Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

Zero-Tolerance Construction

Andy Hunt and Dave Thomas

In *IEEE Software*'s May issue, Steve McConnell asked, "Why doesn't anyone want to talk about software construction?" Well, we want to, and we do!

Why? Because for us, construction is the ultimate embodiment of the software development process. After all, our best architectural ideas, elegant designs, or insightful requirements analyses don't count for much until they are built. How then, could you consider the topic of actually building software in any way less interesting than topics of design, architecture, process improvement, and so on?

Of course, coding is not a neat and tidy subject (which might explain why many avoid writing about it). With our architect hats on, for instance, we can settle into a comfortably pristine universe of coolly elegant designs, where the world of our making will solve the problems we face. And until code exists to implement our architecture, we can live quite nicely in this unvalidated fantasy world.

But as implementation gets underway, life becomes much more challenging. Coding is messy. It's subject to all manner of ugly realities, from defects in compilers and databases to errors in our instructions to the computer. Engineering issues enter the picture—performance, memory usage, memory leaks or garbage-collection problems, synchronization and thread-safety concerns, and so on. The art of programming lies in that nether region between the hopeful wishes of an elegant architecture and the hard grindstone of technical details. What we need to talk about isn't how to solve the low-level engineering problems (there's plenty of Web space and traditional ink devoted to those issues), but instead how to write a program: how to reconcile our mental view of the world with the real world as their interaction unfolds before us.

So that's why we enjoy writing the Software Construction column: it helps us think about techniques and tools that let us create real software in the real world. The last two columns addressed ways of deferring some real-world problems: "Mock Objects" (May/June), which let us unit test components with improved isolation from other components, and "Naked Objects" (July/August), a technique that can help us focus on objects' behavioral completeness—the real functionality of a system—without being distracted by workflow or user interface issues. But there are many distractions from the real world that conspire to undermine our efforts, and we need a general-purpose, effective method to keep encroaching problems at bay.

0740-7459/02/$17.00 © 2002 IEEE

One more wafer-thin fix

In Monty Python's *Meaning Of Life*, the remarkably large Mr. Creosote devours a remarkably large meal. He is offered one final mint to finish off the dinner—but not to worry; it is, after all, only a wafer-thin mint. However, it puts Mr. Creosote over the top, and in fine Monty Python style, he explodes. Apparently you can't judge a wafer by its thickness.

While we can easily be fooled into thinking that an architecture or a design is complete and in good working order, actual code is less forgiving. We can't fool ourselves: the compiler is validating our instructions, and our unit tests and customers are validating the functionality. Lo and behold, we discover that our neat design doesn't work as we intended. "But it's so close—just this one little kludge will fix it. No one will even notice. It's wrong, but I'll add it anyway, and then it will work. Then I can go back and fix the whole thing correctly." So close. But oh, bad luck. One more little fix is required. But this is the last bug, surely. And on it goes.

Unfortunately, you can't judge the effects of hack by its size. One piece of bad code will have a certain negative effect on the system, but two pieces of bad code—however small—will have a negative effect that's more than double. No hack is an island; code's interrelated nature creates a complex system such that any hack that isn't quite right can cause *collateral damage*; that is, damage to seemingly unrelated parts of the program. This damage must then be addressed (usually in the same code-and-fix, hasty manner) and the death spiral begins. Add some time pressure from a demanding client, fast-moving marketplace, or impatient venture capitalist, and you're almost guaranteed to create more bad code than good, quickly dooming the project.

How can we stop this descent into failure?

No Broken Windows

Researchers in the field of crime and urban decay study similar problems in the inner cities. In a well-known study,[1] researchers wanted to understand why some buildings in poor neighborhoods fell into dilapidated, crime-infested ruin, while others—in equally challenged areas—survived. Apparently, all it takes to destroy a building is a single broken window.

The effect works like this: a window is broken, probably by accident, but it is left unrepaired. Then perhaps another window is broken. Maybe it's not an accident this time. Next, graffiti appears, followed by litter. Minor structural damage begins, and tenants start to flee. Major structural damage occurs, and the building's owners abandon it as the cost to repair the building exceeds their investment. Now the criminals move in, and all is lost. Once this escalation begins, its progress is rapid and difficult to stop. However, if you can catch it early, you can save the neighborhood.

