

Work

by Jeffreys Copeland and Haemer



*test (n.) 1. Real users bashing on a prototype long enough to get thoroughly acquainted with it, with careful monitoring and followup of the results. 2. Some bored random user trying a couple of the simpler features with a developer looking over his or her shoulder, ready to pounce on mistakes. Judging by the quality of most software, the second definition is far more prevalent. See also **demo**. – The Jargon File, 4.2.0*

The Art of Software Testing

Zzzz... Huh? What? Oh, sorry. We thought someone said we were going to talk about testing and we dozed off for a minute.

We have tried to stay out of the hustle-and-bustle of high-tech high fashion, but we occasionally find that fashion will not stay out of us. Boring or eccentric areas we were working in that became fashionable before we could get out have included UNIX, PCs, portability, internationalization, POSIX and Linux. Most of our friends and relatives know that we have no taste, and that our early work in these areas just shows that sooner or later everything will become fashionable. *PC Week* is the fulfillment of Andy Warhol's prophecy: "In the future, everyone will be famous for 15 minutes." (Most, but not all. One COBOL programmer has been following us around ever since the last fringe thing we evangelized him with came into vogue: the Internet.)

Unfortunately, it's now happening again. For the past couple of months, our friends have been abuzz with talk

of an alternative to standard development methodologies called "eXtreme Programming" (XP), and last weekend Dave Taenzer gave us a copy of *extreme Programming explained*, by Kent Beck (published by Addison-Wesley Publishing Co., 1999, ISBN 201-61641-6). Since some object-oriented styles promote MixedCaseIdentifiers, and programming books now look like they're printed in German, we suppose one way to make book titles stand out is not to capitalize them.

We read it. For better or worse, it turns out that we already are eXtreme Programmers. We suspect this is because many core XP beliefs come down to one of our core methodologies: eXtreme Laziness (XL). A couple of examples:

- *Use the simplest solution to solve the problems you have right now.*

Don't over-engineer-in features that you think you might need someday. Chances are, you won't. Even if you do, putting them in when you know exactly what you need is still cheaper than

trying to guess now.

The contrasting, severe, standard dogma has long been: "Start your projects with a thorough, initial, analysis and design. This should include, in order, a requirements document, a functional specification, and overall-and-detailed-design phases. The earlier you catch mistakes in the software life cycle, the less expensive it is." Our response to this has always been a cheerful: "The sooner you pick stocks that will shoot up in value, the more money you'll make."

- *Try to get something to actually build. Now.*

This is more than "top-down rapid prototyping." XP extends the rule we first heard as "Don't ship it until it compiles" to "Don't ship it until it integrates."

Eschewing the traditional "Develop, develop, develop...develop. Phew. Integrate, integrate...integrate," XP admonishes us to "Develop, integrate, develop, integrate..." Integration should be so frequent that it's a normal part of every developer's daily work, not a sepa-

rate step, late in the project, after the developers hand over their code to the integrators. XP's catch-phrase for this, in case your management asks, is "continuous integration."

And yes, we integrate a lot. On one of our projects, we integrate three-quarters of a million lines of code nightly. Another project, only about a third the size, but in more active development, gets integrated many times per day by each of its several developers.

Why? We're lazy. When the difference between full system builds is small, finding and fixing build problems gets a lot easier. Anytime we've put off testing and integration to the end of a development cycle, we've been sorry.

In our experience, you can't do this without an easy-to-use, code-configuration management system, such as Concurrent Versions System, or CVS (see the CVS home page at <http://www.cvshome.org>, or our August/September 1997 columns, "Practical CVS, Parts 1 and 2," <http://sw.expert.com/C9/RS.C4.AUG.97.pdf> and <http://sw.expert.com/C9/RS.C4.SEP.97.pdf>, respectively).

