This little book provides you with the newest patterns and methodologies for purposeless PHP residues. It also lists counter arguments should your syntactic salt ever come into question.

Oh and yes; this list is meant to be amusing and partially sarcastic.

Effective PHP Cargo Cult Programming

and enterprisey coding standards

No need to agree. In fact some points are meant to be controversial. The goal of this overdone rant is to make you reassess, not redeem. Many patterns are useful in some situations. But corner cases should not destine the default methodology. WUT

Overview

Weak typing sucks

PHP is actually a weakly typed language. Nowadays we struggle against this unfortunate misfeature.

E_NOTICE is a serious bug

You have to litter your code with isset().

- This totally helps security.
- And it's needed for strict E_STRICTINESS.

E_NOTICE messages are really debugging notes. But as cargo cult programmer you must treat it as high-priority defect and work around. Always.

Frequent use if isset() helps in various ways:

- Makes E_NOTICES go away.
- Adds more code that gives a secure look.
- Hides actual bugs at runtime.

Imagine following code:

```
if (isset($_SERVER["SERVAR_NAME"]) &&
```

Through clever combinations of copy & pasting and hiding the problem with isset, these kind of bugs are more difficult to uncover. Not that the server name was much likely to be unset in any case.

Old-school PHP usage would make the mistake obvious. That's what they're for. But better hunt E_NOTICEs *bugs*.

Must use === or else!

PHP is unreliable. You never know what you get. Hence it's imperative to be explicit.

Yes, always.

Strictly speaking, there are only a few functions (strpos) where you have to differentiate e.g. between boolean errors and integer result values.

But your code gets so much more enterprisey if you take the time to add === identity checks in **all** if constructs:

```
if (headers_sent() === true) { ...
```

You cannot leave boolean evaluation to PHPs weak type system. Override it. Embrace your anger.

But especially PHPs string handling should give you the creeps. If $_GET["what"] contains "0things" this should not be handled as zero or boolean false. This might not be what the user wanted. Check for === "0" and strlen() else. Avoid automatic type juggling, which will **always** do the wrong thing.

OOP only!!!1!

It is imperative that your applications are 100% object-oriented. Only OOP allows programming large-scale, cloud-based, enterprisey, high-scalability, maintainable, AJAX compatible, RESTful and very secure applications.

Oldschool PHP programming with only functions leads to spaghetti code. Ragequit any discussions about the topic by equating procedural code with noodles.

If you are unsure about the benefits of OOP clenchbanger code, just have a visit at phpclasses.not. It's a marvellous collection of supra high quality utility libraries that are totally designed to be classes and not just functions wrapped in them.

- classes are better than functions
- objects are better than procedural code
- classes and objects are the best match for just about everything

And if you have a feature that could be implemented best as function with a simple in-out API, then don't. That would be a wasted opportunity to apply one useful OOP design pattern atop the hip class-based paradigm.

Also, if you don't package everything into classes you will lose credibility in the eyes of your fellow co-developers. They will assume you don't know the OOP if you can't apply it *every time* without any second thought.

Meh, this topic might require a book of its own.

PHP5 is so much better than PHP4

Alas, the differences in the type system and API are mostly gradual. But cargo culters should make a big deal out of these two versions being totally different languages.

Embrace PHP5 constructs without any particular use case.

These are the useful advancements of PHP5:

- unified PDO database interface
- SPL constructs
- revamped XML functionality

Avoid them, and instead strive for these:

- __construct() method name, because it's new
- private and protected attributes / methods
- E_STRICT, your code must be strictly E_STRICT

+20 bonus points if you are one of the developers who dismisses code based on compatibility. "Oh noees, it still uses teh PHP4!!!!"

Access modifiers are more enterprisey

When called into question just say: **En-cap-su-la-tion**. That's the best buzzwordy rebuttal.

The access modifiers `public, private, protected` must be thouroughly used (1/3 each) to proof your code from running under PHP4. These keywords were introduced in PHP5 to overcome complaints that the language is not fully object-oriented. And completely solving all OOP problems they do:

- Prevent stealing of precious precious attributes.
- Malicious co-programmers setting wrong types.
- Secret method sauce stays secrit.

When copying these essential features from Java/C++ it was assured that all inherent advantages from compiled code apply:

- `private` attributes cannot be removed again and are strictly enforced by the JVM PolicyManager.
- Invalid accesses are catched by the compiler and don't just show up at runtime to blow up your application. (But avoid polymorphism and inheritance just in case).

Agreed-on _underscore conventions do totally not suffice! Just look at Python. What a mess! To fend off all your evil co-programmers you need access modifiers. Else APIs get exploited and you can nevermore totally overhaul class interna without leaking useful new functionality. Yuck.

