a document that makes heavy use of pointers to describe a simple, non-recursive parent-child relationship. In the document, all of the parents will come first, followed by all of the children. A typical document would look like this:

```
<document>
  <parent name="parent 1">
    <!-- Some parent stuff -->
  </parent>
  <parent name="parent 2">
    <!-- Some parent stuff -->
  </parent>
  <!-- ... -->
  <parent name="parent N">
    <!-- Some parent stuff -->
  </parent>

  <!-- END parents -->
  <!-- BEGIN children -->

  <!-- "parent 1"'s children -->
  <child name="child 1 of parent 1" parent="parent 1">
    <!-- some child stuff -->
  </child>
  <child name="child 2 of parent 1" parent="parent 1">
    <!-- some child stuff -->
  </child>
  <child name="child 3 of parent 1" parent="parent 1">
    <!-- some child stuff -->
  </child>

  <!-- "parent 2"'s children -->
  <child name="child 1 of parent 2" parent="parent 2">
    <!-- some child stuff -->
  </child>
  <child name="child 2 of parent 2" parent="parent 2">
    <!-- some child stuff -->
  </child>
  <child name="child 3 of parent 2" parent="parent 2">
    <!-- some child stuff -->
  </child>

  <!-- ... -->

  <!-- "parent N"'s children -->
  <child name="child 1 of parent N" parent="parent N">
    <!-- some child stuff -->
  </child>
  <child name="child 2 of parent N" parent="parent N">
    <!-- some child stuff -->
  </child>
  <child name="child 3 of parent N" parent="parent N">
    <!-- some child stuff -->
  </child>
</document>
```

To successfully process this document, you'll need to keep some information about the parents around for processing the children elements. For example, you'll have to verify that each child has a reference to a valid parent (this is called checking for referential integrity). You may also have to access properties of the parent in order to process the children.

If the number of parents is relatively small, the amount of extra space in memory taken up by the parent data probably won't cause your system to start thrashing. It will make the SAX parsing code more complicated, though, since you'll have to write some code to manage that data.

However, a relatively large number of parents could affect the performance of your parsing code, because memory usage will increase linearly with the number of parent elements in the document. In addition, any collections that are used in the parsing code will have to be sized appropriately, otherwise they'll have to be constantly expanded (like vectors or array lists) and potentially rehashed (like hash tables and hash sets), which both takes up memory and a lot of processing time. Inappropriately sized collections can cause a serious performance bottleneck.

The problem with pointers can be aggravated if the referencing elements come before the referenced elements. For example, if you re-wrote our document such that all of the children came before any of the parents, the amount of data that you'd need to keep in memory would potentially be much greater. There are likely to be at least as many, if not more, children as parents. To process the document successfully in this case, you'd need to keep all of that child data in memory until you found and processed the parent for that child. When you find a parent, you'd want to process all of its children, which further complicates the code necessary to complete the parsing process.

You can avoid most of these issues with pointers in situations where your document models one-to-one or one-to-many relationships by using the hierarchical nature of XML. The parsing code and memory overhead of processing hierarchical data is a great deal less than when using pointers. For example, we could rewrite our example parent-child document using a hierarchical structure, which would also make the document a lot cleaner and easier to read:

```
<document>
  <parent name="parent 1">
    <!-- Some parent stuff -->

    <child name="child 1 of parent 1">
      <!-- some child stuff -->
    </child>
    <child name="child 2 of parent 1">
      <!-- some child stuff -->
    </child>
    <child name="child 3 of parent 1">
      <!-- some child stuff -->
    </child>
  </parent>

  <parent name="parent 2">
    <!-- Some parent stuff -->

    <child name="child 1 of parent 2">
      <!-- some child stuff -->
    </child>
    <child name="child 2 of parent 2">
      <!-- some child stuff -->
```

```
      </child>
      <child name="child 3 of parent 2">
        <!-- some child stuff -->
      </child>
    </parent>
    <!-- ... -->
    <parent name="parent N">
      <!-- Some parent stuff -->
      <child name="child 1 of parent N">
        <!-- some child stuff -->
      </child>
      <child name="child 2 of parent N">
        <!-- some child stuff -->
      </child>
      <child name="child 3 of parent N">
        <!-- some child stuff -->
      </child>
    </parent>
  </document>
```

When the document is structured hierarchically, the parent of a given node is implied by its context in the document. Thus, in order to process the children, the only parent you need to keep around is the one currently in context. When the parent goes out of context, there are guaranteed to be no more children for that parent, so there's no need to keep that parent's data around. This kind of structure has the additional advantage of being much more human readable.

Why doesn't Ant use a hierarchical structure? Ant models data that exhibits a many-to-many relationship; any given target can depend on any number of other targets, and any target can be a dependency of any number of other targets. In situations like these, it doesn't make sense to encode the data hierarchically, since there is no generalized ownership pattern of one target to another.

When you're designing your documents, use a hierarchical structure instead of pointers. If you have to model data that requires pointers (which you intend to have parsed using DOM or SAX, since, once again, XSLT does have more robust mechanisms for dealing with pointer resolution), try to design the documents with the parsing code in mind, so that the least possible state is required to be in memory at parsing time.

2.4 Conclusion

That wraps up our discussion on the basics of document design. We covered the two basic document styles, the characteristics of the building blocks of XML, and some important data modeling guidelines. This data has been presented in as general a fashion as possible because document design is a broad and ranging topic; it would be futile to try to cover specifics. The information that we've covered here should serve as a very solid foundation on which to make your own document design decisions based on the requirements and constraints of your own applications.

Throughout the rest of the book, we'll discuss how to design better documents for specific purposes, such as performing XSL transforms, storing data for long-term archival, and using XML as a transmission medium. The information we covered here in this chapter will serve as a basis for much of what we discuss for the rest of the book, so keep it in mind as you read on.