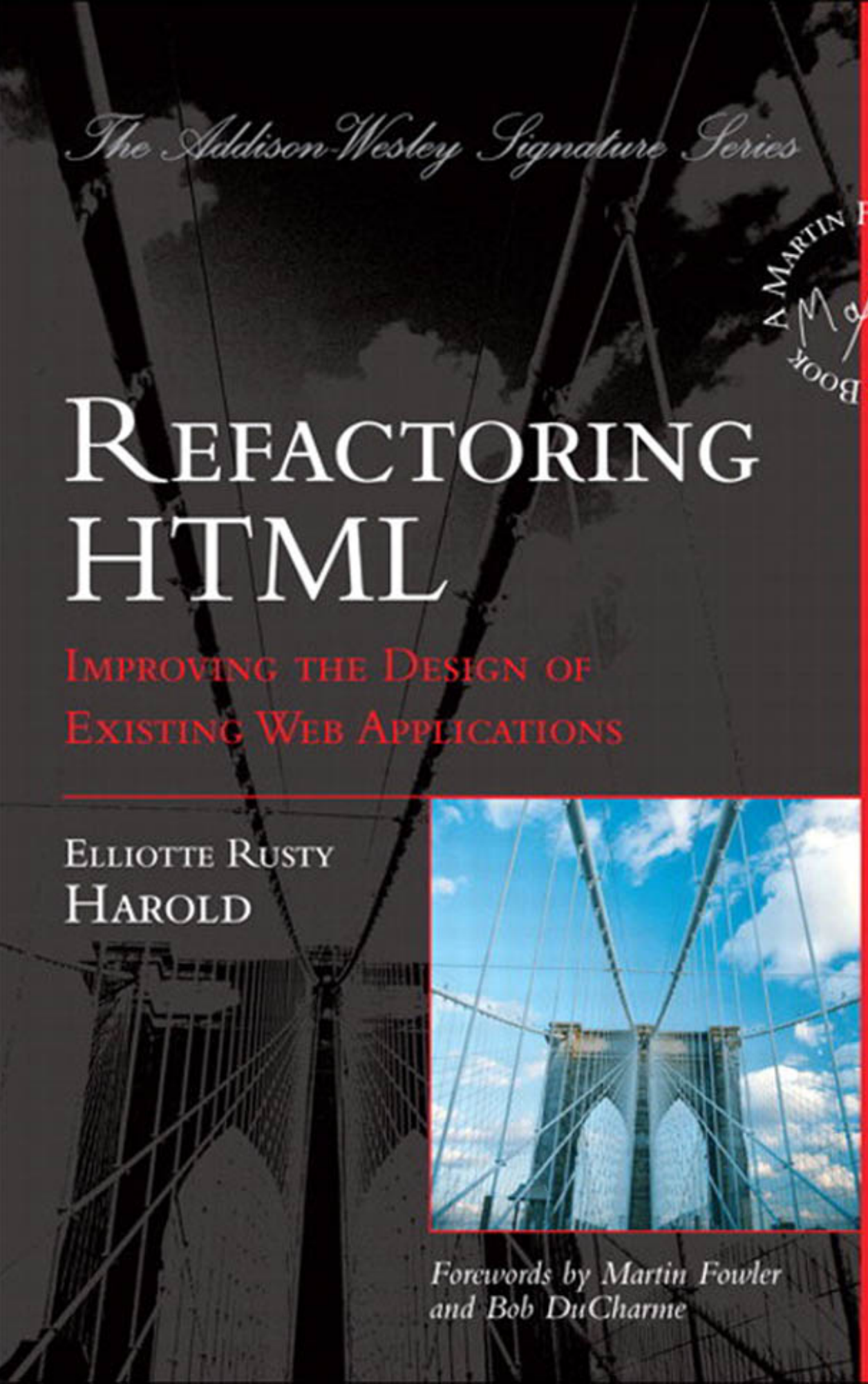

The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Refactoring HTML

Improving the Design of Existing Web Applications

Elliotte Rusty Harold

Forewords by Martin Fowler and Bob DuCharme

Library of Congress Cataloging-in-Publication Data

Harold, Elliotte Rusty.
Refactoring HTML : improving the design of existing Web applications / Elliotte Rusty Harold.
p. cm.
Includes index.
ISBN-13: 978-0-321-50363-3 (pbk. : alk. paper)
ISBN-10: 0-321-50363-5 (pbk. : alk. paper)
1. Web sites—Design. 2. Software refactoring. 3.Web servers—Computer programs. 4. Application software—Development. 5. HTML editors (Computer programs) I. Title.

TK5105.888.H372 2008
006.7'4—dc22 2008008645

ISBN-13: 978-0-321-50363-3
ISBN-10: 0-321-50363-5

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, April 2008

Foreword by Martin Fowler

In just over a decade the Web has gone from a technology with promise to major part of the world's infrastructure. It's been a fascinating time, and many useful resources have been built in the process. But, as with any technology, we've learned as we go how best to use it and the technology itself has matured to help us use it better.

However complex a web application, it finally hits the glass in the form of HTML—the universal web page description language. HTML is a computer language, albeit a very limited and specialized one. As such, if you want a system that you can evolve easily over time, you need to pay attention to writing HTML that is clear and understandable. But just like any computer language, or indeed any writing at all, it's hard to get it right first time. Clear code comes from writing and rewriting with a determination to create something that is easy to follow.

Rewriting code carries a risk of introducing bugs. Several years ago, I wrote about a technique called refactoring, which is a disciplined way of rewriting code that can greatly reduce the chances of introducing bugs while reworking software. Refactoring has made a big impact on regular software languages. Many programmers use it as part of their daily work to help them keep code clear and enhance their future productivity. Tools have sprung up to automate refactoring tasks, to further improve the workflow.

Just as refactoring can make a big improvement to regular programming, the same basic idea can work with HTML. The refactoring steps are different, but the underlying philosophy is the same. By learning how to refactor your HTML, you can keep your HTML clean and easy to change into the future, allowing you to make the inevitable changes more quickly. These techniques can also allow you to bring web sites into line with the improvements in web technologies, specifically allowing you to move toward supporting XHTML and CSS.

Elliotte Rusty Harold has long had a permanent place on my bookshelf for his work on XML technologies and open source software for XML processing. I've always respected him as a fine programmer and writer. With this book he brings the benefits of refactoring into the HTML world.

—Martin Fowler

Foreword by Bob DuCharme

A key to the success of the World Wide Web has always been the ease with which just about anyone can create a web page and put it where everyone can see it. As people create sets of interlinked pages, their web sites become more useful to a wider audience, and stories of web millionaires inspire these web developers to plan greater things.

Many find, though, that as their web sites get larger, they have growing pains. Revised links lead to nowhere, pages look different in different browsers, and it becomes more difficult to keep track of what's where, especially when trying to apply changes consistently throughout the site. This is when many who built their own web site call in professional help, but now with *Refactoring HTML,* you can become that professional. And, if you're already a web pro, you can become a better one.

There are many beginner-level introductions to web technologies, but this book is the first to tie together intermediate-level discussions of all the key technologies for creating professional, maintainable, accessible web sites. You may already be an expert in one or two of the topics covered by this book, but few people know all of them as well as Elliotte, and he's very good at explaining them. (I know XML pretty well, but this book has shown me that some simple changes to my CSS habits will benefit all of the web pages I've created.)

For each recommendation in the book, Elliotte lays out the motivation for why it's a good idea, the potential trade-offs for following the recommendation, and the mechanics of implementing it, giving you a full perspective on the how and why of each tip. For detecting problems, I'll stop short of comparing his use of smell imagery with Proust's, but it's pretty evocative nevertheless.

