Why Most Unit Testing is Waste

By James O Coplien

1.1 Into Modern Times

Unit testing was a staple of the FORTRAN days, when a function was a function and was sometimes worthy of functional testing. Computers computed, and functions and procedures represented units of computation. In those days the dominant design process composed complex external functionality from smaller chunks, which in turn orchestrated yet smaller chunks, and so on down to the level of well-understood primitives. Each layer supported the layers above it. You actually stood a good chance that you could trace the functionality of the things at the bottom, called *functions* and *procedures*, to the requirements that gave rise to them out at the human interface. There was hope that a good designer could understand a given function's business purpose. And it was possible, at least in well-structured code, to reason about the calling tree. You could mentally simulate code execution in a code review.

Object orientation slowly took the world by storm, and it turned the design world upside-down. First, the design units changed from things-that-computed to small heterogeneous composites called objects that combine several programming artefacts, including functions and data, together inside one wrapper. The object paradigm used *classes* to wrap several functions together with the specifications of the data global to those functions. The class became a cookie cutter from which *objects* were created at run time. In a given computing context, the exact function to be called is determined at run-time and cannot be deduced from the source code as it could in FORTRAN. That made it impossible to reason about run-time behaviour of code by inspection alone. You had to run the program to get the faintest idea of what was

going on.

So, testing became *in* again. And it was unit testing with a vengeance. The object community had discovered the value of early feedback, propelled by the increasing speed of machines and by the rise in the number of personal computers. Design became much more data-focused because objects were shaped more by their data structure than by any properties of their methods. The lack of any explicit calling structure made it difficult to place any single function execution in the context of its execution. What little chance there might have been to do so was taken away by polymorphism. So integration testing was out; unit testing was in. System testing was still somewhere there in the background but seemed either to become someone else's problem or, more dangerously, was run by the same people who wrote the code as kind of a grown-up version of unit testing.

Classes became the units of analysis and, to some degree, of design. CRC cards (popularly representing Classes, Responsibilities, and Collaborators) were a popular design technique where each class was represented by a person. Object orientation became synonymous with anthropomorphic design. Classes additionally became the units of administration, design focus and programming, and their anthropomorphic nature gave the master of each class a yearning to test it. And because few class methods came with the same contextualization that a FORTRAN function did, programmers had to provide context before exercising a method (remember that we don't test classes and we don't even test objects — the unit of functional test is a method). Unit tests provided the drivers to take methods through their paces. Mocks provided the context of the environmental state and of the other methods on which the method under test depending. And test environments came with facilities to poise each object in the right state in preparation for the test.

1.2 The Cure is Worse than the Disease

Unit testing is of course not just an issue in object-oriented programming, but the combination of object-orientation, agile software development, and a rise in tools and computing power has made it *de riguer*. As a consultant I often get questions about unit testing, including this real one from a recent client of mine, Richard Jacobs at Sogeti (Sogeti Nederland B.V.):

> My second question is about unit tests. If I remember correctly you said that unit tests are waste. First, I was surprised by that. Today however, my team told me the tests are more complex than the actual code. (This team is not the original team that wrote the code and unit tests. Therefore some unit tests take them by surprise. This current team is more senior and disciplined.) In my opinion, now that's waste... When I was programming on a daily basis, I did make code for testability purposes but I hardly wrote any unit tests. However I was renowned for my code quality and my nearly bug free software. I like to investigate WHY did this work for me?

You'll remember from your trade school education that you can model any program as a Turing tape, and what the program can do is somehow related to the number of bits on that tape at the start of execution. If you want to thoroughly test that program, you need a test with at least the same amount of information: i.e., another Turing tape of at least the same number of bits.

In real practice, the vagaries of programming language make it difficult to achieve this kind of compactness of expression in a test so to do complete testing, the number of lines of code in unit tests would have to be orders of magnitude larger than those in the unit under test. Few developers admit that they do only random or partial testing and many will tell you that they do *complete* testing for some assumed vision of *complete*. Such visions include notions such as: "Every line of code has been reached," which, from the perspective of theory of computation, is pure nonsense in terms of knowing whether the code does what it should. We'll discuss that problem in more detail below. But most programmers think of unit testing this way, which

means that it's doomed to fail from the start.

> ☞ *Be humble about what your unit tests can achieve, unless you have an extrinsic requirements oracle for the unit under test. Unit tests are unlikely to test more than one trillionth of the functionality of any given method in a reasonable testing cycle. Get over it.*

(*Trillion* is not used rhetorically here, but is based on the different possible states given that the average object size is four words, and the conservative estimate that you are using 16-bit words).