You also need an incremental build tool, like *make* (1) (see Peter Collinson's June 1998 UNIX Basics column, "make: Mastermind of the Update," <http://sw.expert.com/C2/SE.C2.JUN.98.pdf> or Jim Fox's August 1998 Q&A column, "Unity Among Web Pages," <http://sw.expert.com/C6/SE.C6.AUG.98.pdf>).

Beck doesn't mention build tools or configuration management, but we're patient: their 15 minutes are coming. It does take several other stands guaranteed to make an academic software engineering professor's fur stand up. Implementation documentation? Useless. Code ownership? Bad idea. Software architects? Testers? Analysts? Don't want 'em. We won't discuss these because we're not writing a column about XP. Besides, the book is mercifully tiny (190 pages), and does a better job promoting its religion than we would. You can also find good information at <http://www.extremeprogramming.org>.

In this column, the XP fad we want to stand up and applaud is eXtreme Testing. This is nothing less than having developers write and run tests from the beginning (mind you, not just test plans—actual tests). Even before there's working code.

We Become Testy

In our experience, continuous integration mostly verifies that everyone's code builds together. This by itself is nothing to sneeze at, but also is a good steppingstone to higher goals. Once you can build frequently, you can test frequently. Once you can build at a moment's notice, you can also test at a moment's notice.

This isn't wishful thinking. Our *makefiles* have productions that update our code with latest versions from CVS, do an incremental build to bring our executables up-to-date and then run an automated test suite.

Once the gear-grinding failure noise subsides to a dull scraping sound, we use *cron* to run the tests nightly, and configure the tools to send us email about failures. When we're notified of failures, we use CVS to figure out who has changed the code since the last successful build, and forward the problems to the guilty parties, who can usually fix them immedi-

ately because the code is fresh in their minds.

Here again, having CVS (or something like it) makes a world of difference. Even when we get a new test that reveals a long-standing bug—and we do—if we have an old version that doesn't have the bug, we can use CVS to do a binary search for the check-in that created the problem. Building and testing a version from two months ago is this one-liner:

```
cvsv update -D'2 months ago'; make test
```

Developers actually love continuous testing, since it gives them a safety net. They can develop with relatively reckless abandon, secure in the knowledge that the test suite will tell them what they've broken—not in a week or a month, but right away. Finding two concurrent bugs is at least five times harder than finding a single bug.



Attempts to improve the quality of the product in one area can cause the quality to slip backwards in another. Continuous testing lets you find and fix these cases as soon as they happen: two steps forward, no steps back.

Dave Taenzer sums it up neatly: "I like to add my bugs one at a time."

Notice that we're not testers in this picture—just mail routers. All that's keeping us in the picture is that we haven't yet figured out how to get the test tools to look in CVS by themselves.

We're often ready to run relatively full suites as soon as we can first integrate, but the suites continue to grow steadily, with new tests added each time we want to add new features and each time we find new bugs.

Incremental integration and testing also subtly changes those conversations with your management that begin, "So you've finished writing the code; that means we can ship tomorrow, right?"

Your answers become, "Yes."

Regression Testing

Right now, we work on projects that run thousands of tests every night. A testing atmosphere like this produces a quality ratchet. We've all seen new code—even simple bug fixes—break old features. Frederick P. Brooks, in his still-wonderful, *The Mythical Man-Month* (published by Addison-Wesley, 1995, ISBN 0-201-83595-9), cites studies showing that a bug fix introduced into a large system has about a 50% chance of breaking something else that used to work: two steps forward and one step back. Attempts to improve the quality of the product in one area can cause the quality to slip backwards in another. Continuous testing lets you find and fix these cases as soon as they happen: two steps forward and no steps back.

Test suites that compare what software does today with what it did yesterday are called “regression tests.” It’s important (though sometimes confusing to managers) to emphasize that regression testing doesn’t test whether something works, only whether it has changed. Each time we fix a bug, we “break” our regression tests. Surprisingly often, a bug fix in one area fixes other tests that we hadn’t guessed it would affect. When this happens, we’ve moved the ratchet forward even more notches.

