Technical Explorations

by Keith R. Bennett

Stealth Conditionals in Ruby

Sep 16, 2012 • keithrbennett

When I first encountered the Ruby language in 2001, after working with Java, C++, and C for several years, I fell in love with it. How expressive, concise, clear, and malleable it is. A few years ago I even named a slide show What I Love About Ruby. I use it for presentations on beginning Ruby for novices.

But there's one thing in Ruby I haven't gotten used to…the widespread use of what I call *stealth conditionals*, conditionals that are "hidden" in the middle of a one line statement, as in:

```
do_something(foo, bar, baz) if some_condition
```

I strongly believe that just as we software developers strive to create user interfaces that communicate structure and content with visual cues to our users, we should do the same for each other in our source code.

The beginning of a line of source code has a special status and importance. As the eyes scan down a section of code, the most prominent text is the text that begins each line.

The opportunity cost of embedding a conditional in the middle of a statement is the lost opportunity to communicate that conditional more clearly and strongly. The result is that the reader is required to work harder to understand the flow of execution.

The classical alternative to the above would be:

```
if some_condition
  do_something(foo, bar, baz)
end
```

Clearly, the presence of the if at the beginning of the line, and the indentation of the function call do a better job of communicating the conditional nature of the code. However, I realize there may be times when a statement is so simple that more than one line feels excessive. In these cases, we do have this as an alternative:

```
if some_condition; do_something(foo, bar, baz) end
```

This is 50 characters long, while the original was 45 characters long. For an extra 5 characters, we get to see the conditional nature of the statement on the left margin of the line. To me, this is a cost that is small compared with its benefit.

That said, sometimes the stealth placement is clearer. For example, below I use the two different strategies in the two unless sections to illustrate. I've intentionally disabled the Ruby color coding because we cannot count on it being present, and we want our code to be maximally understandable either way.

```
# Process the row above
    unless at_top_edge
      neighbors << [row_above, col]
      unless at_left_edge;  neighbors << [row_above, col_to_left]  end
      unless at_right_edge; neighbors << [row_above, col_to_right] end
    end

    # Process the row below
    unless at_bottom_edge
      neighbors << [row_below, col]
```

```
      neighbors << [row_below, col_to_left]   unless at_left_edge
      neighbors << [row_below, col_to_right]  unless at_right_edge
    end
```

In this case, I prefer the lower approach. The special status of the left margin is best used to communicate the commonality of the three lines' operations. The *unless* modifier clause is pretty clear, especially since I've added an extra space to set it apart from the expression before it. By the way, the adding of an extra space or two before a stealth conditional would probably be a good convention to follow.

I do believe the case above to be the exception rather than the rule. Often, the line containing the stealth conditional is one of several in a method, and those lines don't have much in common. In those cases, I recommend putting the *if* or *unless* first.

Published with GitHub Pages