www.phparch.com

September 2021
Volume 20 - Issue 9

php[architect]

It's Really an Upgrade

Why Would Anyone Want to Downgrade Their PHP Code?

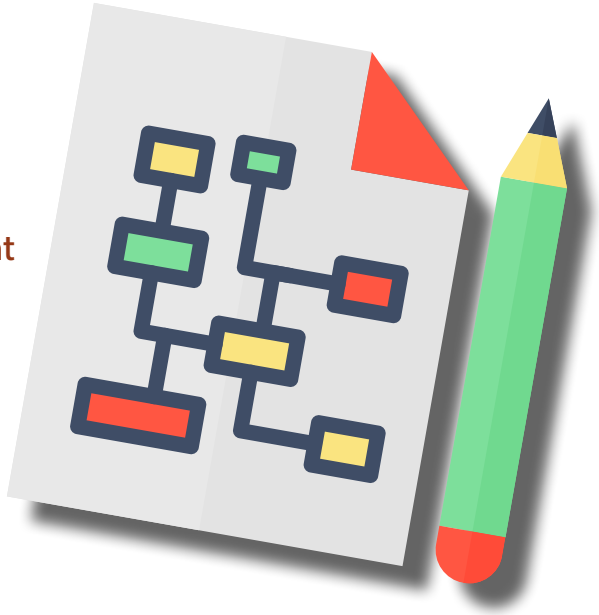
CQRS—Reasoning and Architectural Prospects

Free Sample Article

ALSO INSIDE

PHP is the Worst

Chris Tankersley



I have been programming for nearly twenty years at this point, and I have worked in various languages. At many of my previous jobs, as well as my current one, I have had the pleasure of working with PHP as the core language of my job. Since the first time I started working with PHP, I heard all the complaints about the language, but I also saw the power that PHP has.

PHP is, to say the least, an interesting language. The language and the programs that are built with it fall into two design philosophies, often simultaneously. I do not mean software development lifecycles like waterfall or agile, but rather the fundamental ideas governing what software should look like. These ideas have come to be known as "The Right Way" and "Worse is Better."

PHP encompasses this weird area where, when people complain that the language sucks, they are correct. There are a lot of things that are wrong with the language. There used to be even worse things with the language. The derided "PHP: a fractal of bad design"[1] does have a few correct points, even if those points were out of date at the time of publication over nine years ago.

However, at the same time, PHP allows developers to create structurally "correct" software and embrace ideas from other languages that are considered good practices. You have frameworks like Laminas and Symfony, which use best practices for object-oriented programming to allow developers to write properly structured code.

How does PHP do this? It is because PHP is the worst.

Designing Software

In 1991, Richard P. Gabriel published the essay "Lisp: Good News, Bad News, How to Win Big"[2]. The paper's thesis is that, when it comes to software design and longevity, the idea of "Worse is Better" will be the superior option. He came to this conclusion after realizing that two different schools of program design had emerged, which he designated as the "MIT/Standford Style" or "The Right Way" and what became known as the "New Jersey Style," or "Worse is Better."

The two philosophies were similar in their aims but different in key areas. Each style focused on five key areas of thought: Simplicity, Correctness, Consistency, and Completeness.

The MIT style[3] was described as:

- Simplicity: the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness: the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency: the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness: the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

As for the New Jersey Style[4], Gabriel says that it defines its goals as:

- Simplicity: the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
- Correctness: the design must be correct in all observable aspects. It is slightly better to be simple than correct.
- Consistency: the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases. Still, it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.

1 *"PHP: a fractal of bad design":* https://shorturl.at/agitC

2 *"Lisp: Good News, Bad News, How to Win Big":* https://dreamsongs.com/WIB.html

3 *MIT style:* https://dreamsongs.com/WIB.html

4 *New Jersey Style:* https://dreamsongs.com/WIB.html

- Completeness: the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

The crux of the argument uses LISP and C as examples for why worse is better. To Gabriel, a LISP programmer, LISP was a much better language than C. It was just as fast as C, and Common LISP had spent years being designed, developed, and standardized. The specification that defined the language had taken the best of all the various splinter LISPs, and modern development environments were some of the best around for LISP developers.

LISP was The Right Way

LISP represented "The Right Way" of software development. LISP was considered simple to interface with, and you could interact with it in various ways. Want to call LISP from Fortran? You can invoke LISP from Fortran and pass data in, and vice versa. You could happily use all the modern luxury of LISP while working with your legacy code.

