ALM and Development Tools

Dr. Dobb's Journal

July 2012

Deploying Static Analysis

There's a cheap and easy way to find bugs, but it comes with challenges that tend to be more political than technical

Next

ALSO INSIDE

UBM TechWeb

www.drdobbs.com

Previous Next

Dr. Dobb's Journal

CONTENTS

August 2012

COVER STORY

More on DrDobbs.com

Interview with Alan Kay

The pioneer of object-orientation, co-designer of Smalltalk, and UI luminary opines on programming, browsers, objects, the illusion of patterns, and how Socrates could still make it to heaven..

http://is.gd/ZT8ICC

***Dr. Dobb's* 2012 Salary Survey**

Our survey of nearly 3500 developers and managers shows that while many salaries are flat, they are increasing overall — except for some heavily disfavored niches.

http://is.gd/YIyM4C

The Best of the First Half

The most popular articles on *Dr. Dobb's* for the first half of the year, sprinkled with editors' choices of particularly meritorious pieces. Enjoy!.

http://is.gd/YjBucz

Measuring Performance for Analytics, Decision Support, and the Data Warehouse

The Transaction Processing Council (TPC) produces specifications for benchmarks to compare hardware and software configurations used for applications such as online transaction processing (OLTP) and decision support.

http://is.gd/quzgRO

Asynchronous Java Calls, Part 1

The asynchronous call paradigm results in fast, responsive UIs with a very clean and consistent programming model.

http://is.gd/UdPxep

Mailbag

Alan Kay, more on bad programmers, and defining sofware engineering

The month-long heat wave that has baked most of the US appears to have turned many letter writers downright peevish. We were reassured, however, by reader Thomas Lewis, who wrote in a brief note about the interview with Alan Kay (http://is.gd/ZT8ICC), "It's good to wink sometimes."

Bad Programmers

The June editorial "What Makes Bad Programmers Different?" (http://is.gd/D1cwSk) was met with some impassioned responses:

I am not a professional programmer. I was a professor of Mechanical Engineering and Mathematics for five years, and before that, I worked as a researcher. I have had to work with many so-called programmers in industry before I went into my Ivory Tower. It always got to me, when someone with a so-called degree from a "Mickey Mouse" College or school of some kind, said they were programmers. They never even came close to the education that I have had in computer science, let alone writing all the code I had to for all of my main core courses, such as heat transfer. Those good old punch cards still bring back memories. Nowadays, the kids that think they can program can check run their programs in a flash, yet they have no idea what computer science is all about, let alone the advanced mathematics that it takes to write good code. Nor do they have the background in other fields of study, even though they profess to know it all. But they sure come out of these schools thinking they know it all.

Prof. Jake Trexel
Ph.D Mechanical Engineering / Ph.D. Mathematics
Knoxville, Tenn.

Andrew Binstock replies: *I understand your point, but I'm not convinced. When I was in IT, I recall managers discussing hiring comp-sci graduates who could happily write algorithms for running elevators efficiently, but couldn't figure out how to code a multi-way control break in a report. Some of the best programmers I've met were self-educated. The only thing they tended to lack was the broad exposure to programming that an enforced curriculum provides.*

A bigger problem is the bad manager/team lead/decision maker. The decision makers mistakenly assume that it takes more time to do it right than to do it wrong. Therefore, if they do it wrong, they are more likely to come in on-time, on-budget. Everyone — good and bad programmers, testers, DBAs, etc. — sees that the dominant personalities who control the decisions do not value doing it right. This has been

consistently true in my 30 years in IT at Fortune 100, and in my 20 years consulting business prior to IT. It is our culture. The bad programmers do not improve themselves because they see no reason to change.

In IT, there has always been a shortage of IT workers willing and able to do the work. The result is that companies hire immigrants or offshore. And the #1 reason for companies doing it is because non-Americans of certain other cultures are more motivated than Americans to do what is needed to be done.

