Extracted from:

The Rails View

Creating a Beautiful and Maintainable User Experience

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Rails View

Create a Beautiful and Maintainable User Experience

John Athayde
and Bruce Williams

Edited by Brian P. Hogan

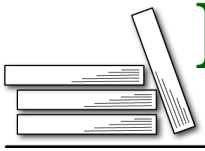
The Facets of Ruby Series

The Rails View

Creating a Beautiful and Maintainable User Experience

John Athayde
Bruce Williams

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian Hogan (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-687-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

2.2 Standardizing Coding Practices

It's important that teams follow consistent, sane rules when it comes to writing templates and style sheets, from indentation standards to the ordering of CSS properties.

Say we've been away on vacation for the past week, soaking up the sun somewhere suitably tropical and devoid of Internet access. We've come back and found a *mess*. Our team has been working furiously on new features, and the templates look like a bowl of half-cooked spaghetti.

What does half-cooked spaghetti look like in code? Avoiding ASCII art, it's really what happens when we've let our code go during development. Line length is all over the place, indents are haphazard, tags are not balanced, and we don't know what we're closing when we have a closing . It's hard to read, hard to maintain, and hard to extend. Obviously we need to do something.

Indenting Without Hard Tabs

One of the first things we discover is that some members of our team are using hard tabs ("hard" refers to using tab characters, as opposed to "soft" tabs, which are spaces that emulate tabs). This causes portability problems across editors and other tools and is inconsistent with the way Ruby developers write code.

Tabs vs. spaces is somewhat of a holy war in programming circles. We at ArtFlow Industries Inc. use spaces because we have many programmers, some of whom prefer tab stops of differing sizes. This becomes a major issue when we agree on, say, a four-space indent per line. Each individual has tab stops set differently. For example, Frank uses two-space tabs. So each time he indents a line with tabs, he inserts two tab characters to achieve the required four-space indent. Sam likes four-space tab stops, so he only tabs once. When Frank opens Sam's recent commits, the indentation is all wrong in his text editor because it renders tabs as two spaces. Likewise, when Sam opens Frank's commits, things are overindented, like in Figure 9, *Indenting going horribly wrong*, on page 6.

This gets really bad when Sam decides to reflow his code to make it look better on his editor. He then commits this to our source code repository, and when we try to discover who has made specific changes to the file, it looks as if Sam's modified the entire file (because, well, he has), as we see in Figure 10, *Minor whitespace changes can look like major modifications*, on page 7.

```
class AssetsController < ApplicationController
¬
        def create¬
                if @asset.valid?¬
                        flash[:notice] = "Asset created!"¬
                else¬
                        flash.now[:alert] = "Could not save asset!"¬
                end¬
                # Redirect¬
        end¬
¬
end¬
```

Figure 9—Indenting going horribly wrong

This causes a big problem; let's fix it. We replace all tabs with standard two-space indentation, and we make sure our team members are using editors that transparently support soft tabbing.

Indenting for Logic and Markup

For our ArtFlow app/views/clients/index.html.erb template we have a listing of clients that breaks an important rule on page ?:

```
<ul>
<% @clients.each do |client| %>
<li><%= link_to client.name, client %></li>
<% end %>
</ul>
```

The contents of *any* pair of tags—either HTML opening and closing tags or the start and end tags for an ERB block—should be indented a level to indicate hierarchy. Things inside a pair of tags are effectively children of the parent tag. The purpose of indentation is to visually indicate hierarchy and nesting, no matter which types of tags are involved.

Here it would be helpful to immediately see that our ERB loop is *inside* the <ul> tag merely by scanning the template. At first glance here, it appears to be a sibling. Let's change the indenting to be cleaned up the way we want it.

artflow/readability/app/views/clients/index.html.erb

```
<ul>
  <% @clients.each do |client| %>
    <li><%= link_to client.name, client %></li>
  <% end %>
</ul>
```

