The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition
Topic 38: Programming by Coincidence

Do you ever watch old black-and-white war movies? The weary soldier advances cautiously out of the brush. There's a clearing ahead: are there any land mines, or is it safe to cross? There aren't any indications that it's a minefield—no signs, barbed wire, or craters. The soldier pokes the ground ahead of him with his bayonet and winces, expecting an explosion. There isn't one. So he proceeds painstakingly through the field for a while, prodding and poking as he goes. Eventually, convinced that the field is safe, he straightens up and marches proudly forward, only to be blown to pieces.

The soldier's initial probes for mines revealed nothing, but this was merely lucky. He was led to a false conclusion—with disastrous results.

As developers, we also work in minefields. There are hundreds of traps waiting to catch us each day. Remembering the soldier's tale, we should be wary of drawing false conclusions. We should avoid programming by coincidence—relying on luck and accidental successes—in favor of programming *deliberately*.

How to Program by Coincidence

Suppose Fred is given a programming assignment. Fred types in some code, tries it, and it seems to work. Fred types in some more code, tries it, and it still seems to work. After several weeks of coding this way, the program suddenly stops working, and after hours of trying to fix it, he still doesn't know why. Fred may well spend a significant amount of time chasing this piece of code around without ever being able to fix it. No matter what he does, it just doesn't ever seem to work right.

Fred doesn't know why the code is failing because he didn't know why it worked in the first place. It seemed to work, given the limited "testing" that Fred did, but that was just a coincidence. Buoyed by false confidence, Fred charged ahead into oblivion. Now, most intelligent people may know

someone like Fred, but we know better. We don't rely on coincidences—do we?

Sometimes we might. Sometimes it can be pretty easy to confuse a happy coincidence with a purposeful plan. Let's look at a few examples.

Accidents of Implementation

Accidents of implementation are things that happen simply because that's the way the code is currently written. You end up relying on undocumented error or boundary conditions.

Suppose you call a routine with bad data. The routine responds in a particular way, and you code based on that response. But the author didn't intend for the routine to work that way—it was never even considered. When the routine gets "fixed," your code may break. In the most extreme case, the routine you called may not even be designed to do what you want, but it seems to work okay. Calling things in the wrong order, or in the wrong context, is a related problem.

Here it looks like Fred is desperately trying to get something out on the screen using some particular GUI rendering framework:

```
paint();
invalidate();
validate();
revalidate();
repaint();
paintImmediately();
```

But these routines were never designed to be called this way; although they seem to work, that's really just a coincidence.

To add insult to injury, when the scene finally does get drawn, Fred won't try to go back and take out the spurious calls. "It works now, better leave well enough alone…."

It's easy to be fooled by this line of thought. Why should you take the risk of messing with something that's working? Well, we can think of several reasons:

- It may not really be working—it might just look like it is.
- The boundary condition you rely on may be just an accident. In different circumstances (a different screen resolution, more CPU cores), it might behave differently.
- Undocumented behavior may change with the next release of the library.
- Additional and unnecessary calls make your code slower.

- Additional calls increase the risk of introducing new bugs of their own.

For code you write that others will call, the basic principles of good modularization and of hiding implementation behind small, well-documented interfaces can all help. A well-specified contract (see Topic 23, Design by Contract) can help eliminate misunderstandings.

For routines you call, rely only on documented behavior. If you can't, for whatever reason, then document your assumption well.

Close Enough Isn't

We once worked on a large project that reported on data fed from a very large number of hardware data collection units out in the field. These units spanned states and time zones, and for various logistical and historical reasons, each unit was set to local time. As a result of conflicting time zone interpretations and inconsistencies in Daylight Savings Time policies, results were almost always wrong, but only off by one. The developers on the project had gotten into the habit of just adding one or subtracting one to get the correct answer, reasoning that it was only off by one in this one situation. And then the next function would see the value as off by the one the other way, and change it back.

