Writing cleaner code

By Cal Henderson

2009-05-11: These days I would recommend jumping straight to a framework (or at least a templating engine) to separate the logic from the template. But you already know that.

This article is the first in a series that looks at bits of real-world code and explains ways to improve it: making code clearer and faster.

Why bother changing code if it works? At some point, you're going to want to change something. This is inevitable. If might be tomorrow, it might be in a few years time. Someone else might take over from you and want to change something. You should always aim to make your code as *maintainable* as possible. If you come back to it in a year's time, will you remember how it works? No - and you'll need to read the code to figure it out. If you give the code to someone else, they'll need to do the same. More time is spent during maintainance just trying to figure out what's going on, than the time it would have taken to write cleaner code to begin with. Writing maintainable code is working smart - a little extra effort now saves you from eternal tourment further down the road.

There's another reason for it too - you only *assume* your code works. Sure, it might pass a few tests, but what about edge cases? What about people using the code in ways you hadn't thought about. The clearer your code is, the easier it is to spot and fix bugs. And your code will always have bugs: don't imagine you're immune, because even the best programmers make mistakes.

This is our example snippet. Taken from a public code repository, it shows a way to alternate background colors for rows of data taken from a table:

```
 <table border="1">
<?php
 $row_count = mysql_num_rows($result);
 $x = 1;

 while ($x <= $row_count) {
  if($x%2): $color = "#FFFFFF"; else: $color = "#CCCCCC"; endif;
  $row = mysql_fetch_array($result);
  print "<tr><td style=\"background-color: ".$color."\">Row #".$row[id]."</td></tr>\n";

  $x++;
 }
?>
</table>
```

We only have 8 lines of code here, but there's much that can be done. Let's start with the main fetch loop:

```
 $row_count = mysql_num_rows($result);
$x = 1;

while ($x <= $row_count) {
 $row = mysql_fetch_array($result);
 #stuff
 $x++;
}
```

This code loops, pulling rows out of a database. We have already executed a mysql_query() call at

this point and the query handle is held in $result, all ready to return some data to us. There's a neat trick we can use to clean this up alot - mysql_fetch_array() returns undefined when it's run out of rows. This allows us to do this:

```
 while ($row = mysql_fetch_array($result)) {
  #stuff
}
```

Already we've cut 6 lines down to 2. The code is much more readable and you can easily tell what's going on. We have less variables polluting the name space, and less things to go wrong.

The problem we face now is that the code that decided the background color used $x, which we no longer have. Well, we could add it back in, or do something more sensible:

```
 $color = 0;
while ($row = mysql_fetch_array($result)) {
 $color = ($color == $cfg['col1']) ? $cfg['col2'] : $cfg['col1'];
 #stuff
}
```

At first this looks complicated, but really it's much easier than before. The first line initialises $color to 0. What we initialise it to is not important, as long as it doesn't contain a color value. Then we use the ternary operator to assign a new value to $color, based on the current value in $color. These two statements are equivilent:

```
 $a = ($b == $c) ? $d : $e;

if ($b == c){
 $a = $d;
}else{
 $a = $e;
}
```

So we are checking if $color contains the first row color already. If it does, assign the second color, else assign the first color. When the code is executed the first time, the value is 0, doesn't match the first row color and so the first row color is assigned to $color. From then on, the code alternates the value in $color.

You might also notice that i've moved from having literal color strings to using variables, from a hash called $cfg. This is my hash of configuration options - if you want to change your row colors across your whole application, it helps to have them all in one place.

Now onto the final stage - the output itself.

```
 <table border="1">
<?php
 $color = 0;
 while ($row = mysql_fetch_array($result)) {
  $color = ($color == $cfg['col1']) ? $cfg['col2'] : $cfg['col1'];
?>
 <tr class="<?php echo $color; ?>">
```

```
   <td>
    Row #<?php echo $row['id']; ?>
   </td>
  </tr>
 <?php
  }
 ?>
 </table>
```

A few things have changed here. Firstly, we're now using class names instead of color values. Your application's colors should be set in a stylesheet, so all we need to do is alternate class names.

The other change is that we are no longer using echo statements to output the table row. Why not? Well, if your echo statement contains variables that need to be interpolated then it has to be a double-quotish string. And if you have HTML in your echo statement then you'll almost certainly need to use double quotes. Having to escape them is troublesome and messy, so we use php as it was intended and escape out of the code into HTML for a while.

Argh! The code has actually got bigger. But this is not an exercise in writing small code - it's about writing clear code that's easy to maintain. And whilst we have more lines, we actually have less statements - the code is now simple as well as clear. This should always be your aim when writing code - clear, simple, maintainable.

(end of article)