I've read several of Elliotte's books, but not all of them. When I heard that *Refactoring HTML* was on the way, I knew right away that I'd want to read it, and I was glad to get an advanced look. I learned a lot, and I know that you will, too.

—Bob DuCharme
Solutions Architect, Innodata Isogen

Chapter 3
Well-Formedness

The very first step in moving markup into modern form is to make it well-formed. Well-formedness is the basis of the huge and incredibly powerful XML tool chain. Well-formedness guarantees a single unique tree structure for the document that can be operated on by the DOM, thus making it the basis of reliable, cross-browser JavaScript. The very first thing you need to do is make your pages well-formed.

Validity, although important, is not nearly as crucial as well-formedness. There are often good reasons to compromise on validity. In fact, I often deliberately publish invalid pages. If I need an element the DTD doesn't allow, I put it in. It won't hurt anything because browsers ignore elements they don't understand. If I have a blockquote that contains raw text but no elements, no great harm is done. If I use an HTML 5 element such as `m` that Opera recognizes and other browsers don't, those other browsers will just ignore it. However, if the page is malformed, the consequences are much more severe.

First, I won't be able to use any XML tools, such as XSLT or SAX, to process the page. Indeed, almost the only thing I can do with it is view it in a browser. It is very hard to do any reliable automated processing or testing with a malformed page.

Second, browser display becomes much more unpredictable. Different browsers fill in the missing pieces and correct the mistakes of malformed pages in different ways. Writing cross-platform JavaScript or CSS is hard enough without worrying about what tree each browser will construct from ambiguous HTML. Making the page well-formed makes it a lot more likely that I can make it behave as I like across a wide range of browsers.

What Is Well-Formedness?

Well-formedness is a concept that comes from XML. Technically, it means that a document adheres to certain rigid constraints, such as every start-tag has a matching end-tag, elements begin and end in the same parent element, and every entity reference is defined.

Classic HTML is based on SGML, which allows a lot more leeway than does XML. For example, in HTML and SGML, it's perfectly OK to have a `<br>` or `<li>` tag with no corresponding `</br>` and `</li>` tags. However, this is no longer allowed in a well-formed document.

Well-formedness ensures that every conforming processor treats the document in the same way at a low level. For example, consider this malformed fragment:

```
<p>The quick <strong>brown fox</p>
jumped over the
<p>lazy</strong> dog.</p>
```

The `strong` element begins in one paragraph and ends in the next. Different browsers can and do build different internal representations of this text. For example, Firefox and Safari fill in the missing start- and end-tags (including those between the paragraphs). In essence, they treat the preceding fragment as equivalent to this markup:

```
<p>The quick <strong>brown fox</strong></p>
<strong>jumped over the </strong>
<p><strong>lazy</strong> dog.</p>
```

This creates the tree shown in Figure 3.1.

By contrast, Opera places the second `p` element inside the `strong` element which is inside the first `p` element. In essence, the Opera DOM treats the fragment as equivalent to this markup:

```
<p>The quick
  <strong>brown fox jumped over the
    <p>lazy dog.</p>
  </strong>
</p>
```

This builds the tree shown in Figure 3.2.

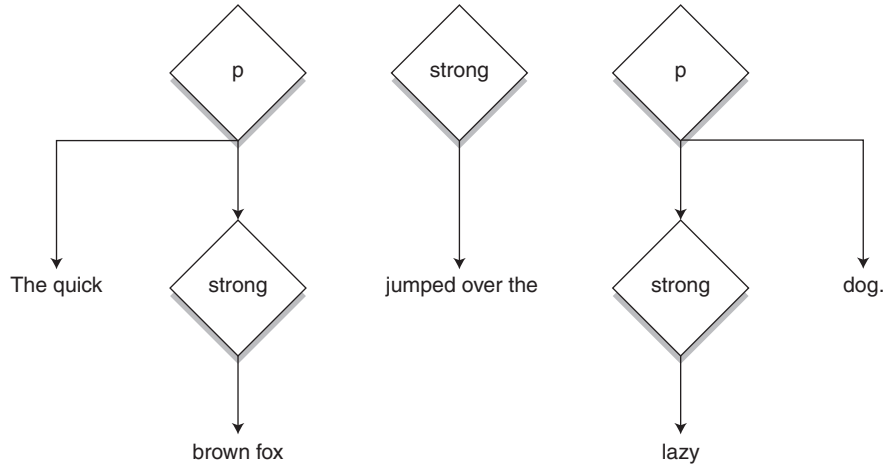

FIGURE 3.1 An overlapping tree as interpreted by Firefox and Safari

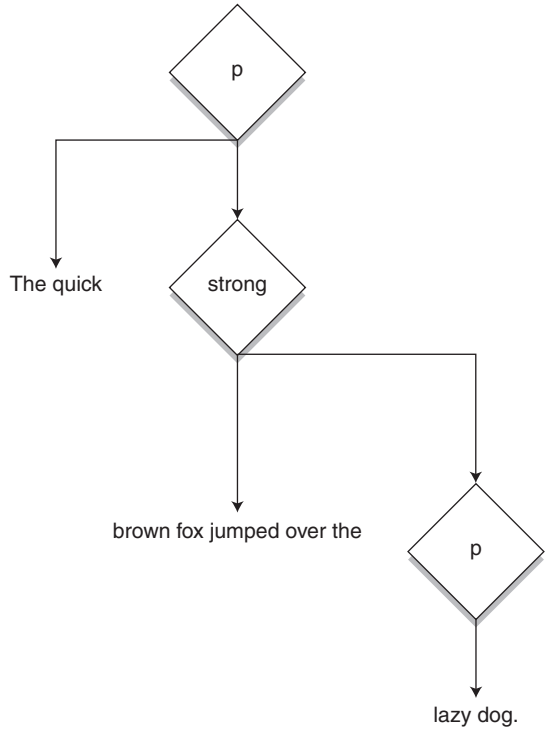

FIGURE 3.2 An overlapping tree as interpreted by Opera

If you've ever struggled with writing JavaScript code that works the same across browsers, you know how annoying these cross-browser idiosyncrasies can be.

By contrast, a well-formed document removes the ambiguity by requiring all the end-tags to be filled in and all the elements to have a single unique parent. Here is the well-formed markup corresponding to the preceding code:

```
<p>…foo<strong>…</strong></p> <p><strong>…bar</strong> </p>
```

This leaves no room for browser interpretation. All modern browsers build the same tree structure from this well-formed markup. They may still differ in which methods they provide in their respective DOMs and in other aspects of behavior, but at least they can agree on what's in the HTML document. That's a huge step forward.

Anything that operates on an HTML document, be it a browser, a CSS stylesheet, an XSL transformation, a JavaScript program, or something else, will have an easier time working with a well-formed document than the malformed alternative. For many use cases such as XSLT, this may be critical. An XSLT processor will simply refuse to operate on malformed input. You must make the document well-formed before you can apply an XSLT stylesheet to it.

Most web sites will need to make at least some and possibly all of the following fixes to become well-formed.

- Every start-tag must have a matching end-tag.
- Empty elements should use the empty-element tag syntax.
- Every attribute must have a value.
- Every attribute value must be quoted.
- Every raw ampersand must be escaped as `&`.
- Every raw less-than sign must be escaped as `<`.
- There must be a single root element.
- Every nonpredefined entity reference must be declared in the DTD.

In addition, namespace well-formedness requires that you add an `xmlns="http://www.w3.org/1999/xhtml"` attribute to the root `html` element.

