CHAPTER 5

Attributes, Empty Tags, and XSL

In This Chapter

Attributes

Attributes versus elements

Empty tags

XSL

You can encode a given set of data in XML in nearly an infinite number of ways. There's no one right way to do it although some ways are more right than others, and some are more appropriate for particular uses. In this chapter, we explore a different solution to the problem of marking up baseball statistics in XML, carrying over the baseball example from the previous chapter. Specifically, we will address the use of attributes to store information and empty tags to define element positions. In addition, since CSS doesn't work well with content-less XML elements of this form, we'll examine an alternative — and more powerful — style sheet language called XSL.

Attributes

In the last chapter, all data was categorized into the name of a tag or the contents of an element. This is a straightforward and easy-to-understand approach, but it's not the only one. As in HTML, XML elements may have attributes. An attribute is a name-value pair associated with an element. The name and the value are each strings, and no element may contain two attributes with the same name.

You're already familiar with attribute syntax from HTML. For example, consider this <IMG> tag:

```
<IMG SRC=cup.gif WIDTH=89 HEIGHT=67 ALT="Cup
of coffee">
```

It has four attributes, the `SRC` attribute whose value is `cup.gif`, the `WIDTH` attribute whose value is `89`, the `HEIGHT` attribute whose value is `67`, and the `ALT` attribute whose value is `Cup of coffee`. However, in XML-unlike HTML-attribute values must always be quoted and start tags must have matching close tags. Thus, the XML equivalent of this tag is:

```
<IMG SRC="cup.gif" WIDTH="89" HEIGHT="67" ALT="Cup of coffee">
</IMG>
```

Note

Another difference between HTML and XML is that XML assigns no particular meaning to the `IMG` tag and its attributes. In particular, there's no guarantee that an XML browser will interpret this tag as an instruction to load and display the image in the file cup.gif.

You can apply attribute syntax to the baseball example quite easily. This has the advantage of making the markup somewhat more concise. For example, instead of containing a `YEAR` child element, the `SEASON` element only needs a `YEAR` attribute.

```
<SEASON YEAR="1998">
</SEASON>
```

On the other hand, `LEAGUE` should be a child of the `SEASON` element rather than an attribute. For one thing, there are two leagues in a season. Anytime there's likely to be more than one of something child elements are called for. Attribute names must be unique within an element. Thus you should not, for example, write a `SEASON` element like this:

```
<SEASON YEAR="1998" LEAGUE="National" League="American">
</SEASON>
```

The second reason `LEAGUE` is naturally a child element rather than an attribute is that it has substructure; it is subdivided into `DIVISION` elements. Attribute values are flat text. XML elements can conveniently encode structure-attribute values cannot.

However, the name of a league is unstructured, flat text; and there's only one name per league so `LEAGUE` elements can easily have a `NAME` attribute instead of a `LEAGUE_NAME` child element:

```
<LEAGUE NAME="National League">
</LEAGUE>
```

Since an attribute is more closely tied to its element than a child element is, you don't run into problems by using `NAME` instead of `LEAGUE_NAME` for the name of the attribute. Divisions and teams can also have `NAME` attributes without any fear of confusion with the name of a league. Since a tag can have more than one attribute (as long as the attributes have different names), you can make a team's city an attribute as well, as shown below:

```
<LEAGUE NAME="American League">
  <DIVISION NAME="East">
   <TEAM NAME="Orioles" CITY="Baltimore"></TEAM>
   <TEAM NAME="Red Sox" CITY="Boston"></TEAM>
   <TEAM NAME="Yankees" CITY="New York"></TEAM>
   <TEAM NAME="Devil Rays" CITY="Tampa Bay"></TEAM>
   <TEAM NAME="Blue Jays" CITY="Toronto"></TEAM>
  </DIVISION>
</LEAGUE>
```

Players will have a lot of attributes if you choose to make each statistic an attribute. For example, here are Joe Girardi's 1998 statistics as attributes:

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2">
</PLAYER>
```

Listing 5-1 uses this new attribute style for a complete XML document containing the baseball statistics for the 1998 major league season. It displays the same information (i.e., two leagues, six divisions, 30 teams, and nine players) as does Listing 4-1 in the last chapter. It is merely marked up differently. Figure 5-1 shows this document loaded into Internet Explorer 5.0 without a style sheet.

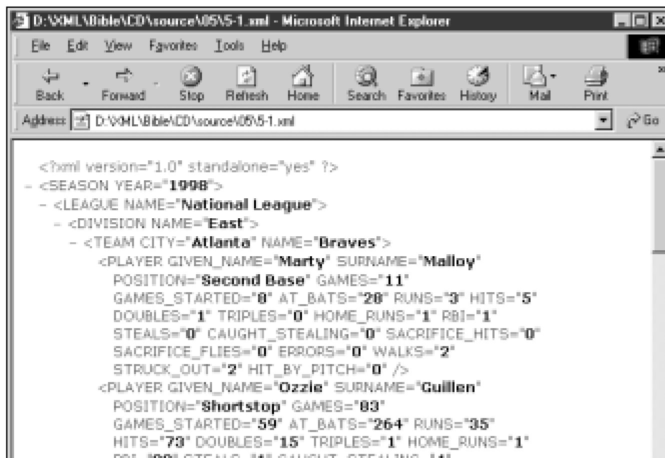

Figure 5-1: The 1998 major league baseball statistics using attributes for most information.

Listing 5-1: **A complete XML document that uses attributes to store baseball statistics**

```
<?xml version="1.0" standalone="yes"?>
<SEASON YEAR="1998">
  <LEAGUE NAME="National League">
    <DIVISION NAME="East">
      <TEAM CITY="Atlanta" NAME="Braves">
        <PLAYER GIVEN_NAME="Marty" SURNAME="Malloy"
         POSITION="Second Base" GAMES="11" GAMES_STARTED="8"
         AT_BATS="28" RUNS="3" HITS="5" DOUBLES="1"
         TRIPLES="0" HOME_RUNS="1" RBI="1" STEALS="0"
         CAUGHT_STEALING="0" SACRIFICE_HITS="0"
         SACRIFICE_FLIES="0" ERRORS="0" WALKS="2"
         STRUCK_OUT="2" HIT_BY_PITCH="0">
        </PLAYER>
        <PLAYER GIVEN_NAME="Ozzie" SURNAME="Guillen"
         POSITION="Shortstop" GAMES="83" GAMES_STARTED="59"
         AT_BATS="264" RUNS="35" HITS="73" DOUBLES="15"
         TRIPLES="1" HOME_RUNS="1" RBI="22" STEALS="1"
         CAUGHT_STEALING="4" SACRIFICE_HITS="4"
         SACRIFICE_FLIES="2" ERRORS="6" WALKS="24"
         STRUCK_OUT="25" HIT_BY_PITCH="1">
        </PLAYER>
        <PLAYER GIVEN_NAME="Danny" SURNAME="Bautista"
          POSITION="Outfield" GAMES="82" GAMES_STARTED="27"
          AT_BATS="144" RUNS="17" HITS="36" DOUBLES="11"
          TRIPLES="0" HOME_RUNS="3" RBI="17" STEALS="1"
          CAUGHT_STEALING="0" SACRIFICE_HITS="3"
          SACRIFICE_FLIES="2" ERRORS="2" WALKS="7"
          STRUCK_OUT="21" HIT_BY_PITCH="0">
        </PLAYER>
        <PLAYER GIVEN_NAME="Gerald" SURNAME="Williams"
         POSITION="Outfield" GAMES="129" GAMES_STARTED="51"
         AT_BATS="266" RUNS="46" HITS="81" DOUBLES="18"
         TRIPLES="3" HOME_RUNS="10" RBI="44" STEALS="11"
         CAUGHT_STEALING="5" SACRIFICE_HITS="2"
         SACRIFICE_FLIES="1" ERRORS="5" WALKS="17"
         STRUCK_OUT="48" HIT_BY_PITCH="3">
        </PLAYER>
        <PLAYER GIVEN_NAME="Tom" SURNAME="Glavine"
         POSITION="Starting Pitcher" GAMES="33"
         GAMES_STARTED="33" WINS="20" LOSSES="6" SAVES="0"
         COMPLETE_GAMES="4" SHUT_OUTS="3" ERA="2.47"
         INNINGS="229.1" HOME_RUNS_AGAINST="13"
         RUNS_AGAINST="67" EARNED_RUNS="63" HIT_BATTER="2"
         WILD_PITCHES="3" BALK="0" WALKED_BATTER="74"
         STRUCK_OUT_BATTER="157">
        </PLAYER>
        <PLAYER GIVEN_NAME="Javier" SURNAME="Lopez"
         POSITION="Catcher" GAMES="133" GAMES_STARTED="124"
         AT_BATS="489" RUNS="73" HITS="139" DOUBLES="21"
         TRIPLES="1" HOME_RUNS="34" RBI="106" STEALS="5"
```