Bob Schmidt

Software Engineers

To the editorial "Software Engineers All!" (http://is.gd/ssn8qq), which discussed how the term "software engineer" has become a synonym for "programmer":

The confusion between programmers and software engineers was settled forty year ago. The origins of this issue go back to the controversial Department of Labor rules governing overtime payment for exempt and non-exempt workers. Programmers were considered non-exempt and considered eligible for overtime pay; software engineers and analysts were considered exempt and not automatically entitled to overtime payment. Why are you trying to stir this up again?

What is software? Harlan Mills, early pioneer in the application of structured programming, observed, "Software permits the harmonious cooperation between people and machines." This cannot be improved upon as a starting point for the definition of software.

What is software engineering?

- The term software engineering was coined at the NATO Software Engineering Conference held in Garmisch, Germany in 1968 to focus on the software crisis.
- In 1969, conference attendee Frederick L. Bauer provided a definition for software engineering, "The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines." At IBM's Federal Systems Division in 1988, Terry Baker defined software engineering as, "The application of conceptual models, basic structures, and unit operations to the systematic design, development, and management of software products which solve real world problems."
- *Scientific American*, September 1994, provided a definition of software engineering, "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software."
- Influenced by TRW's Barry Boehm, the IEEE definition for software engineering states, "The systematic approach to the development, operation, maintenance, and retirement of software."

Tell us something we don't already know.

Don O'Neill
Montgomery Village, MD

Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.

The Shrinking Role of QA

As developers do more of the testing and deliver working software on every iteration, the role of QA is becoming smaller and more ill-defined.

By Eli Lopian with Andrew Binstock

The Industrial Revolution began about 250 years ago and machines started to take over for human labor in factories, fields, and mines. This led to major economic growth, but machines started to replace the average worker, who could not get another job or acquire a new skill, which hurt many people. The similarity of the situation QA testing finds itself in currently is uncanny. QA was a savior during the big internet boom — when budgets were virtually unlimited and software was being produced at an exponential rate. However, as the economy bottomed out, budgets deflated, Agile development became more widespread, and automated testing began taking over. Just as some forms of manual labor became obsolete over time due to the Industrial Revolution, manual QA testing finds itself facing the same predicament.

Let's take a step back and examine how things used to work. Waterfall has been the method of choice for many software development teams since the 1950s. This approach allowed for developers to design everything upfront, then focus on their code, pass it on to QA for testing, and get it back with a list of bugs to fix. Over time, systems were upgraded, development methods got better and faster, and we found ourselves producing so much software at such a rate that we needed more testers. More testers were hired and developers got lazier knowing that QA would be there to catch their mistakes.

In February 2001, as the dot-com bubble was bursting, the Agile Manifesto was published and a new way of developer thinking started to emerge. Agile development methodologies breathed new life into the developer world, where adapting to ever-changing situations and rapidly deploying working software became the focus of development teams. Agile is more involved in each working part of a team and the code, with a special focus on developer testing rather than QA testing. As Agile continues to become more widespread and effective, QA is needed less, and that means it has one foot out the door.

More QA, More Problems

Enterprise software development is an expensive and complex endeavor. When using the Waterfall model, it is common to find that the original product goals are not reached — at which point, management typically makes one of three choices:

- Add Budget: More money can be thrown at a project. This might make it possible to complete the project on time, taking into account the law of diminishing returns. But this is not a preferred scenario for management.
- Trim Features: Neither developers nor management are keen on giving customers less than they are paying for. This generally is not an option for many companies.
- Lower Quality: Quality is expensive, and thus is often the first thing to be cut back.

What we are left with is a sense of imbalance. Developers create code that lacks the necessary quality, while at the same time, management is cutting down on QA. There is a fundamental flaw here. This is where Agile fundamentals can come into play.

It's An Agile Thing

After the dot-com bubble burst and the economy was gradually working its way back, companies knew that they needed to create good software without the huge costs. Fortunately, Agile development is about developers testing their own code. While QA can still test fringe or flow cases, developers recognize the need to take responsibility for their code. One of the pillars of Agile development is delivery of working software. Some Agile methodologies include TDD and unit testing performed by the developers. Unit testing is about developers doing their part to make the whole better with the benefit of continuous, instant feedback that flushes bugs out quickly and cheaply. Without good unit testing coverage, it is very hard to stay Agile because, as a design continuously changes, the unit tests can flag problematic consequences of the rolling changes.