1.3 Tests for their Own Sake and Designed Tests

I had a client in northern Europe where the developers were required to have 40% code coverage for Level 1 Software Maturity, 60% for Level 2 and 80% for Level 3, while some where aspiring to 100% code coverage. No problem! You'd think that a reasonably complex procedure with branches and loops would have provided a challenge, but it's just a matter of *divide et impera*. Large functions for which 80% coverage was impossible were broken down into many small functions for which 80% coverage was trivial. This raised the overall corporate measure of maturity of its teams in one year, because you will certainly get what you reward. Of course, this also meant that functions no longer encapsulated algorithms. It was no longer possible to reason about the execution context of a line of code in terms of the lines that precede and follow it in execution, since those lines of code are no longer adjacent to the one you are concerned about. That sequence transition now took place across a polymorphic function call — a hyper-galactic GOTO. But if all you're concerned about is branch coverage, it doesn't matter.

> ☞ *If you find your testers splitting up functions to support the testing process, you're destroying your system*

architecture and code comprehension along with it. Test at a coarser level of granularity.

And that's just code mass. You can get the application code mass down, but that code contains loops that "cheat" information theory by wrapping up many lines of code in a small space. That means that tests have to be at least as *computationally complex* as code. You not only have many tests but very long-running tests. To test any reasonable combination of loop indices in a simple function can take centuries.

One brute-force attack on this problem is to run tests continuously. People confuse automated tests with unit tests: so much so that when I criticise unit testing, people rebuke me for critising automation.

☞ *If you write a test to cover as many possibilities as possible you can dedicate a rack of machines to running the tests 24 hours a day, 7 days a week, tracking the most recent check-in.*

Remember, though, that automated crap is still crap. And those of you who have a coroporate Lean program might note that the foundations of the Toyota Production System, which were the foundations of Scrum, were very much against the automation of intellectual tasks (http://www.computer.org/portal/web/buildyourcareer/Agile-Careers/-/blogs/autonomation). It's more powerful to keep the human being in the loop, as is more obvious in exploratory testing. If you're going to automate, automate something of value. And you should automate the mundane stuff. You'll probably get better return on your investment by automating integration tests, bug regression tests, and system tests than by automating unit tests.

A smarter approach would reduce the test code mass through formal test design: that is, to do formal boundary-condition

checking, more white-box testing, and so forth. That requires that the unit under test be *designed* for testability. This is how hardware engineers do it: designers provide "test points" that can read out values on a J-Tag pin of a chip, to access internal signal values of the chip — tantamount to accessing values between intermediate computations in a computational unit. I advocate doing this at the system level where the testing focus should lie; I have never seen anyone achieve this at the unit level. Without such hooks you are limited to black-box unit testing.

I might believe in formalized unit test design if the behavior can be formalized — that is, if there is some absolute, formal oracle of correctness from which the test can be derived. More on that below. Otherwise, it is just the programmer's guess.

> ☞ *Tests should be* designed *with great care. Business people, rather than programmers, should design most functional tests. Unit tests should be limited to those that can be held up against some "third-party" success criteria.*

1.4 The Belief that Tests are Smarter than Code Telegraphs Latent Fear or a Bad Process

Programmers have a tacit belief that they can think more clearly (or guess better) when writing tests when writing code, or that somehow there is more information in a test than in code. That is just formal nonsense. The psychological perspective is instructive here, and it's important because that — rather than any computational property — most drives developer behaviour.

If your coders have more lines of unit tests than of code, it probably means one of several things. They may be paranoid about correctness; paranoia drives out the clear thinking and innovation that bode for high quality. They may be lacking in analytical mental tools or in a discipline of thinking, and they want the machine to do their thinking for them. Machines are good at repeating mechanical tasks but test design still requires

careful thought. Or it may be that your process makes it impossible to integrate frequently, because of bad process design or bad tools. The programmers are doing their best to compensate by creating tests in an environment where they have some control over their own destiny.

> ☞ *If you have a large unit test mass, evaluate the feedback loops in your development process. Integrate code more frequently; reduce the build and integration times; cut the unit tests and go more for integration testing.*

Or the problem may be at the other end: developers don't have adequately refined design skills, or the process doesn't encourage architectural thinking and conscientious design. Maybe the requirements are so bad that developers wouldn't know what to test if they had to, so they make their best guess. Software engineering research has shown that the most cost-effective places to remove bugs are during the transition from analysis and design, in design itself, and in the disciplines of coding. *It's much easier to avoid putting bugs in than to take them out.*

> ☞ *If you have comprehensive unit tests but still have a high failure rate in system tests or low quality in the field, don't automatically blame the tests (either unit tests* or *system tests). Carefully investigate your requirements and design regimen and its tie to integration tests and system tests.*

But let's be clear that there will always be bugs. Testing will not go away.