```
     CAUGHT_STEALING="3" SACRIFICE_HITS="1"
     SACRIFICE_FLIES="8" ERRORS="5" WALKS="30"
     STRUCK_OUT="85" HIT_BY_PITCH="6">
    </PLAYER>
    <PLAYER GIVEN_NAME="Ryan" SURNAME="Klesko"
     POSITION="Outfield" GAMES="129" GAMES_STARTED="124"
     AT_BATS="427" RUNS="69" HITS="117" DOUBLES="29"
     TRIPLES="1" HOME_RUNS="18" RBI="70" STEALS="5"
     CAUGHT_STEALING="3" SACRIFICE_HITS="0"
     SACRIFICE_FLIES="4" ERRORS="2" WALKS="56"
     STRUCK_OUT="66" HIT_BY_PITCH="3">
    </PLAYER>
    <PLAYER GIVEN_NAME="Andres" SURNAME="Galarraga"
     POSITION="First Base" GAMES="153" GAMES_STARTED="151"
     AT_BATS="555" RUNS="103" HITS="169" DOUBLES="27"
     TRIPLES="1" HOME_RUNS="44" RBI="121" STEALS="7"
     CAUGHT_STEALING="6" SACRIFICE_HITS="0"
     SACRIFICE_FLIES="5" ERRORS="11" WALKS="63"
     STRUCK_OUT="146" HIT_BY_PITCH="25">
    </PLAYER>
    <PLAYER GIVEN_NAME="Wes" SURNAME="Helms"
     POSITION="Third Base" GAMES="7" GAMES_STARTED="2"
     AT_BATS="13" RUNS="2" HITS="4" DOUBLES="1"
     TRIPLES="0" HOME_RUNS="1" RBI="2" STEALS="0"
     CAUGHT_STEALING="0" SACRIFICE_HITS="0"
     SACRIFICE_FLIES="0" ERRORS="1" WALKS="0"
     STRUCK_OUT="4" HIT_BY_PITCH="0">
    </PLAYER>
  </TEAM>
  <TEAM CITY="Florida" NAME="Marlins">
  </TEAM>
  <TEAM CITY="Montreal" NAME="Expos">
  </TEAM>
  <TEAM CITY="New York" NAME="Mets">
  </TEAM>
  <TEAM CITY="Philadelphia" NAME="Phillies">
  </TEAM>
</DIVISION>
<DIVISION NAME="Central">
  <TEAM CITY="Chicago" NAME="Cubs">
  </TEAM>
  <TEAM CITY="Cincinnati" NAME="Reds">
  </TEAM>
  <TEAM CITY="Houston" NAME="Astros">
  </TEAM>
  <TEAM CITY="Milwaukee" NAME="Brewers">
  </TEAM>
  <TEAM CITY="Pittsburgh" NAME="Pirates">
  </TEAM>
  <TEAM CITY="St. Louis" NAME="Cardinals">
  </TEAM>
</DIVISION>
```

Continued

Listing 5-1 *(continued)*

```
    <DIVISION NAME="West">
      <TEAM CITY="Arizona" NAME="Diamondbacks">
      </TEAM>
      <TEAM CITY="Colorado" NAME="Rockies">
      </TEAM>
      <TEAM CITY="Los Angeles" NAME="Dodgers">
      </TEAM>
      <TEAM CITY="San Diego" NAME="Padres">
      </TEAM>
      <TEAM CITY="San Francisco" NAME="Giants">
      </TEAM>
    </DIVISION>
  </LEAGUE>
  <LEAGUE NAME="American League">
    <DIVISION NAME="East">
      <TEAM CITY="Baltimore" NAME="Orioles">
      </TEAM>
      <TEAM CITY="Boston" NAME="Red Sox">
      </TEAM>
      <TEAM CITY="New York" NAME="Yankees">
      </TEAM>
      <TEAM CITY="Tampa Bay" NAME="Devil Rays">
      </TEAM>
      <TEAM CITY="Toronto" NAME="Blue Jays">
      </TEAM>
    </DIVISION>
    <DIVISION NAME="Central">
      <TEAM CITY="Chicago" NAME="White Sox">
      </TEAM>
      <TEAM CITY="Kansas City" NAME="Royals">
      </TEAM>
      <TEAM CITY="Detroit" NAME="Tigers">
      </TEAM>
      <TEAM CITY="Cleveland" NAME="Indians">
      </TEAM>
      <TEAM CITY="Minnesota" NAME="Twins">
      </TEAM>
    </DIVISION>
    <DIVISION NAME="West">
      <TEAM CITY="Anaheim" NAME="Angels">
      </TEAM>
      <TEAM CITY="Oakland" NAME="Athletics">
      </TEAM>
      <TEAM CITY="Seattle" NAME="Mariners">
      </TEAM>
      <TEAM CITY="Texas" NAME="Rangers">
      </TEAM>
     </DIVISION>
  </LEAGUE>
</SEASON>
```

Listing 5-1 uses only attributes for player information. Listing 4-1 used only element content. There are intermediate approaches as well. For example, you could make the player's name part of element content while leaving the rest of the statistics as attributes, like this:

```
<P>
  On Tuesday <PLAYER GAMES="78" AT_BATS="254" RUNS="31"
  HITS="70" DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRIKE_OUTS="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2">Joe Girardi</PLAYER> struck out twice
  and...
</P>
```

This would include Joe Girardi's name in the text of a page while still making his statistics available to readers who want to look deeper, as a hypertext footnote or tool tip. There's always more than one way to encode the same data. Which way you pick generally depends on the needs of your specific application.

Attributes versus Elements

There are no hard and fast rules about when to use child elements and when to use attributes. Generally, you'll use whichever suits your application. With experience, you'll gain a feel for when attributes are easier than child elements and vice versa. Until then, one good rule of thumb is that the data itself should be stored in elements. Information about the data (meta-data) should be stored in attributes. And when in doubt, put the information in the elements.

To differentiate between data and meta-data, ask yourself whether someone reading the document would want to see a particular piece of information. If the answer is yes, then the information probably belongs in a child element. If the answer is no, then the information probably belongs in an attribute. If all tags were stripped from the document along with all the attributes, the basic information should still be present. Attributes are good places to put ID numbers, URLs, references, and other information not directly or immediately relevant to the reader. However, there are many exceptions to the basic principal of storing meta-data as attributes. These include:

- ✦ Attributes can't hold structure well.
- ✦ Elements allow you to include meta-meta-data (information about the information about the information).
- ✦ Not everyone always agrees on what is and isn't meta-data.
- ✦ Elements are more extensible in the face of future changes.

Structured Meta-data

One important principal to remember is that elements can have substructure and attributes can't. This makes elements far more flexible, and may convince you to encode meta-data as child elements. For example, suppose you're writing a paper and you want to include a source for a fact. It might look something like this:

```
<FACT SOURCE="The Biographical History of Baseball,
Donald Dewey and Nicholas Acocella (New York: Carroll &amp;
Graf Publishers, Inc. 1995) p. 169">
  Josh Gibson is the only person in the history of baseball to
  hit a pitch out of Yankee Stadium.
</FACT>
```

Clearly the information "The Biographical History of Baseball, Donald Dewey and Nicholas Acocella (New York: Carroll & Graf Publishers, Inc. 1995) p. 169" is meta-data. It is not the fact itself. Rather it is information about the fact. However, the `SOURCE` attribute contains a lot of implicit substructure. You might find it more useful to organize the information like this:

