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

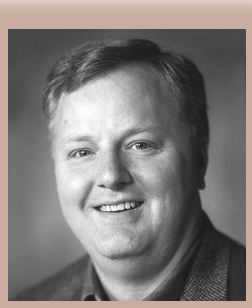
Three Legs, No Wobble

Andy Hunt and Dave Thomas

We begin this month with a look at how to ruin a software project in just three easy steps (there are many more, of course, but we'll start with these):

1. Never look back.

Once you've finished writing a piece of code, you're done. You're an excellent coder—you don't need any feedback from the code itself. Just throw the code over your shoulder for some other poor slob to test. Don't bother to check if the code is doing what you think it's doing, except maybe for one pass right "down the middle" with normal, expected values. Print the results out and if they look okay, you're done. Shovel out the code, adding it to the nice compost pile of rising technical debt for the whole team.



2. Never let go.

Never let go of code. Once written, stick with it—in its current form—forever. It's okay to keep the worst code forever, after all, because the replacement might turn out even worse—or take even longer to complete. The code was hard enough to write the first time, so hang on to it and don't try to change it no matter what happens. Take a lesson from the fellow in the news who dropped his cell phone down the public toilet on the Metro-North commuter railroad. He decided on a bold course of action and tried to rescue the lost phone. Granted, he got his arm stuck in the u-bend of the toilet and had to be rescued with the Jaws of Life, disrupting rush hour trains for millions. Now, the transit authority is seeking damages against him as well, but he had the right idea. Despite the costs, never let go.

3. Do it differently.

Whether you are compiling, testing, creating a release, or doing end-user product installation, do it differently every time. The easiest way to do this is to create a sheet of cryptic instructions for everyone to follow. People aren't as consistent as computers and so will tend to do it just a little bit differently (or forget a step) every time. While it's true that you'll see some weird failures on some machines now and then, the good news is that you'll see your project actually work on some machines some of the time, too.

Three legs of support

These approaches will kill a project every time. Fortunately, three simple practices can

 0740-7459/04/$20.00 © 2004 IEEE

save a project from these and other common mishaps: version control, unit testing, and automation.

Despite the obviousness of these practices, not everyone does them and even fewer teams do them well. According to one informal survey by *SD Times*, close to 40 percent of project teams in the US don't use any form of version control at all. Their policy is "last one in, wins." Many other teams invest in expensive version control systems that they don't use fully because of steep learning curves or intensive administration requirements. These end up as useless "shelfware." Similarly, unit testing is often given mere lip service (if even that). Automation generally stops with integrated development or build environments.

We feel that these practices are so important that we've written a new series of books to cover them, with one book devoted to each practice. In this column, we'll look at what's important about each of these areas and list commonly used, freely available open source tools that can help.

Version control

The world's fanciest word processor would be almost useless if it didn't have a Backspace key. At the end of a lengthy document, you'd start typing v-e-r-y s-l-o-w-l-y to avoid mistakes. A version control system is a project's Undo button. With proper version control in place, developers have the freedom to experiment, knowing they can always go back to any previous version of code if necessary.

Developers need to save their code into the version control repository and get fresh copies of other team member's changes frequently. When working on code (or even books or columns like this one), we tend to synchronize our code with the repository several times an hour, all day long. This helps prevent integration nightmares and keeps everyone on the team on the same page.

When looking for a version control system, consider the following features: it should be able to operate across distributed clients, so that developers can work from home during a blizzard or team members in separate locations can use it; it should work across all the operating systems and platforms on which you develop. We recommend systems that feature optimistic locking, not strict locking; strict locking lets only one person edit a file at a time, and in practice this seriously limits the team's throughput without any real benefit.

Finally, consider how the version control system will work and play with your chosen methodology. It might provide hooks or tie-ins to notify people when code is ready for review or testing, and so on.

The most popular open source version control tools include the Concurrent Versions System (www.cvshome.org) and its up-and-coming successor Subversion (http://subversion.tigris.org).

Unit testing

Unit testing is poorly named, as it has little to do with testing per se. We use unit testing as the primary source of developer feedback. It's really the same thing as the typical developer practice of using temporary print statements or checking values in a debugger, except that we arrange to have the computer check these values for us, and we save our work so that we can run this same test over and over again.

Close to 40 percent of project teams in the US don't use any form of version control at all. Their policy is "last one in, wins."