LISP had a consistent design, thanks to its specification. If you look at a modern language like Python, a specification goes a long way in providing multiple backends and compilers that all interpret or compile the code in the same way. The tooling was top-notch, and 1991 LISP had all the creature comforts we still enjoy today, like step debugging, data inspection, and fancy editors.

As a language, LISP was complete. It had an advanced object-oriented programming layer, multiple inheritance, first-class objects as well as functions, and typing. LISP looked like the language that developers would want.

In 1991 LISP, as a language, was in probably the best shape it had ever been. This technical correctness was not borne out by actual usage. LISP development shops were on a decline. Years of bad press and positioning missteps had hindered LISP's external reputation. It was no longer being looked at as a way to deliver software to end-users.

"In 1991 LISP, as a language, was in probably the best shape it had ever been. This technical correctness was not borne out by actual usage.

In development terms, LISP tends to represent many of the same ideals as "Big Design Up Front." If you have worked with design methodologies like the Waterfall Model[5] you can already see where some of the issues crop up. "The Right Way" heavily stresses consistency, correctness, and ensuring that all conceivable issues are thought of.

LISP itself was also not a singular language but a family of languages. While Common LISP was meant to be a standard, LISP itself existed as a variety of actual implementations based on the work needed to be done. An article on Lockless Inc's website[6] calls out this fragmentation as one of the defining reasons that LISP ultimately failed. Even with LISP adhering to the "Right Way" of software design, the fragments were distinct enough that code maintenance and portability suffered.

C and Unix Were the Wrong Way

In the meantime, C was gaining ground as the preferred way to develop software, thanks to Unix. C was designed for Unix, and Unix was designed from C. Its developers did not take the same design stance as LISP and its authors at MIT.

In 1972, C was designed as a simple language. By 1991 it had changed somewhat, but the fundamentals of C had not changed. Features were added based on what developers needed and what Unix needed. Writing a compiler and programs was easy because the language was so simple. While the language did not stop you from doing complex programming, C had an estimated 50-80% of what programmers should want compared to LISP.

C was, however, incredibly portable. It also ran on underpowered hardware compared to what would usually be used for LISP software and environments. This factor opened it up to being able to compile and run software on a broader range of machines. C software, and Unix, were so easy to run, Gabriel considered Unix and C viruses.

Development of C occurred as Dennis Ritchie designed and built Unix. Unix was also easily distributed to various other users thanks to Bell Laboratories not being allowed to formally enter the computer space[7]. These other users helped patch Unix for their needs. Dennis Ritchie was able to incorporate those patches as needed versus having to think about those needs upfront.

Unlike LISP, C is still used quite a bit. While higher-level and interpreted languages like PHP, JavaScript, and Python are the go-to's for many developers, C is used to develop many of those higher-level languages. C is still used in smaller, lower-powered devices even with competitors like Rust starting to gain ground.

PHP is the Worst

> *Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be*

5 *Waterfall Model:* https://w.wiki/k2M

6 *Lockless Inc's website:* https://phpa.me/lockless-lisp-failed

7 *computer space:* https://w.wiki/43QN

> *improved to a point that is almost the right thing.*
>
> *– Richard Gabriel*

A few years after this revelation, Rasmus Lerdorf started working on Personal Home Page/Forms Interpreter, which we now know as PHP. PHP/FI was born out of a need for Lerdorf to maintain his home page and interact with forms and databases. PHP/FI was not even designed as an actual programming language but as a layer of scripts and functions on top of C.

PHP is Simple

> *The design must be simple, both in implementation and interface.*

Under the hood, PHP uses the C language, which we have already established is "worst." However, this brings a few advantages, with the most significant being that a more simple underlying language can make it easier to extend. While Hack/HHVM went with a more C++ approach, PHP itself is still in C.

Learning the internals of the language can be done in just a few hours. Elizabeth Smith has an excellent talk on PHP Extensions[8] that can be absorbed in a single sitting, and there is a wealth of information on how the internals work. The language itself borrows from other C-style languages, making it easy to read and switch to from other languages in the C-style family.

Much of PHP's interface, or standard library, is considered simple because much of the core functionality only wraps various C libraries and exposes them almost as-is. While this leads to some inconsistencies in the interface, it provides a familiar environment for developers coming from C or C++.