Although it's easy to find and fix some of these problems manually, you're unlikely to catch all of them without help. As discussed in the preceding chapter, you can use xmllint or other validators to check for well-formedness. For example:

```
$ xmllint --noout --loaddtd http://www.aw.com
http://www.aw-bc.com/:118: parser error : Specification
mandate value for attribute nowrap
<TD class="headerBg" bgcolor="#004F99" nowrap align="left">
                                                ^
http://www.aw-bc.com/:118: parser error : attributes construct error
<TD class="headerBg" bgcolor="#004F99" nowrap align="left">
                                                ^
http://www.aw-bc.com/:118: parser error : Couldn't find end
of Start-tag TD line 118
<TD class="headerBg" bgcolor="#004F99" nowrap align="left">
                                                ^
…
```

TagSoup or Tidy can handle many of the necessary fixes automatically. However, they don't always guess right, so it pays to at least spot-check some of the problems manually before fixing them. Usually it's simplest to fix as many broad classes of errors as possible. Then run xmllint again to see what you've missed.

The following sections discuss the mechanics and trade-offs of each of these changes as they usually apply in HTML.

Change Name to Lowercase

Make all element and attribute names lowercase. Make most entity names lowercase, except for those that refer to capital letters.

```
<BLOCKQUOTE CITE=
  'http://www.gutenberg.org/dirs/etext00/dvlft10.txt'>
<P>
```

(continued)

```
It was, then, with <EM>considerable</EM> surprise that I
received a telegram from Holmes last Tuesday&MDASH;he has
never been known to write where a telegram would
serve&MDASHin the following terms:
</P>

<P>
Why not tell them of the Cornish horror&MDASH;strangest
case I have handled.
</P>
</BLOCKQUOTE>
```

```
<blockquote cite=
  'http://www.gutenberg.org/dirs/etext00/dvlft10.txt'>
<p>
It was, then, with <em>considerable</em> surprise that I
received a telegram from Holmes last Tuesday&mdash;he has
never been known to write where a telegram would
serve&mdash;in the following terms:
</p>

<p>
Why not tell them of the Cornish horror&MDASH;strangest
case I have handled.
</p>
</blockquote>
```

Motivation

XHTML uses lowercase names exclusively. All elements and attributes are written in lowercase. For example, `<table>` is recognized but not `<TABLE>` or `<Table>`. In XHTML mode, lowercase is required.

Entity Names

Entity names are sometimes case-sensitive, even in classic HTML. For instance, `&eacute;` resolves to é, but `&Eacute;` resolves to É. It's important to get these right, too, as an XHTML browser will recognize `&eacute;` and `&Eacute;` but not `&EAcute;` or `&EACUTE;`. Even a browser operating in HTML mode can guess wrong if you don't have the right case in the entity reference.

Generic XML tools don't care about case but do care that it matches. That is, a `<table>` start-tag is closed by a `</table>` end-tag but not by `</TABLE>` or `</Table>`. The `id` attribute has the type ID as defined in the XHTML DTD and can be used as a link anchor. However, the `id` attribute does not and cannot.

Potential Trade-offs

There are relatively few trade-offs for converting to lowercase. All modern browsers support lowercase tag names without any problems. A few *very* old browsers that were never in widespread use, such as HotJava, only supported uppercase for some tags. The same is true of early versions of Java Swing's built-in HTML renderer. However, this has long since been fixed.

It is also possible that some homegrown scripts based on regular expressions may not recognize lowercase forms. If you have any scripts that screen-scrape your HTML, you'll need to check them to make sure they're also ready to handle lowercase tag names. Once you're done making the document well-formed, it may be time to consider refactoring those scripts, too, so that they use a real parser instead of regular expression hacks. However, that can wait. Usually it's simple enough to change the expressions to look for lowercase tag names instead of uppercase ones, or to not care about the case of the tag names at all.

Mechanics

The first rule of well-formedness is that every start-tag has a matching end-tag. The *matching* part is crucial. Although classic HTML is case-insensitive, XML and XHTML are not. `<DIV>` is not the same as `<div>` and a `</div>` end-tag cannot close a `<DIV>` start-tag.

For purely well-formedness reasons, all that's needed is to normalize the case. All tags could be capitalized or not, as long as you're consistent. However, it's easiest for everyone if we pick one case convention and stick to it. The community has chosen lowercase for XHTML. Thus,

the first step is to convert all tag names, attribute names, and entity names to lowercase. For example:

- <P> to <p>
- <Table> to <table>
- </DIV> to
- <BLOCKQUOTE CITE="http://richarddawkins.net/article,372,n,n"> to <blockquote cite="http://richarddawkins.net/article,372,n,n">
- © to ©

There are several ways to do this.

The first and the simplest is to use TagSoup or Tidy in XHTML mode. Along with many other changes, these tools will convert all tag and attribute names to lowercase. They will also change entity names that need to be in lowercase.

You also can accomplish this with regular expressions. Because HTML element and attribute names are composed exclusively of the Latin letters *A* to *Z* and *a* to *z,* this isn't too difficult. Let's start with the element names. There are likely to be thousands, perhaps millions, of these, so you don't want to fix them by hand.

Tags are easy to search for. This regular expression will find all start-tags that contain at least one capital letter:

```
<[a-zA-Z]*[A-Z]+[a-zA-Z]*
```

This regular expression will find all end-tags that contain at least one capital letter:

```
</[a-zA-Z]*[A-Z]+[a-zA-Z]*>
```

Entities are also easy. This regular expression finds all entity references that contain a capital letter other than the initial letters:

```
&[A-Za-z] [A-Za-z] [A-Z]+[A-Za-z]*;
```

I set up the preceding regular expression to find at least three capital letters to avoid accidentally triggering on references such as Ω that should have a single initial capital letter and on references such as

Æ that have two initial capital letters. This may miss some cases, such as &Amp; and &AMp;, but those are rare in practice. Usually entity references are either all uppercase or all lowercase. If any such mixed cases exist, we'll find them later with xmllint and fix them by hand.

Attributes are trickier to find because the pattern to find them (=name) may appear inside the plain text of the document. I much prefer to use Tidy or TagSoup to fix these. However, if you know you have a large problem with particular attributes, it's easy to do a search and replace for individual ones—for instance, HREF= to href=. As long as you aren't writing about HTML, that string is unlikely to appear in plain text content.

Sometimes your initial find will discover that only a few tags use uppercase. For instance, if there are lots of uppercase table tags, you can quickly change <TD> to <td>, </TD> to </td>, <TR> to </tr>, and so forth without even using regular expressions. If the problem is a little broader, consider using Tidy or TagSoup. If that doesn't work, you'll need a tool that can replace text while changing its case. jEdit can't do this. However, Perl and BBEdit can. Use \L in the replacement pattern to convert all characters to lowercase. For example, let's start with the regular expression for start-tags:

```
(<[a-zA-Z]*[A-Z]+[a-zA-Z]*)
```

This expression will replace it with its lowercase equivalent:

```
\L\1
```

Quote Attribute Value

Put quotes around all attribute values.

```
<div id=speech1>
<span class=speaker>PROSPERO</span>
<blockquote cite=
http://www-tech.mit.edu/Shakespeare/tempest/tempest.4.1.html>
```

