How to write unmaintainable PHP code [2009]

September 18th, 2015. Tagged: php, Uncategorized

> Originally published at `http://phpadvent.org/` in 2009. The site is no more, but I found a copy on my hard drive. I'm not sure but I do hope it's the final version because Sean Coates (@coates) and Chris Shiflett (@shiflett) did a really nice job editing the pice.
>
> Also hope it's still funny...

With the unemployment rates lately being at the levels that they are, everybody realizes that job security is important. And what's the best way to keep a job but to be irreplaceable, one way or another. The simple truth is that if no one can maintain the code you write, you have a job for life. Writing unmaintainable code is a special skill that, strangely enough, seem to come quite naturally to certain developers. But for the rest of you, here are some tips and hints to get you started.

First things first

It all starts with the job posting. You should look for the right company, where you can spread your wings and achieve your unmaintainable potential. You don't necessarily need to be *the* PHP guru in the company but it sure helps. Look for job descriptions where they mention migrating to PHP from something else (so you know you'll be calling the shots). Or maybe search misguided jobs posts that require for example 10 years of PHP5 experience. Plus fluency in FrontPage and Netscape Composer.

Once you land the golden opportunity, be vocal from day one. Speak up at meetings, let your opinion be heard. Talk about object-oriented design architectures, enterprise, shifting paradigms, how "good enough" is not good enough, and, of course, your personal commitment to excellence. Make sure everyone consults your opinion on the important coding initiatives.

The pillars of unmaintainability

Inspired by the most excellent piece "Writing unmaintainable code" (must-read for anyone interested in keeping their job), here are the two important concepts you need to grasp and master:

1. You should make it impossible for someone to change something easily without breaking something else.

   Understand that the maintainer doesn't have time to understand your code. Maintainable code means being able to quickly locate a specific piece in the mountains of code, understand how it works right away and change it without breaking stuff. You can't have that. No one should be able to simply search for something and just find it where they expect.

2. Your code shouldn't *look* unmaintainable (because someone will suspect something), it just has to *be* unmaintainable.

The code should look normal to the maintainers, but take them by surprise when they least expect.

Best practices

1. Ban coding conventions. Endless flamewars have been fought over coding and naming conventions. You can't have that in your fine organization. You have awesome projects to build and you can't afford to spend countless hours in discussion of tabs vs. spaces. Plus, conventions are restrictive. If a new hire comes and is not used to your conventions, they'll be miserable. Unhappy programmer is unproductive programmer. Explain that to anyone who asks. Let everybody write in their favorite style-du-jour. As for your own code - rotate your conventions. Go `camelCase` on Monday, `all_lowercase` on Tuesday, mix-and-match Friday and Hungarian on every February 29th.
2. No comments. Your code is beautiful, it doesn't need comments. If someone can't understand it, well maybe they are not so good programmers after all. If, by any chance, you're forced to write comments, then simply overdo it. Elaborately describe the most obvious and trivial code, skip the rest.

```
// in the following block
// we add two variables:
// named variable a and variable b
// both of them integers

// declare variable a
// and assign integer 1 to it
$a = 1;
// declare variable b
$b = 2;
// sum the two variables a and b
// declared and initialized above
// and assign the result to
// a new variable c
$c = $a + $b;
```

3. Standardize on Notepad. Or any other editor without source highlighting. Let the others suffer and eventually leave the team. You don't need to listen to them complaining all the time. And if someone asks why Notepad, be ready to explain: it comes with Windows (the only OS for today's productive programmer), no training is necessary, it doesn't cost anything. I'm sure you'll find references on the web how you can use any program, including Word, to write code for web pages, but Notepad is for real gurus. And, after all, your company hires no one but the gurus.
4. No unit testing. Explain to anyone who asks that you are hired to write high-quality code that doesn't have bugs (ergo, needs no testing). Why would anyone in their right mind spend any time writing useless tests that confirm the obvious - that the code works? Some things in life just *are* - the sky is blue, the sun rises from the East and your code works, thank you very much. Move on. (Like with comments, if forced to do tests, be prepared to overtest the obvious and skip the rest)
5. No templating engine. Templating engines help separate business logic from presentation. It may make the code maintainable and you cannot allow that. Quote Rasmus Lerdorf: "PHP *is* a templating engine". Even if you're forced to use a templating engine, find a way to misuse it and put little pieces of business logic in the template, as well as carefully crafted mix of HTML (and CSS and JavaScript) into the database access layer for example.

   In general, strive for a healthy mix of PHP, HTML, CSS and JavaScript, possibly on the same line of code. Write PHP that creates JavaScript that creates HTML with inline styles. If asked, call this pattern "encapsulation" - your code is just fully

responsible for itself.