Shallow setters and getters

Restricting all object attribute access is also important so you have a reason to introduce the most important OOP concept of all:

```
function getTitle($title) {
        return $this->title;
}
function setTitle($title) {
        $this->title = $title;
}
```

You see, they wouldn't strictly be necessary. And as cargo culter you must ensure that they are completely purposeless. Remember that they are needed just in case you wanted to implement any checks. At all. Or Later.

For application performance it's however inadvisable to do any type checks or apply format constraints. The setter must always copy the data verbatim. The getter should have side-effects however and if possible.

Note that you *could* use the magic __set() method to easily implement the type checking support that's amiss in PHP, e.g. by using a $_typemap = [name=>type] array.

But remember that this is way too scripting languageish, even discouraged in Python. And it's why we use setters and getters - which were originally introduced for Java Beans. (Something with RMI and data hiding and binary representation, a dire necessity in PHP.)

'Microoptimizations'

This would be no decent PHP recommendation without educating dear readers about serious ~~SEO~~ runtime and performance optimization techniques.

13.8%

Yes that's right. **Single quotes** give a whopping 13-14% speedup for your PHP scripts.

That is, if you have more then 100M strings in your application and constantly pipe new ones into eval(). Or that's at least how I got to measure it. The effect takes some time to show. (Yes, I've really run that test.)

But there you have it. It's btw not the compiler, but the "tokenizer" which 'tokenizes' strings. And as a matter of fact, 'single' quotes require significantly less work than "double" quotes. These things are called so for a reason.

So before you start denormalizing your database, or reduce database queries for that matter, begin with the optimization technique that trumps them all.

Single quotes should be in your enterprisey coding standard. It's such an obvious scalability gain!

Security

Too much of a good thing...

mysql_query is top-notchy

What could possibly be wrong with a method that ***all*** PHP tutorials since 1997 recommend?

mysql_query is the de-facto standard for interfacing with databases in PHP. Hence it must be used. Everyone does.

The only thing you can improve on it is adding an object wrapper and call it database abstraction. Certainly unneeded are any fancy methods to avoid SQL exploits.

Security by cautiousness. It's sufficient to write mysql_real_escape_string() once in a while. You'll certainly take care to be diligent with it. No better method for security in that area has yet been invented.

Except PDO. But that's entirely unreasonable to be used. Too much overhead. And parameterized SQL is certainly hard to get a grasp on, when mixing up commands and values into an SQL string are that simple in PHP.

Trust the herd.

XHTML-strict all the way

You know what's holding the web application business back? That's right, procedural HTML.

It should be common knowledge by now that HTML is not a real standard, or at least not based on real standards (lets disregard SGML as too fancy). It's only XML from here on out.

Must use XHTML-Strict with E_STRICTINESS in XPHP.

The obvious advantage of XHTML is that it's hip. It adds some other things that we must take care to use right:

- The MIME type must still be text/html, so our XHTML-strict is just a fancy serialization but easily understood by all browsers and those that can't.
- It's forbidden to use any other XMLNS in our XHTML. We just like the X, not the XML use cases.
- Use target="_blank", complain about it's absence in the XHTML-strict standard, yet deploy it.
- Embed some Google Adsense script tags that use document.write in our "XHTML" document.
- Since just not utilizing it or using the right MIME type doesn't add any benefit, it's imperative to add a shiny XHTML badge. To show off competency.

Outright disregard HTML4. It's like totally old and PHP4.

And HTML5 won't be usable before 2023 when it's 120% standardized. And IE9 is the yardstick for everything.

Upcoming Topics

- patterns
  - MVC MVC MVC, Beetlejuice Beetlejuice Be...
  - fake singletons, unproductive factories

Design patterns are a great way to enterprise up your applications. The more the merrier. If your coding style starts to look like J2EE and has lots of Fecades and AbstractFactory classes you are cloning it right. It also helps if the problems you're trying to solve don't exist.

Enterprisey coding standards

No publication on PHP was complete if the author didn't give you well-reasoned coding style advises.

someCamelMethods

Java uses it all over the place. Must be good. Let's too.

RunningWordsTogether is not only an established practice in the Wiki scene , but also a defacto standard for enterprise software development. And that's the glamour we should strive for.

The main reason for its use is saving a single character. Using underscores in method_names only brings a disregardable readability enhancement, so let's disregard it. Also it's not sensible to orientate method names on the programming language base functions. It's OOP, so it must use something better! addSomeOtherRemarkHere();

\deeply\nested\name\spaces

A long overdue language construct was recentely introduced. Let's use it real quick and ferociously!

PHP.net developers looked to Java and C++, but adopted neither syntax. It really came down to parser fixing woes rather than cogitated syntax considerations. Some people now call PHP a laughing stock, but we prefer the term "special needs language".