(continued)

```
<span class=verse id=a4s1v1>If I have too austerely
punish'd you,</span>
<span class=verse id=a4s1v2>Your compensation makes amends,
for I</span>
<span class=verse id=a4s1v3>Have given you here a third
of mine own life,</span>
<span class=verse id=a4s1v4>Or that for which I live;
who once again</span>
<span class=verse id=a4s1v5>I tender to thy hand:
all thy vexations</span>
<span class=verse id=a4s1v6>Were but my trials of
 thy love and thou</span>
<span class=verse id=a4s1v7>Hast strangely stood the
 test here, afore Heaven,</span>
</blockquote>
</div>
```

```
<div id="speech1">
<span class="speaker">PROSPERO</span>
<blockquote cite=
  "http://www-tech.mit.edu/Shakespeare/tempest/tempest.4.1.html">
<span class="verse" id="a4s1v1">If I have too austerely
punish'd you,</span>
<span class="verse" id="a4s1v2">Your compensation makes amends,
for I</span>
<span class="verse" id="a4s1v3">Have given you here a third
of mine own life,</span>
<span class="verse" id="a4s1v4">Or that for which I live;
who once again</span>
<span class="verse" id="a4s1v5">I tender to thy hand:
all thy vexations</span>
<span class="verse" id="a4s1v6">Were but my trials of
 thy love and thou</span>
<span class="verse" id="a4s1v7">Hast strangely stood the
 test here, afore Heaven,</span>
</blockquote>
</div>
```

Motivation

In XHTML, all attribute values are quoted, even those that don't contain whitespace.

Potential Trade-offs

Absolutely no browsers are in the least bit confused by a properly quoted attribute value.

This can add roughly two bytes per attribute value to the file size. If you're Google and are counting every byte on your home page because you serve gigabytes per second, this may matter. This should not concern anybody else.

Mechanics

Manually, all you have to do is type a single or double quote before and after the attribute value. For example, consider this start-tag:

```
<a class=q href=http://www.example.com>
```

You simply turn that into this:

```
<a class="q" href="http://www.example.com">
```

Or this:

```
<a class='q' href='http://www.example.com'>
```

There's no reason to prefer single or double quotes. Use whichever one you like. Mechanically, both Tidy and TagSoup will fill these quotes in for you. It's probably easiest to let them do the work.

Regular expressions are a little tricky because you also need to consider the case where there's whitespace around the equals sign. For instance, you don't just have to handle the preceding examples. You have to be ready for this:

```
<a class = q href = http://www.example.com>
```

And even this:

```
<a class=
     q href
     = http://www.example.com>
```

Finding the cases without whitespace is not too hard. This will do it:

```
[a-zA-Z]+=[^'"><\s]+
```

However, the preceding code snippet will also find lots of false positives. For instance, it will find this tag because of `item=15314` in the query string:

```
<a href="http://www.cybout.com/cgi-bin/product_info?item=15314">
```

We can improve this a little bit by requiring whitespace before the name, like so:

```
\w+[a-zA-Z]+=[^'"><\s]+
```

You may discover a few cases where the attribute value contained whitespace and was not quoted. Similarly, you may find a few places where the initial quote is present but the closing quote is not. These are problematic, and you need to fix them. Browsers do not always interpret these as you might expect, and different browsers handle them differently. What makes no difference in Internet Explorer may cause Firefox to hide content and vice versa.

Fill In Omitted Attribute Value

Add values to all valueless attributes.

```
<input type="radio" name="p" value="debit" checked></input>
<input name="generator" value="system78" readonly></input>
<input name="date" value="2007-12-17" disabled></input>
<a href="http://example.com/imagemap/library">
<img src="duane.png" ismap></a>
```

```
<input type="radio" name="p" value="debit" checked="checked" />
<input name="generator" value="system78" readonly='readonly' /
<input name="date" value="2007-12-17" disabled="disabled"></input>
<a href="http://example.com/imagemap/library">
<img src="duane.png" ismap="ismap"></a>
```

Motivation

XHTML does not support the attribute name-only syntax.

Potential Trade-offs

Minimal. Browsers are perfectly happy to read the values you supply.

Mechanics

Omitted attribute values are fairly rare in practice. The only place they're at all common is in forms and image maps. It may well be possible to manually fix all occurrences on a site without a great deal of effort. Alternatively, you can just use Tidy or TagSoup.

To fix one, just set the attribute value to the name. For example, change this:

```
<input type="radio" name="p" checked>
```

into this:

```
<input type="radio" name="p" checked='checked'>
```

Only a few elements and attributes support valueless attributes in the first place:

- `input`: `checked`, `disabled`, `readonly`, `ismap`
- `optgroup`: `disabled`
- `option`: `selected`, `disabled`
- `textarea`: `disabled`, `readonly`
- `button`: `disabled`
- `script`: `defer`
- `img`: `ismap`, `controls`
- `area`: `nohref`
- `dl`: `compact`
- `ul`: `compact`
- `ol`: `compact`
- `ul`: `compact`, `plain`

- frame: noresize
- table: border
- marquee: truespeed
- link: disabled
- style: disabled
- applet: mayscript
- select: disabled, multiple
- object: declare

Because many of the words involved can appear in plain text, it is not safe to use regular expressions to replace these. However, you can use a simple search to find them and then verify and fix them manually. For the attributes that aren't actually English words, such as ismap and mayscript, just search for those words. For the attributes that are, such as border and compact, use a regular expression such as this:

```
<.*\s+compact\s+.*>
```

Then search for the case where the valueless attribute is the last attribute:

```
<.*\s+compact\s*>
```

Replace Empty Tag with Empty-Element Tag

Change elements such as <br> to <br class='empty' />.

```
Polonius<hr>
You shall do marvelous wisely, good Reynaldo,<br>
Before you visit him, to make inquire<br>
Of his behavior.<br>
```

```
Polonius<hr class='empty' />
You shall do marvelous wisely, good Reynaldo,<br class='empty' />
Before you visit him, to make inquire<br class='empty' />
Of his behavior. <br class='empty' />
```

Motivation

XML parsers require that every start-tag have a matching end-tag. There can be no without a corresponding . Similarly, there can be no <br> without a corresponding </br>. Alternatively, you can use empty-element tag syntax, such as <br/> and <hr/>. This is usually simpler for elements that are guaranteed to be empty and more compatible with legacy browsers.

Potential Trade-offs

Although most modern browsers have no problem with empty-element tags, a few older ones you'll still find installed here or there, such as Netscape 3, do. For example, some will treat <br/> as an element whose name is br/ and will not insert the necessary break. Others will take
</br> as a double break, rather than a single break. The content will still be present, but it may not be styled properly.

Mechanics

Classic HTML defines 12 empty elements:

-
- <hr>
- <meta>
- <link>
- <base>
- <img>
- <embed>
- <param>
- <area>
- <frame>
- <col>
- <input>

In addition, a few other elements from various proprietary browser extensions may also appear:

- `<basefont>`
- `<bgsound>`
- `<keygen>`
- `<sound>`
- `<spacer>`
- `<wbr>`

Although XML and XHTML allow these tags to be written either with a start-tag/end-tag pair such as `<br></br>` or with an empty-element tag such as `<br />`, the latter is much friendlier to older browsers and to human authors. There's little reason not to prefer the empty-element tag.

However, even an empty-element tag such as `<br/>` can confuse some older browsers that actually read this as an unknown element with the name `br/` instead of the known element `br`. Maximum compatibility is achieved if you add an attribute and a space before the final slash. The `class` attribute is a good choice. For example:

```
<br class="empty" />
<hr class="empty" />
```

I picked `empty` as the class to be clear why I inserted it. However, the value of the `class` attribute really doesn't matter. If you have reason to assign a different class to some or all of these elements, feel free.

TagSoup and Tidy will convert these elements as part of their fix-up. However, neither adds the `class="empty"` attributes. You can add those with an extra search and replace step at the end, or you can just make the entire change with search and replace. I would start with the `<br/>` element. You can simply search for all `<br>` tags and replace them with `<br class="empty" />`.