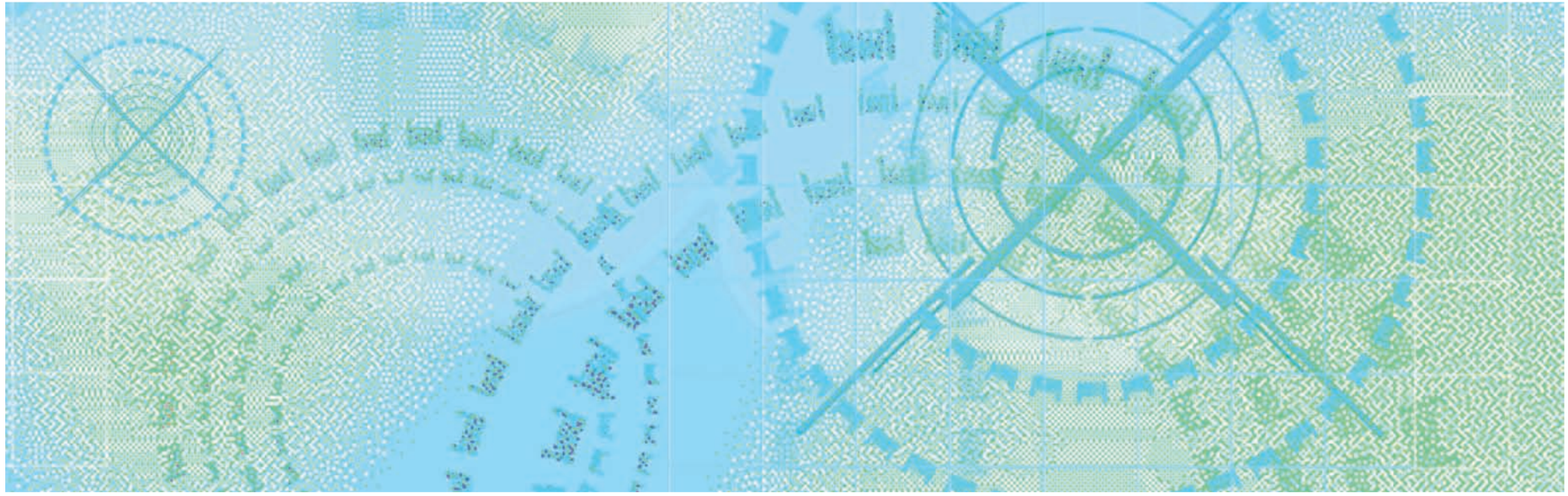
Unit Testing: Possible QA Killer

It has been shown that unit testing can properly check 90% of code and, unlike QA Automation tools, unit tests that are built correctly evolve with your codebase. As a platform for testing code (and creating abstract design), unit testing costs less and gets better quality software out the door quicker than competing approaches that rely on QA to find all the defects. And, as mentioned previously, automated unit testing and TDD enable developers to adapt their code to new features and other changes on the fly. When combined with other Agile techniques, these approaches rely on delivering working software every day. The question is what role, if any, QA departments will be left to handle?

At many companies, when working software passes UATs, it is good enough for deployment for internal use. This leaves a rapidly shrinking role for QA departments. Obviously, companies with products for external consumption or with tight safety or regulatory requirements will need extensive support from QA and ancillary testers, but increasingly, those might well be the last redoubts of the formerly universal QA departments. And even their fate will hinge upon how much testing developers take on, which itself is a function of how much developers can automate the testing of their own code.

— Eli Lopian is the CEO of Typemock, the Unit Testing Company. A well-known figure in the agile and test-driven-development arenas, Eli has over 17 years of R&D experience at large companies such as AMDOCS (NYSE:DOX) and Digital Equipment Corporation (DEC). Andrew Binstock is the editor in chief of **Dr. Dobb's.**

Comment



Deploying Static Analysis

Static analysis is a cheap and easy way to find bugs, but it offers important challenges that tend to be more political than technical.