Anyway, the namespace backslash helps in various ways.

- It semantically encourages directoritis, mapping classes 1:1 onto files. (Java does it too, right?)
- It hasn't really Java-like semantics. But that's what the recommended namespacing schemes mimick.

Originally namespaces were intended to reduce conflicts. A single namespace would suffice for that. But as cargo culters we're not happy until the problem is oversolved in a major eye-straining way. And rightly so, instead of the old **Long_Class_Names** we've got:

```
\vendor\Short\Class\Names;
```

Ideally namespaced classes could be aliased and become shorter than the_old_variants. But you see, we only want namespaces to choose unimaginative and highly generic class names within. So the 'use' advantage for practical purposes is out of the window, once every namespace repeats the same set of undescriptive class names. Instead these dull class names are to be used with their fully qualified namespaces everywhere. Whooho!

FAQ and Excuse Chapter

I'd probably get too much hatemail, so I'll rather add some relativizations right away. Less sarcasm from here.

Why so serious?

No. This is mostly a fun list. It highlights a few current PHP fallacies.
It's about thoughtless construct copy&pasting and buzzword-reasoned syntax usage. Not about forbidding them.

But I must use isset!

Sure you do. NULL is a proper value and should play a role in your application logic. The section on isset discusses its officious use. Once the isset construct decorates all variable references, you have defeated the purpose of the debug error level (which E_NOTICE is). It then likely obfuscates real problems and can obstruse debugging. And it's syntactic salt if used under the assumption of raising security or code cleanliness.

But it can't be false if *everyone* does it?

I hope note. Actually I'm sure not *everyone* has crowd-sourced critical thinking and sensible coding practices.

It's a common misconception that widespread equals ok.

How dare you not like OOP ?!!

Well why, in fact I do. It's just the spilling misdesigns and purposeless applications I don't.

Why shouldn't encapsulation require access modifiers?

Access modifiers have their origin in compiled languages. Bar pointer workarounds and allowed reflection they are enforcable there. In all-sourcy PHP they might as well be comment decorators. In fact most other scripting languages run well on conventions like the underscoritis prevalent in Python and PHP4 before. This is sufficient for programmers behaving as adults. A resoned explanation or useful API better dissuade from object interna access.

Encapsulation is about functionality compounds, not attribute misuse paranoia. Type constraints are rareley employed (and wouldn't make much sense in a weakly typed language anyway). And the presence of shallow setters speaks for itself.

It's only mentioned here because public/private/protected are treated like semicolons by many PHP programmers, when the safety reasoning is mostly hypothetical.

Like the word "enterpisey" much?

But it sounds so nice.

And it's the best description of what some syntactically raped coding patterns look like.

So what's the mysql_query bashing about?

I've recentely seen a lot of uninformed denial around PDO. Many people claim it's difficult to use or understand or some other random excuse. But as a matter of fact, it frequently simplifies queries. It particularily obviates manual and bug-prone escaping. Why people refuse to migrate off oldschool SQL concatenation is beyond me. It's clearly a clinging to outdated practices and an ingrained worse-is-better mentality.

Can you knock off the namespace syntax hate?

Not. This syntax choice was retarded back then when it was done in that afternoon IRC session. And it remains retarded three years later. Valid complaints don't vanish from sitting it out. Surprise.

Meanwhile magic-quotes-2.0 outcomes start to show. Framework developers are clearly obsessed with the syntax. And mapping directories onto ···► code and class identifiers is an implementation smell.
Humdrum class names became the standard, use isn't used, and name aliases can't alleviate it sensibly. It all adds mostly ambiguity atop an ill-conceived workaround.

What's "cargo cult programming" anyway?

Mimicking other peoples coding methodologies without understanding the original intention or if it's applicable to the current environment. WikiPedia has pictures.

Afterword

Oh my. The FAQ wasn't really toned down. So let me iterate the main point of this rant again.
I realize it reads like flamebait when critizizing these pretty widespread coding conventions. Also the name calling doesn't help much either. - Even though "cargo cult" means pretty much unreasoned pattern adoptions.

Anyway, there is no point in getting all worked up about the topics. If you are using them in any of your applications, fine. At least it follows common practices and its prevalence makes it therefore easier to read.

However it's a good idea to understand where some constructs came from, what kind of problems they were originally intended to solve. And also placebo usage isn't wrong per se.

In conclusion, if you want to comment or counter argue, please go ahead. This fun ebook is meant to start an overdue controversy.
It's helpful if you can dig out some nice example code, explaining where a pattern is actually useful. If you need half an hour to find an example, it's however very likely not relevant. Corner cases aren't helpful as reference for standard aspiring coding patterns.

And thanks for an open mind and reading that far!

Yeah, all to the bottom.

That's here.

End.