1.5 Low-Risk Tests Have Low (even potentially negative) Payoff

I told my client that I could guess that many of their tests might be tautological. Maybe all a function does is sets X to 5, and I'll

bet there's a test of that function to see if the value of X is 5 after it runs. Good testing, again, is based on careful thought and on basic principles of risk management. Risk management is based on statistics and information theory; if the testers (or at least the test manager) don't have at least rudimentary skills in this area, then you are likely to do a lot of useless tests.

Let's dissect a trivial example. The purpose of testing is to create information about your program. (Testing does not increase quality; programming and design do. Testing just provides the insights that the team lacked to do a correct design and implementation.) Most programmers want to "hear" the "information" that their program component works. So when they wrote their first function for this project three years ago they wrote a unit test for it. The test has never failed. The question is: How much information is in that test? That is, if "1" is the passing of a test and "0" is the failing of a test, how much information is in this string of test results:

11111111111111111111111111111

There are several possible answers depending on which formalism you apply, but most of the answers are wrong. The naive answer is 32, but that is the bits of *data*, not of *information*. You could be an information theorist and say that the number of bits of information in a homogeneous binary string is the binary log of the length of the string, which in this case is 5. But that isn't what I want to know: in the end I want to know how much information I get from a single run of this test. Information is based on probability. If the probability of the test passing is 100%, then there is *no* information — by definition, from information theory. There is almost no information in any of the 1s in the above string. (If the string were infinitely long then there would be exactly zero bits of information in each test run.)

Now, how many bits of information in this string of test runs?

10110110001101011010001101011 01

The answer is... a lot more. Probably 32. That means that there's a lot more information in each test run. If we can't predict at the outset whether a test will pass or fail then each test run contains a full bit of information, and you can't get better than that. You see, developers love to keep around tests that pass because it's good for their ego and their comfort level. But the information comes from *failed* tests. (Of course, we can take the other extreme:

00000000000000000000000000000000

where there really is no information, either, at least about the process of quality improvement.)

> ☞ *If you want to reduce your test mass, the number one thing you should do is look at the tests that have never failed in a year and consider throwing them away. They are producing no information for you — or at least very little information. The value of the information they produce may not be worth the expense of maintaining and running the tests. This is the first set of tests to throw away — whether they are unit tests, integration tests, or system tests.*

Another client of mine also had too many unit tests. I pointed out to them that this would decrease their velocity, because every change to a function should require a coordinated change to the test. They informed me that they had written their tests in such a way that they didn't have to change the tests when the functionality changed. That of course means that the tests weren't testing the functionality, so whatever they were testing was of little value.

Don't underestimate the intelligence of your people, but don't underestimate the collective stupidity of many people working

together in a complex domain. You probably think you would never do what the team above did, but I am always finding more and more things like this that almost defy belief. It's likely that you have some of these skeletons in you closet. Hunt them out, have a good laugh at yourself, fix them, and move on.

> ☞ *If you have tests like this, that's the second set of tests to throw away.*

The third tests to throw away the tautological ones. I see more of these than you can imagine — particularly in shops following what they call test-driven development. (Testing for `this` being non-null on entry to a method is, by the way, not a tautological test — and can be very informative. However, *as with most unit tests*, it's better to make this an assertion than to pepper your test framework with such checks. More on that below.)

In most businesses, the only tests that have business value are those that are derived from business requirements. Most unit tests are derived from programmers' fantasies about how the function should work; that has no provable value. There were methodologies in the 1970s and 1980s based on traceability that tried to reduce system requirements all the way down to the unit level. In general, that's an NP-hard problem (unless you are doing pure procedural decomposition) so I'm very skeptical of anyone who says they can do that. So one question to ask about every test is: *If this test fails, what business requirement is compromised?* Most of the time, the answer is, "I don't know." If you don't *know* the value of the test, then the test theoretically could have zero business value. The test *does* have a cost: maintenance, computing time, administration, and so forth. That means the test could have *net negative value*. That is the fourth category of tests to remove. These are tests which, though they may even do some amount of verification, do no validation.