But the fact that it was "only" off by one some of the time was a coincidence, masking a deeper and more fundamental flaw. Without a proper model of time handling, the entire large code base had devolved over time to an untenable mass of +1 and -1 statements. Ultimately, none of it was correct and the project was scrapped.

Phantom Patterns

Human beings are designed to see patterns and causes, even when it's just a coincidence. For example, Russian leaders always alternate between being bald and hairy: a bald (or obviously balding) state leader of Russia has succeeded a non-bald ("hairy") one, and vice versa, for nearly 200 years.

But while you wouldn't write code that depended on the next Russian leader being bald or hairy, in some domains we think that way all the time. Gamblers imagine patterns in lottery numbers, dice games, or roulette, when in fact these are statistically independent events. In finance, stock and bond trading are similarly rife with coincidence instead of actual, discernible patterns.

A log file that shows an intermittent error every 1,000 requests may be a difficult-to-diagnose race condition, or may be a plain old bug. Tests

that seem to pass on your machine but not on the server might indicate a difference between the two environments, or maybe it's just a coincidence.

Don't assume it, prove it.

Accidents of Context

You can have "accidents of context" as well. Suppose you are writing a utility module. Just because you are currently coding for a GUI environment, does the module have to rely on a GUI being present? Are you relying on English-speaking users? Literate users? What else are you relying on that isn't guaranteed?

Are you relying on the current directory being writable? On certain environment variables or configuration files being present? On the time on the server being accurate—within what tolerance? Are you relying on network availability and speed?

When you copied code from the first answer you found on the net, are you sure your context is the same? Or are you building "cargo cult" code, merely imitating form without content?

Finding an answer that happens to fit is not the same as the right answer.

Tip 62: Don't Program by Coincidence

Implicit Assumptions

Coincidences can mislead at all levels—from generating requirements through to testing. Testing is particularly fraught with false causalities and coincidental outcomes. It's easy to assume that X causes Y, but as we said in Topic 20, Debugging: don't assume it, prove it.

At all levels, people operate with many assumptions in mind—but these assumptions are rarely documented and are often in conflict between different developers. Assumptions that aren't based on well-established facts are the bane of all projects.

How to Program Deliberately

We want to spend less time churning out code, catch and fix errors as early in the development cycle as possible, and create fewer errors to begin with. It helps if we can program deliberately:

- Always be aware of what you are doing. Fred let things get slowly out of hand, until he ended up boiled, like the frog here.
- Can you explain the code, in detail, to a more junior programmer? If not, perhaps you are relying on coincidences.

- Don't code in the dark. Build an application you don't fully grasp, or use a technology you don't understand, and you'll likely be bitten by coincidences. If you're not sure why it works, you won't know why it fails.
- Proceed from a plan, whether that plan is in your head, on the back of a cocktail napkin, or on a whiteboard.
- Rely only on reliable things. Don't depend on assumptions. If you can't tell if something is reliable, assume the worst.
- Document your assumptions. Topic 23, Design by Contract, can help clarify your assumptions in your own mind, as well as help communicate them to others.
- Don't just test your code, but test your assumptions as well. Don't guess; actually try it. Write an assertion to test your assumptions (see Topic 25, Assertive Programming). If your assertion is right, you have improved the documentation in your code. If you discover your assumption is wrong, then count yourself lucky.
- Prioritize your effort. Spend time on the important aspects; more than likely, these are the hard parts. If you don't have fundamentals or infrastructure correct, brilliant bells and whistles will be irrelevant.
- Don't be a slave to history. Don't let existing code dictate future code. All code can be replaced if it is no longer appropriate. Even within one program, don't let what you've already done constrain what you do next—be ready to refactor (see Topic 40, Refactoring). This decision may impact the project schedule. The assumption is that the impact will be less than the cost of not making the change.[53]

So next time something seems to work, but you don't know why, make sure it isn't just a coincidence.