However, there are a few things to watch out for. The first is whether someone has already done this. Check to see whether there are any `</br>` elements in the source. If any appear, first remove them, as they're no longer necessary.

The remaining concern is `br` tags with attributes, such as `<br clear="all">`. You can find these by searching for "`<br`."

If there aren't too many of these, I might just open the files and fix them manually. If there are a lot of them, you can automate the process, but this will require a slightly more complicated regular expression. The search expression is:

```
<br\s+([^>]*)=([^>]+[^/])>
```

The replace expression is:

```
<br \1=\2 />
```

When you're done, run your test suite to make sure all is well and you haven't accidentally broken something.

The `hr` element is handled almost identically. The `meta` and `link` elements are trickier because they almost always have attributes, so you need to use the more complicated form of the regular expressions. Of course, Tidy and TagSoup are also options.

Add End-tag

Close all paragraphs, list items, table cells, and other nonempty elements.

```
It is intended to include all the industries of the United
States concerned in French trade under the following
classifications:<p>

<ol>
<li>Machine-Tools, Wire, Transmission and Textiles
<li>Milling Machinery
<li>Electrical Apparatus
<li>Transportation
<li>Importers
<li>Synthetic Products based on chemical processes
```

(continued)

```
<li>Bankers
<li>Factory Architects, Engineers and Contractors
</ol>
```

```
<p>It is intended to include all the industries of the
United States concerned in French trade under the following
classifications:</p>

<ol>
<li>Machine-Tools, Wire, Transmission and Textiles</li>
<li>Milling Machinery</li>
<li>Electrical Apparatus</li>
<li>Transportation</li>
<li>Importers</li>
<li>Synthetic Products based on chemical processes</li>
<li>Bankers</li>
<li>Factory Architects, Engineers and Contractors</li>
</ol>
```

Motivation

The first motivation is simply XML compatibility. XML parsers require that each start-tag be matched by a corresponding end-tag.

However, there's a strong additional reason. Many documents do not display as intended in classic HTML when the end-tags are omitted. The problem is not that the browsers do not know how or where to insert end-tags. It's that authors often do not arrange the tags properly. All too often, the boundaries of an unclosed HTML element do not fall where the author expects. The result can be a document that appears quite different from what is expected. Indentation problems are the most common symptom (elements are not indented that should be, or elements are indented too far). However, all sorts of display problems can result. CSS is extremely hard to create and debug in the face of improperly closed elements.

Potential Trade-offs

Few and minimal. The resultant documents may be slightly larger. If you're not serving gigabytes per day, this is not worth worrying about.

Mechanics

Manually, you simply need to inspect each file and determine where the end-tags belong. For example, consider this table modeled after one in the HTML 4 specification:

```
<table>
<tr>
  <th rowspan="2">
  <th colspan="2">Average
  <th rowspan="2">Blond Hair
<tr><th>Height<th>Weight
<tr><th>Boys<td>1.4<td>58<td>28%
<tr><th>Girls<td>1.3<td>34.5<td>17%
</table>
```

Only the `</table>` end-tag is present. All the other end-tags are implied. A browser can probably figure this out. A human author might not and is likely to insert new content in the wrong place. Add end-tags after each element, like so:

```
<table>
  <tr>
    <th rowspan="2"></th>
    <th colspan="2">Average</th>
    <th rowspan="2">Blond Hair</th>
  </tr>
  <tr>
    <th>Height</th>
    <th>Weight</th>
  </tr>
  <tr>
    <th>Boys</th>
    <td>1.4</td>
    <td>58</td>
    <td>28%</td>
  </tr>
  <tr>
    <th>Girls</th>
    <td>1.3</td>
    <td>34.5</td>
    <td>17%</td>
</table>
```

Paragraphs are worth special attention here. When paragraph tags are omitted, the `<p>` start-tag usually serves as an end-tag rather than a

start-tag. You'll commonly see content such as this tidbit from *Through the Looking Glass:*

```
Alice didn't like this idea at all:  so, to change the
subject, she asked 'Does she ever come out here?'
<p>

'I daresay you'll see her soon,' said the Rose.  'She's one
of the thorny kind.'
<p>

'Where does she wear the thorns?' Alice asked with some
curiosity.
<p>
```

When encountering text such as this, you'll want to turn each into a , and then add the missing start-tags like so:

```
<p>Alice didn't like this idea at all:  so, to change the
subject, she asked 'Does she ever come out here?'
</p>

<p>'I daresay you'll see her soon,' said the Rose.
'She's one of the thorny kind.'
</p>

<p>'Where does she wear the thorns?' Alice asked with some
curiosity.
</p>
```

Tidy and TagSoup can fix this. However, they usually incorrectly guess the proper location of the start-tag and produce markup such as this:

```
Alice didn't like this idea at all:  so, to change the
subject, she asked 'Does she ever come out here?'
<p>

'I daresay you'll see her soon,' said the Rose.
'She's one of the thorny kind.'
</p>

<p>'Where does she wear the thorns?' Alice asked with some
curiosity.
</p>
<p>
</p>
```

Tidy doesn't add the closing empty paragraph, but it still fails to find the start of the first paragraph. You can tell Tidy to wrap para-

graphs around orphan text blocks using the `--enclose-block-text` option with the value `y`:

```
$ tidy -asxhtml --enclose-block-text y  endtag.html
```

This doesn't matter for basic browser display, but it matters a great deal if you've assigned any specific CSS style rules to the `p` element. Furthermore, it can apply special formatting intended for the first paragraph of a chapter or section to the second instead.

Usually this happens only to the first paragraph in a section. However, if the runs of paragraphs are interrupted by a `div`, `table`, `blockquote`, or other element, there is likely such a block after each such block-level element.

Consequently, after running TagSoup over a page, search for empty paragraphs. Anytime you find one, it means there's probably a paragraph-less block of text earlier in the document that you should enclose in a new `p` element. However, this is tricky because often the start-tag and end-tag are on different lines. The following regular expression will find most occurrences:

```
<p>\s*</p>
```

This expression will find any empty paragraphs that have attributes:

```
<p\s[^>]*>\s*</p>
```

However, such paragraphs weren't created by Tidy or TagSoup, so you'll probably want to leave them in.

Remove Overlap

Close every element within its parent element.

```
This is <strong><em>very important</strong></em>!
<p>Sarah answered, <q>I'm really not sure about this.</p>
<p>Maybe you should ask somebody else?</q> Then she
```

(continued)

```
sat down.
</p>
```

```
This is <strong><em>very important</em></strong>!
<p>Sarah answered, <q>I'm really not sure about this.</q>
</p>
<p><q>Maybe you should ask somebody else?</q> Then she
sat down.
</p>
```

Motivation

Different browsers do not build the same trees from documents containing overlapping elements. Consequently, JavaScript can work very differently than you expect between browsers.

Furthermore, small changes in a document with overlap can make radical changes in the DOM tree that even a single browser builds. Consequently, JavaScript built on top of such documents is fragile. CSS is likewise fragile. JavaScript, CSS, and other programs that read a document's DOM are hard to create, debug, and maintain in the face of overlapping elements.

Potential Trade-offs

Sometimes the nature of the text really does call for overlap—for instance, when a quote begins in one paragraph and ends in another. This comes up frequently in Biblical scholarship, for instance. Not all text fits neatly into a tree.

Unfortunately, HTML, XML, and XHTML cannot handle overlap in any reasonable fashion. If you're doing scholarly textual analysis, you may need something more powerful still. However, this is rarely a concern for simple web publication. You can usually hack around the problem well enough for browser display by using more elements than may logically be called for.