By Flash Sheridan

Static source code analysis for bug finding ("static analysis" for short) is the process of detecting bugs via an automated tool that analyzes source code without executing it. The idea goes back at least to Lint, which was invented at Bell Labs in the 1970s, but static analysis has undergone a revolution in effectiveness and usability in the last decade. The initial focus of static analysis tools was on the C and C++ programming languages. Such tools are particularly necessary given C/C++'s notorious flexibility and susceptibility to low-level bugs. More recently, tools have flourished for Java and/or Web applications; these are needed because of the prevalence of easily exploitable network vulnerabilities. When using such tools, it is all too easy to deploy them in a way that looks good superficially, but misses important defects, shows many false positives, and brings the tool into disrepute. This article is a guide to the process of deploying a static analysis tool in a large organization while avoiding the worst organizational and technical pitfalls.

Leading commercial static analysis tools with which I am familiar include Coverity, Fortify (now owned by Hewlett-Packard), and Klocwork. Klocwork and Coverity both initially focused on C/C++, although they came from opposite origins: Klocwork from the telephone equipment company Nortel, and Coverity from Stanford University. Fortify's initial focus was on security for Web applications in languages such as Java and PHP. All three companies are now encroaching on each others' territories, but it remains to be seen how well they will do outside of their core competencies.

An excellent, free, but limited, academic static Java byte code analysis tool is FindBugs. Its lack of an integrated database for defect suppression makes its large-scale use difficult in sizable organizations, but its use by individual developers within the Eclipse development environment can be extremely valuable. Similarly, recent versions of Apple's Xcode and Microsoft's Visual Studio development environments contain integrated static analysis tools for C/C++. These are useful for finding relatively shallow bugs while an individual developer is writing code; their short feedback loop bypasses the difficulties of broader deployment of tools that perform deeper analysis. A longer list of tools is provided in Figure 1 (which is taken from "Magic Quadrant for Static Application Security Testing," by Gartner Inc.), albeit with a strong bias towards security and adherence to Gartner's strategy recommendations.

There is no general-purpose introductory textbook on the subject; the best general introduction is a short article by Dawson Engler, the

inventor of Coverity, et al. (http://is.gd/NKxpI4). Two of the leaders at Fortify have written an introductory textbook (http://is.gd/6YkJTO), but it focuses primarily on their tool and on security, and skimps on key static analysis concepts. There is a rigorous academic textbook on the more general concept of static analysis (http://is.gd/y7AOQM), but it preceded the revolution in static analysis for bug finding.

