# BASECODE

a field guide to writing code that lasts

by Jason McCreary

# BaseCode

*A field guide to writing lasting code*

by Jason McCreary

*“To those about to read, we salute you...”*

# Table of Contents

# *Bootstrap*

## *A tale of two quotes*

I was traveling to the lake with the family. Although I normally use this time to unplug, I take a book to read during the drive – which is normally related to programming. What can I say, I love what I do. I'm also a slave to efficiency.

At the time I was reading Implementation Patterns by Kent Beck. It was one of the shorter books on The Reading List recommended while interviewing with companies in Silicon Valley. I had no idea it would contain one of the universal truths about programming. One which would forever change the way I write code.

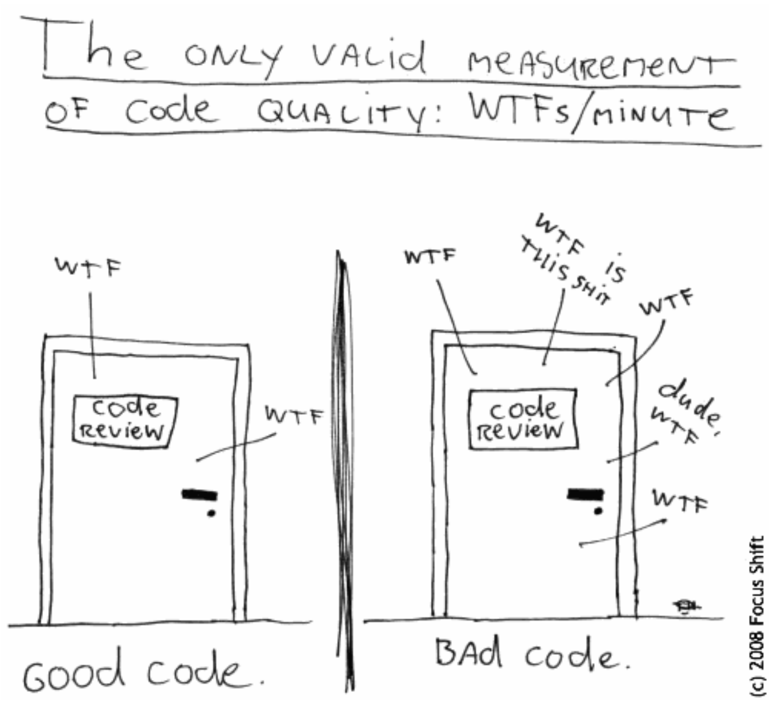
> *Programs are read more often than they are written.*

All of my experience supported this. Every time I reviewed an answer on *StackOverflow* or browsed the source code of a project on GitHub I was **reading code**. Even when I wrote code, I read several lines of exiting code to determine where to insert the new code.

I often paraphrase the original quote by simply stating:

> *We read code more than we write code.*

Given this universal truth, how could I improve the code I write? Well, I could prioritize *readability*. Instead of playing code golf with fancy one-liners I could be more expressive. After all, another programmer is going to *read* that code in the future and ask, “WTF does this do?”



*Readability* has become my measure of code. It influences all the code I write. If I alter code in any way it is for **readability**. Not how many abstractions or patterns are used. But how readable the code is. When someone asks, “which code is better”, the deciding factor is readability.

All through college I interned as a web programmer. I did all the grunt work like markup web forms and save their data to a database. Although most of this was simple content it was before the days of WordPress or modern full-stack frameworks.

So I did what any young programmer would do - I wrote my own. It was not just a CMS, it was a **CMS generator**. I created a YAML-esque domain language which I parsed to create all the things. It generated the HTML and PHP for all the cruddy web forms. It generated the MySQL `CREATE TABLE` statements. It injected the forms into templates to provide all the admin interfaces. I made it an active generator, so when a new field was added all I had to do was update the config file and I could regenerate entire sections of the CMS.

It was awesome. I was the best programmer alive.

I believe every programmer needs to experience this at least once. It's really the only way to realize how insane it is. Now I look back and ask myself:

- Why did I need my own domain language?
- Why did I need an *active* generator?
- Why did I need to support every HTML input type?

The truth is **I didn't**. You rarely *need* complexity. In the moment, it's easy to believe you do. But in retrospect I would've been far more productive with a simple script to generate a majority of the web form fields and coded the rest manually.