```
<SOURCE>
  <AUTHOR>Donald Dewey</AUTHOR>
  <AUTHOR>Nicholas Acocella</AUTHOR>
  <BOOK>
    <TITLE>The Biographical History of Baseball</TITLE>
    <PAGES>169</PAGES>
    <YEAR>1995</YEAR>
  </BOOK>
</SOURCE>
```

Furthermore, using elements instead of attributes makes it straightforward to include additional information like the authors' e-mail addresses, a URL where an electronic copy of the document can be found, the title or theme of the particular issue of the journal, and anything else that seems important.

Dates are another common example. One common piece of meta-data about scholarly articles is the date the article was first received. This is important for establishing priority of discovery and invention. It's easy to include a `DATE` attribute in an `ARTICLE` tag like this:

```
<ARTICLE DATE="06/28/1969">
  Polymerase Reactions in Organic Compounds
</ARTICLE>
```

However, the `DATE` attribute has substructure signified by the `/`. Getting that structure out of the attribute value, however, is much more difficult than reading child elements of a `DATE` element, as shown below:

```
<DATE>
  <YEAR>1969</YEAR>
  <MONTH>06</MONTH>
  <DAY>28</DAY>
</DATE>
```

For instance, with CSS or XSL, it's easy to format the day and month invisibly so that only the year appears. For example, using CSS:

```
YEAR {display: inline}
MONTH {display: none}
DAY {display: none}
```

If the `DATE` is stored as an attribute, however, there's no easy way to access only part of it. You must write a separate program in a programming language like ECMAScript or Java that can parse your date format. It's easier to use the standard XML tools and child elements.

Furthermore, the attribute syntax is ambiguous. What does the date "10/11/1999" signify? In particular, is it October 11th or November 10th? Readers from different countries will interpret this data differently. Even if your parser understands one format, there's no guarantee the people entering the data will enter it correctly. The XML, by contrast, is unambiguous.

Finally, using `DATE` children rather than attributes allows more than one date to be associated with an element. For instance, scholarly articles are often returned to the author for revisions. In these cases, it can also be important to note when the revised article was received. For example:

```
<ARTICLE>
  <TITLE>
    Maximum Projectile Velocity in an Augmented Railgun
  </TITLE>
  <AUTHOR>Elliotte Harold</AUTHOR>
  <AUTHOR>Bruce Bukiet</AUTHOR>
  <AUTHOR>William Peter</AUTHOR>
  <DATE>
    <YEAR>1992</YEAR>
    <MONTH>10</MONTH>
    <DAY>29</DAY>
  </DATE>
  <DATE>
    <YEAR>1993</YEAR>
    <MONTH>10</MONTH>
    <DAY>26</DAY>
  </DATE>
</ARTICLE>
```

As another example, consider the ALT attribute of an IMG tag in HTML. This is limited to a single string of text. However, given that a picture is worth a thousand words, you might well want to replace an IMG with marked up text. For instance, consider the pie chart shown in Figure 5-2.

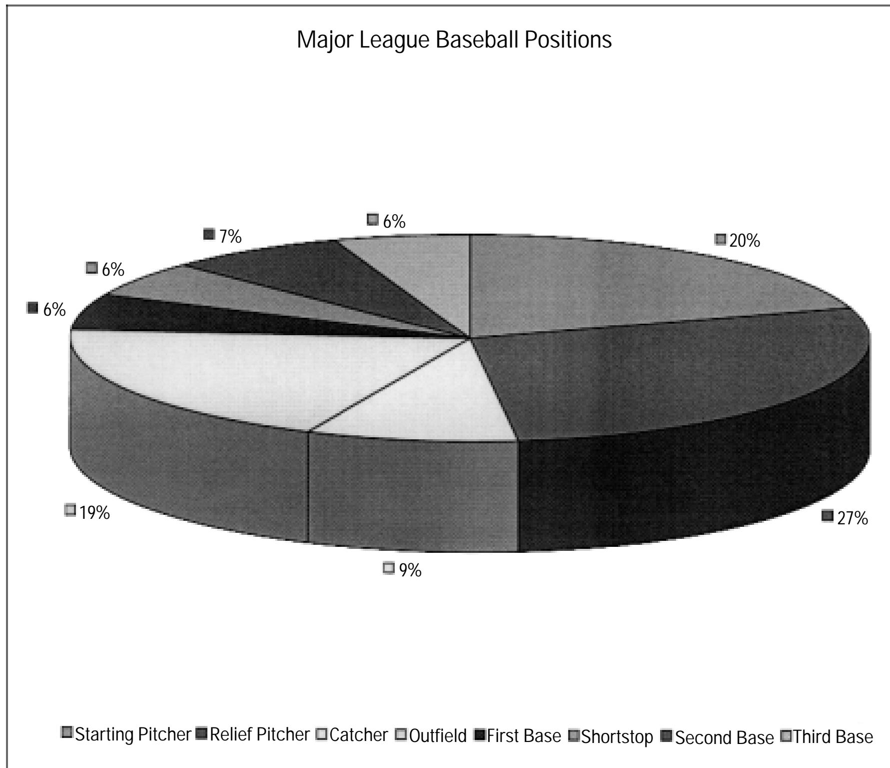

Figure 5-2: Distribution of positions in major league baseball

Using an ALT attribute, the best description of this picture you can provide is:

```
<IMG SRC="05021.gif"
     ALT="Pie Chart of Positions in Major League Baseball"
     WIDTH="819" HEIGHT="623">
</IMG>
```

However, with an ALT child element, you have more flexibility because you can embed markup. For example, you might provide a table of the relevant numbers instead of a pie chart.

```
<IMG SRC="05021.gif" WIDTH="819" HEIGHT="623">
  <ALT>
    <TABLE>
      <TR>
        <TD>Starting Pitcher</TD> <TD>242</TD> <TD>20%</TD>
      </TR>
      <TR>
        <TD>Relief Pitcher</TD> <TD>336</TD> <TD>27%</TD>
      </TR>
      <TR>
        <TD>Catcher</TD> <TD>104</TD> <TD>9%</TD>
      </TR>
      <TR>
        <TD>Outfield</TD> <TD>235</TD> <TD>19%</TD>
      </TR>
      <TR>
        <TD>First Base</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
      <TR>
        <TD>Shortstop</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
      <TR>
        <TD>Second Base</TD> <TD>88</TD> <TD>7%</TD>
      </TR>
      <TR>
        <TD>Third Base</TD> <TD>67</TD> <TD>6%</TD>
      </TR>
    </TABLE>
  </ALT>
</IMG>
```

You might even provide the actual Postscript, SVG, or VML code to render the picture in the event that the bitmap image is not available.

Meta-Meta-Data

Using elements for meta-data also easily allows for meta-meta-data, or information about the information about the information. For example, the author of a poem may be considered to be meta-data about the poem. The language in which that author's name is written is data about the meta-data about the poem. This isn't a trivial concern, especially for distinctly non-Roman languages. For instance, is the author of the Odyssey Homer or ______? If you use elements, it's easy to write:

```
<POET LANGUAGE="English">Homer</POET>
<POET LANGUAGE="Greek">______</POET>
```

However, if POET is an attribute rather than a child element, you're stuck with unwieldy constructs like this:

```
<POEM POET="Homer" POET_LANGUAGE="English"
 POEM_LANGUAGE="English">Homer
  Tell me, O Muse, of the cunning man...
</POEM>
```

And it's even more bulky if you want to provide both the poet's English and Greek names.