Regression tests aren’t the only kinds of tests we run. Let’s look at some other useful test types.

Conformance Tests

Conformance tests ask, “How close are we to where we’re trying to go?” (These are the tests that your manager probably confuses with the regression tests—that conformance tests often serve as the core of a good regression test suite doesn’t help clarify things.)

For example, if you want to ask whether your operating system is POSIX.1-conforming (or whether it’s not), you can run the POSIX Conformance Test Suite (PCTS), from the U.S. Department of Commerce’s National Institute of Standards and Technology (NIST). The PCTS tests a series of assertions about POSIX.1, listed in ANSI/IEEE 2003.1, and reports which assertions pass and which fail on your system. (Actually, the possible outcomes are PASS, FAIL, UNRESOLVED, UNTESTED and UNSUPPORTED, each of which the standard defines precisely.)

NIST also supplies conformance test suites for the following (see <http://www.itl.nist.gov/div897/ctg/software.htm>):

- XML technologies (XML, DOM)
- VRML
- X3D (VRML97 Profile)
- CGM
- COBOL85
- FORTRAN78
- PHIGS
- POSIX
- SQL

All these are “free,” which means that we all pay for the COBOL85 conformance test suite: Your tax dollars at work.

At the other end of the spectrum, capitalism brings us vendors that develop test suites for commercially important *de facto* and *de jure* standards. For example, QualityLogic Inc. (<http://www.qualitylogic.com>) makes a good living producing

useful and extensive conformance tests for printer languages, like PostScript and PCL. (For those of you involved with printers, QualityLogic is Genoa Technology’s new name.)

What if you can’t get third-party conformance tests? Sometimes you can generate them automatically. A personal example will illustrate this nicely.

Minolta-QMS recently added a Portable Document Format (PDF) interpreter to its printers. During this project, Haemer needed a PDF-conformance test suite, but none were commercially available. At first, he thought about generating an array of PDF files by hand, but he worried that one person working with one set of tools might not provide enough PDF variety. The solution? A 43-line shell script that used Perl and its `WWW::Search` and `LWP::Simple` modules to find and retrieve random PDF files from the Web.

But what if automatic generation won’t work in your case, and you still lack the tests you need?

You ask your fellow developers to create them. Wait. The *developers?*

Correct. For reasons we alluded to above, developers quickly get addicted to continuous testing. Once they’re hooked, we find it’s easy to get them to generate useful tests.

Performance Tests

We don’t know who first said, “First, make it work, then make it fast,” but we can now say it quickly.

Thinking carefully about this aphorism leads to an epiphany: Even if the latest changes haven’t broken anything, your product may not be working very fast anymore.

Performance tests ask, “Just how slow are we?” As soon as you are doing regular regression testing, you can time how long it takes to run each test.

You could use a stopwatch for performance testing, but this, too, can be automated. POSIX’s `times()` module lets you track the amount of user and system time that a command and its children take. These times are largely independent of the system load. You can often collect these times while you’re collecting the regression data.

Those of you working in Perl may also be able to make good use of the `Benchmark` module.

Progression Tests

We confess. We just made the term “progression tests” up. We wanted a term to describe a form of testing that we do and never see mentioned in the testing literature. Here’s the problem it solves:

Imagine you have a suite of conformance/performance tests that announces the presence of 1,000 failures, hence, bugs.

Your boss says, "Bugs? BUGS? Fix them!"

After a year's steady work, you drop the number of failures to 500; after a second, you drop it to 100; after a third, you drop it to zero. You also fix the failures in a handful of new regression tests that someone's added in along the way.

"At last!," your boss tells the CEO, "Our code is defect-free."

"At last!," marketing tells the PR firm, "Our code is defect-free."

"At last!," other programmers who read your company's ads say, "Proof that all those other companies are run by idiots, too."

Cutting the number of known failures from 1,000 to 500 suggests you've cut the number of bugs substantially. But once you've removed most test suite failures, the suite is only a regression test; fixes to the remaining bugs no longer measure much of anything. You need a measure for bugginess that isn't tainted by people just working to make the measurements look good.