Only through reflection on your own similar experience can you realize the next universal truth:

> *Most programs are too complicated – that is, more complex than they need to be to solve their problems efficiently.*

Rob Pike said this in one of his essays in 1989. Things have only gotten more complicated since. When I interned, it was common for web programmers to write an entire site in a basic text editor using only HTML. Now web programmers need to know JavaScript and CSS (really pre-processed, supersets of JavaScript and CSS), package managers and build tools, and multiple frameworks.

Neither approach is right or wrong. But one is obviously more complex than the other. Complexity is an easy sell to programmers, and for whatever the reason, we buy in bulk. We pre-architect entire systems before writing a single line of code. We reach for using the new shiny tool. We force design patterns or data structures into our programs.

We need to remind ourselves **most code is too complex**.

Start simple and keep it simple for as long as you possibly can. It's likely you aren't going to need what you thought (YAGNI). In the rare cases you do, the solution is often self-evident, without requiring any speculation.

When I started writing `BaseCode` I wanted to focus solely on *readability*. Ironically, to keep it simple as a single motivation. Yet, when we put readability and complexity side by side, we notice a subtle distinction - simple code is not necessarily readable and, conversely, complex code is not necessarily unreadable.

In the end, while readability is the primary motivation, we need to balance complexity as well. All of the practices within `BaseCode` aim to improve *readability* and reduce *complexity*.

## Why these practices matter

`BaseCode` takes a bottom up approach. The goal is to get back to the basics. Because of this some might dismiss these practices as too trivial. They think they don't matter. This is a mistake.

From the same essay on complexity, Rob Pike goes on to say, "programs are often complicated at the microscopic level". He's referring to the fundamental aspects of the code.

All code consists of the same fundamental elements: variables, control structures, etc. Complexity at a low level bubbles up to complexity at a higher level. By adopting practices which focus on the elements of all code (the *base code*), we can write any code to last.

## Why a field guide

Reading all of the books that went into `BaseCode` would take years. Robert C. Martin wrote a great book on Clean Code that alone is 412 pages. I've read it as well as many other books on *The Reading List*.

That's not to say you shouldn't read these books. More that most books contain fluff. I mean no disrespect. I just wanted `BaseCode` to be very focused. I intentionally kept it short. As such, I didn't feel right calling it a *book*. `BaseCode` is more like the last chapters of Refactoring – a catalog of specific code practices. I landed on *field guide* because this feels more akin to an army manual or survival guide.

The practices in `BaseCode` are the result of the books I've read, the programmers I've paired with, and the code I've written over 20 years. I spent a lot of time distilling each practice to its purest form to avoid overlap. Each one is pragmatic so you can start applying them immediately to your code.

## *How to read* `BaseCode`

You can read this cover to cover like any other book. It's short enough you may be able to get through it in a day. You can start with the first practice, apply it, then move on to the next practice.

However, you don't need to read it cover to cover. You are welcome to choose your own adventure. If you have a codebase with a lot of comments, read *Removing Comments*. Need inspiration for finding better names, read *Naming Things*. Been copy and pasting a lot of code, reread Rule of Three.

The practices don't necessarily build upon the previous. However, they are progressive. The first set of practices are easy – ones you can apply immediately. The next set take a little more work. The final few practices may take a lifetime to master.

No matter the order, I would suggest reading one practice at a time. Once you have, apply it. Implement it into your codebase. Have your teammates review it. Discuss it. Challenge it. If you get stuck reread the chapter. If you get really stuck, email me or message me on Twitter (seriously). I believe each one of these practices improves code readability and decreases code complexity. If it doesn't, I want to know about it.

I also encourage you to read the entire field guide. All of these practices work together in harmony. Missing any one leaves opportunity for *noise* within your code. Over time that may grow louder and louder, drowning out the others. Keeping your code readable and avoiding complexity takes a lot of discipline. You must apply these practices consistently.

## *Accept the challenge*

These practices are opinions which deal with subjective matters. I don't expect you to agree with all of them. You may even find some controversial. I know we're passionate programmers. As such, we can take things personal and become defensive of the code we write.

I remember a discussion regarding separation of concerns with a respected peer. As the new guy on team, I challenged his opinion. This was fine. What wasn't fine was making it a zero-sum game. There had to be a *right* and *wrong*. He recognized this and ended the discussion by saying, "I was where you were a year ago. You'll get there." This, of course, pissed me off. Now it was personal - after all he was challenging my experience. A year later, my view aligned almost exactly with his. It wasn't personal. He was just a little farther down the road than me.

We're all on a journey. My goal is to share these practices as I believe they will improve the code you write. They will challenge the way you write code. As such, they will challenge you. It's not personal. They're so that you may take your journey a little farther.

