subscribe to the Embedded Muse.

A Guide to Code Inspections

Version 2. August 2009. Complete rewrite
Version 2.1 February 2010 minor edits

What's the cheapest way to get rid of bugs?

Don't put them in in the first place.

That seemingly trite statement is the idea behind the entire quality revolution that reinvented manufacturing during the 70s. Design quality in rather than try to fix a lot of problems during production.

Most people under 40 have no memory of the quality problems US automotive vendors inflicted on their customers for many years. I remember my folks buying cars in the 60s. With five kids on a single-income engineer's salary my dad's primary decision parameters (mom was never consulted on such a purchase) were size (a big station wagon) and price. Choices were mostly limited to the Big Three. With the exception of the even-then ubiquitous VW Beetle, foreign manufacturers had made few inroads into this market. But Detroit's offerings were always plagued with problems, from small nuisance issues to major drivetrain troubles. Consumers had no recourse since *all* of the vendors offered the same poor quality. Perhaps foreshadowing today's low expectations about commercial software, car buyers forty years ago accepted the fact that vehicles were full of problems, and many trips to the dealer to get these cleared up was simply part of the process of acquiring a new car.

About the same time Japanese products had a well-deserved bad reputation. "Made in Japan" and "junk" were synonymous. But Japanese managers became students of quality guru W. Edwards Deming, who showed how a single-minded focus on cost at the expense of quality was suicidal. They eventually shifted production to a low-waste system with an unyielding focus on designing quality in. The result: better autos at a lower cost. Detroit couldn't compete. (Of course, many other factors contributed to the US firms' 1970s' woes. But cash-strapped American buyers found the lure of lower-cost high-quality foreign cars irresistible.)

US vendors scrambled to compete using, at first, the marketing rather than substance. "Quality is Job One" became Ford's tagline in 1975. Buyers continued to flock to less self-aggrandizing manufacturers who spoke softly but carried few defects. But by the very early eighties Ford was spewing red ink at an unprecedented rate. A division quality manager hired Deming to bring the Japanese miracle to Detroit. Eventually the quality movement percolated throughout the automotive industry, and today it might be hard to find much of a difference in fit and finish between any manufacturer. Tellingly, Ford abandoned "Quality is Job One" as a mantra in 1998. The products demonstrated their success and marketing sleight of hand was no longer needed to dodge an inconvenient truth.

Lean manufacturing perhaps got its name from a 1989 book (Lean Thinking by James Womack and Daniel Jones) but its roots trace back to at least Ben Franklin and later to Henry Ford. Waste means there's a problem in the process, whether the waste is from rework (errors) or practices that lead to full garbage pails. Wastage is a sure indicator that something is wrong with any process. And it's an equally-vital red flag that a software development group is doing something wrong.

For some reason the lean revolution by and large hasn't made it into software engineering. Bugs plague our efforts, and are as expected as any other work product. Most projects get bogged down in a desperate debugging phase that can consume half the schedule. I guess that means we can call the design and coding line items the "bugging" phase.

Software is hard and bugs will always be our bane. But I take exception to any zeitgeist that accepts bugs.

Bugs are like disease: though surely part of the human condition they are an abomination we must seek to prevent. But how can one preempt them? Is there an inoculation for defects?

When, in 1796 Edward Jenner rubbed cowpox on eight year old James Phipps' arms he wasn't fixing a fever; the boy was perfectly healthy. Rather, Jenner knew some 60% of the population was likely to get smallpox and so was looking for a way to *prevent* the disease before it occurred. That idea was revolutionary in a time when illness was poorly understood and often attributed to vapors or other imaginary effects of devils or magic.

The pre-Jenner approach parallels software engineering with striking similarity. The infection is there; with enough heroics it might be possible to save the patient, but the toll on both the victim and doctor leaves both weakened. Jenner taught us to anticipate and eliminate sickness. Lean manufacturing and the quality movement showed that defects indicate a problem with the *process* rather than the product. Clearly, if we can minimize waste the system will be delivered faster and with higher quality.

In other words, cut bugging to shorten debugging. The best tool we have to reduce bugging is the code inspection.

The statistics are dramatic. Most code starts life with around five to ten bugs per hundred lines of code. Without inspections figure on finding about 95% of those pre-shipping. For a 100KLOC program that means shipping with hundreds of defects! (Note that some sources, such as Stan Rifkin, "The Business Case for Software Process Improvement", Fifth SEPG National Meeting, 26-29 April 1993, claim far worse numbers, on the order of 6 shipped bugs per hundred lines of code - six thousand in a 100KLOC project. My unscientific observations, coupled with data from private conversations with Capers Jones, suggest that firmware generally does not sink to that abysmal level.)