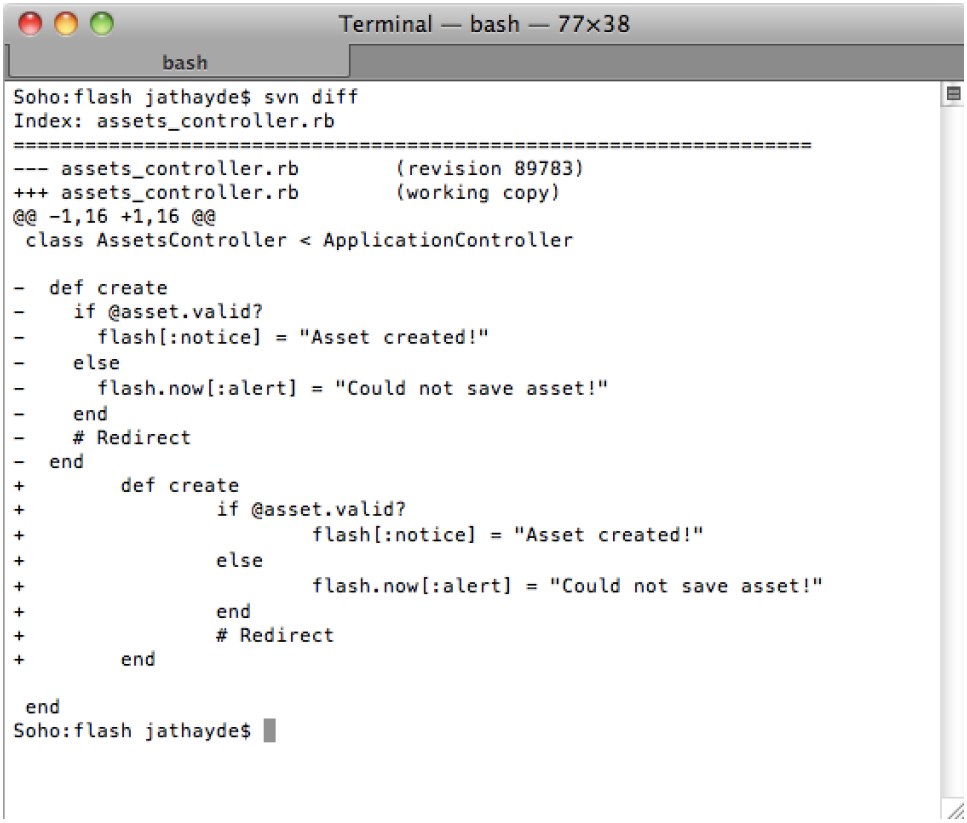


Figure 10—Minor whitespace changes can look like major modifications

A good way to think about this is to imagine that the ERB tags are *inserted* into the hierarchy between the <ul> and its child <li> tags. Following this simple rule will help make the logical and physical structure of templates more obvious.

Don't worry about how the generated HTML looks. Browsers are the ones that do the reading (or during debugging, Firebug and the Chrome developer tools are the ones that clean things up[2]).

Policing Line Length

As developers, we've all had the experience where we see code go off the right edge of the editor screen.

2. http://getfirebug.com/ and http://code.google.com/chrome/devtools/, respectively.

Joe asks:

But My Whole Team Has Massive Monitors. Why Can't We Use Line Lengths More Than 80 Characters Wide?

Even with extra-large monitors in many development environments, the reality is that we still hit code from a variety of devices and with a variety of preferences. Many users bump up the font size so that while it may be 1600 pixels wide, it's still only 80–100 characters wide.

Using 78–80 characters is a standard for editing that will work on almost any system, because that is the default width of terminals, including older VAX systems. Your team can decide to go longer, but our rule for this team will be 80.

For more guidance on line length (and many other concepts in this chapter), we recommend *Clean Code: A Handbook of Agile Software Craftsmanship* [Mar08], by Robert C. Martin.

Anything over eighty characters will potentially float off in the ether if someone views it from a terminal window, and scrolling horizontally (either physically in the window or with our eyes on a high resolution screen) back and forth makes for slow reading. It's easier to read a narrow block of text than one that stretches across the width of the screen. If we break it into multiple lines, we can see everything at once much easier.

Soft wrapping might sound like a solution, but it's a Band-Aid that makes line editing more difficult and doesn't work well when our developers use command-line utilities.

Some text editors, such as TextMate, have a column counter and a wrap column setting so you automatically know when you hit your determined character limit. They also have macros and other tools to help with reformatting large blocks of text.

Lining Up Tags and Attributes

When tag contents span multiple lines, take care to line up the opening and closing tags horizontally and indent the contents one level. Working on the marketing copy for ArtFlow, you can see our `<p>` and `<em>` tags are lined up correctly and their contents set a level deeper.

Let's look at a snippet from ArtFlow's homepage as an example:

Joe asks:
A New Line for Every Element?

If there's an inline markup tag, such as <b>, <i>, , , <abbr>, <dfn>, and similar tags, we do not always kick them to a new line. It's about improving readability. If putting one word on a new line doesn't improve readability, don't do it.

artflow/readability/app/views/pages/home.html.erb

```
<p>
  Have a file, store a file.
  Then change it, tag it, and share it or send it.
  <em>
    This isn't your father's asset management application.
  </em>
</p>
```

When we stack lots of classes or have long ID and class names, we can end up with a long line just for one HTML element. We fix this with a newline between attributes (and some care to line them up):

```
<p id="product-description"
   class="important blurb rounded">
  Have a file, store a file.
  Then change it, tag it, and share it or send it.
  <em>
    This isn't your father's asset management application.
  </em>
</p>
```

We can add ERB comments to our template for TODOs, placemarkers, or short descriptions for complex markup. They look like normal ERB tags and start with a # character, just like Ruby comments do:

```
<%# TODO: Add list of articles. -BW 2011-11-01 %>
```

While adding ERB comments can clarify and illuminate, at some point they can also make a page messy and more cluttered, and just like code comments, it's easy to let them get out-of-date. Less is more!

Now that we have some basic formatting rules to serve as our foundation, let's dig into the way we're actually building up our markup to see if we can simplify things and make our template more readable.

Joe asks:

Why ERB Comments Instead of HTML Comments?

HTML comments are present in your generated markup and visible to users of your application (if they're curious and click View Source). ERB comments get stripped out long before your page ever gets to a browser. This doesn't matter so much for smaller apps, but a few kilobytes here and there can start to add up over millions of users (just ask Twitter).

Use ERB comments unless you really want to whisper something to the geekiest of your users—or as a *temporary* debugging technique.