# *Formatting*

Let's jump right in, shall we… When it comes to formatting there are a few kinds of programmers. There are those who don't care. They just write code. Aside from the chaos, this might actually be better than the other two.

The next are those who make an effort to format their code. However, they may not apply it consistently across the codebase. After all, we're human.

The final kind is those who really care. I mean they really, really care. I was this latter kind. If the code wasn't formatted, I'd spend time formatting it before I began writing new code. If another programmer didn't format the code, oh man, I'd let them know.

I remember being team lead and we had adopted a *modified* version of PSR-2 as our PHP standard. We were in a review meeting and I was, of course, criticizing the formatting. One of my teammates replied with "why does it matter?".

At the time, this pissed me off. From my perspective, we had agreed upon a standard as a team and they still didn't follow it.

I'm less anal about it now. Nonetheless formatting remains a controversial topic. It leads to holy wars such as tabs versus spaces and K&R versus Allman.

You have to let go. I know that can be hard. After all, formatting is one of those personal marks we imprint on the code. But is this really the lasting mark you want to leave on the code?

## *Why formatting actually matters*

My teammates question still lingers. It deserves an answer. At the time, I was so focused on the action I never thought about the reason behind it. The reason is the exact motivation behind `BaseCode` – improving readability.

Malformed code is incredibly hard to read. So, I'd take the time to format it in an effort improve its readability. But this was misguided. I'd often spend more time reformatting the code than I would writing new code.

The trolls will argue code format is subjective. A format one programmer finds readable another programmer may not. This may appear true, but only on the surface.

The underlying process of reading is **not** subjective. We all learn how to read. We train our brains to turn letters into words, words into sentences, and so on. Over time we rely on a certain cues, such as formatting, to make the process of reading easier. If I were to say, start m e s s i n g with the fo rm at ,it would bemuch harder to read.

Reading code is no different. In fact, Douglas Crockford states this in JavaScript: The Good Parts:

> *It turns out that style matters in programming for the same reason that it matters in writing. It makes for better reading.*

As programmers, we unconsciously rely on similar cues to make the process of reading code easier. We recognize structures like assignment statement, if statements, and method blocks. When the code is well formatted we can recognize these structures easily. If the code format is malformed or the format constantly changes, it makes it harder for us to recognize these structures.

Too often programmers focus on the syntax. When talking about formatting, we need to focus on readability, not syntax. We need a way to represent the code as we see it *unconsciously*. Kevlin Henney demonstrates this through a visual representation of code.

To do so, he replaces characters with X and eliminates punctuation (like line ending semicolons). What we are left with is a visual representation of the code. Or how our minds *see* the code.

So if we have the following simple assignment statement:

```
variable = value;
```

Its visual representation would be:

```
XXXXXXXX X XXXXX
```

Let’s look at another visual representation:

```
XX XXXXXXX XX XXXXXXXXXXXXXXXXXXXXX XXXXXX XXXXX
XXXX XXXXXX XXXXXX
```

What is this code? What does it do? How much do we know simply from the visual representation? Maybe you can identify something, maybe you can’t…

Now let’s add some formatting with simple whitespace adjustments:

```
XX XXXXXXX XX XXXXXXXXXXXXXXXXXXXXX
    XXXXXX XXXXX
XXXX
    XXXXXX XXXXXX
```

Have we learned more about the code? It’s the same code, just formatted differently. Even though we don’t see the actual syntax, as a programmer we pick up on certain cues. We see paired blocks of code. We see indentation levels. We see the top levels start with two characters and four characters. In a fraction of a second, we unconsciously identified this structure as an `if`/`else` block.

There’s no denying the second format relays more information than the first. All we did was format the code in a *standard* way. Formatting the code has the largest impact on readability. Code that is properly and consistently formatted has what Kevlin Henney calls “visual honesty”. That is the visual representation and the actual representation align.

We now realize format is not as much about *what we like*, but *how we see*. By letting go of our own personal format and adopting a standard format, we can make it easier for every programmer to read the code.

## *What to do*

Code formats can vary from person to person and language to language. They can change over time. So what do we do?

First, understand programming is a team sport, even if the only members of the team are you and future you. You have to let go of your individual formatting for the benefit of the *team*. Being anal benefits no one. You'll go mad formatting all yourself or drive your team crazy making them adhere to your format.

Second, the tiny nuances of the formatting don't matter. Tabs versus spaces or K&R versus Allman, they don't *really* matter. What does matter is a format is applied consistently so we can improve the readability by increasing the "visual honesty".