It's important to understand that testing, as implemented by most companies, just does not work. Many studies confirm that tests exercise only about half of the code. Exception handlers are notoriously problematic. Also consider nested conditionals: `IFs` nested five or six levels deep can yield hundreds of possible conditions. How many test regimes check all of those possibilities?

Cyclomatic complexity, among other things, tells us the minimum number of tests needed to completely exercise a function. I found that one function in the Apache web server scored a complexity index of 725. Who constructs nearly a thousand tests for a single function?

Another analysis of the testing problem is scarier. In "The Software Quality Challenge" (Crosstalk, June 2008, http://www.stsc.hill.af.mil/crosstalk/2008/06/0806Humphrey.html) Watts Humphrey shows that a program with 100 decisions can have nearly 200,000 possible paths. With 400 decisions that explodes to 10^{11} paths, each requiring a test! Though he describes a pathological worst-case situation, the implications are important.

Some approaches to test are much more effective. Code coverage, used in the avionics and some other industries, can approach 100% testing, albeit with huge costs.
Test Driven Development advocates also claim good results, though reliable supporting studies are hard to find. However, anecdotal evidence does suggest that most of the agile approaches, which have a disciplined focus on testing, greatly improve on the 50% code test ratio. The agile approaches demand, correctly, automated tests that cost little to run. But in the embedded space that's hard. Someone has to watch the display and press the buttons. I've seen some very cool ways to automate those activities, and some agilists successfully build mock objects to simulate the user interaction. So-called virtual prototypes, too, are gaining market share (see the products from Virtutech or VaST, for instance).

The bottom line is that test is a huge problem. However, a well-run inspection program will eliminate 70 to 80% of the bugs *before* any testing starts.

When I was a young engineer my boss demanded that all schematics go through a formal design review. It was expensive to respin boards, and even costlier to troubleshoot and repair design mistakes. At first I found this onerous: a colleague was going to critique my work? How mortifying when someone found errors in my circuits! But a little time spent reviewing saved many hours downstream. Even better was that I was able to tap into the brains of some really smart engineers. One was a whiz at manufacturability issues, and he brought this insight to each review. What did college teach about that subject? Nothing, but Ken's probing dissections of my designs was a crash course in the art of creating products that were buildable and maintainable. Another engineer always eerily spotted those ghostly race conditions that are so tough to find. The old saw that many brains being better than one was proven many times in these reviews, and the same idea applies to inspecting code.

Plenty of evidence suggest well-run inspections are some 20 times cheaper than debugging. That's a staggering number. Since the average project burns about half the schedule in the debugging phase, any technique with a 20X improvement nearly halves the schedule. I'm generally skeptical of any dramatic claim, from the Ginzu knives to politicians' promises to clean up Washington, but even if you cut that number by an order of magnitude inspections still shave a third from the schedule.

Higher quality code, for less money. What a deal!

Inspections are a quality gate, a filter for bugs. They must take place after one gets a clean compile (i.e., let the tools find the easiest bugs) and before any testing starts. Yes, most developers prefer to inspect code they've beaten into submission with the debugger for a while, but given the 20X efficiency factor, doing any testing before inspecting is like burning stacks of $100 bills. It's simply bad business. Even worse, early testing leads to the demise of the inspections. Few of us are strong enough to resist the siren call of "But it works!" We're all busy, claims of "works," which may mean little depending on the efficacy of the tests, generally result in the inspection meeting being dismissed.

One complaint about inspections is that they may find few bugs. That, of course, is exactly our goal. If I write software no one will see, and then write the same function given the knowledge that my peers will be digging deep into its bowels, be sure the code will be very different. No one wants his errors to be pointed out. The inspections are therefore sort of an audit that immediately raises the program's quality.

But remember this: after examining 4000 software projects, Capers Jones found the number one cause of schedule problems was quality. (Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Yourdon Press, 1994.) Bugs. Inspections don't slow the project. Bugs do.

The Process

Code inspections are nothing new; they were first formally described by Michael Fagan in his seminal 1976 paper "Design and Code Inspections to Reduce Errors in Program Development" (Fagan, Michael E, IBM Systems Journal, Vol. 15, No. 3, 1976, available on-line at http://www.research.ibm.com/journal/sj/153/ibmsj1503C.pdf.) But even in 1976 engineers had long employed design reviews to find errors before investing serious money in building a PCB, so the idea of having other developers check one's work before "building" or testing it was hardly novel.