```
<POEM POET_NAME_1="Homer" POET_LANGUAGE_1="English"
 POET_NAME_2="______" POET_LANGUAGE_2="Greek"
 POEM_LANGUAGE="English">Homer
  Tell me, O Muse, of the cunning man...
</POEM>
```

What's Your Meta-data Is Someone Else's Data

"Metaness" is in the mind of the beholder. Who is reading your document and why they are reading it determines what they consider to be data and what they consider to be meta-data. For example, if you're simply reading an article in a scholarly journal, then the author of the article is tangential to the information it contains. However, if you're sitting on a tenure and promotions committee scanning a journal to see who is publishing and who is not, then the names of the authors and the number of articles they've published may be more important to you than what they wrote (sad but true).

In fact, you may change your mind about what's meta and what's data. What's only tangentially relevant today, may become crucial to you next week. You can use style sheets to hide unimportant elements today, and change the style sheets to reveal them later. However, it's more difficult to later reveal information that was first stored in an attribute. Usually, this requires rewriting the document itself rather than simply changing the style sheet.

Elements Are More Extensible

Attributes are certainly convenient when you only need to convey one or two words of unstructured information. In these cases, there may genuinely be no current need for a child element. However, this doesn't preclude such a need in the future.

For instance, you may now only need to store the name of the author of an article, and you may not need to distinguish between the first and last names. However, in the future you may uncover a need to store first and last names, e-mail addresses, institution, snail mail address, URL, and more. If you've stored the author of the article as an element, then it's easy to add child elements to include this additional information.

Although any such change will probably require some revision of your documents, style sheets, and associated programs, it's still much easier to change a simple element to a tree of elements than it is to make an attribute a tree of elements. However, if you used an attribute, then you're stuck. It's quite difficult to extend your attribute syntax beyond the region it was originally designed for.

Good Times to Use Attributes

Having exhausted all the reasons why you should use elements instead of attributes, I feel compelled to point out that there are nonetheless some times when attributes make sense. First of all, as previously mentioned, attributes are fully appropriate for very simple data without substructure that the reader is unlikely to want to see. One example is the `HEIGHT` and `WIDTH` attributes of an `IMG`. Although the values of these attributes may change if the image changes, it's hard to imagine how the data in the attribute could be anything more than a very short string of text. `HEIGHT` and `WIDTH` are one-dimensional quantities (in more ways than one) so they work well as attributes.

Furthermore, attributes are appropriate for simple information about the document that has nothing to do with the content of the document. For example, it is often useful to assign an `ID` attribute to each element. This is a unique string possessed only by one element in the document. You can then use this string for a variety of tasks including linking to particular elements of the document, even if the elements move around as the document changes over time. For example:

```
<SOURCE ID="S1">
  <AUTHOR ID="A1">Donald Dewey</AUTHOR>
  <AUTHOR ID="A2">Nicholas Acocella</AUTHOR>
  <BOOK ID="B1">
    <TITLE ID="B2">
      The Biographical History of Baseball
    </TITLE>
    <PAGES ID="B3">169</PAGES>
    <YEAR ID="B4">1995</YEAR>
  </BOOK>
</SOURCE>
```

`ID` attributes make links to particular elements in the document possible. In this way, they can serve the same purpose as the `NAME` attribute of HTML's `A` elements. Other data associated with linking — `HREF`s to link to, `SRC`s to pull images and binary data from, and so forth — also work well as attributes.

You'll see more examples of this when XLL, the Extensible Linking Language, is discussed in Chapter 16, *XLinks*, and Chapter 17, *XPointers*.

Attributes are also often used to store document-specific style information. For example, if `TITLE` elements are generally rendered as bold text but if you want to make just one `TITLE` element both bold and italic, you might write something like this:

```
<TITLE style="font-style: italic">Significant Others</TITLE>
```

This enables the style information to be embedded without changing the tree structure of the document. While ideally you'd like to use a separate element, this scheme gives document authors somewhat more control when they cannot add elements to the tag set they're working with. For example, the Webmaster of a site might require the use of a particular DTD and not want to allow everyone to modify the DTD. Nonetheless, they want to allow them to make minor adjustments to individual pages. Use this scheme with restraint, however, or you'll soon find yourself back in the HTML hell XML was supposed to save us from, where formatting is freely intermixed with meaning and documents are no longer maintainable.

The final reason to use attributes is to maintain compatibility with HTML. To the extent that you're using tags that at least look similar to HTML such as `<IMG>`, `<P>`, and `<TD>`, you might as well employ the standard HTML attributes for these tags. This has the double advantage of enabling legacy browsers to at least partially parse and display your document, and of being more familiar to the people writing the documents.

Empty Tags

Last chapter's no-attribute approach was an extreme position. It's also possible to swing to the other extreme — storing all the information in the attributes and none in the content. In general, I don't recommend this approach. Storing all the information in element content — while equally extreme — is much easier to work with in practice. However, this section entertains the possibility of using only attributes for the sake of elucidation.

As long as you know the element will have no content, you can use empty tags as a short cut. Rather than including both a start and an end tag you can include one empty tag. Empty tags are distinguished from start tags by a closing `/>` instead of simply a closing `>`. For instance, instead of `<PLAYER></PLAYER>` you would write `<PLAYER/>`.

Empty tags may contain attributes. For example, here's an empty tag for Joe Girardi with several attributes:

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
```

```
    SACRIFICE_FLY="1" SACRIFICE_HIT="8"
    HIT_BY_PITCH="2"/>
```

XML parsers treat this identically to the non-empty equivalent. This PLAYER element is precisely equal (though not identical) to the previous PLAYER element formed with an empty tag.

```
<PLAYER GIVEN_NAME="Joe" SURNAME="Girardi"
  GAMES="78" AT_BATS="254" RUNS="31" HITS="70"
  DOUBLES="11" TRIPLES="4" HOME_RUNS="3"
  RUNS_BATTED_IN="31" WALKS="14" STRUCK_OUT="38"
  STOLEN_BASES="2" CAUGHT_STEALING="4"
  SACRIFICE_FLY="1" SACRIFICE_HIT="8"
  HIT_BY_PITCH="2"></PLAYER>
```

The difference between <PLAYER/> and <PLAYER></PLAYER> is syntactic sugar, and nothing more. If you don't like the empty tag syntax, or find it hard to read, you don't have to use it.

XSL

Attributes are visible in an XML source view of the document as shown in Figure 5-1. However, once a CSS style sheet is applied the attributes disappear. Figure 5-3 shows Listing 5-1 once the baseball stats style sheet from the previous chapter is applied. It looks like a blank document because CSS styles only apply to element content, not to attributes. If you use CSS, any data you want to display to the reader should be part of an element's content rather than one of its attributes.

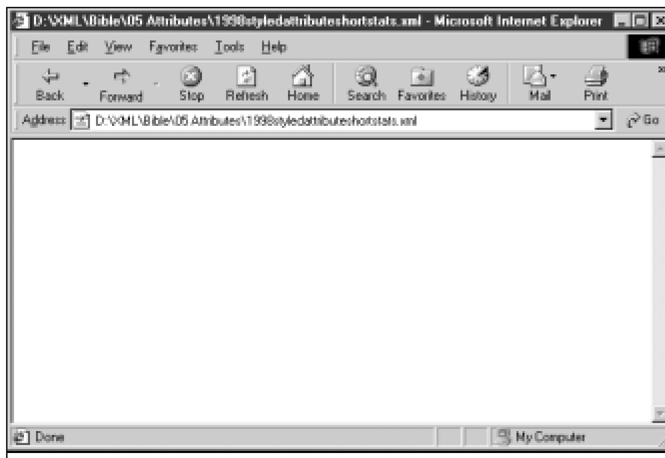

Figure 5-3: A blank document is displayed when CSS is applied to an XML document whose elements do not contain any character data.

However, there is an alternative style sheet language that does allow you to access and display attribute data. This language is the Extensible Style Language (XSL); and it is also supported by Internet Explorer 5.0, at least in part. XSL is divided into two sections, transformations and formatting.

The transformation part of XSL enables you to replace one tag with another. You can define rules that replace your XML tags with standard HTML tags, or with HTML tags plus CSS attributes. You can also do a lot more including reordering the elements in the document and adding additional content that was never present in the XML document.

The formatting part of XSL defines an extremely powerful view of documents as pages. XSL formatting enables you to specify the appearance and layout of a page including multiple columns, text flow around objects, line spacing, assorted font properties, and more. It's designed to be powerful enough to handle automated layout tasks for both the Web and print from the same source document. For instance, XSL formatting would allow one XML document containing show times and advertisements to generate both the print and online editions of a local newspaper's television listings. However, IE 5.0 and most other tools do not yet support XSL formatting. Therefore, in this section I'll focus on XSL transformations.

XSL formatting is discussed in Chapter 15, *XSL Formatting Objects*.

