

# Tame the Name

by Mike Clark

I REMEMBER WHEN MY DAD, who builds houses, told me that there's a right way and a wrong way to cut trim. The right way requires a bit more thought and a few more measurements, but it works better even if the person paying for the work doesn't immediately notice the difference. So you do it the right way—at least until you're good enough to know when to make an exception. "How did he know that?" I wondered. It was only after cutting a few lengths of trim myself, and struggling to rework existing trim, that I understood Dad's wisdom.

Even though I was a dedicated apprentice of my dad's work, I never put the time into becoming a master. Instead, I discovered that I had a passion for another craft: writing code. Like the smell of fresh oak to a woodworker, I'm drawn to a blank editor window. Consequently, I write a lot of code, and read even more. And with my passion for coding comes a desire to share what I've learned along the way, knowing that it will further hone my skills. To that end, this new series of articles is about writing good code as a craft, rather than code being merely one turn of the crank after a good design.

## Responding to Change

Why focus on the craft of coding? A craft is characterized by internal standards of quality that aren't always obvious to the customer. If the code works as expected, the customer may not care about the code's internal quality. It's only when the code must be changed to make room for a new feature or to fix a bug that the cost of poorly written code becomes noticeable. And it's the ability to respond to change economically, which the customer does care about, that separates good code from bad.

If it will take until the heat death of the universe to retrofit the next feature, you're working with bad code. Where you put your curly braces and how your code ranks in terms of static metrics is largely irrelevant. What is relevant is the speed at which the code can be changed. So, to continually keep the cost of change at a minimum, we need to be thinking of axes of change all the time we're coding.

Like any skill, crafting code that can be easily understood and modified takes



practice. In this series, we'll practice the craft techniques involved in writing code that tolerates change. We won't shy away from the fundamentals because they lay a crucial foundation for the other techniques. And there's nothing more fundamental about crafting good software than communication.

## Call It By Name

One of the joys of being a creator is that you get to name things. It's a power graciously bestowed on all programmers, regardless of their experience. And yet picking names for classes, methods, and variables is a big responsibility because these names are the language of the software. A name laid down once in a flurry of keystrokes and left unchanged will be read many times over by those who follow. And

for all of its virtues, the compiler offers no help when it comes to choosing names.

Choosing meaningful names for classes, methods, and variables is the single-most important thing you can do to make your code understandable. Simply put, you can change code that you understand in less time than code that's convoluted with cryptic and misleading names. This seems so obvious, but having reviewed a goodly amount of code over the years, and having personally struggled with the name game, I know that naming bears repeating.

But it's not enough to repeat the obvious. Developing good naming skills takes practice. Thankfully, writing code affords us many opportunities to practice the naming technique. What should I call this thing? Better question: What is it I'm creating? Indeed, there's a parallel between

choosing good names and test-first development: They both force you to think before you code. Whenever I'm unsure what to call something, I find it beneficial to take my hands off the keyboard to think of a good name before the moment passes me by. It's in that quiet moment that I consider whether the name is pronounceable, expressive, and contextual—the three basic qualities of a good name.

### What Did He Say?

One effective way to write code for humans to read is to choose names that you, a human, can pronounce . . . in public. It's a basic litmus test that often has surprising results. For example, you're bound to raise a few eyebrows if you walk about the office talking about the `frgnky` variable. Usually buying a vowel or two (`foreignkey`) does wonders for your pronunciation. The code will thank you, too.

### What Did He Mean?

Good names are expressive, but they need not be verbose. A person who practices the coding craft learns how to strike a balance to come up with a name that's clear and concise.

Choose class names that express the purpose of the class as a noun or noun phrase. The `CalcLifeExpAndPrem` class, for example, was likely a function in its previous life.

It often helps to write a sentence describing the class, and then pick nouns you thought of when you weren't thinking of code. When it comes to code, we programmers tend to have an expanded vocabulary and we're quick to show it off. For example, our first thought might be to rename the class as `CalcSessionFacadeEJB`. Unfortunately, the name primarily describes the design pattern and technology used at the time the class was born (modern-day Hungarian notation) but says nothing about the class's purpose in life. Simply calling it a `LifeInsuranceCalculator` and placing the class in an appropriate package reveals its purpose, while not exposing the secrets of its inner workings. The new name also makes the class a beacon of refuge for wayward functions. And when it's no longer cool to be

named after a pattern, the class won't be relentlessly teased by its peers.

Choose method names that use a verb phrase to describe what the method does, not how it does it. This implies that you know what the method will do before choosing a name. This is a good thing. If you pick a name before thinking through the method's purpose, you usually end up with a generic method name such as `doIt()` or `process()`. Methods with these ambiguous names lack cohesion and become dumping grounds for stray bits of code. You can go overboard by being too descriptive, as in the case of the `sortMessageListByNameForDisplay()` method. It's expressive, but unclear because it leaves us with a question: What happens if we call it but we don't display the sorted message list?

Variables should describe the data they hold or the object they reference. If variables don't help you remember what they represent, then they need to be renamed. Two confused variables, `actn` and `actno`, were once found wandering aimlessly. When questioned, they responded that they represented "action" and "account number," respectively. At least their creator knew what they meant, right up until he went insane.

### What Was He Thinking?

Context matters when it comes to naming things. If the programmers need a generic class that represents a queue of objects, then call it a (wait for it) `Queue`. That name speaks volumes to a coder. However, when you use an instance of the `Queue` class in business logic (code that's specific to the problem domain) choose a variable name in the business vocabulary. For example, creating a `pendingOrders` variable that references a `Queue` object makes sense in the context of your order fulfillment software.

But don't go too far with context by embedding contextual information into names. If your `Queue` class can hold any object, then don't diminish its usefulness across your system by calling it the `OrderQueue`. And while you may be quite fond of your project's `Queue` implementation, there's no need to name it `AcmeProjectQueue`. This extra context may be meaningful today,

but it becomes meaningless when the project name changes. Putting the `Queue` class in an appropriate package prevents any namespace collisions.

### Second Chances

Of course, you won't always choose a good name the first time. Often the right name comes only after you've read the code a few times. You live and learn, but that's no excuse to walk away, having carelessly wielded your naming power. Skilled code artisans don't think twice about changing names to improve clarity, because to do otherwise would cause their work to quickly decay. And with good tool support, renaming is both safe and inexpensive.

Human-readable code is paramount to preserving the value of your code over time. All the other coding craft techniques we'll practice throughout this series will build on good naming skills. So take the time to think up good names, even if it means taking a few extra measurements. Naming your creations, and taming those created by others, is not always easy or immediately noticeable, but it pays for itself every time the code is modified. Until next time . . . **{end}**

---

*Mike Clark is a consultant, author, speaker, and programmer. He is the author of Pragmatic Project Automation (The Pragmatic Bookshelf, 2004), a frequent speaker at software development conferences, and the creator of several popular open source tools. Mike helps teams build better software faster through his company, Clarkware Consulting, Inc. (<http://clarkware.com>). Contact him at [mike@clarkware.com](mailto:mike@clarkware.com).*

### Don't Stop Now!

Log on to **StickyMinds.com** and join Mike Clark and your peers in a conversation about this issue's topic. At the end of the digital column, add your views or just read what others have to say.