



First, Do No Harm

Diomidis Spinellis

LET'S FACE IT: not all software developers are superstar programmers (and, trust me, not all luminary developers program in a sane way). This means that when we maintain existing code, we must be very careful to avoid breaking or degrading the system we work on. Why? Because a failure of a running system

offer the required functionality. We developers often get carried away implementing complex functionality when the customer actually requires (or can agree to) a much simpler implementation. The reason behind this could be technological machismo, the desire to learn new stuff, the not-invented-here syndrome, or a simple

documented, simply derive the conventions by examining similar code. And when you see code with style flaws, coordinate with your colleagues to fix them (in a commit separate from your other changes, please).

Program defensively. When you add new code, you'll often find yourself wondering how exactly the existing system behaves. Counteract this uncertainty by verifying the assumptions you make through runtime assertions and rigorous error checking.

Maintain backward compatibility. Your interfaces to the external world are a contract you're obliged to keep: it's often difficult to coordinate with everyone who depends on those interfaces. This means you can't change APIs, file formats, schemas, and service endpoint URLs arbitrarily. Ensure that you make such changes in a way that's transparent to existing clients. (In some cases, you might even need to remain bug-compatible with the system you replace.) Make the new functionality a superset of the old one, perhaps through an adaptation layer. Versioning of interfaces and the gradual deprecation of older ones through warnings can help you manage backward compatibility over longer time scales.

Preserve architectural properties. When you stay within the realm of a single module and use code constructs similar to those that are already there, you're probably treading on safe ground. Otherwise, you must

Inconsistently styled code is distracting to read and a pain to edit.

can affect operations, people, profits, property, and sometimes even lives. Here are the rules.

Development

Have the design and change scope reviewed. Discuss with both your manager and your users what you intend to implement, and how you plan to

misunderstanding. Through a design review, you can minimize the chance that you'll waste effort and burden the software with needless complexity. For example, your reviewer might point out a simpler implementation that will work well enough or an existing library you can reuse to save days of work.

Follow code style rules. Inconsistently styled code is distracting to read and a pain to edit. Worse, the careless attitude it exudes can give rise to ghastly design and implementation sins. When developers see code that's all over the place, they get the message that any programming sin is fair game. Therefore, find the style rules adopted in your setting and follow them religiously. If these aren't

Post your comments online
by visiting the column's blog:

www.spinellis.gr/tools

verify that your code doesn't breach any existing architectural properties. For instance, your code might be breaking encapsulation by exposing functionality that was private to a module or violating layering through calls to higher-level layers.

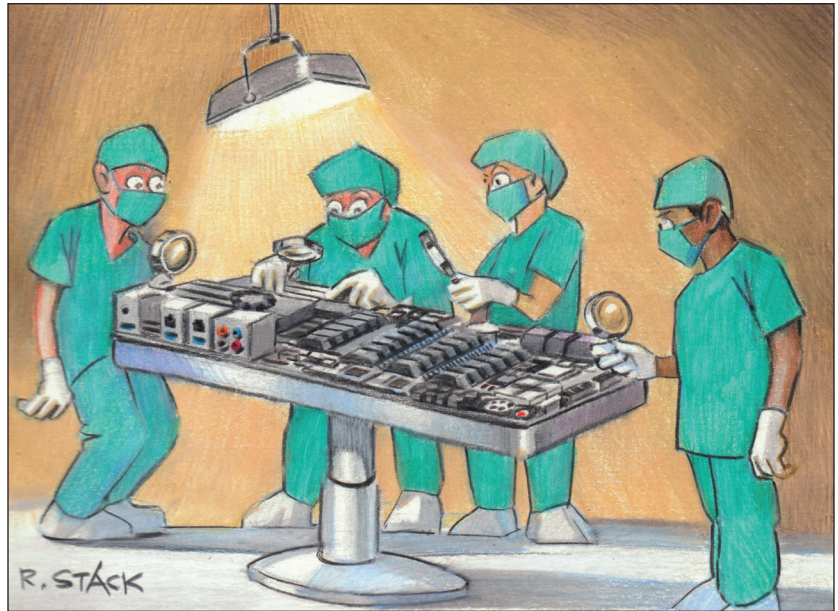
Testing

Have your code reviewed. A second pair of eyes will look at it from a different perspective, question any incorrect implicit assumptions you might have made, notice errors you may have overlooked, pinpoint style and architectural infractions you might have brushed aside, and keep you honest about the amount of testing that's required. Don't skimp on it; record how you close each raised issue, and repeat the reviews until you and your reviewer agree that you've resolved all the spotted issues.