XSL Style Sheet Templates

An XSL style sheet contains templates into which data from the XML document is poured. For example, one template might look something like this:

```
<HTML>
  <HEAD>
    <TITLE>
      XSL Instructions to get the title
    </TITLE>
  </HEAD>
  <H1>XSL Instructions to get the title</H1>
  <BODY>
    XSL Instructions to get the statistics
  </BODY>
</HTML>
```

The italicized sections will be replaced by particular XSL elements that copy data from the underlying XML document into this template. You can apply this template to many different data sets. For instance, if the template is designed to work with the baseball example, then the same style sheet can display statistics from different seasons.

This may remind you of some server-side include schemes for HTML. In fact, this is very much like server-side includes. However, the actual transformation of the source XML document and XSL style sheet takes place on the client rather than on the server. Furthermore, the output document does not have to be HTML. It can be any well-formed XML.

XSL instructions can retrieve any data stored in the elements of the XML document. This includes element content, element names, and, most importantly for our example, element attributes. Particular elements are chosen by a pattern that considers the element's name, its value, its attributes' names and values, its absolute and relative position in the tree structure of the XML document, and more. Once the data is extracted from an element, it can be moved, copied, and manipulated in a variety of ways. We won't cover everything you can do with XML transformations in this brief introduction. However, you will learn to use XSL to write some pretty amazing documents that can be viewed on the Web right away.

Chapter 14, *XSL Transformations*, covers XSL transformations in depth.

The Body of the Document

Let's begin by looking at a simple example and applying it to the XML document with baseball statistics shown in Listing 5-1. Listing 5-2 is an XSL style sheet. This style sheet provides the HTML mold into which XML data will be poured.

Listing 5-2: An XSL style sheet

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>
        <H1>Major League Baseball Statistics</H1>

        <HR></HR>
        Copyright 1999
        <A HREF="http://www.macfaq.com/personal.html">
         Elliotte Rusty Harold
        </A>
```

Continued

Listing 5-2 *(continued)*

```
        <BR />
        <A HREF="mailto:elharo@metalab.unc.edu">
         elharo@metalab.unc.edu
        </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

It resembles an HTML file included inside an `xsl:template` element. In other words its structure looks like this:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    HTML file goes here
  </xsl:template>

</xsl:stylesheet>
```

Listing 5-2 is not only an XSL style sheet; it's also a well-formed XML document. It begins with an XML declaration. The root element of this document is `xsl:stylesheet`. This style sheet contains a single template for the XML data encoded as an `xsl:template` element. The `xsl:template` element has a `match` attribute with the value `/` and its content is a well-formed HTML document. It's not a coincidence that the output HTML is well-formed. Because the HTML must first be part of an XSL style sheet, and because XSL style sheets are well-formed XML documents, all the HTML in a XSL style sheet must be well-formed.

The Web browser tries to match parts of the XML document against each `xsl:template` element. The `/` template matches the root of the document; that is the entire document itself. The browser reads the template and inserts data from the XML document where indicated by XSL instructions. However, this particular template contains no XSL instructions, so its contents are merely copied verbatim into the Web browser, producing the output you see in Figure 5-4. Notice that Figure 5-4 does not display any data from the XML document, only from the XSL template.

Attaching the XSL style sheet of Listing 5-2 to the XML document in Listing 5-1 is straightforward. Simply add a `<?xml-stylesheet?>` processing instruction with a `type` attribute with value `text/xsl` and an `href` attribute that points to the style sheet between the XML declaration and the root element. For example:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="5-2.xsl"?>
<SEASON YEAR="1998">
...
```

This is the same way a CSS style sheet is attached to a document. The only difference is that the `type` attribute is `text/xsl` instead of `text/css`.

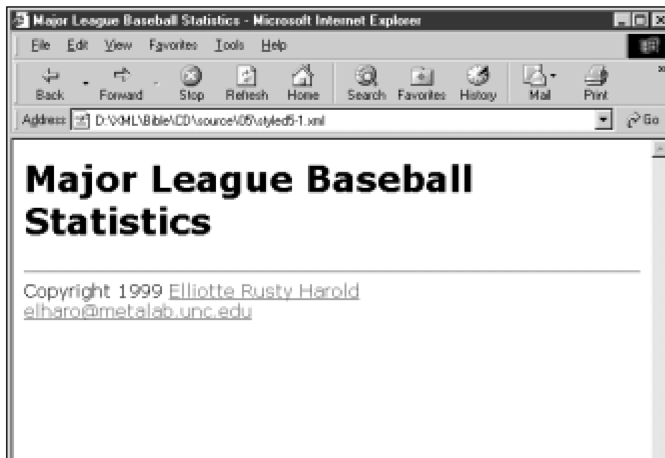

Figure 5-4: The data from the XML document, not the XSL template, is missing after application of the XSL style sheet in Listing 5-2.

The Title

Of course there was something rather obvious missing from Figure 5-4 — the data! Although the style sheet in Listing 5-2 displays something (unlike the CSS style sheet of Figure 5-3) it doesn't show any data from the XML document. To add this, you need to use XSL instruction elements to copy data from the source XML document into the XSL template. Listing 5-3 adds the necessary XSL instructions to extract the `YEAR` attribute from the `SEASON` element and insert it in the `TITLE` and `H1` header of the resulting document. Figure 5-5 shows the rendered document.

Listing 5-3: An XSL style sheet with instructions to extract the SEASON element and YEAR attribute

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>
          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>
      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

The new XSL instructions that extract the `YEAR` attribute from the `SEASON` element are:

```
<xsl:for-each select="SEASON">
  <xsl:value-of select="@YEAR"/>
</xsl:for-each>
```

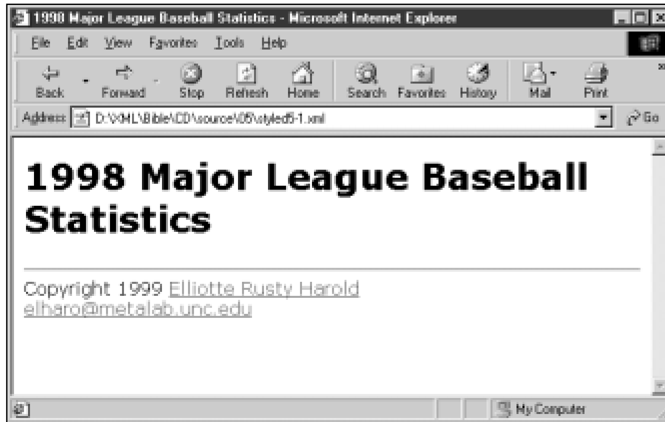

Figure 5-5: Listing 5-1 after application of the XSL style sheet in Listing 5-3

These instructions appear twice because we want the year to appear twice in the output document-once in the `H1` header and once in the `TITLE`. Each time they appear, these instructions do the same thing. `<xsl:for-each select="SEASON">` finds all `SEASON` elements. `<xsl:value-of select="@YEAR"/>` inserts the value of the `YEAR` attribute of the `SEASON` element — that is, the string "1998" — found by `<xsl:for-each select="SEASON">`.

This is important, so let me say it again: `xsl:for-each` selects a particular XML element in the source document (Listing 5-1 in this case) from which data will be read. `xsl:value-of` copies a particular part of the element into the output document. You need both XSL instructions. Neither alone is sufficient.

XSL instructions are distinguished from output elements like `HTML` and `H1` because the instructions are in the `xsl` namespace. That is, the names of all XSL elements begin with `xsl:`. The namespace is identified by the `xmlns:xsl` attribute of the root element of the style sheet. In Listings 5-2, 5-3, and all other examples in this book, the value of that attribute is `http://www.w3.org/TR/WD-xsl`.

Cross-Reference

Namespaces are covered in depth in Chapter 18, *Namespaces*.

Leagues, Divisions, and Teams

Next, let's add some XSL instructions to pull out the two `LEAGUE` elements. We'll map these to `H2` headers. Listing 5-4 demonstrates. Figure 5-6 shows the document rendered with this style sheet.