Fagan's approach is in common use, though many groups tune it for their particular needs. I think it's a bit too "heavy" for most of us, though those working on safety-critical devices often use it pretty much unmodified. I'll describe a practical approach used by quite a few embedded developers.

First, the objective of an inspection is to find bugs. Comments like "man, this code sucks!" are really attacks on the author and are inappropriate. Similarly, metrics one collects are not to be used to evaluate developers.

Secondly, *all new code gets inspected*. There are no exceptions. Teams that exclude certain types of routines because either they're hard ("no one understands DMA here") or because only one person really understands what's going on will find excuses to skip inspections on other sorts of code, and will eventually give up inspections altogether. If only Joe understands the domain or the operation of a particular function, the process of inspection spreads that wisdom and lessens risk if Joe were to get sick or quit.

We only inspect *new* code because there just isn't time to pour over a million lines of stuff inherited in an acquisition.

In an ideal world an inspection team has four members plus maybe a newbie or two that attend just to learn how the products work. In reality it might be hard to find four people so fewer folks will participate. But there are four roles, all of which must be filled, even if it means one person serves in two capacities:

- A moderator runs the inspection process. He finds a conference room, distributes listings to the participants, and runs the meeting. That person is one of us, not a manager, and smart teams insure everyone takes turns being moderator so one person isn't viewed as the perennial bad guy.
- A reader looks at the code and translates it into an English-language description of the statement's intent. The reader doesn't say: `"if (tank_press>max_press)dump();"` After all, the program is nothing more than a translation of an English-language spec into computerese. The reader converts the C back into English, in this case saying something like: "Here we check the tank pressure to see if it exceeds the maximum allowed, which is a constant defined in max_limits.h. If it's too high we call dump, which releases the safety valve to drop the pressure to a safe value." Everyone else is reading along to see if they agree with the translation, and to see if this is indeed what the code should do.
- The author is present to provide insight into his intentions when those are not clear (which is a sign there's a problem with either the code or the documentation). If a people shortage means you've doubled up roles, the author may not also be the reader. In writing prose it has long been known that editing your own work is fraught with risk: you see what you thought you wrote, not what's on the paper. The same is true for code.
- A recorder logs the problems found.

During the meeting we don't invent fixes for problems. The author is a smart person we respect who will come up with solutions. Why tie up all of these expensive people in endless debates about the best fix?

We do look for testability issues. If a function looks difficult to test then either it, or the test, needs to be rethought.

Are all of the participants familiar enough with the code to do a thorough inspection? Yes, since before the meeting each inspects the code, alone, in the privacy of his or her own office, making notes as necessary. Sometimes it's awfully hard to get people other than the author to take the preparation phase seriously, so log each inspector's name in the function's header block. In other words, we all own this code, and each of us stamped our imprimatur on it.

The inspection takes place after the code compiles cleanly. Let the tools find the easy problems. But do the inspection before any debugging or testing takes place. Of course everyone wants to do a bit of testing first just to insure our colleagues aren't pointing out stupid mistakes. But since inspections are some 20 times cheaper than test, it's just a waste of company resources to test first. Writing code is a business endeavor; we're not being paid to have fun - though I sure hope we do - and such inefficient use of our time is dumb.

Besides, in my experience when test precedes the inspection the author waltzes into the meeting and tosses off a cheery "but it works!" Few busy people can resist the temptation to accept "works", whatever that means, and the inspection gets dropped.

My data shows that an ideal inspection rate is about 150 lines of code per hour. Others suggest slightly different numbers, but it's clear that working above a few hundred lines of code per hour is too fast to find most errors. 150 LOC/hr is an interesting number: if you've followed the rules and kept functions to no more than 50 LOC or so, you'll be inspecting perhaps a half-dozen functions per hour. That's 2.5 LOC/minute, slow enough to really put some thought into the issues. The cost is around $2/LOC, which is in the noise compared to the usual $20 to $40/LOC cost of most firmware. Of course, the truth is that inspections save time so have a negative cost.

It's impossible to inspect large quantities of code. After an hour or two one's brain turns to pudding. Ideally, limit inspections to an hour a day.

