www.phparch.com

php[architect]

July 2020
Volume 19 - Issue 7

Warp Driven Development

Event-Driven Design With Drift PHP

Getting Started with Test-Driven Development

The Zen of Ego-Free Presenting

ALSO INSIDE

Education Station: Writing Concise Code

Community Corner: PHP 8 Release Managers: Interview with Sara Golemon and Gabriel Caruso, Part 1

The Workshop: Twig, Bulma, and CodeIgniter 4

Sustainable PHP: We Got Robbed

Security Corner: Information Tokenization

PHP Puzzles: Generating Random Loot

finally{}: The Dangers of Intellectual (Sounding) Arguments

Writing Concise Code

Chris Tankersley

There is a huge emphasis put on the code maintainability, and for a good reason. The programming industry is rife with what we like to term “legacy code,” which boils down to code that solves a business problem, but we, as the current maintainers, do not understand. No language is safe from this problem.

As a codebase grows older, the lessons learned during construction get lost to the aether. Unless we’re diligent in updating documentation and capturing intended behavior via tests, the business decisions that led to why a block of code is written a certain way disappear with the writers. The current maintainers—us—are left not to put together a puzzle, but understand how that square peg managed to fit in the round hole.

The first step any developer can take is the idea of “Self-Documenting Code.”[1], and making sure the code indicates what it is doing. Self-documenting code helps reduce the amount of technical documentation which needs to be maintained by following rules that make code more explicit. Doing so can be achieved through well-named classes, methods, and variables, and leaving rationales as comments. Being explicit is preferred over being clever.

An aspect of this that gets somewhat overshadowed is the idea of concise code. Concise code should result in less code to maintain, so I don’t believe it goes against the “be explicit” rule.

I do not mean we should begin writing the shortest amount of code possible (remember, do not try and be clever). We can do plenty of things in PHP to help us write less code, but still be explicit in what we are doing.

The Ternary Operator

The ternary operator (`?:`) is generally the first concise operator many developers encounter. We call this operator “ternary” as it operates on three expressions to produce its result:

1. a condition expression,
2. an expression to invoke if the condition is `true`,
3. and an expression to invoke if the condition evaluates to `false`.

```
(expr1) ? (expr2) : (expr3)
```

`expr1` is used as a testing expression, and this should evaluate to either a `true` or `false` value. `expr1` can be a variable that is a Boolean value, a comparison expression, or a function call that returns a value. What matters most is that its goal is to provide a `true` or `false` condition.

1 *“Self-Documenting Code.”*: *https://phpa.me/wikip-self-documenting*

`expr2` is the expression returned if `expr1` evaluates to `true`. It can be a value, another expression, or a function call to do more work. It is considered a best practice to keep this expression short and as uncomplicated as possible, which helps avoid readability problems and odd resolution order bugs.

`expr3` works the same as `expr2`; however, it is only evaluated if `expr1` is considered false.

```
$state = true;
echo $state? 'Valid' : 'Invalid';
// True
```

In PHP, we frequently use the ternary operator as shorthand for an `if/else` block. This practice helps clean up code when performing small operations. For example, if you are writing a number guessing game and want to output a Win or Lose message, you could write it with an if/else statement:

```
$win = 10;
if ($guess === $win) {
    echo 'You Win!';
} else {
    echo 'You Lose!';
}
```

Since the logic between the `if` and `else` statements are relatively simple, we can turn that into a ternary. We can move the comparison into `expr1`, the winning output into `expr2`, and the losing output to `expr3`.

```
$win = 10;
echo $guess === $win ? 'You Win!' : 'You Lose!';
```

The ternary operator is a great way to help shorten simple logic. If we need to do a lot more work between the `if` and `else` flows, then the ternary operator tends to get very complicated. Only use the ternary operator for simple operations.

Yes, you could chain together ternaries. I would heavily caution against doing so, as readability suffers. Also, the actual execution flow becomes quite tricky to keep track of. While it may seem like a good idea because the chain looks simple, the long-term maintenance cost goes through the roof exponentially.