Mechanics

A validator will report all areas where overlap is a problem. However, overlap is so confusing to tools that they may not diagnose it properly or in an obvious fashion. Different validators will report problems in different locations, and a single validator may report several errors related to one occurrence. Sometimes the problem will be indicated as an unclosed element or an end-tag without a start-tag, or both. For example:

```
overlap.html:10: parser error : Opening and ending tag
mismatch: q line 10 and p
<p>Sarah answered, <q>I'm really not sure about this.</p>
                                                      ^
overlap.html:11: parser error : Opening and ending tag
mismatch: p line 11 and q
<p>Maybe you should ask somebody else?</q> Then she
```

Furthermore, an overlap problem may cause a parser to miss the starts or ends of other elements, and it may not be able to recover. It is very common for overlap to cause a cascading sequence of progressively more serious errors for the rest of the document. Thus, you should start at the beginning and fix one error at a time. Often, fixing an overlap problem eliminates many other error messages.

Repairing overlap is not hard. Sometimes the overlap is trivial, as when the end-tag for the parent element immediately precedes the end-tag for the child element. Then you just have to swap the end-tags. For example, change this:

```
<strong><em>very important</strong></em>
```

to this:

```
<strong><em>very important</em></strong>
```

If the overlap extends into another element, you close the overlapping element inside its first parent and reopen it in the last. For example, suppose you have these two paragraphs containing one quote:

```
<p>Sarah answered, <q>I'm really not sure about this.</p>
<p>Maybe you should ask somebody else?</q> Then she
sat down.</p>
```

Change them to two paragraphs, each containing a quote:

```
<p>Sarah answered,
  <q>I'm really not sure about this.</q>
</p>
<p>
  <q>Maybe you should ask somebody else?</q>
  Then she sat down.
</p>
```

If there are intervening elements, you'll need to create new elements inside those as well.

Tidy and TagSoup can fix technical overlap problems but not especially well, and the result is usually not what you would expect. For example, Tidy will not always reopen an overlapping element inside the next element. For instance, it turns this:

```
<p>Sarah answered, <q>I'm really not sure about this.</p>
<p>Maybe you should ask somebody else?</q> Then she
sat down.</p>
```

into this:

```
<p>Sarah answered, <q>I'm really not sure about this.</p>
<p>Maybe you should ask somebody else? Then she
sat down.</p>
```

It completely loses the quote in the second paragraph. TagSoup keeps the quote in the second paragraph but introduces a quote around the boundary whitespace between the two paragraphs:

```
<p>Sarah answered, <q>I'm really not sure about this.</p>
<q></q>
<p><q>Maybe you should ask somebody else?</q> Then she
sat down.</p>
```

Consequently, I prefer to fix these overlap problems by hand if there aren't too many of them. You're more likely to reproduce the original intent that way.

Convert Text to UTF-8

Reencode all text as Unicode UTF-8.

Motivation

Pages that use any content except basic ASCII have cross-platform display problems. Windows encodings are not interpreted correctly on the Mac and vice versa. Web browsers guess what encoding they think a page is in, but they often guess wrong.

UTF-8 is a standard encoding that works across all web browsers and is supported by all major text editors and other tools. It is reasonably fast, small, and efficient. It can support all Unicode characters and is a good basis for internationalization and localization of pages.

Potential Trade-offs

You need to be able to control your web server's HTTP response headers to properly implement this. This can be problematic in shared hosting environments. Bad tools do not always recognize UTF-8 when they should.

Mechanics

There are two steps here. First, reencode all content in UTF-8. Second, tell clients that you've done that. Reencoding is straightforward, provided that you know what encoding you're starting with. You have to tell Tidy that you want UTF-8, but once you do, it will do the work:

```
$ tidy -asxhtml -m --output-encoding utf8 index.html
```

TagSoup you don't have to tell. It just produces UTF-8 by default.

A number of command-line tools and other programs will also save content in UTF-8 if you ask, such as GNU recode (www.gnu.org/software/recode/recode.html), BBEdit, and jEdit. You should also set your editor of choice to save in UTF-8 by default.

The next step is to tell the browsers that the content is in UTF-8. There are three parts to this.

- Add a byte order mark.
- Add a meta tag.
- Specify the Content-type header.

The byte order mark is Unicode character 0xFEFF, the zero-width space. When this is the first character in a document, the browser should recognize the byte sequence and treat the rest of the content as UTF-8. This shouldn't be necessary, but Internet Explorer and some other tools are more reliable if they have it. Some editors add this automatically and some require you to request it.

The second step is to add a meta tag in the `head`, such as this one:

```
<meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8" />
```

The `charset=UTF-8` part warns browsers that they're dealing with UTF-8 if they haven't figured it out already.

Finally, you want to configure the web server so that it too specifies that the content is UTF-8. This can be tricky. It requires access to your server's configuration files or the ability to override the configuration locally. This may not be possible on a shared host, but it should be possible on a professionally managed server. On Apache, you can do this by adding the following line to your httpd.conf file or your .htaccess file within the content directory:

```
AddDefaultCharset utf-8
```

You really shouldn't have to do all three of these. One should be enough. However, in practice, some tools recognize one of these hints but not the others, and the redundancy doesn't hurt as long as you're consistent.

I do not recommend adding an XML declaration. XML parsers don't need it, and it will confuse some browsers.

Escape Less-Than Sign

Convert < to `<`.

```
x < y ==> y > x
```

```
x &lt; y ==> y > x
```

Motivation

Although some browsers can recover from an unescaped less-than sign some of the time, not all can. An unescaped less-than sign is more likely than not to cause content to be hidden in the browser. Even if you aren't transitioning to full XHTML, this one is a critical fix.

Potential Trade-offs

None. This change can only improve your web pages. However, you do need to be careful about embedded JavaScript within pages. In these cases, sometimes the less-than sign cannot be escaped. You can either move the script to an external document where the escaping is not necessary or reverse the sense of the comparison.

Mechanics

Because this is a real bug that does cause problems on pages, it's unlikely to show up in a lot of places. You can usually find all the occurrences and fix them by hand.

I don't know one regular expression that will find all cases of these. However, a few will serve to find most. The first thing to look for is any less-than sign followed by whitespace. This is never legal in HTML. This regular expression will find those:

```
<\s
```

If you're not using any embedded JavaScript, you can search for `<(\s)` and replace it with `<\1`. However, if you're using JavaScript, you need to be more careful and should probably let Tidy or TagSoup do the work.

If your pages involve mathematics at all, it's also worth doing a search for a < followed by a digit:

```
<\d
```

However, a validator such as xmllint or HTML Validator should easily find all cases of these, along with a few cases the simple search will mix.

Embedded JavaScript presents a problem here. JavaScript does not recognize `<` as a less-than sign. Inside JavaScript, you have to use the literal character. A less-than sign can usually be recast as a greater-than sign with arguments reversed. For example, instead of writing

```
if (x < 7)
```

you write

```
if (7 > x)
```

However, I normally just rely on placing the script in an external file or an XML comment instead:

```
<script type="text/javascript" language="javascript">
<!--
if (location.host.toLowerCase().indexOf("example.com") < 0 &&
location.host.toLowerCase().indexOf("example.org") <= 0) {
    location.href="http://www.example.org/";
}// -->
</script>
```

This is a truly ugly hack and one I cringe to even suggest, but it is what seems to work and what browsers expect and deal with, and it is well-formed.