Pass existing tests. If the system you modify features a testing infrastructure, verify it works correctly before you apply your changes, fix it if it doesn't, and then use it to verify the system's functionality after you make your changes.

Create unit tests for the implementation of the new functionality. A legacy system could well lack a comprehensive unit test suite, and creating one from scratch is often impractical. However, by adding unit tests for the new code you write, you can increase your confidence that it works as intended and will play well with the rest of the system. For extra points, try adding unit tests for any existing routines that you modify.

Beware of changed components. When you upgrade the libraries, the compiler, the framework, the middleware, the storage engine, the operating system, or the hardware on which your software runs, bad things can happen, especially if their developers



haven't followed this column's advice. Don't assume your software will run just fine under the new configuration—test it thoroughly.

Check for performance regressions. Evaluate the system's performance in terms of time, storage, and (perhaps) power and bandwidth before and after your changes. Keep an eye out for performance regressions that are outside acceptable limits. Also, make sure that you haven't accidentally degraded an important operation's time or space complexity (for example, changing the time's complexity class from linear time to quadratic).

Operations

Perform a phased rollout. First deploy on a test system to avoid breaking production systems with any serious errors that might have crept through. Once you're satisfied, deploy so-called canaries to a small percentage of the production systems, and carefully monitor how your system performs in that envi-

ronment. Only when you're completely satisfied, roll out to the entire user base.

Have a back-off plan. Even a well-tested systems and careful deployment can go catastrophically wrong. Develop a plan to quickly pull out the new system from production when problems arise. Test the plan, and don't hesitate to abort deployment at the first sign of trouble. We developers are often optimistic, thinking that our code is always correct, and that the problems showing up in deployment are just minor teething pain. With objective criteria for when to back off, you prevent (bad) judgment decisions under stress.

"Test as you fly, fly as you test." It's often tempting to push a tiny change to production without going through the arduous and time-consuming process of testing and phased rollout. It's also very risky, and this is why the aerospace community operates by a simple rule: all systems that fly should be tested, and

only the tested systems should fly. Repeating this motto in my head every time I had the urge to take a testing shortcut has saved me on countless occasions. I suggest you do the same.


Deploy in off-peak periods. Colleagues who work at telecoms tell me that they plan disruptive upgrades for 4 am, yet they still get calls from irate customers who complain about outages. Imagine what would have happened at a peak time! Improper timing of a release can overwhelm your support personnel, alienate customers, and cloud your decision-making ability. To avoid the temptation and external pressure to deploy during a peak period, consider setting a policy on the hours or times of the year when new deployments can (or can't) be performed.

Coordinate with support teams. Let everybody who's supporting your software, from system engineers to helpdesk staff, know your deployment and back-off plan, your changes, and who to contact when things go wrong. Present your plan in a meeting to make certain you're all on the same page and that you've addressed any concerns support teams may have.

Have key people on standby. The main developer of a popular open source system used to issue new releases on the evening before he left for vacation. Today, this practice just doesn't cut it. Coordinate with key developers and other engineers so that they'll be available during deployment to advise the team on how to handle any snafus.

As you can see, “doing no harm” can be a tall order. Yet, following this guideline is a sign of professionalism and respect for your colleagues and customers. 📺

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of the books *Code Reading* and *Code Quality: The Open Source Perspective* (Addison-Wesley, 2003, 2006). Contact him at dds@aueb.gr.



See www.computer.org/software-multimedia for multimedia content related to this article.



IEEE Software offers pioneering ideas, expert analyses, and thoughtful insights for software professionals who need to keep up with rapid technology change. It's the authority on translating software theory into practice.

www.computer.org/software/subscribe

SUBSCRIBE TODAY