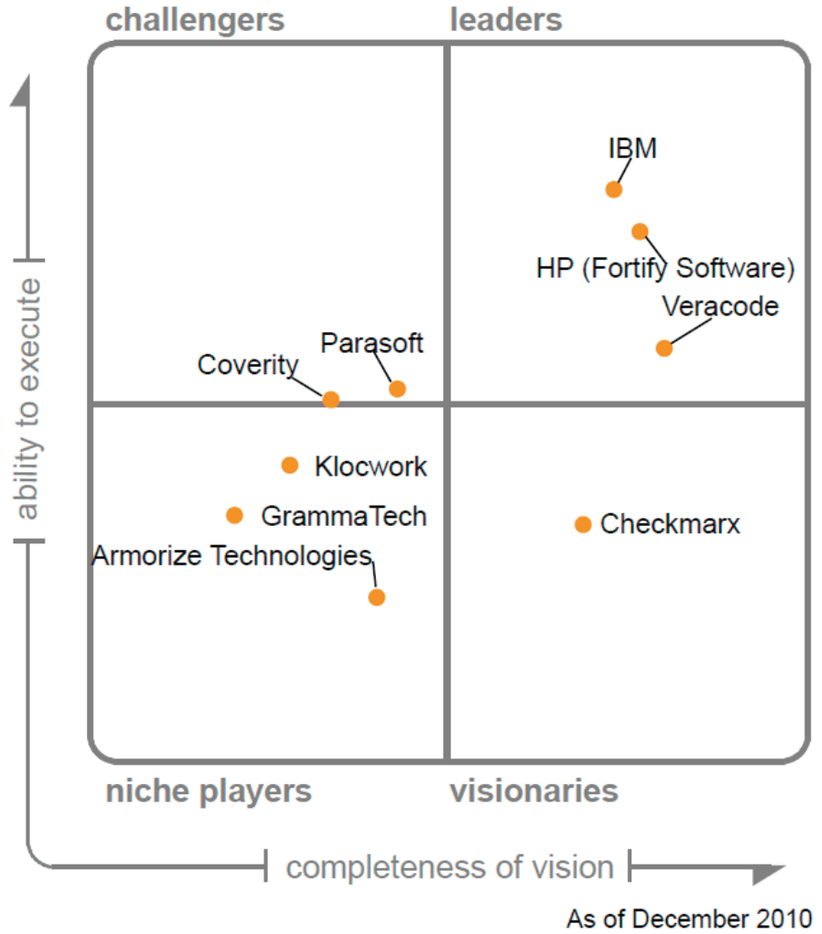
Getting Started: The Politics

The first question to ask before deciding to do static analysis in an organization is not what tool to buy, nor even whether you should do static analysis at all. It's "why?"

If your purpose is genuinely to help find bugs and get them fixed, then your organizational and political approach must be different from the more usual, albeit unadmitted, case: producing metrics and procedures that will make management look good. (Fixing bugs should actually be your second goal: An even higher goal is preventing bugs in the first place by making your developers learn from their mistakes. This also contraindicates outsourcing the evaluation and fixing of defects, tempting though that may be.)

Political Issues to Settle in Advance

Get buy-in from your testing/quality assurance department. They must support the project and will have authority over quality-related issues, even if they inconvenience the other stakeholders. Quality has a much smaller constituency than the schedule or the smooth running of internal procedures, but it must be the final arbiter for crucial quality-related decisions (see Chapter 22 of *Joel On Software* at http://is.gd/YBAqJn for more on this topic).



Source: Gartner (December 2010)

Figure 1.

Give some thought to what part of the organization, if any, should be in charge of running the tool once it's set up. If your organization has a tools team, it might seem the obvious owner, but this does need careful consideration. Static analysis for bug finding is probably not your

organization's core competency, and you will need to worry about the Iron Law of Bureaucracy: Your tools team's institutional interest will be in the smooth running of the tool, not in the messy changes necessary for finding bugs. Even if you're reluctant to outsource the rest of the process, administration and configuration may be more flexible if done by external players rather than an internal team with its own interests, habits, and procedures. It may also be more productive to hire an expensive consultant for a few hours, rather than a lesser-paid internal resource full-time; an external resource may be more flexible and less prone to establishing an entrenched bureaucracy.

Engineering Management

The conventional wisdom is that getting most developers to use a static analysis tool requires a high-ranking management champion to preach its benefits, ensure that the tool is used, and keep the focus on finding bugs and getting them fixed. The flip side to this is that any attempt to herd cats (or to get programmers to adhere to best practices) will cause a backlash. Your tool must withstand scrutiny from developers looking for excuses to stop using it.

Get buy-in from engineering that they will make time in the schedule to review and fix bugs found, even if they are disinclined to do so, which they will be once they see the first false positive. (Or even the first false false positive; more on this later.) Ensure that it's not the least-effective engineers whose time is allotted for reviewing and fixing static analysis bugs (more on this momentarily). You'll also need agreement from the security team that sales personnel will get access to your real source code; more on that is also to follow.

Smart Programmers Add More Value — and Subtract Less

Handling static analysis defects is not something to economize on. Writing code is hard, finding bugs in professional code should be hard, and evaluating possible mistakes in alleged bugs is even harder. Learning to evaluate static analysis defects, even in a developer's own code, requires training and supervision. It is necessary to tread delicately around the polite pretense that the code owner is an infallible authority on the behavior of that code. Misunderstandings about the actual behavior of unsigned integers and assertions are, for instance, regrettably common in my experience.

It can be tempting to delegate the triage (that is, initial screening) of static analysis defects to someone other than the code owner. This is much more hazardous than it first appears. It's quite easy for unsupervised screeners to subtract much of the value you could get from static analysis. The conventional wisdom operates here: A smart programmer adds more value than several cheap programmers, and subtracts vastly less value. If an unqualified screener triages half of the genuine defects away as false false positives, and marks half the false positives as genuine, then the engineer responsible for fixing them won't see many of the genuine defects, and may use the erroneously marked false positives as an excuse to speed-mark the rest as false positives, too.