A lot of these problems can spread out across a site when the site is dynamically generated from a database and the scripts or templates that generate it do not sufficiently clean the data they're working with. A typical SQL database has no trouble storing a string such as x > y in a VARCHAR field. However, when you take data out of a database you have to clean it first by escaping any such characters. Most major templating languages have functions for doing exactly this. For instance, in PHP the `htmlspecialchars` function converts the five reserved characters (>, <, &, ', and ") into the equivalent entity references. Just make sure you use it. Even if you think there's no possible way the data can contain reserved characters such as <, I still recommend cleaning it. It doesn't take long, and it can plug some nasty security holes that arise from people deliberately injecting weird data into your system.

Note

You do not need to escape greater-than signs, although you can. The only situation where this is mandatory is when the three-character string `]]>` appears in regular content. This is likely to happen only if you're writing an XML tutorial. (That's the CDATA section closing delimiter.) Nonetheless, if you're worried about someone attempting to inject bad data into your system, you can use a similar approach to change > to `>`.

Escape Ampersand

Convert & to `&`.

```
<a href="/discipline/470.html">Health & Kinesiology</a>
<img src="text.gif" alt="Texts & Technology" />
```

```
<a href="/discipline/470.html">Health &amp; Kinesiology</a>
<img src="text.gif" alt="Texts &amp; Technology" />
```

Motivation

Although most browsers can handle a raw ampersand followed by whitespace, an ampersand not followed by whitespace confuses quite a few. An unescaped ampersand can hide content from the reader. Even if you aren't transitioning to full XHTML, this refactoring is an important fix.

Potential Trade-offs

None. This change can only improve your web pages.

However, you do need to be careful about embedded JavaScript within pages. In these cases, the ampersand usually cannot be escaped. Sometimes you instead can use an external script where the escaping is not necessary. Other times, you can hide the script inside comments where the parser will not worry about the ampersands.

Mechanics

Because this is a bug that results in visible problems, there usually aren't many cases of this. You can typically find all the occurrences and fix them by hand.

I don't know one regular expression that will find all unescaped ampersands. However, a few simple expressions will usually sniff them all out. First, look for any ampersand followed by whitespace. This is never legal in HTML. This regular expression will find those:

```
&\s
```

If the pages don't contain embedded JavaScript, simply search for `&(\s)` and replace it with `\&\1`. A validator such as xmllint or HTML Validator will easily find all cases of these, along with a few cases the simple search will mix. However, if pages do contain JavaScript, you must be more careful and should let Tidy or TagSoup do the work.

Embedded JavaScript presents a special problem here. JavaScript does not recognize `&` as an ampersand. JavaScript code must use the literal & character. I normally place the script in an external file or an XML comment instead:

```
<script type="text/javascript" language="javascript">
<!--
if (location.host.toLowerCase().indexOf("example.com") < 0 &&
location.host.toLowerCase().indexOf("example.org") <= 0) {
    location.href="http://www.example.org/";
}// -->
</script>
```

If a site is dynamically generated from a database, this problem can become more frequent. A SQL database has no trouble storing a string such as `"A&P"` in a field, and indeed it is the unescaped string that should be stored.

When you receive data from a database or any other external source, clean it first by escaping these ampersands. For example, in a Java environment, the Apache Commons library includes a `StringEscapeUtils` class that can encode raw data using either XML or HTML rules.

Do not forget to escape ampersands that appear in URL query strings. In particular, a URL such as this:

```
http://example.com/search?name=detail&uid=165
```

must become this:

```
http://example.com/search?name=detail&amp;uid=15
```

This is true even inside `href` attributes of `a` elements:

```
<a href=
"http://example.com/search?name=detail&amp;uid=16">
 Search</a>
```

Escape Quotation Marks in Attribute Values

Convert " to `&quot;` or ' to `&apos;` in attribute values.

```
<blockquote cite='Jane's Fighting Ships 2007-2008,
 Stephen, R.N. Saunders, p. 32'>
<a title="How the Supreme Court "elected"
  George W. Bush president">
```

```
<blockquote cite='Jane&apos;s Fighting Ships 2007-2008,
 Stephen, R.N. Saunders, p. 32'>
<blockquote cite="Jane's Fighting Ships 2007-2008,
 Stephen, R.N. Saunders, p. 32">
<a title='How the Supreme Court "elected" George W. Bush'>
<a title="How the Supreme Court &quot;elected&quot;
  George W. Bush president">
```

Motivation

A quotation mark that appears inside an attribute value delimited with the same style of quotation mark prematurely closes the value. Different browsers deal differently with this situation, but the result is almost never anything you want. Even if you aren't transitioning to full XHTML, this refactoring is an important fix.

Potential Trade-offs

None. This change can only improve your web pages.

Mechanics

Because this is a real bug that does cause problems on pages, it's unlikely to show up in a lot of significant places. You can usually fix all the occurrences by hand fairly easily.

Because the legality or illegality of any one quote mark depends on others, it's not easy to check for this problem using regular expressions. However, well-formedness testing will find this problem. Indeed, you may need to fix this one before fixing other, lesser problems because it's likely to hide other errors.

As with < and &, this problem is most often caused by blindly copying data from a database or other external source without first scanning it for reserved characters. Be sure to clean the data using a function such as PHP's `htmlspecialchars` to convert quotation marks and apostrophes into the equivalent entity references before inserting them into attribute values.

Contrary to popular belief, you do not need to escape all quotation marks, only those inside attribute values. You can escape quote marks in plain text if you want to, but this is superfluous. I usually don't bother. Even inside attribute values, you only need to escape the kind of quote that delimits the attribute value. Because different authors, editors, and tools differ in whether they prefer single or double quote marks, I usually escape both to be safe.

Tidy and TagSoup cannot reliably fix quotation marks inside attribute values. For example, Tidy turned this:

```
<blockquote cite='Jane's Fighting Ships 2007-2008,
 Stephen, R.N. Saunders, p. 32'>
```

into this:

```
<blockquote cite='Jane' s="" fighting="" ships=""
  r.n.="" p.="">
```

You shouldn't encounter a lot of these problems, though, so it's best to fix them by hand once a validator points them out.

Introduce an XHTML DOCTYPE Declaration

Insert an XHTML DOCTYPE declaration at the start of each document.

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

Motivation

The DOCTYPE declaration points to the DTD that is used to resolve entity references. Without it, the only entity references you can use are `&`, `<`, `>`, `&apos;`, and `&quot;`. Once you've added it, though, you can use the full set of HTML entity references: `&copy;`, ` `, `&eacute;`, and so forth.

The DOCTYPE declaration will also be important in the next chapter when we begin to make documents valid, not merely well-formed.

Potential Trade-offs

Adding an XHTML DOCTYPE declaration has the side effect of turning off quirks mode in many browsers. This can affect how a browser renders a document. In general, this is a good thing, because nonquirks mode is much more interoperable. However, if you have old stylesheets that depend on quirks mode for proper appearance, adding a DOCTYPE may break them. You might have to update them to be standards conformant first. This is especially true for stylesheets that do very precise layout calculations.

Mechanics

You can use three possible DTDs for XHTML: frameset, transitional, and strict.

- The frameset DTD allows pages to contain frames.
- The transitional DTD retains deprecated presentational elements such as `i`, `b`, `u`, `iframe`, and `applet`.
- The strict DTD removes all deprecated presentational elements and attributes that should be replaced with CSS. It also tightens up the content model of many elements. For instance, in strict XHTML, blockquotes and bodies cannot contain plain text, only other block-level elements.