Listing 5-4: An XSL style sheet with instructions to extract LEAGUE elements

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>
        </xsl:for-each>

      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

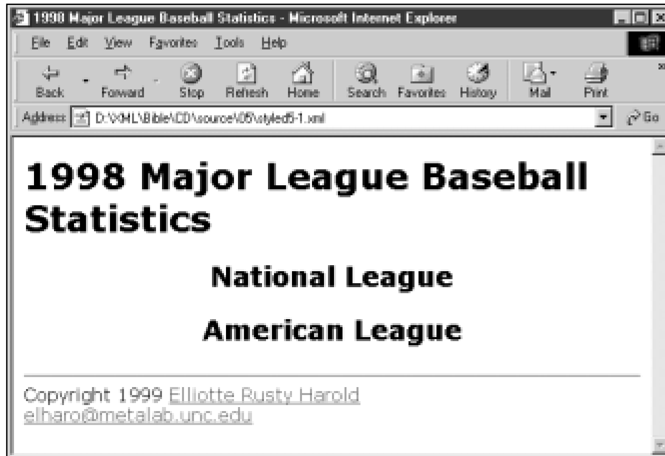

Figure 5-6: The league names are displayed as H2 headers when the XSL style sheet in Listing 5-4 is applied.

The key new materials are the nested `xsl:for-each` instructions

```
<xsl:for-each select="SEASON">
  <H1>
    <xsl:value-of select="@YEAR"/>
    Major League Baseball Statistics
  </H1>

  <xsl:for-each select="LEAGUE">
    <H2 ALIGN="CENTER">
      <xsl:value-of select="@NAME"/>
    </H2>
  </xsl:for-each>

</xsl:for-each>
```

The outermost instruction says to select the `SEASON` element. With that element selected, we then find the `YEAR` attribute of that element and place it between `<H1>` and `</H1>` along with the extra text `Major League Baseball Statistics`. Next, the browser loops through each `LEAGUE` child of the selected `SEASON` and places the value of its `NAME` attribute between `<H2 ALIGN="CENTER">` and `</H2>`. Although there's only one `xsl:for-each` matching a `LEAGUE` element, it loops over all the `LEAGUE` elements that are immediate children of the `SEASON` element. Thus, this template works for anywhere from zero to an indefinite number of leagues.

The same technique can be used to assign `H3` headers to divisions and `H4` headers to teams. Listing 5-5 demonstrates the procedure and Figure 5-7 shows the document rendered with this style sheet. The names of the divisions and teams are read from the XML data.

Listing 5-5: An XSL style sheet with instructions to extract DIVISION and TEAM elements

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>

          <xsl:for-each select="DIVISION">
            <H3 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
            </H3>

            <xsl:for-each select="TEAM">
              <H4 ALIGN="CENTER">
              <xsl:value-of select="@CITY"/>
              <xsl:value-of select="@NAME"/>
              </H4>
            </xsl:for-each>
          </xsl:for-each>

        </xsl:for-each>
      </xsl:for-each>

      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
```

```
    Elliotte Rusty Harold
   </A>
   <BR />
   <A HREF="mailto:elharo@metalab.unc.edu">
    elharo@metalab.unc.edu
   </A>

   </BODY>
  </HTML>
 </xsl:template>

</xsl:stylesheet>
```

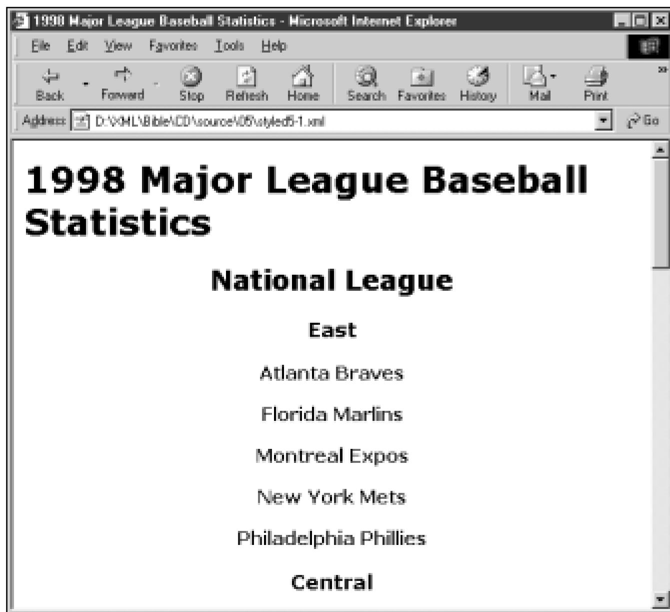

Figure 5-7: Divisions and team names are displayed after application of the XSL style sheet in Listing 5-5.

In the case of the `TEAM` elements, the values of both its `CITY` and `NAME` attributes are used as contents for the `H4` header. Also notice that the nesting of the `xsl:for-each` elements that selects seasons, leagues, divisions, and teams mirrors the hierarchy of the document itself. That's not a coincidence. While other schemes are possible that don't require matching hierarchies, this is the simplest, especially for highly structured data like the baseball statistics of Listing 5-1.

Players

The next step is to add statistics for individual players on a team. The most natural way to do this is in a table. Listing 5-6 shows an XSL style sheet that arranges the players and their stats in a table. No new XSL elements are introduced. The same `xsl:for-each` and `xsl:value-of` elements are used on the `PLAYER` element and its attributes. The output is standard HTML table tags. Figure 5-8 displays the results.

Listing 5-6: An XSL style sheet that places players and their statistics in a table

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

      <xsl:for-each select="SEASON">
        <H1>
          <xsl:value-of select="@YEAR"/>
          Major League Baseball Statistics
        </H1>

        <xsl:for-each select="LEAGUE">
          <H2 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
          </H2>

          <xsl:for-each select="DIVISION">
            <H3 ALIGN="CENTER">
            <xsl:value-of select="@NAME"/>
            </H3>

            <xsl:for-each select="TEAM">
              <H4 ALIGN="CENTER">
              <xsl:value-of select="@CITY"/>
              <xsl:value-of select="@NAME"/>
              </H4>

              <TABLE>
```

```
    <THEAD>
     <TR>
      <TH>Player</TH><TH>P</TH><TH>G</TH>
      <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
      <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
      <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
      <TH>E</TH><TH>BB</TH><TH>SO</TH><TH>HBP</TH>
     </TR>
    </THEAD>
   <TBODY>
    <xsl:for-each select="PLAYER">
     <TR>
      <TD>
       <xsl:value-of select="@GIVEN_NAME"/>
       <xsl:value-of select="@SURNAME"/>
      </TD>
      <TD><xsl:value-of select="@POSITION"/></TD>
      <TD><xsl:value-of select="@GAMES"/></TD>
      <TD>
        <xsl:value-of select="@GAMES_STARTED"/>
      </TD>
      <TD><xsl:value-of select="@AT_BATS"/></TD>
      <TD><xsl:value-of select="@RUNS"/></TD>
      <TD><xsl:value-of select="@HITS"/></TD>
      <TD><xsl:value-of select="@DOUBLES"/></TD>
      <TD><xsl:value-of select="@TRIPLES"/></TD>
      <TD><xsl:value-of select="@HOME_RUNS"/></TD>
      <TD><xsl:value-of select="@RBI"/></TD>
      <TD><xsl:value-of select="@STEALS"/></TD>
      <TD>
       <xsl:value-of select="@CAUGHT_STEALING"/>
      </TD>
      <TD>
       <xsl:value-of select="@SACRIFICE_HITS"/>
      </TD>
      <TD>
       <xsl:value-of select="@SACRIFICE_FLIES"/>
      </TD>
      <TD><xsl:value-of select="@ERRORS"/></TD>
      <TD><xsl:value-of select="@WALKS"/></TD>
      <TD>
       <xsl:value-of select="@STRUCK_OUT"/>
      </TD>
      <TD>
       <xsl:value-of select="@HIT_BY_PITCH"/>
      </TD>
     </TR>
    </xsl:for-each>
   </TBODY>
  </TABLE>

</xsl:for-each>
```

Continued

Listing 5-6 *(continued)*

```
            </xsl:for-each>

          </xsl:for-each>
        </xsl:for-each>

        <HR></HR>
        Copyright 1999
        <A HREF="http://www.macfaq.com/personal.html">
         Elliotte Rusty Harold
        </A>
        <BR />
        <A HREF="mailto:elharo@metalab.unc.edu">
         elharo@metalab.unc.edu
        </A>

        </BODY>
      </HTML>
    </xsl:template>

  </xsl:stylesheet>
```

Separation of Pitchers and Batters

One discrepancy you may have noted in Figure 5-8 is that the pitchers aren't handled properly. Throughout this chapter and Chapter 4, we've always given the pitchers a completely different set of statistics, whether those stats were stored in element content or attributes. Therefore, the pitchers really need a table that is separate from the other players. Before putting a player into the table, you must test whether he is or is not a pitcher. If his `POSITION` attribute contains the string "pitcher" then omit him. Then reverse the procedure in a second table that only includes pitchers-`PLAYER` elements whose `POSITION` attribute contains the string "pitcher".