Even more importantly, the most valuable benefit from static analysis, greater even than fixing bugs, is preventing future bugs by educating the developer about his or her mistakes. If someone else is choosing the defects, the developer never sees the mistakes and cannot learn from them.

Not Our Problem

If your software included third-party code, it's an unpleasant political reality that you may include it in your product without fixing it. Whether this is a good idea for your customers and/or your organization is outside the scope of this article; but if you're not going to fix it,

then you shouldn't spend too much time investigating static analysis defects in it. On the other hand, you should nag your suppliers to use static analysis to find and fix their bugs.

Preparing for an Embarrassment of Riches

The Embarrassment of Riches problem means that a modern commercial static analysis tool generally finds more bugs than the user has resources, or willingness, to fix. Political resistance to static analysis bugs is sometimes warranted, sometimes mere laziness, but sometimes deeper and cultural: Avoiding the kinds of bugs that static analysis finds is largely a matter of discipline, which is sometimes unpopular among programmers.

Fixing these bugs, and verifying that your organization has done so, will require adaptability and judgment. Attempts to design simple rules and metrics for this are, in my opinion, at best premature, and perhaps impossible.

Non-Goals and Metrics

Beware of two fallacies. The first is one of the standard risks of measurement: over-optimization of single factor measurement the factor you measure may be all that gets optimized, displacing efforts towards your genuine goal. (See section 26.2 of Steve McConnell's *Rapid Development* at http://is.gd/T2yHsD for more information.) The second is what I call the "Management Metrics Fallacy":

If it can't be measured, it can't be managed.
What's easy to measure is all that's important.

Stable metrics are the enemy of quality: If your tool must produce numbers that won't oscillate or upset people, then it can't change rapidly in order to catch real bugs or to stop reporting fake ones. This will bring static analysis into disrepute within your organization, and give engineers an excellent excuse for ignoring (or de-prioritizing) static analysis defects.

Conversely, it's easy to track a number that makes management look good, but doesn't get important bugs fixed; for example, the number of projects analyzed or the number of unexamined defects. And even though getting bugs fixed is the goal, simply tracking that number may still be misleading (I have been guilty of this myself). If you put too much emphasis on the number of defects fixed, the developers being measured may spend more time than is warranted on unimportant (or even obsolete) areas of the code, where static analysis defects may be more common. Some code should be excluded from analysis and ignored, rather than fixed. And, sadly, some shipping code will be excluded from being fixed.

What counts is a hypothetical, and hence impossible to measure with certainty: How many serious bugs did you prevent from reaching customers? This is related to a secondary consideration: How many bugs did you prevent from reaching manual testing where bugs start to be expensive?

False Negatives

No static analysis tool will find all bugs in any significant codebase. Part of the revolution in static analysis was giving up on even limited attempts to do so, in favor of heuristics to find actionable bugs. Definite numbers are hard to come by, but Coverity's Analysis Architect estimates that it probably finds less than 20% of bugs present (see http://is.gd/RNTYc5, registration required). It seems unlikely that any current tool can do much better except at the cost of an unwieldy

number of false positives. (Vendor claims of completeness are generally so restricted as to be impractical for normal development.) This is less of a disadvantage than it seems, since the Embarrassment of Riches problem means that a modern commercial static analysis tool generally finds more bugs than the user has time to fix. Thus, prioritization is crucial.

Living with False Positives

Conversely, no significant static analysis tool is immune to false positives; the number of these tends also to be, very roughly, 20%. This number, however, is usually eclipsed by the number of technically correct defects that are of little interest for other reasons, such as being in a part of the code that no one cares about, or being (in the opinion of the code owner) unlikely to occur in the field. (You must also be prepared for a high false false positive ratio, though this varies widely among developers.) Even with a high false positive ratio, static analysis is still vastly more efficient than other forms of bug finding, since it takes little time for a code owner to dismiss an irrelevant defect. (Triaging false positives can be more time-consuming for people not familiar with the code, however.) It is important to manage expectations, nonetheless, so that someone tasked with examining static analysis defects is not discouraged by false positives from persevering in resolving real bugs.