The Shorthand Ternary

There is an alternate syntax (since 5.3) for the ternary operator, commonly called the Shorthand Ternary (which not only is a longer name but turns it into a binary operator—we can ignore the existential crisis this causes for the ternary operator). In this format, `expr1` is just returned in place of `expr2`.

```
(expr1) ?: (expr3)
// Expands to  this: (expr1) ? (expr1) : (expr3)
```

This operator is useful when your `expr1` returns a “truthy” value, or a value PHP considers `true`. PHP is a dynamic language and still holds to the idea that different values can be regarded as `true` or `false` for many comparisons that are not strict (using `===`). The PHP documentation has a type comparison[2] table that helps describe this.

If you have an ecommerce site, one of the things you may want to show to the user is the number of items in their basket. If the user has no items, you want to show the statement “no items” instead of “0 items” to lend it more of a conversational tone. We can take advantage of the fact an empty array counts to 0, and 0 is considered falsey.

In Listing 1, `count($basket)` is evaluated and returns either `2` or `0`. Since PHP considers non-zero integers as `true`, the shortened ternary returns `2` in the first case, as it is considered “truthy.” In the second case, `0` is evaluated as `false`, so it shows the word “no.”

Listing 1

```
<?php
$basket = [
    ['item' => 'Plant', 'cost' => '12.99'],
    ['item' => 'Shovel', 'cost' => '23.99'],
];
echo 'You have ' . (count($basket) ?: 'no')
    . ' items in your basket';
// You have 2 items in your basket

$basket = [];
echo 'You have ' . (count($basket) ?: 'no')
    . ' items in your basket';
// You have no items in your basket
```

The Null Coalescing Operator

PHP 7 introduced another similar syntax, called the Null Coalescing Operator (`??`). Null coalescing is close to the shorthand ternary in use, but it has a slight distinction in that, instead of testing for True or False, it tests for existence and `null` values. This becomes useful when looking to see if things like array elements or return values are defined, versus whether they test to a Boolean.

```
(expr1) ?? (expr2)
```

The syntax for this is succinct. If `expr1` is defined and is not null, return `expr1`. Otherwise, return `expr2`. This code is very close to the shorthand ternary, but `expr1` is not evaluated for truthy-ness, just existence and value. `expr1` can evaluate to a false value and still be returned.

A typical example is hydrating an object from an array. The object can have a method named something like `fromArray()`, which takes an array and assigns the values of the array to properties of the object. Since incoming array keys may not exist, you have to do a lot of checking to make sure they do, generally with the ternary operator, as in Listing 2.

Listing 2

```
class MyObject
{
   protected $id;
   protected $name;

   public function fromArray(array $data) {
      $this->id = isset($data['id']) ? $data['id'] : uniqid();
      $this->name = isset($data['name']) ? $data['name'] : '';
   }
}
```

Listing 3

```
class MyObject {
    protected $id;
    protected $name;

    public function fromArray(array $data) {
        $this->id = $data['id'] ?? uniqid() ;
        $this->name = $data['name'] ?? '';
    }
}
```

While the lines of code do not change, we can cut some of the boilerplate code needed by switching to the Null Coalesce Operator. It makes the code easier to scan and read (see Listing 3).

I use it all the time when checking to see if array elements exist and defining default values. Since it acts like `isset()`, it can save a lot of additional typing and makes it clear we are pulling the value of the array element versus something else.

2 *type comparison: https://php.net/types.comparisons*

The Null Coalescing Assignment Operator

A close cousin to the Null Coalescing Operator is its assignment operator (??=), which dropped in PHP 7.4. It allows you to shorthand assignment of a variable if it does not exist, and works in combination with arrays with the implicit `isset()`.

```
(var1) ??= (expr1)
```

With the Null Coalescing Assignment Operator, if `var1` does not exist, it is created and assigned the value of `expr1`. If it does exist, it is left alone. This behavior is slightly different in that it creates a variable, where the standard Null Coalesce Operator returns a value.

If we have a function that takes a set of parameters, we can use the Null Coalescing Assignment Operator to set defaults on the values passed in, in case the user forgets to add them. See Listing 4.