> ☞ *If you cannot tell how a unit test failure contributes to product risk, you should evaluate whether to throw the test*

away. There are better techniques to attack quality lapses in the absence of formal correctness criteria, such as exploratory testing and Monte Carlo techniques. (Those are great and I view them as being in a category separate from what I am addressing here.) Don't use unit tests for such validation.

Note that there are some units for which there is a clear answer to the business value question. One such set of tests is regression tests; however, those rarely are written at the unit level but rather at the system level. We know what bug will come back if a regression test fails — by construction. Also, some systems have key algorithms — like network routing algorithms — that are testable against a single API. There is a formal oracle for deriving the tests for such APIs, as I said above. So those unit tests have value.

☞ *Consider whether the bulk of your unit tests should be those that test key algorithms for which there is a "third-party" oracle for success, rather than one created by the same team that writes the code. "Success" here should reflect a business mandate rather than, say, the opinion of a team member called "tester" whose opinion is valued only because it is independent. Of course, an independent evaluation perspective is also important.*

1.6 Complex Things are Complicated

There is a dilemma here, and that is that in some software, most of the interesting quality data are in the tails of the test result distributions, and conventional approaches to statistics tell you the wrong things. So a test may pass 99.99% of the time but the one test in ten thousand that fails kills you. Again, borrowing from the hardware world, you can design for a given probability of failure or you can do *worst-case analysis* (WCA) to reduce the probability of failure to arbitrarily low levels. Hardware people typically use WCA during asynchronous system design to guard against "glitches" in signal arrivals that wander outside

the design parameters one in every100 million times. In hardware, such a module would be said to have a FIT rate of 10 — ten Failures In a Trillion.

The client that I mentioned near the start of this article was puzzled about why tests weren't working in his team, because they had worked for him in an earlier job. I sent him an earlier version of this paper and he replied,

> It is a pleasure to read it while it makes clear why things did work for me (and the rest of the team). As you might know, I am an avionics engineer whose career started as an embedded software developer with one foot in the hardware development. That is how I started testing my software, with a hardware mindset. (It was a four men team: 3 electrical engineers from Delft University (incl. me specialized in avionics) and one software engineer (The Hague University). We were highly disciplined while we were working on security systems for banks, penitentiaries, fire houses, police stations, emergency services, chemical plants, etc. It had to right the first time *all* the time.)

Given reasonable assumptions, you can do WCA in hardware largely because cause-and-effect relationships are easily traceable: we can look at the wiring to see what causes a memory element to change state. The states in a Von Neumann change as a side effect of function execution and it is in general impossible to trace the cause of a given state change, or even if a given state is reachable. Object-orientation makes it worse. It is impossible to know, for a given use of some state value within a program, what instruction last modified that state.

Most programmers believe that source line coverage, or at least branch coverage, is enough. No. From the perspective of computing theory, worst-case coverage means investigating every possible *combination* of *machine* language sequences, ensuring that each instruction is reached, and proving that you have reproduced every possible configuration of bits of data in the program at every value of the program counter. (It is insufficient to reproduce the state space for just the module or

class containing the function or method under test: generally, any change anywhere can show up anywhere else in a program and requires that the entire program can be retested. For a formal proof, see the paper: Perry and Kaiser, "Adequate Testing and Object–oriented Programming," *Journal of Object-Oriented-Programming 2*(5), Jan. 1990, p. 13). For a smallish program we are already into a test inventory way beyond the number of molecules in the universe. (My definition of code coverage is the percent of all possible pairs, {*Program Counter*, *System State*} that your test suite reproduces; anything else is a heuristic, and you'll probably be hard-pressed to find any rationale for it.) Most undergraduate CS graduates will recognize the Halting Problem in most variants of this exercise and know that it is impossible.

1.7 Less is More, or: You are Not Schizophrenic

There's another gotcha here, specifically with respect to the initial question from my client. The naïve tester will try to tease data from the tails by keeping all the tests around or even by adding more tests; that leads exactly to the situation my client found himself in, with more complexity (or code mass or choose-your-favourite-measure) in the tests than in the code. The classes he was testing are code. The tests are code. Developers write code. When developers write code they insert about three system-affecting bugs per thousand lines of code. If we randomly seed my client's code base — which includes the tests — with such bugs, we find that the tests will hold the code to an incorrect result more often than a genuine bug will cause the code to fail!