The PHP language is also heavily focused on web development. Taking a concept from HTTP and finding an analog in the language is usually straightforward. Want the headers for a request? `get_headers()`[9] has you covered. Getting request information is as simple as reading `$_GET` and `$_POST` global variables.

PHP keeps the developer interface simple and keeps internals as simple as possible.

PHP is (mostly) Correct

> *The design must be correct in all observable aspects. It is slightly better to be simple than correct.*

Here, PHP tends to err on the side of "simple" versus correct. Up until the advent of HHVM, there was no specification on how the language looked or functioned. The Zend Interpreter itself was the specification, and the language was always "correct" in how it behaved (excluding actual bugs). If you wanted to replace the PHP engine with something else, it would need to implement all the quirks of the existing engine.

8 *PHP Extensions:* https://www.slideshare.net/auroraeosrose

9 `get_headers()`: https://php.net/get-headers

Many of the core functions' lax function parameters and return types adhere to making working with the system more accessible. A function like `strpos()`[10] that returns either an integer or a boolean is slightly easier to handle than having a method that returns an integer or throws an exception.

Looking at how the language is evolving, almost all new functionality is based on things developers need versus a strict idea of "fix this because it's wrong." A larger focus on strict typing and exceptions-over-errors is a more correct way of doing things. Still, other things like short arrow functions, attributes, and enums are things that developers want to simplify their code.

PHP Does Not Have to be Consistent

> *The design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases.*

I am not even going to pretend that PHP is consistent, but it is consistent enough. People may complain about needle/haystack parameter order when it comes to arrays versus string functions. However, in general, array functions are consistent, and string functions are consistent. It is simpler to be consistent with the underlying C libraries than being consistent in the language.

PHP is consistent enough in other ways. As I mentioned with `strpos()`, PHP tends to be fairly consistent in returning `FALSE` for functions that encounter errors. That is not necessarily correct, but it is consistent. Function names with underscores and without underscores tend to match their underlying libraries.

PHP, the language, sacrifices consistency for simplicity, but even without a specification, it tries to be consistent where it makes sense.

PHP is as Complete as it Needs to Be

> *The design must cover as many important situations as is practical.*

At any given time, PHP is as complete as it needs to be to do what it was designed to do—write web applications. PHP was never designed to be a language that covered every single problem in the world of programming. Still, its simplicity lends itself to being used in situations outside of the web. Its initial focus on working with the web helped shape what features were initially needed, and that trend continues today.

Changes to the core language tend to be primarily driven by developer needs. The community at large puts forth changes, the community votes, and the new features are rejected, changed, or accepted. Much of the innovation in the language comes from the need to do our jobs quicker. Even when we are cribbing features from other languages, it is because it makes our development lives easier, and rarely is it because another language does it "more correct."

PHP allows you to build web applications today. It will still allow you to make web applications in five years, just with some new features. The language itself, however, is as complete as it needs to be today. We can always add to or change the language if we need to.

Is Worse Better?

Gabriel admits that the idea of "worse-is-better" is designed in such a way to look bad and probably should not be the better option. The only problem is that when he looks at the two ideas, "worse-is-better" still ends up being the more flexible option and "has better survival characteristics" compared to the MIT/"right-way" design philosophy. If we look at PHP, it corroborates the idea that worse is better.

In the intervening years, Gabriel has admitted that he waffled between which is actually better[11]. PHP as a community constantly argues whether we should do things correctly or continue to do things simply. We have frameworks like Laminas that build libraries in a classic computer science way, and then we have frameworks like Laravel that focus directly on developer experience and speed. PHP itself allows both.

The next time someone wants to rag on PHP, own it. The language sucks. But in many ways, the longevity and widespread use of PHP is a testament to the fact that doing things "the right way" is not always better than doing things the "worst" way. If someone complains about the framework you are using, understand that it does not matter in the long run. Pick a design philosophy that you feel is comfortable for you, and be happy knowing that being worse might actually be better.

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank

Related Reading

- *The Business Of PHP*
  by Sherri Wheeler, April 2020.
  https://phpa.me/business-php-wheeler
- *finally{}: The Seven Deadly Sins of Programming: Envy*
  by Eli White, March 2019.
  https://phpa.me/finally-mar-19
- *finally{}: 25 Years of PHP*
  by Eli White, August 2019.
  https://phpa.me/finally-aug-19

10 `strpos()`: https://php.net/strpos

11 *actually better:* https://dreamsongs.com/WorseIsBetter.html