So, to prevent major, catastrophic loss, we must focus on preventing the triggering mechanism from occurring. If we can fix the little problems as they occur, then we'll have fewer large problems with which to contend. New York City's police department adopted the Broken Windows theory as part of former New York City mayor Rudy Giuliani's crackdown on crime. This approach of attacking the small problems—graffiti, littering, and pan-handling—led to a dramatic decrease in major crimes. From 1993 to 1997, felonies in New York City dropped 44 percent, murders 60 percent, and robberies and burglaries decreased by nearly 50 percent.[2] Minor crime provides a fertile breeding ground for major crime; eliminate the minor crime and you cut down the major crime.

To see similar benefits on projects, we adhere to our pragmatic practice of No Broken Windows. We must support and encourage the notion that there is no "later." However small and trivial a bug or design defect might seem, it has a negative effect on developers that will only grow larger over time and will begin to affect other areas of the project as well. The time to fix it—and fix it properly—is right now. If this requires some modification to the schedule, then so be it. The alternative is a slow, spreading code rot and attendant developer malaise that can quickly exceed our capacity to repair. There are some fixes, however, that can't be addressed in a timely manner. In that case, we suggest you "board it up." Make it clear to everyone involved that this feature is known to be broken, and put up enough "plywood and police tape" so that no one trips over the broken item or is in any way misled into thinking that it might work. It's as important to show the team—and yourself—that you're on top of the situation as it is to ensure no one else gets hurt. Just as with the apartment building, it doesn't take much for a few broken pieces of the project to get out of control, rendering the situation unrecoverable.

Many of the agile methodologies support this notion of minimizing technical debt and of maintaining a zero-tolerance policy toward ugly hacks. In *The Pragmatic Programmer*,[3] we describe this principle using the Broken Windows metaphor. Extreme Programming (XP)[4] relies on constant refactoring to maintain the quality of the code base at consistently high levels. With Scrum[5] you know that the software on which you are working will be released at the

"But it's so close—just this one little kludge will fix it. No one will even notice."

end of the 30-day iteration. There is no "later" for the feature you are working on. At the end of the iteration, the features that are shipped are solid, possibly at the expense of others that will be deferred to the next iteration.

XP recommends that you never let the sun set on bad code. That is, if at the end of the day the piece that you are working on (and any other hacks you had to add) doesn't measure up to the system's quality expectations, you should abandon it. Throw away the code and try again tomorrow. While this approach might sound extreme, it does help to keep ill-conceived, broken code out of the system in the first place. Fred Brooks told us long ago to plan on throwing out our first version of software, and the industry continues to ignore that advice at our collective peril.

Not just construction

Of course, the No Broken Windows principle applies to more than just code. Anything that is perceived as broken—whether it is a bug in a user-visible feature, part of the development infrastructure (including automatically building and testing the project), part of requirements or documentation, a bad design decision, or a poor technological choice—must be fixed before it can cause more damage and grow even larger. In fact, problems with process and teams can escalate even faster than problems in code: software doesn't have morale to damage, but teams do. Once damaged, morale is considerably harder to fix than process or code.

The wider applicability of No Broken Windows demonstrates another reason why it's difficult to talk about construction alone: construction never happens by itself. Architecture, design, code, and requirements are different facets of the same activity. As an industry, we cannot assume that we've fixed all the coding problems and move on to the more interesting problems. It's all the same problem, and we need to address all its facets to be successful. The best way to stay on top of the project's facets is to not let problems get out of hand. Fix broken windows as soon as they occur, and your project will thrive.

References

1. J.Q. Wilson and G. Kelling, "The Police and Neighborhood Safety," *The Atlantic Monthly*, vol. 249, no. 3, Mar. 1982, pp. 29–38.
2. J.A. Greene, "Zero Tolerance: A Case Study of Police Policies and Practices in New York City," *Crime & Delinquency*, vol. 45, no. 2, Apr. 1999, p. 171–188.
3. A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, Boston, 2000.
4. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, 1999.
3. K. Schwaber and M. Beedle, Agile Software Development with Scrum, Prentice-Hall, Englewood Cliffs, NJ, 2001.

Andy Hunt and **Dave Thomas** are partners in The Pragmatic Programmers, LLC. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. They also offer training in modern development techniques to programmers and their management. Contact them via www.pragmaticprogrammer.com.