Unit testing allows for more aggressive code rewriting—or replacing—because you can determine easily if the new code is fulfilling the old code's obligations. Along with a version control system, this gives you great freedom to fix, correct, rewrite, and experiment.

Unit test code itself should have the following attributes:

- *Professional.* Test code should be well written, modular, well factored, and decoupled—as professional as the code that's shipped to users. You should have at least as much test code as production code.
- *Automatic.* Test code should check all results itself and not leave decisions open to human interpretation. It should be equally easy to run one test or all the tests in the system.
- *Independent.* You should be able to run any test at any time and in any order. Individual tests should test just one thing, without dragging in large portions of the rest of the system (that's more suited for functional testing, not unit tests). Mock objects (that is, simplistic "stand-ins" for real objects) can help here.
- *Repeatable.* Tests shouldn't depend on any external resources that you don't control directly. You must get the same results every time, time after time.

The xUnit frameworks are available for virtually every programming language and environment under the sun, including Java, C#, C++, Ruby, Perl, SQL, HTML, and so on (www.xprogramming.com/software.htm). Standard mock objects, mock object generators, and other related products are freely available as well (see http://mockobjects.com, http://easymock.org, http://junitdoclet.org, and http://cunitgen.org).

Automation

Any repetitive procedure must be automated. Typically, this includes activities such as building code, running unit tests, creating a release, and end-user

Continued on p. 22

common functionality. Combined with our inexperience with ASP.NET, this led to a lot of duplication in the presentation layer.

We only internationalized output, not input, but it still took one pair about a week and a half. The culprit was duplication. Rather than change one method that handled all input and output, we had to find all places that output was generated and change that. As with the security retrofit, this was tedious, but not particularly hard.

Why does it work?

On all these projects, the difficulty of making changes directly related to specific design qualities. The most obvious is duplication: when a change we wanted was localized in a single class, it was trivial. When we had to modify similar code over and over, the change was tedious and took a lot longer.

Other design qualities also affected our ability to make changes. Simplicity was important. With simpler designs, we were less likely to encounter existing code. When adding features, we were better off when there was no preexisting design to handle that feature. Adding code that doesn't exist is easy; fixing someone's preconceptions about a feature first is more costly. The sidebar lists a number of other design qualities that have made our projects easier to maintain and change.

Before you try

My experiments with continuous design have been very successful. I recommend that you try it on your projects. Before you begin, though, look at your current process. Software processes oriented around up-front design might not be friendly to continuous design. At a minimum, you'll need automated tests, a team-based approach to changes (such as collective code ownership), and commitment to continuously evaluating and improving your design in the face of schedule pressure.

You might wish to experiment with continuous design by mixing it with up-front design. If you do, be aware that continuous design requires specific design goals (see the sidebar). In particular, up-front designs often include "extensibility hooks" for future design changes. This approach makes continuous design harder and should be avoided.

On my projects, continuous design's focus on simplicity and continuous improvement has made the code better and more maintainable over time, rather than less. After experimenting with continuous design for so long, I'm convinced that it's harder to paint yourself into a corner than it is with up-front design. Try it for yourself, and let me know how it worked for you.

Jim Shore is the founder of Titanium I.T., a Portland, Oregon, consultancy specializing in Extreme Programming. He'd like to hear about your experiences with continuous design. Contact him at jshore@titanium-it.com.

Continued from p. 19

installation. But any task that a developer has to perform more than three times is a good candidate for automation.

Implementing automation can be as simple as writing a shell script or batch file, or a macro in your integrated development environment. Or, you might add additional rules or targets to an existing build script (such as Ant or Make would use). It might require an entire program itself, written in Ruby or Java. However it's implemented, make sure that the automation code is kept in version control and is advertised and available for the entire team's use.

Automation gives the team consistency, reliability, and repeatability across different developers and environments. New developers can get on board and be productive much faster if all they have to do is push a button or type a command, even if they're building on a different platform than they're accustomed to.

Products to coordinate compilation and building include old standbys such as Ant (ant.apache.org) or Make (www.gnu.org/software/make). Systems such as AntHill (www.cs.unibo.it/projects/anthill/index.html), Cruise Control (http://cruisecontrol.sourceforge.net), or Dartboard (http://public.kitware.com/Dart) perform continuous build and integration.

With these three legs in place, you'll have a firm base from which to build great code.

Andy Hunt and Dave Thomas are partners in The Pragmatic Programmers and authors of the new *The Pragmatic Starter Kit* book series. Contact them via www.PragmaticProgrammer.com.