Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Job Security

Diomidis Spinellis

My colleague, who works for a major equipment vendor, was discussing how his employer was planning to lay off hundreds of developers over the coming months. “But I’m safe,” he said, “as I’m one of the two people in our group who really understand the code.” It seems that writing code that nobody else can comprehend can be a significant job security booster. Here’s some advice.

Unreadable Code

Start by focusing on your code’s low-level details. Nothing puts off maintainers trying to take over your job than code that brings tears to their eyes. Be inconsistent in all aspects of your code: naming, spacing, indenting, commenting, style. Every time there are multiple ways to implement something, throw dice and choose at random. Avoid writing similar code in comparable situations. Spend time coming up with coding tricks that nobody has ever used. Why write `a = 0` when you can write `a ^= a`? Apply this advice liberally in the way you format expressions and statements. There’s only one generally accepted way to space between operators and operands; avoid it. Control flow statements are more fun because there are two schools on where to put braces; randomly switch between them to throw people off.

Unfortunately, these tricks won’t get you far, because beautifiers can readily bring your code up to scratch. However, when naming your variables, methods, fields, and classes, your choices can persist for decades; think of the Unix `creat` (sic) system call. Some languages, such as Java, have well-established naming conventions regarding capitalization and the joining of words. View them as an opportunity; these rules were designed to be broken. In other languages, such as C++, naming conventions are already severely broken or nonexistent. In this case, you can make your mark by using new names for existing concepts. For instance, name the methods for an iterator’s range `start` and `finish`, rather than `begin` and `end`. Further innovate by making those ranges symmetric rather than following the customary asymmetric style.

You might think that simply avoiding comments is the way to go, but you can do a lot better than that. If you change already-commented code, leave the existing comments in place without updating them. This is a sure way to send your code’s hapless readers on a wild-goose chase. Surprisingly, IDEs can also help you here. Many IDEs insert boilerplate comments at the beginning of each method and class. Keeping them there unfilled occupies valuable screen real estate, making your code harder to follow. Even better, this orderly boilerplate gives the initial impression that the code is well commented, thus increasing the unavoidable subsequent disappointment.

Painful Changes

Regrettably, many of the tricks I’ve discussed so far can be overcome by the unfortunate practice of refactoring. This allows a determined killjoy to slowly but surely improve your code’s quality. Guard against such accidents, while making the code even more unmaintainable, by ensuring that code changes really hurt. Modern languages have brought with them the disturbing habit of declaring a specialized type for each different entity you want to model in the code. Worryingly for you, this can make changes particularly easy, because after a change the compiler will automatically

Post your comments online by visiting the column’s blog: **www.spinellis.gr/tools** detect any nonmatching types. You can code around this problem by representing most data as plain numbers and strings. When you need to group more entities together, just separate them in the string with a delimiter or twiddle an integer's bits using the language's binary operators. Also, give a special meaning to negative numbers and, of course, zero.

Some programmers have developed the nasty habit of adding assertions in their code. Avoid these constructs like the plague. They tend to provide an early warning when an algorithm or a class's state has gone south, depriving everybody of the opportunity for countless interesting hours of debugging. The same goes for providing unit-testing support.

Puzzling Interactions

An additional way to ensure your code is and remains unmaintainable is to booby-trap it, so that nobody (but, hopefully, you) can foresee a change's effects. The keyword here is *coupling*—the more, the better. Some types of coupling between modules are truly devious. For instance, you can have two classes implicitly share knowledge of data formats and protocols, or have one class modify another's internal workings. For this to work, it helps if you declare all variables, fields, and methods with the widest possible visibility. Use globally visible static fields to good effect, communicating through them as if they were global variables, and make your code change its behavior depending on their value. For added points, have a class's methods behave differently depending on the order in which they're called, and pass data around in large chunks, even if a method requires only a tiny bit of it. This will prompt your code's readers to come to you to find out which part of the data is really needed.

Byzantine Design

There's also ample scope to tie your job security right into the code's design. Deep and wide inheritance trees; useless abstraction layers (if anyone dares to ask, claim they might be needed in the future); and incestuous, seemingly random, interactions between classes are your tools of the trade. Also, lumping together many responsibilities in each class will make it easier for you to create surreptitious links between seemingly unrelated elements. For added effect, make dependencies between packages follow the path of unstable dependencies: everybody should depend on packages whose interfaces change at the slightest provocation. Earn bonus points by introducing some cyclical dependencies so that changes cascade in a loop.

Icing the Cake

You can ensure that nobody will want to take your job, even without messing with your code. If the project takes hours to build (doesn't support incremental builds), other developers will stay away. Similarly, lack of a test infrastructure will make even the most trivial changes a risky proposition. Nevertheless, if some brave souls manage to build and (manually) test your code, ensure that they won't be able to release it. Make the project's release a manual, lengthy, undocumented, and highly complex procedure that only you can pull off.

By now, you surely realize that providing any external documentation, especially those types that are automatically kept up to date, is a big no-no. People should come to you for help. Using a version control system is also problematic because this can leak valuable information regarding the code's evolution.

Team Effort

Creating bad code often requires team effort. By hiring people who write awfully, you can increase the magnitude of the code base on which your job security depends a lot more effectively than you could on your own. Even better, these developers typically also have difficulty understanding well-written code, further strengthening the case for retaining you. If you're lucky, they'll bring their similarly mediocre friends to the team, institutionalizing the practice of writing code that nobody wants to touch. If you're not responsible for hiring decisions, make sure your manager understands that hiring many low-paid, dreadful developers is vastly preferable to hiring a few good ones.

Don't waste any time or resources training, tutoring, or mentoring young recruits. Instead, intimidate them by dumping a mountain of undocumented code on them and let them grapple with it. When they turn to you for help or, even better, leave for another job, you'll appear even more indispensable.

Of course, the effort toward job security can go too far. A company facing unmaintainable code can simply abandon the specific product (firing its obviously appalling development team) or, failing to keep up with the competition, even go under. The pity is that nobody will shed a tear for the horrible code that will be left behind.

Creating bad code often requires team effort.

Diomidis Spinellis is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business. He's also a four-time winner of the International Obfuscated C Code Contest and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.