Ranking and Prioritization

Most static analysis tools present defects in essentially random order (for example, alphabetical by checker name or by file). This is unwise. If the first defect in a given engineer's queue is unimpressive, you may have lost him. This is particularly disappointing because there are techniques in the academic literature for ranking defects by reliability, relevance, and estimated importance. (Two papers by Ted Kremenek and Dawson Engler dig into this topic more deeply; see http://is.gd/eZqEVg and http://is.gd/shvdR8).

One common facility is ranking defects by the checker that finds them: However, an unreliable defect with little impact is less important than a definite bug with bad consequences, regardless of which checkers found them. I have no general solution to offer users of existing tools, beyond the advice to follow on experimenting, measuring, and adapting. Don't dismiss even unambitious checkers too hastily. Engler's Third Law is that no bug is too foolish to check for and this is depressingly well confirmed by experience, at least for C/C++. (Egregious bugs may be harder to find in Java.) In code under development, the simplest checks for the most painfully obvious bugs are often the most effective, such as use of a variable immediately after `free()`, etc.

Handling Static Analysis Defects

Identifying bugs and their fixes via static analysis is the easy part; the tool goes right to the heart of a defect and highlights the problem in clear, bright colors. Difficulties usually arise after the fix is identified, either because it is misunderstood (which I discuss below), or because there is resistance to making the change. Given that a tool is good enough to find real bugs, what determines whether the bugs get fixed is the sociology of the organization and its politics.

The conventional wisdom is that static analysis tools have depressingly little overlap; but missing bugs is not the end of the world. As long as the pipeline is full of genuine and significant bugs, a good static analysis tool will do a lot of good for the code and the customers. Conversely, some genuine bugs found by the tool will be ignored, either as incorrectly deemed false positives, annoyances swept under the rug, or simply not gotten to in time due to the Embarrassment of Riches problem.

Keep a close eye on the ratio of false positives; anything much above 20% indicates a configuration error, user errors, or (more rarely) a bad checker that should be disabled.

Novice static analysis users need to accept that, when they disagree with the tool, they will usually (but not always) be wrong. This is not to say that any tool is perfect; once a user understands the kinds of mistakes it makes (and doesn't make), human supervision can begin to add value. So, do not make the opposite mistake and fix all defects blindly; unnecessary code changes carry their own risks and I have seen mistakes (detected by later static analysis) in code required to silence earlier static analysis warnings. In particular, a strict policy of requiring immediate fixes for newly checked-in static analysis defects might have unintended consequences. A programmer who is in a hurry to get home after a check-in is unlikely to be in the best state of mind for analyzing and fixing static analysis defects.

A grayer area is defects where the tool is technically correct, but the developer believes that the code path is not worth worrying about. This is a judgment call, and hence hard for anyone but the code owner to second-guess, which also makes it impractical to judge statistically.

Practicalities

Most major tools present found defects in a browser. This makes the user interface of your particular browser of considerable importance and necessitates a delicate balance between habitual usability and specific features needed for examining static analysis defects.

Symbol Highlighting

Firefox's Highlight All feature is invaluable for quickly highlighting all occurrences of the currently selected text (such as a variable or function name).

If you will be examining many defects, it's worth practicing the key sequence until it's instinctive. (On Macintosh Firefox, one of the keyboard equivalents is missing, so a keyboard macro program is necessary.)

Most browser-based tools have similar functionality that's symbol-based, which has advantages, disadvantages, and limitations. Some instances of the symbol can be missed, but on the other hand, a symbol that is a superstring of the selected symbol will be properly ignored by symbol-based search, but improperly highlighted by a browser's string-based search.

Quick highlighting can reduce some types of defects from a cognitive problem to simple pattern recognition. For instance, a buffer overrun defect with a parameter array index can often be recognized merely by highlighting the index on the line where the tool reports the defect. The pattern of color between there and the function head makes it obvious where there are checks, if any, on the parameter.