We solve the problem with an old statistics trick: we sequester a random collection of test failures that we never fix. Well, at least, not on purpose. We use these progression tests to measure our progress over time.



The speed of disappearance of introduced errors can be used to estimate many things, including the efficiency of testers in finding bugs and the distribution and volume of naturally occurring errors.

Here's how it works: Suppose you take the 1,000-failure test suite and set aside half the tests, chosen at random, as progression tests. The rest are conformance tests. Two years later, you've fixed all 500 of the conformance tests bugs. And, without ever addressing them directly, two-thirds of the 500 former failures in the progression suite now pass, too.

"At last!," your boss tells the CEO, "Our code is defect-free."

"Nope," you interject, "but we have fixed all the problems that our conformance suite found. If you want a reasonable guess, our progression tests say we only have about one-third as many bugs in our code as we did two years ago. At this rate, I'd expect that to drop to one-ninth in another two years."

Students of testing will see a similarity to "error seeding," which starts by having a third party put random errors into the code before testing starts. The speed of disappearance of introduced errors can be used to estimate many things, including the efficiency of testers in finding bugs and the distribution and volume of naturally occurring errors.

Suppose we introduce 100 "random" bugs and, after a year, the testers find 10% of them. Suppose also, in that same period, they find 100 bugs we didn't know about. If the intro-

duced bugs are typical of all bugs in the code, then the other 100 bugs represent about 10% of all the bugs that we didn't know about, and there are roughly 900 others that we didn't put in left for us to find.

The trick with this is the phrase, "If the introduced bugs are typical of all bugs in the code." Error seeding requires a way to introduce representative "random errors." Progression testing sidesteps this by choosing preexisting natural errors.

Also, because the progression suite includes tests that initially pass, we can measure the frequency of newly introduced bugs. (Our regression tests are completely incapable of letting us measure this because we fix any newly introduced bugs in the regression tests immediately.)

In our experience, the hardest thing about maintaining a progression test suite is convincing fellow developers *not* to fix reproducible bugs for which you have perfectly good test cases.

There are many ways to label tests—application tests, unit tests, system tests, stress tests, acceptance tests and so on—but these are our four favorites: conformance tests, performance tests, regression tests and progression tests.

Let us put this vocabulary to good use and write some eXtremely Fashionable (XF) test code with Perl's `Test` and `Test::Harness` modules. But not this month.

We usually put lots of code in our columns. This time, just setting the stage brought us over our word limit.

We would be negligent, while talking about testing, not to mention Don Libes' `Expect`—a Tcl application that lets you write scripts to drive interactive programs. If you have an interactive program that you need to test, `Expect` is the tool of choice. Other languages now provide `Expect`-like extensions, but `Expect` takes best-of-show. (Though Kevin Cohen, at MapQuest Inc., tells us that Perl's `Expect.pm` is The Realization Of The Full Potential Of Perl.) Don, not coincidentally, works for NIST, where they do lots of testing. Like the COBOL85 conformance test suite, `Expect` is free. You can find more information at <http://expect.nist.gov>.

Finally, before we go, the winner of our June contest was Paul Livesey of Boa FP Systems, Ltd, in Berkshire, England. Paul was the first reader—despite trans-Atlantic postal latency—to tell us that the late Robert Coveyou, a mathematician at Oak Ridge National Laboratory and eight-time Tennessee state chess champion, was the source of the quote "The generation of random numbers is too important to be left to chance." Keep those cards and letters coming, folks.

Until next month, happy trails. ✍

Jeffrey Copeland (copeland@alumni.caltech.edu) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.

Jeffrey S. Haemer (jsh@usenix.org) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for Minolta-QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.

Note: The software from this and past *Work* columns is available at <http://alumni.caltech.edu/~copeland/work> or alternately at <ftp://ftp.cpg.com/pub/Work>.