To do this, you have to add additional code to the `xsl:for-each` element that selects the players. You don't select all players. Instead, you select those players whose `POSITION` attribute is not pitcher. The syntax looks like this:

```
<xsl:for-each select="PLAYER[(@POSITION != 'Pitcher')">
```

But because the XML document distinguishes between starting and relief pitchers, the true answer must test both cases:

```
<xsl:for-each select="PLAYER[(@POSITION != 'Starting Pitcher')
  $and$ (@POSITION != 'Relief Pitcher')]">
```

Figure 5-8: Player statistics are displayed after applying the XSL style sheet in Listing 5-6.

For the table of pitchers, you logically reverse this to the position being equal to either "Starting Pitcher" or "Relief Pitcher". (It is not sufficient to just change *not equal* to *equal*. You also have to change *and* to *or*.) The syntax looks like this:

```
<xsl:for-each select="PLAYER[(@POSITION = 'Starting Pitcher')
 $or$ (@POSITION = 'Relief Pitcher')]">
```

Note

Only a single equals sign is used to test for equality rather than the double equals sign used in C and Java. That's because there's no equivalent of an assignment operator in XSL.

Listing 5-7 shows an XSL style sheet separating the batters and pitchers into two different tables. The pitchers' table adds columns for all the usual pitcher statistics. Listing 5-1 encodes in attributes: wins, losses, saves, shutouts, etc. Abbreviations are used for the column labels to keep the table to a manageable width. Figure 5-9 shows the results.

Listing 5-7: **An XSL style sheet that separates batters and pitchers**

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

  <xsl:template match="/">
    <HTML xmlns:xsl="http://www.w3.org/TR/WD-xsl">
      <HEAD>
        <TITLE>
          <xsl:for-each select="SEASON">
            <xsl:value-of select="@YEAR"/>
          </xsl:for-each>

          Major League Baseball Statistics
        </TITLE>
      </HEAD>
      <BODY>

        <xsl:for-each select="SEASON">
          <H1>
            <xsl:value-of select="@YEAR"/>
            Major League Baseball Statistics
          </H1>

          <xsl:for-each select="LEAGUE">
            <H2 ALIGN="CENTER">
              <xsl:value-of select="@NAME"/>
            </H2>

            <xsl:for-each select="DIVISION">
              <H3 ALIGN="CENTER">
              <xsl:value-of select="@NAME"/>
              </H3>

              <xsl:for-each select="TEAM">
                <H4 ALIGN="CENTER">
                <xsl:value-of select="@CITY"/>
                <xsl:value-of select="@NAME"/>
                </H4>

                <TABLE>
                 <CAPTION><B>Batters</B></CAPTION>
                 <THEAD>
                  <TR>
                   <TH>Player</TH><TH>P</TH><TH>G</TH>
                   <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
                   <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
                   <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
                   <TH>E</TH><TH>BB</TH><TH>SO</TH>
                   <TH>HBP</TH>
                  </TR>
                 </THEAD>
```

```
 <TBODY>
  <xsl:for-each select="PLAYER[(@POSITION
    != 'Starting Pitcher')
   $and$ (@POSITION != 'Relief Pitcher')]">
   <TR>
    <TD>
     <xsl:value-of select="@GIVEN_NAME"/>
     <xsl:value-of select="@SURNAME"/>
    </TD>
    <TD><xsl:value-of select="@POSITION"/></TD>
    <TD><xsl:value-of select="@GAMES"/></TD>
    <TD>
      <xsl:value-of select="@GAMES_STARTED"/>
    </TD>
    <TD><xsl:value-of select="@AT_BATS"/></TD>
    <TD><xsl:value-of select="@RUNS"/></TD>
    <TD><xsl:value-of select="@HITS"/></TD>
    <TD><xsl:value-of select="@DOUBLES"/></TD>
    <TD><xsl:value-of select="@TRIPLES"/></TD>
    <TD>
      <xsl:value-of select="@HOME_RUNS"/>
    </TD>
    <TD><xsl:value-of select="@RBI"/></TD>
    <TD><xsl:value-of select="@STEALS"/></TD>
    <TD>
     <xsl:value-of select="@CAUGHT_STEALING"/>
    </TD>
    <TD>
     <xsl:value-of select="@SACRIFICE_HITS"/>
    </TD>
    <TD>
     <xsl:value-of select="@SACRIFICE_FLIES"/>
    </TD>
    <TD><xsl:value-of select="@ERRORS"/></TD>
    <TD><xsl:value-of select="@WALKS"/></TD>
    <TD>
     <xsl:value-of select="@STRUCK_OUT"/>
    </TD>
    <TD>
     <xsl:value-of select="@HIT_BY_PITCH"/>
    </TD>
   </TR>
  </xsl:for-each>  <!— PLAYER —>
 </TBODY>
</TABLE>

<TABLE>
  <CAPTION><B>Pitchers</B></CAPTION>
  <THEAD>
   <TR>
    <TH>Player</TH><TH>P</TH><TH>G</TH>
    <TH>GS</TH><TH>W</TH><TH>L</TH><TH>S</TH>
```

Continued

Listing 5-7 *(continued)*

```
<TH>CG</TH><TH>SO</TH><TH>ERA</TH>
  <TH>IP</TH><TH>HR</TH><TH>R</TH><TH>ER</TH>
  <TH>HB</TH><TH>WP</TH><TH>B</TH><TH>BB</TH>
  <TH>K</TH>
 </TR>
</THEAD>
<TBODY>
 <xsl:for-each select="PLAYER[(@POSITION
  = 'Starting Pitcher')
  $or$ (@POSITION = 'Relief Pitcher')]">
  <TR>
   <TD>
    <xsl:value-of select="@GIVEN_NAME"/>
    <xsl:value-of select="@SURNAME"/>
   </TD>
   <TD><xsl:value-of select="@POSITION"/></TD>
   <TD><xsl:value-of select="@GAMES"/></TD>
   <TD>
     <xsl:value-of select="@GAMES_STARTED"/>
   </TD>
   <TD><xsl:value-of select="@WINS"/></TD>
   <TD><xsl:value-of select="@LOSSES"/></TD>
   <TD><xsl:value-of select="@SAVES"/></TD>
   <TD>
    <xsl:value-of select="@COMPLETE_GAMES"/>
   </TD>
   <TD>
    <xsl:value-of select="@SHUT_OUTS"/>
   </TD>
   <TD><xsl:value-of select="@ERA"/></TD>
   <TD><xsl:value-of select="@INNINGS"/></TD>
   <TD>
   <xsl:value-of select="@HOME_RUNS_AGAINST"/>
   </TD>
   <TD>
    <xsl:value-of select="@RUNS_AGAINST"/>
   </TD>
   <TD>
    <xsl:value-of select="@EARNED_RUNS"/>
   </TD>
   <TD>
    <xsl:value-of select="@HIT_BATTER"/>
   </TD>
   <TD>
     <xsl:value-of select="@WILD_PITCH"/>
   </TD>
   <TD><xsl:value-of select="@BALK"/></TD>
   <TD>
    <xsl:value-of select="@WALKED_BATTER"/>
   </TD>
   <TD>
```

```
                  <xsl:value-of select="@STRUCK_OUT_BATTER"/>
                  </TD>
                 </TR>
                </xsl:for-each>  <!— PLAYER —>
               </TBODY>
              </TABLE>

            </xsl:for-each> <!— TEAM —>
          </xsl:for-each> <!— DIVISION —>
      </xsl:for-each> <!— LEAGUE —>
    </xsl:for-each> <!— SEASON —>


      <HR></HR>
      Copyright 1999
      <A HREF="http://www.macfaq.com/personal.html">
       Elliotte Rusty Harold
      </A>
      <BR />
      <A HREF="mailto:elharo@metalab.unc.edu">
       elharo@metalab.unc.edu
      </A>

      </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

Figure 5-9: Pitchers are distinguished from other players after applying the XSL style sheet in Listing 5-7.