Some people tell me that this doesn't apply to them since they take more care in writing tests than in writing the original code. First, that's just poppycock. (The ones that really make me laugh are the ones who tell me they are able to forget the assumptions they made while coding and bring a fresh, independent set to their testing effort. Both of them have to be schizophrenic to do that.) Watch what your developers do when

running a test suite: they're *doing*, not *thinking* (like most of the Agile Manifesto, by the way). There was a project at my first job in Denmark heavily based on XP and unit testing. I faithfully tried to bring up the software build on my own machine and, after many struggles with Maven and other tools finally succeeded in getting a clean build. I was devastated when I found that the unit tests didn't pass. I went to my colleagues and they said, "Oh, you have to invoke Maven with this flag that turns off those tests — they are tests that no longer work because of changes in the code, and you need to turn them off."

If you have 200 tests — or 2000, or 10,000 — you're not going to take time to carefully investigate and (ahem) re-factor each one every time it fails. The most common practice — which I saw at a startup where I used to work back in 2005 — is to just overwrite the old test golds (the expected output or computational results on completion of a given test) with the new results. Psychologically, the green bar is the reward. Today's fast machines give the illusion of being able to supplant the programmer's thinking; their speed means I don't take the time to think. In any case, if a client reports a fault, and I hypothesize where the actual bug lies and I change it so the *system* behavior is now right, I can easily be led to believe that the function where I made the fix is now right. I accordingly overwrite the gold for that function. But that's just bad science and is rooted in the witchcraft that correlation is causality. It's necessary to re-run all the regressions and system tests as well.

Second, even if it *were* true that the tests were higher quality than the code because of a better process or increased attentiveness, I would advise the team to improve their process so they take the smart pills when they write their code instead of when they write their tests.

1.8 You Pay for Tests in Maintenance — and Quality!

The point is that code is part of your system architecture. Tests are modules. That one doesn't deliver the tests doesn't relieve

one of the design and maintenance liabilities that come with more modules. One technique commonly confused with unit testing, and which uses unit tests as a techique, is Test-Driven Development. People believe that it improves coupling and cohesion metrics but the empirical evidence indicates otherwise (one of several papers that debunk this notion with an empirical basis is Janzen and Saledian, "Does Test-Driven Development Really Improve Software Design Quality?" *IEEE Software 25*(2), March/April 2008, pp. 77 - 84.) To make things worse, you've introduced coupling — coordinated change — between each module and the tests that go along with it.

When I look at most unit tests — especially those written with JUnit — they are *assertions* in disguise. When I write a great piece of software I sprinkle it with assertions that describe promises that I expect the callers of my functions to live up to, as well as promises that function makes to its clients. Those assertions evolve in the same artefact as the rest of my code. Most environments have provisions to administratively neuter those assertions when you ship.

An even more professional approach is to leave the assertions in the code when you ship, and to automatically file a bug report on behalf of the end user and perhaps to try to re-start the application every time an assertion fails. At that same startup I mentioned above I had a boss who insisted that we not do this. I pointed out to him that an assertion failure meant that something in the program was very wrong and that it was likely that the program would produce the wrong result. Even the tiniest error in the kind of software we were building could cost a client $5 million in rework. *He said it was more important that the company avoid the appearance of having done something wrong than that we stop before producing an incorrect result.* I left the company. Maybe you are one of his clients today.

> ☞ *Turn unit tests into assertions. Use them to feed your fault-tolerance architecture on high-availability systems.*

This solves the problem of maintaining a lot of extra software modules that assess execution and check for correct behavior; that's one half of a unit test. The other half is the driver that executes the code: count on your stress tests, integration tests, and system tests to do that.

Almost last, there are some unit tests that just reproduce system tests, integration tests, or other tests. In the early days of computing when computers were slow, unit tests gave the developer more immediate feedback about whether a change broke the code instead of waiting for system tests to run. Today, with cheaper and more powerful computers, that argument is less persuasive. Every time I make a change to my Scrum Knowsy® app, I test at the system level. Developers should be integrating *continuously* and doing system testing *continuously* rather than focusing on their unit tests and postponing integration, even by an hour. So get rid of unit tests that duplicate what system tests already do. If the system testing level is too expensive, then create subunit integration tests. Rex feels that "the next great leap in testing is to design unit tests, integration tests, and system tests such that inadvertent gaps and overlap are removed."

☞Check your test inventory for replication; you can fund this under your Lean program. Create system tests with good feature coverage (not code coverage) — remembering that proper response to bad inputs or other unanticipated conditions is part of your feature set.