So, here's what you do about a formatting:

1. Choose it
2. Automate it
3. Forget it

### *Choosing a format*

I will not dictate a specific code format. As noted, the format naturally varies between programmers and languages. Therefore, it's highly unlikely you'll find one to fit all programmers or all languages. You must come to terms with the fact that whatever the format you choose may not be the one you want or the one you use in other languages. That may be difficult at first. It was for me. But sooner or later you'll realize your time is more valuable than to waste formatting code.

I strongly recommend adopting a **standard** format. Nearly all languages have a common (popular) code format. Remember every detail of this format may not match your own. That's OK. You are welcome to attempt to adjust it, but as before I think you will find that to be a waste of time.

For example I often write PHP. As such I adopted the PSR-2 code style. For JavaScript I use StandardJS. For Objective-C and Swift I use the style applied by my IDE (Xcode).

Adopting a common standard not only relinquishes the governance of formatting, but also means my code aligns with other code written in that language.

## *Automate the format*

You have better things to do with your time than format code. Let me say that again because it took me a long time to realize this. You have better things to do with your time than format code.

Choosing a format is the hardest step. The next step is applying and maintaining that format. First find a tool to automate this process. Your IDE is good. Being able to run a script or service is even better. For example, when writing PHP, I run PHPCodeSniffer to verify I adhere to the PSR-2 code style. If the format you have chosen cannot be applied in an automated way, you should reconsider the format you chose.

Once you have found a tool that can automate your format, run it across your entire codebase. Check in all of those changes as a single commit (yes, you're using version control). Some programmers will not like this. Their common reason is that it messes up the commit history, specifically when running commands like `git blame`. While having formatted code far outweighs placing blame on a previous code change, this can be remedied. If you are this programmer or have this programmer on your team, you can pass a commit reference to `git blame` to run blame against the history before the mass formatting.

## *Forget it*

Now that you have chosen a format and applied it to your codebase everything is great, right? Well not really. Over time your code will fall back into an unformatted mess. This is inevitable. The result of laziness and new programmers.

Violations in format should be treated as a syntax error. Programmers should not be able to merge code unless it is formatted to the chosen standard. Again, you can automate this with a validator. Most of these tools can also correct any violations.

However, this should not be something a programmer has to think about. Once you've done the two previous steps you should be able to forget about formatting. It's something that just happens.

## *Closing example*

Let's close with some real world code I encountered in my last project. We'll continue to clean up more of this code through other practices. For now, we'll focus on formatting.

```
function check($scp, $uid){
  if (Auth::user()->hasRole('admin')){
    return true;
  }
  else {
  switch ($scp) {
    case 'public':
      return true;
      break;
    case 'private':
      if (Auth::user()->id === $uid)
        return true;
      break;
    default: return false;
  }
  return false;
  }
}
```

Of course, I can muddle through this code. But it's not easy to read. The formatting is nonexistent, or at least inconsistent. Even though it's a simple `switch` statement, it's "visually dishonest". We can see this through the visual representation:

```
XXXXXXXX XXXXXXXXXXX XXXXX
  XX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    XXXXXX XXXX

  XXXX
  XXXXXX XXXXXX
    XXXX XXXXXXXXX
      XXXXXX XXXX
      XXXXX
    XXXX XXXXXXXXXX
      XX XXXXXXXXXXXXXXXXX XXX XXXXX
        XXXXXX XXXX
      XXXXX
    XXXXXXXX XXXXXX XXXXX

  XXXXXX XXXXX
```

By automatically applying a standard format we can immediately improve the readability with very little effort. Before looking at the code, we can see

```
XXXXXXXX XXXXXXXXXXX XXXXX

    XX XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXX XXXX
    XXXX
        XXXXXX XXXXXX
            XXXX XXXXXXXXX
                XXXXXX XXXX
                XXXXX
            XXXX XXXXXXXXXX
                XX XXXXXXXXXXXXXXXXX XXX XXXXX
                    XXXXXX XXXX

                XXXXX
            XXXXXXXX
                XXXXXX XXXXX

        XXXXXX XXXXX
```

From the visual representation it’s still a bit busy. However, it is nonetheless formatted. This reveals other opportunities to clean up the code. By not wasting our time focusing formatting, we are free to focus on other more important things, such as improving the code through other practices.

For now, here’s the formatted code.

```php
function check($scp, $uid)
{
    if (Auth::user()->hasRole('admin')) {
        return true;
    } else {
        switch ($scp) {
            case 'public':
                return true;
                break;
            case 'private':
                if (Auth::user()->id === $uid) {
                    return true;
                }
                break;
            default:
                return false;
        }
        return false;
    }
}
```