Listing 4

```
function makeRequest(string $uri, array $options) {
   $options['timeout'] ??= 1;
   $options['method'] ??= 'GET';
   $options['limit'] ??= 10;
   $options['page'] ??= 1;

   return $client = HTTPClient::makeRequest(
      $uri,
      $options['method'],
      [
         'timeout' => $options['timeout'],
         'page' => $options['page'],
         'limit' => $options['limit'],
      ]
   );
}
```

Arrow Functions

PHP 7.4 brought not only Null Coalesce Assignment but also the idea of the arrow function[3]. This is an alternative syntax for writing single-line anonymous functions in a vaguely similar syntax to JavaScript's arrow function[4] expressions.

```
fn(<params>) => <single line of logic>;
```

Let's look at a simple function that echoes out "Hello" and then some other word. We can write this up as a little three-line anonymous function without much effort:

```
$fn = function($name) {
    return 'Hello ' . $name;
}
echo $fn('World'); // Hello World
```

Arrow functions allow us to take small anonymous functions that only have a single unit of logic to them and rewrite them as a one-liner. Just like any other function, arrow functions generate their own scope, can pass and return data by reference or value, and can take advantage of variadics.

```
$fn = fn($name) => 'Hello ' . $name;
echo $fn('World'); // Hello World
```

There are a few things to keep in mind with arrow functions, however. The first is that they automatically return whatever value they generate. In the case of our example, it produces the string `"Hello ${name}"` and returns it, even without the `return` keyword. Arrow functions are meant to manipulate parameters and variables, and immediately return data.

The second is the implicit value binding. In a normal anonymous function, you can bind an external variable to the function with the `use()` construct. With arrow functions, any variables within the defining scope are automatically passed, by value, into the arrow function. You cannot bind via reference, for that you need to use a full anonymous function.

```
$name = "Bob";
$fn = fn() => 'Hello ' . $name;
echo $fn(); // Hello Bob
```

Where is a good case to use arrow functions? They are often used with methods that take a callback to manipulate data, like many of the array manipulation functions. Arrow functions can be cleaner than having the additional lines a full anonymous function generates.

```
$array = [1,2,43,5,];
$val = array_reduce($array, fn($carry, $item) => $carry + $item);
echo $val; // 51
```

Avoid Removing Curly Brackets

There are some syntaxes you should avoid because they cause problems in the long run. Remember, concise code is meant to help remove code where it makes sense, not necessarily to save yourself time or keystrokes. If your goal in programming is to write the fewest characters, re-evaluate your priorities.

One of the oldest "shortcuts" is one-liner `if/else` statements. These are `if/else` statements where we can remove the curly brackets if the code block is a single line of code. Here, PHP follows how many curly bracket languages work. However, it is also very easy to introduce bugs with it.

```
$val = true;
if ($val)
    echo "It's True!";
else
    echo "It's False!"
```

The problem with this syntax is that without the brackets, it is effortless to allow logic to seep out of the `if/else` statements, and even introduce syntax errors. If we add one additional line of logic to the `else` portion and forget to add the curly

3 *arrow function: https://php.net/functions.arrow*

4 *JavaScript's arrow function: https://phpa.me/mozdev-arrow-functions*

brackets, the new line runs regardless of the value of `$val`.

```
$val = true;
if ($val)
    echo "It's True!";
else
    echo "It's False!"
    die("Only die if we are false");
```

Since PHP does not care about whitespace, the interpreter sees the `if/else` without curly brackets, associates one the single line to those control blocks, and the `die()` statement, being the second line not included. It runs every single time.

Concise Does Not Mean Clever

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

– John Woods

The goal of any maintainable system should be clarity of the code, but that does not mean we should have to write out everything long-form. There are some delightful syntax structures PHP has developed/stolen over the years that can help.

If you are interested in strictly writing the fewest characters of code, I recommend looking into Code Golf[5]. There are various implementations, but the idea is to write the smallest physical amount of code possible to solve a problem. The code does not need to be intelligible; it just needs to function.

Outside of code golf, make sure you follow the general rule of self-documenting code. If you do not think you will understand what a specific block of code does in six months, rewrite it to be more apparent. All the better if you can shorten code without making it less clear.

5 *Code Golf: https://phpa.me/wikip-code-golf*

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

- *Education Station: Writing DRY, SOLID FOSS OOP CRUD Code* by Chris Tankersley, August 2019. https://phpa.me/education-aug-2019
- *The Life-Changing Magic of Tidying Your Code* by Bryce Embry, May 2018. https://phpa.me/embry-tidying-code
- *Building Software that Lasts* by Susanne Moog, October 2017. http://phparch.com/magazine/2017-2/october

CONTENTS
Mutant Testing
24
29
Leveling Up: Ensuring Your Tests are Valuable
Columns
ALSO INSIDE