Last: I once heard an excuse from someone that they needed a unit test because it was impossible to exercise that code unit from any external testing interface. If your testing interfaces are well-designed and can reproduce the kinds of system behaviours you see in the real world, and you find code like this that is unreachable from your system testing methodology, then.... **delete the code**! Seriously, reasoning about your code in light of system tests can be a great way to find dead code. That's even

more valuable than finding unneeded tests.

1.9 Wrapup

Back to my client from Sogeti. At the outset, I mentioned that he said:

> When I was programming on a daily basis, I did make code for testability purposes but I hardly did write any unit tests. However I was renowned for my code quality and my nearly bug free software. I like to investigate WHY did this work for me?

Maybe Richard is one of those rare people who know how to think instead of letting the computer do your thinking for him — be it in system design or low-level design. I tend to find this more in Eastern European countries, where the lack of widely available computing equipment forced people to think. There simply weren't enough computers to go around. When I first visited Serbia back in 2004, the students at FON (the faculty where one learned computing) could get to a computer to access the Internet *once a week*. And the penalty for failure is high: if your code run doesn't work, you have to wait another week to try again.

I fortunately was raised in a programming culture like this, because my code was on punch cards that you delivered to the operator for queuing up to the machine and then you gathered your output 24 hours later. That forced you to think — or fail. Richard from Sogeti had a similar upbringing: They had a week to prepare their code and just one hour per week to run it. They had to do it *right first time*. By all means, a learning project should assess the cost impediments and remove another one every iteration, focusing on ever-increasing value. Still, one of my favourite cynical quotes is, “I find that weeks of coding and testing can save me hours of planning.” What worries me most about the *fail-fast* culture is much less in the *fail* than the *fast*. My boss Neil Haller told me years ago that debugging isn’t what you do sitting in front of your program with a debugger; it’s

what you do leaning back in your chair staring at the ceiling, or discussing the bug with the team. However, many supposedly agile nerds put processes and JUnit ahead of individuals and interactions.

The best example was one I heard last year, from a colleague, Nancy Githinji, who used to run a computing company with her husband in Kenya; they both now work at Microsoft. The last time she was back home (last year) she encountered some kids who live out in the jungle and who are writing software. They get to come into town once a month to get access to a computer and try it out. *I want to hire those kids!*

As an agile guy (and just on principle) it hurts me a little bit to have to admit that Rex is right now and then ☺, but he put it very eloquently: "There's something really sloppy about this 'fail fast' culture in that it encourages throwing a bunch of pasta at the wall without thinking much… in part due to an over-confidence in the level of risk mitigation that unit tests are achieving." The fail-fast culture can work well with very high discipline, supported by healthy skepticism, but it's rare to find these attitudes surviving in a dynamic software business. Sometimes failure requires thinking, and that requires more time than would be afforded by failing fast. As my wife Gertrud just reminded me: no one wants a failure to take a long time…

If you hire a professional test manager or testing consultant, they can help you sort out the issues in the bigger testing picture: integration testing, system testing, and the tools and processes suitable to that. It's important. But don't forget the Product Owner perspective in Scrum or the business analyst or Program Manager: risk management is squarely in the center of their job, which may be why Jeff Sutherland says that the PO should conceive (and at best *design*) the system tests as an input to, or during, Sprint Planning.

As for the Internet: it's sad, and frankly scary, that there isn't

much out there. There's a lot of advice, but very little of it is backed either by theory, data, or even a model of why you should believe a given piece of advice. Good testing begs skepticism. Be skeptical of yourself: measure, prove, retry. Be skeptical of *me* for heaven's sake. Write me at jcoplien@gmail.com with your comments and copy Rex at the address at the front of this newsletter.

In summary:

- *Keep regression tests around for up to a year — but most of those will be system-level tests rather than unit tests.*
- *Keep unit tests that test key algorithms for which there is a broad, formal, independent oracle of correctness, and for which there is ascribable business value.*
- *Except for the preceding case, if X has business value and you can text X with either a system test or a unit test, use a system test — context is everything.*
- *Design a test with more care than you design the code.*
- *Turn most unit tests into assertions.*
- *Throw away tests that haven't failed in a year.*
- *Testing can't replace good development: a high test failure rate suggests you should shorten development intervals, perhaps radically, and make sure your architecture and design regimens have teeth*
- *If you find that individual functions being tested are trivial, double-check the way you incentivize developers' performance. Rewarding coverage or other meaningless metrics can lead to rapid architecture decay.*
- *Be humble about what tests can achieve. Tests don't improve quality: developers do.*