These are indicated by one of the following three DOCTYPE declarations:

```
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

In the short run, it doesn't matter which you pick. In the long run, you'll probably want to migrate your documents to the strict DTD, but for now you can use the frameset DTD on any pages that contain frames and the transitional DTD for other documents.

Browsers look at the public identifier to determine what flavor of HTML they're dealing with. However, they will not actually load the DTD from the specified URL. In essence, they already know what's there and don't need to load it every time.

Other, non-HTML-specific tools such as XSLT processors may indeed load the DTD. In this case, you may wish to replace the remote URLs with local copies. For example:

```
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "dtd/xhtml1-strict.dtd">

<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "dtd/xhtml1-transitional.dtd">

<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
 "dtd/xhtml1-frameset.dtd">
```

As long as the public identifiers are the same, the browsers will still recognize these.

Some documents on a site may already have DOCTYPE declarations, either XHTML or otherwise. Many tools have added these by default over the years, even though browsers never paid much attention to them. Thus, the first step is to find out what you've already got. Do a multifile search for `<!DOCTYPE`. Unless you're writing HTML or XML tutorials, any hits you get are almost certain to be preexisting DOCTYPE declarations. In most cases, though, they will not be the right one. Usually, there are only a few variants, so you can do a constant string multifile search and replace to upgrade to the newer XHTML DOCTYPE. Any that don't fit the pattern can be fixed by hand.

Documents that don't have a DOCTYPE are also easy to fix. The DOCTYPE always goes immediately before the `<html>` start-tag. Thus, all you have to do is search for `<html\w` and replace it with the following:

```
<!DOCTYPE html
 PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "dtd/xhtml1-strict.dtd">
<html
```

You should also take this opportunity to configure your authoring tools to specify the XHTML DOCTYPE by default. Often it's a simple checkbox in a preference pane somewhere.

```
$ tidy -asxhtml --doctype strict file.htmlTagSoup does not add DOCTYPE
declarations. You'll need to insert these by hand. Tidy adds a
transitional DOCTYPE by default. However, you can request strict instead
with the --doctype strict option:
```

Terminate Each Entity Reference

Place semicolons after entity references.

```
&copy 2007 TIC Corp.
if (i &lt 7) {
Ben &amp Jerry's Ice Cream
```

```
&copy; 2007 TIC Corp.
if (i &lt; 7) {
Ben &amp; Jerry's Ice Cream
```

Motivation

XML requires that each entity reference end with a semicolon.

Web browsers can usually work around a missing semicolon, but only if the entity name is followed by whitespace. For instance, most browsers can handle `"Ben &amp Jerry's"` but not `"A&ampP"`.

Potential Trade-offs

None. All browsers recognize entity references that end with semicolons.

Mechanics

To find cases such as this, search for any entity reference where whitespace precedes the next semicolon:

```
&[^;]*\s
```

Because the next character after the entity reference is unpredictable, you're better off replacing it manually or letting Tidy or TagSoup do the work. They can both fix most of these problems.

This search will also find a number of purely unescaped ampersands. This is especially common in two places: JavaScript and URLs.

Validation should find any remaining cases, and you can fix those by hand. Sometimes manual inspection is necessary to see exactly where the entity boundary lies.

Replace Imaginary Entity References

Make sure all entity references used in the document are defined.

```
&copyright; 2007 TIC Corp.
```

```
&copy; 2007 TIC Corp.
```

Motivation

Occasionally, authors begin to use entity references that simply don't exist. Sometimes it's a simple typo, such as `&apm;` instead of `&`. Sometimes it's misremembered code, such as `&tm;` instead of `&trade;` or `&copyright;` instead of `&copy;`. Either way, this causes display problems for all browsers and should be fixed.

Potential Trade-offs

None. This is only good.

Mechanics

The hardest problem is finding these imaginary entity references, because there's not necessarily any rhyme or reason to them. Often, the

first time you realize there's a problem is while browsing your site. If you're lucky it will appear in the plain text like this:

```
&copyright; 2007 TIC Corp.
```

If not, the browser will just drop it out completely:

```
2007 TIC Corp.
```

The same mistakes do tend to repeat themselves, so once you've noticed a problem, a straight search and replace will usually find and fix all other occurrences.

Otherwise, validation (or at least well-formedness checking) is necessary to identify these issues. Once a validator finds such imaginary entity references, you can fix them by hand if they aren't too numerous, or with a targeted search and replace if they are.

Occasionally, you'll find someone has invented an entity reference that perhaps should exist but doesn't: `&yen;` for ¥ or `&bet;` for the Hebrew letter ב. Although it's theoretically possible to define new entity references such as these in the internal DTD subset or external DTD, I do not recommend this. XML parsers can handle this, but browsers cannot. Either replace the references with the actual characters (especially if you already reencoded the document in UTF-8) or use a numeric character reference such as `&#xA5;` or `&#1489;`.

Introduce a Root Element

Make sure every document has an `html` root element.

Motivation

XML requires every document to have a single root element. XHTML requires that this element be `html`.

Browsers interpret `html`-less (and headless and bodyless) documents differently. Adding the proper root element will synchronize their behavior.

Potential Trade-offs

None.

Mechanics

Search for documents that don't contain `<html`. Because of DOCTYPE declarations, comments, whitespace, and byte order marks, this often isn't exactly the first thing in the document, but it's usually pretty near the start. It's very unusual to find this string in any document that doesn't have an `html` root element.

This problem isn't a common one, but I have seen it more frequently than I'd expect. Fixing it is straightforward: Just put `<html>` at the start (though after the DOCTYPE) and `</html>` at the end.

Documents that are missing `html` tags are often missing head and/or body elements, too. You may need to add these as well. The head is not technically required, but you really want to have one with at least a title. The body element is required if you have any content at all: text, paragraphs, tables, anything.

Introduce the XHTML Namespace

Add an `xmlns="http://www.w3.org/1999/xhtml"` attribute to every `html` element.

```
<html>
```

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

Motivation

XSLT and other XML-based tools can treat the same element differently, depending on its namespace. XML-based XHTML tools expect to find HTML elements in the XHTML namespace and will usually not function correctly if they are in no namespace instead.

Furthermore, many browser extensions such as XForms, SVG, and MathML operate correctly only when embedded inside a properly namespaced XHTML document.

Potential Trade-offs

None. This will not affect browser display.

Mechanics

This can mostly be fixed with search and replace. The most common `html` start-tag is simply `<html>` with no attributes. Without even using regular expressions, you can do a multifile search and replace that converts this into `<html xmlns="http://www.w3.org/1999/xhtml">`.

However, you may also encounter some other additional attributes on the `html` element. The `lang` attribute is particularly common, but other possibilities include `id` and `dir`. For example:

```
<html lang='en-UK'>
```

Thus, as a first step, I suggest searching for `<html\s`—that is, `<html` followed by any whitespace character. If there are a few of them, you can fix them manually. If there are a lot of them, most likely some person, tool, or program made a common practice of adding some particular attribute to the `html` start-tag. If so, this is likely to be consistent across the site. For example, you may need to search for `<html lang='en'>` instead of just `<html>`.

The only thing you need to be careful of is that no one has already changed some (but not all) of the HTML documents to use the

XHTML namespace. You may wish to do a search for this first. Thus, the order is

1. Search for `http://www.w3.org/1999/xhtml`. If no results are found, continue. Otherwise, exclude the files containing this string from future replacements.
2. Search for `"<html\s"` and replace it with `"<html xmlns='http://www.w3.org/1999/xhtml' "`.
3. Search for `<html>` and replace it with `<html xmlns='http://www.w3.org/1999/xhtml'>`.

When you're done, set your validator to check for XHTML specifically. It should warn you of any lingering problems that you missed.