6. Version control. It's hard to avoid it, but it's worth to try talking yourself out of any form or shape of version control. Demonstrate how it improves communication between team members if you talk about stuff, instead of relying on cold-blooded version control software to resolve conflicts. When you fail convincing anybody, do not despair. You don't have to commit it *all* to source control. Keep some code to yourself. Small but deadly pieces, which will break the project if someone other than you tries to build and deploy. If caught, explain how this code was not yet fit-to-print, after all, you only commit code that can educate junior team members on exceptional quality and clever solutions. These boys and girls look up to you and expect nothing but the best!
7. Build a framework. Then you inevitably become The Architect and your authority is unquestionable. Plus it lets you add secret conventions (and lots of them, sometimes contradicting ones) that will trip even the most experienced maintainer. Your framework will take care of everything, no one should bother understanding it, they should be happy you're single-handedly making development easier and more productive for the whole company. Never release the framework as open-source because a) the framework is an asset to the company and the company have invested heavily, and b) the open-source community will poke fun at you and that will be the end of your bluff.

Naming stuff

Your variables should be named mysteriously, often with one letter only. The goal is to make it impossible for anyone to find anything with a simple search.

Class names and functions could be one character names too. When you do actually decide to use a normal name for a change, use it all the time - remember sometimes the best way to hide information is to have too much of it. When reusing the same name (call it "subject matter-oriented programming"), it also helps if you place parentheses and curly braces on new lines - to claim you improve readability and also to make your teammates brush up on their regexp if they want to find anything in your code. Consider this:

```
$noodles = 1;
class
noodles
{
    var $noodles = 2;
    function
        noodles
        ()
    {
        $noodles['noodles'] = 'noodles';
    }
}
function
    noodles() {
        return new noodles;
    }
$noodles = noodles();
var_dump($noodles);
```

You can also use strange charsets when naming variable. Cyrillic letters are quite appropriate, because some letters look just like the roman letters, but they aren't (all of these: xopekacMEBCTAKXOPH). So what will the following echo:

```
$alert = 1;
$аlert = 2;
echo $alert;
```

2? Not if the second `alert` starts with a Cyrillic "a"!

Referencing stuff

Even if you define something normally, doesn't mean you cannot use it in interesting ways. The weapons to master include:

- `eval()`
- variable variables
- variable class names, e.g. `$strudels = "noodles"; $noo = new $strudels;`
- `call_user_func()`

Basically any language construct that allows you to treat code as strings is your friend.

```
    // calling abc();
    $z = 'A';
    call_user_func($z .'bC');
```

Capitalization

Master letter cases. Function names are case insensitive. Abuse this.

```
function abc(){
    echo "abc";
}
AbC();
```

On the other hand array keys are sensitive. Abuse this as well.

```
$a['UseConvetionsOnlyTobreakThem'] = 1;
if (isset($a['UseConvetionsOnlyToBreakThem'])) {
    // ?? capital B !!1!
}
```

Overwrite

Overwrite globals where least expected. Especially superglobals. Overwrite elements in the `$_GET` array early, overwrite them often. Same for `$_POST`. Sprinkle that with some quiet `$_REQUEST` overwrites. If WTF-ed, explain you're sanitizing user input to prevent XSS, injections, infections, and other diseases.

Control structures

Use, mix and match all the alternative `if`, `while`, `for`, `foreach`, `switch` syntax. If asked why, explain that you're training the new hires to really learn the language, all of it.

```
if ($a > 5):
  if ($a > 4) {
      while ($a > 0):
        echo --$a;
      endwhile;
  }
endif;
```

Nest ternary operators, nothing like nice concise code:

```
// guess the output
echo true ? 'true' : false ? 't' : 'f';
```

Inside of the body of `for` loops, increment the `$i` once again to keep everybody alert. Alternatively, surprise folks by not using `$i` for loop increment. Never.

Nest loops. Deeply. Then suddenly break out of them. Statements like `break 2` and `break 3` are a pure joy to figure out, especially when mixed with strangely indented code.

That's a start!

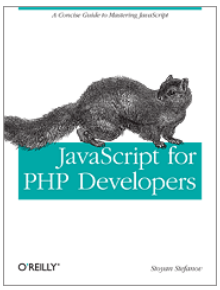
And that's all for today. I hope by now you're convinced that you can do it, you too can write unmaintainable code. Now your future is in your hands! Can you take the chance of being replaceable by writing predictable code?

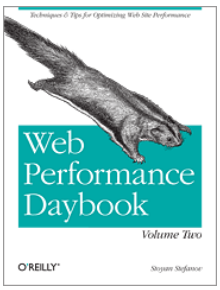
Tell your friends about this post: Facebook, Twitter, Google+

« From JSXTransformer to Babel

Sorry, comments disabled and hidden due to excessive spam. Working on restoring the existing comments...

Meanwhile, hit me up on twitter @stoyanstefanov





phpied.com is powered by WordPress, RSS feed

Search