The best teams measure the effectiveness of their inspection process, and, in fact, measure how well they nab bugs pre-shipping in every phase of the development project. Two numbers are important: the defect potential, which is the total number of bugs found in the project between starting development and 90 days after shipping to customers, and the defect removal efficiency. The latter is a fancy phrase that is simply the percentage of bugs found pre-shipping.

Clearly you can't measure defect potential unless you track every bug found during development, whether discovered in an inspection or when Joe is furiously debugging his code. Being just a count it's painless to track.

If your inspections aren't finding at least 60% of all of the bugs there's something wrong with the process. Tune as needed. 70% is excellent, and some companies achieve 80%.

(To put more numbers around the process, Capers Jones showed in his paper "Measuring Defect Potentials and Defect Removal Efficiency" Crosstalk magazine, June, 2008, www.stsc.hill.af.mil/crosstalk/2008/06/0806Jones.html, that the industry averages an 85% defect removal efficiency. At the usual 5 bugs injected per 100 LOC that means shipping with 7.5 bugs per KLOC. Achieve 60% efficiency just in inspections and figure on 4.5 per KLOC. Jones found that the lowest development costs occur at a defect removal efficiency of 95%, or 2.5 bugs/KLOC. According to a vast library of data he sent me, projects run at 95% cost nearly 50% less to develop than ones idling along at 85% yet they ship with three times fewer bugs. And he calls an efficiency of 75% "professional malpractice.")

Inspections are irrevocably tied to the use of firmware standards. If a team member complains about a stylistic issue, the moderator says: "does this conform to the standard?" If the answer is "yes," the discussion is immediately closed. If no, the proper response is a simple: "fix it." No debate, no expensive discussion. Don't use inspections and the team will surely drift away from the firmware standard. The two processes are truly synergistic

Managers have an important role: stay away. Enable the process, provide support, demand compliance, but never use the inspections as a way to evaluate people. This is all about the code, and nothing else.

Managers must always insure the team uses inspections and all of the other processes even when the delivery date looms near. Can you imagine getting on a plane, walking in the door, only to overhear the pilot say to the co-pilot: "We're running late today; let's skip the pre-takeoff checklist." What amateurs! Management can't let the threat of a schedule dismember proper processes . . . which, after all, shorten delivery times.

Inspections inspire plenty of debate, but few who have looked at the data doubt their importance. In today's agile world some advocate a less formal approach, though all buy into the many-brains-is-better-than-one theory. For an example of lighter-weight reviews see http://www.methodsandtools.com/archive/archive.php?id=66.

The agile community buys into inspections. As Kent Beck wrote in eXtreme Programming Explained: "When I first articulated XP, I had the mental image of knobs on a control board. Each knob was a practice that from experience I knew worked well. I would turn all the knobs up to 10 and see what happened. I was a little surprised to find that the whole package of practices was stable, predictable, and flexible." One of those knobs is code inspections, embodied in XP's pair programming.

The most complete work on inspections is Software Inspection, by Tom Gilb and Dorothy Graham (Addison-Wesley, 1993, ISBN 978-0201631814). It's a great cure for insomnia. Or consider Peer Reviews in Software, by Karl E. Wiegers (Addison-Wesley, 2001, ISBN 978-0201734850). Written in an engaging style, rather like the Microsoft Press books, it's an accessible introduction to all forms of inspections, covering more than the traditional Fagan versions. One of my favorite books about inspections is a free one available from SmartBear Software called Best Kept Secrets of Peer Code Review (www.smartbearsoftware.com).

Then there's Software Inspection: An Industry Best Practice by Bill Brykczynski (Editor), Reginald N., Jr. Meeson (Editor), David A. Wheeler (Editor), IEEE Press, 1996, ISBN 978-0818673405. Unhappily it's out of print but used copies are usually available on Amazon. The book is a collection of papers about successful and failed inspections. But it, but don't read it. When I can't stand the thought of yet another inspection I pull it off the shelf and read one paper at random. Like a tent revival meeting it helps me get back on the straight and narrow path.

Inspections yield better code for less money. You'd think anything offering that promise would be a natural part of everyone's development strategy. But few embedded groups use any sort of disciplined inspection with regularity. The usual approach is to crank some code, get it to compile (turning off those annoying warning messages) and load the debugger. Is it any surprise projects run late and get delivered loaded with bugs?

But consider this: why do we believe (cue the choir of angels before uttering these sacred words) open-source software is so good? Because with enough eyes, all bugs are shallow.

Oddly, many of us believe this open-source mantra with religious fervor but reject it in our own work.