Element Contents and the select Attribute In this chapter, I focused on using XSL to format data stored in the attributes of an element because it isn't accessible when using CSS. However, XSL works equally well when you want to include an element's character data rather than (or in addition to) its attributes. To indicate that an element's text is to be copied into the output document, simply use the element's name as the value of the `select` attribute of the `xsl:value-of` element. For example, consider, once again, Listing 5-8:

```
Listing 5-8greeting.xml<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="greeting.xsl"?>
<GREETING>
Hello XML!
</GREETING>
```

Let's suppose you want to copy the greeting "Hello XML!" into an `H1` header. First, you use `xsl:for-each` to select the `GREETING` element:

```
<xsl:for-each select="GREETING">
   <H1>
   </H1>
</xsl:for-each>
```

This alone is enough to copy the two `H1` tags into the output. To place the text of the `GREETING` element between them, use `xsl:value-of` with no `select` attribute. Then, by default, the contents of the current element (`GREETING`) are selected. Listing 5-9 shows the complete style sheet.

Listing 5-9: **greeting.xsl**

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <HTML>
      <BODY>
        <xsl:for-each select="GREETING">
          <H1>
            <xsl:value-of/>
          </H1>
        </xsl:for-each>
      </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

You can also use `select` to choose the contents of a child element. Simply make the name of the child element the value of the `select` attribute of `xsl:value-of`. For instance, consider the baseball example from the previous chapter in which each player's statistics were stored in child elements rather than in attributes. Given this structure of the document (which is actually far more likely than the attribute-based structure of this chapter) the XSL for the batters' table looks like this:

```
<TABLE>
  <CAPTION><B>Batters</B></CAPTION>
  <THEAD>
   <TR>
    <TH>Player</TH><TH>P</TH><TH>G</TH>
    <TH>GS</TH><TH>AB</TH><TH>R</TH><TH>H</TH>
    <TH>D</TH><TH>T</TH><TH>HR</TH><TH>RBI</TH>
    <TH>S</TH><TH>CS</TH><TH>SH</TH><TH>SF</TH>
    <TH>E</TH><TH>BB</TH><TH>SO</TH><TH>HBP</TH>
   </TR>
  </THEAD>
 <TBODY>
  <xsl:for-each select="PLAYER[(POSITION
   != 'Starting Pitcher')
   $and$ (POSITION != 'Relief Pitcher')]">
   <TR>
    <TD>
     <xsl:value-of select="GIVEN_NAME"/>
     <xsl:value-of select="SURNAME"/>
    </TD>
    <TD><xsl:value-of select="POSITION"/></TD>
    <TD><xsl:value-of select="GAMES"/></TD>
    <TD>
      <xsl:value-of select="GAMES_STARTED"/>
    </TD>
    <TD><xsl:value-of select="AT_BATS"/></TD>
    <TD><xsl:value-of select="RUNS"/></TD>
    <TD><xsl:value-of select="HITS"/></TD>
    <TD><xsl:value-of select="DOUBLES"/></TD>
    <TD><xsl:value-of select="TRIPLES"/></TD>
    <TD><xsl:value-of select="HOME_RUNS"/></TD>
    <TD><xsl:value-of select="RBI"/></TD>
    <TD><xsl:value-of select="STEALS"/></TD>
    <TD>
     <xsl:value-of select="CAUGHT_STEALING"/>
    </TD>
    <TD>
     <xsl:value-of select="SACRIFICE_HITS"/>
    </TD>
    <TD>
     <xsl:value-of select="SACRIFICE_FLIES"/>
    </TD>
    <TD><xsl:value-of select="ERRORS"/></TD>
```

```
          <TD><xsl:value-of select="WALKS"/></TD>
          <TD>
           <xsl:value-of select="STRUCK_OUT"/>
          </TD>
          <TD>
           <xsl:value-of select="HIT_BY_PITCH"/>
          </TD>
         </TR>
        </xsl:for-each>  <!— PLAYER —>
       </TBODY>
      </TABLE>
```

In this case, within each PLAYER element, the contents of that element's GIVEN_NAME, SURNAME, POSITION, GAMES, GAMES_STARTED, AT_BATS, RUNS, HITS, DOUBLES, TRIPLES, HOME_RUNS, RBI, STEALS, CAUGHT_STEALING, SACRIFICE_HITS, SACRIFICE_FLIES, ERRORS, WALKS, STRUCK_OUT and HIT_BY_PITCH children are extracted and copied to the output. Since we used the same names for the attributes in this chapter as we did for the PLAYER child elements in the last chapter, this example is almost identical to the equivalent section of Listing 5-7. The main difference is that the @ signs are missing. They indicate an attribute rather than a child.

You can do even more with the select attribute. You can select elements: by position (for example the first, second, last, seventeenth element, and so forth); with particular contents; with specific attribute values; or whose parents or children have certain contents or attribute values. You can even apply a complete set
of Boolean logical operators to combine different selection conditions. We will explore more of these possibilities when we return to XSL in Chapters 14 and 15.

CSS or XSL?

CSS and XSL overlap to some extent. XSL is certainly more powerful than CSS. However XSL's power is matched by its complexity. This chapter only touched on the basics of what you can do with XSL. XSL is more complicated, and harder to learn and use than CSS, which raises the question, "When should you use CSS and when should you use XSL?"

CSS is more broadly supported than XSL. Parts of CSS Level 1 are supported for HTML elements by Netscape 4 and Internet Explorer 4 (although annoying differences exist). Furthermore, most of CSS Level 1 and some of CSS Level 2 is likely to be well supported by Internet Explorer 5.0 and Mozilla 5.0 for both XML and HTML. Thus, choosing CSS gives you more compatibility with a broader range of browsers.

Additionally, CSS is more stable. CSS level 1 (which covers all the CSS you've seen so far) and CSS Level 2 are W3C recommendations. XSL is still a very early working

draft, and probably won't be finalized until after this book is printed. Early adopters of XSL have already been burned once, and will be burned again before standards gel. Choosing CSS means you're less likely to have to rewrite your style sheets from month to month just to track evolving software and standards. Eventually, however, XSL will settle down to a usable standard.

Furthermore, since XSL is so new, different software implements different variations and subsets of the draft standard. At the time of this writing (spring 1999) there are at least three major variants of XSL in widespread use. Before this book is published, there will be more. If the incomplete and buggy implementations of CSS in current browsers bother you, the varieties of XSL will drive you insane.

However, XSLis definitely more powerful than CSS. CSS only allows you to apply formatting to element contents. It does not allow you to change or reorder those contents; choose different formatting for elements based on their contents or attributes; or add simple, extra text like a signature block. XSL is far more appropriate when the XML documents contain only the minimum of data and none of the HTML frou-frou that surrounds the data.

With XSL, you can separate the crucial data from everything else on the page, like mastheads, navigation bars, and signatures. With CSS, you have to include all these pieces in your data documents. XML+XSL allows the data documents to live separately from the Web page documents. This makes XML+XSL documents more maintainable and easier to work with.

In the long run XSL should become the preferred choice for real-world, data-intensive applications. CSS is more suitable for simple pages like grandparents use to post pictures of their grandchildren. But for these uses, HTML alone is sufficient. If you've really hit the wall with HTML, XML+CSS doesn't take you much further before you run into another wall. XML+XSL, by contrast, takes you far past the walls of HTML. You still need CSS to work with legacy browsers, but long-term XSL is the way to go.

Summary

In this chapter, you saw examples of creating an XML document from scratch. Specifically, you learned:

- ✦ Information can also be stored in an attribute of an element.
- ✦ An attribute is a name-value pair included in an element's start tag.
- ✦ Attributes typically hold meta-information about the element rather than the element's data.
- ✦ Attributes are less convenient to work with than the contents of an element.

- Attributes work well for very simple information that's unlikely to change its form as the document evolves. In particular, style and linking information works well as an attribute.
- Empty tags offer syntactic sugar for elements with no content.
- XSL is a powerful style language that enables you to access and display attribute data and transform documents.

In the next chapter, we'll specify the exact rules that well-formed XML documents must adhere to. We'll also explore some additional means of embedding information in XML documents including comments and processing instructions.