Incompatibilities and Difficulties

Some tools cause difficulties with some browsers. Coverity 5.x's heavy-duty Web pages, for instance, do not work properly with some versions of Firefox and will not open separate pages for function definitions; this is not a problem with Coverity 4.x. (My workaround for the latter limitation is to take quick notes via text drag and drop, but this requires support from both the browser and your text editor.) Fortify's Web interface uses Adobe Flash, which does not support standard text selection behavior, so I recommend avoiding it in favor of their standalone application (which has text interface difficulties of its own.)

Improper configuration, in particular errors in locating and meta-compiling your source, can silently ruin your analysis while leaving the illusion of doing useful work. Build and parsing errors can make most defects false positives, or miss many of the genuine defects the

tool is capable of finding. The upside is that a few simple (though perhaps hard-to-find) fixes to the build configuration can convert a worse-than-useless analysis, consisting of mostly false positives, into something worth getting your engineers to look at. But you must set up the procedure so that those configuration fixes are made promptly: Letting a bogus build count as a success or setting up a slow bureaucracy to approve changes will bring the tool into disrepute, and give engineers an excuse to ignore even legitimate bugs found by the tool.

Integration with Defect Tracking

Most commercial tools come with a defect-tracking system, which is essential for suppressing false positives. This is essentially a bug database, but without many essential features, such as status tracking. (The first thing a quality assurance professional checks in the morning is changed bugs in the database, for instance, to see what developers and managers have been marking as unimportant.) None of the bundled databases known to me matches the features of professional bug-tracking databases; I have sometimes been reduced to archiving bug lists as CSV files, and diffing them as time goes by.

Sometimes it is useful to link an organization's real bug-tracking database with a static analysis defect database. I don't advise importing defects automatically. In my opinion, a real bug database should be reserved for issues that a human has judged to be genuine. One strategy I have found useful is to file one bug report per component into your existing bug database, with a link in each bug report to a live query with suspected defects in the component in the current build, with separate counts for high- and low-priority checkers.

Keep Measuring and Adapting

As I have mentioned previously, metrics — applied blindly — make a very bad master; but applied wisely, they can be a helpful servant. Track the numbers of important (and unimportant) defects found, and the checkers that found them. Even good checkers might not find the sort of defects that your organization is, realistically, going to fix. Track the defects that actually are fixed, preferably by keeping an eye on source code management check-ins. (Pretend not to see check-in comments that minimize the importance of the defect fixed; developers are often reluctant to admit that an automated tool found significant bugs in their code. These imperfect developers are your best customers, not your enemy.) Given finite resources, it is crucial to prioritize defects; sadly, this sometimes means ignoring some components or disabling some checkers.

Also keep an eye on static analysis defects that are not fixed. Sometimes the problem will be with the tool, and some checkers are not worth the resources they consume. More often, the problem will be with the reaction to the tool, and sometimes education is necessary. That is perhaps the most valuable result of static analysis: In the long run, making developers think about their code and learn about what it actually does, is even more important than fixing the current version of their code.

— Flash Sheridan was a static source code analysis consultant at Bell Labs when writing this article, and has also done static analysis consulting at Qualcomm and Palm. He is currently testing a combined compiler and static analyzer for a major technology company.

Comment

Previous Next

IN THIS ISSUE

Shrinking Role of QA >>
Static Analysis >>
Building QuickBooks >>
Unethical Behavior >>
Letters >>
Links >>
Table of Contents >>

Building QuickBooks: How Intuit Manages 10 Million Lines of Code

Continuous integration and delivery based on automated software configuration management are the keys for developing a major retail product.

By J.D. Hildebrand and Andrew Binstock

Intuit Inc. was founded in 1983 as the publisher of Quicken, an MS-DOS financial-management system for individuals. Quicken's success led to a host of follow-up products, including QuickBooks, which was launched in 1992 and quickly became the industry's top small-business accounting solution. Intuit describes QuickBooks as the backbone of its "small business ecosystem," a market that accounts for a major share of the company's $3.9 billion in annual revenues. It is easily the best-selling retail software for small-business accounting worldwide.

Dr. Dobb's spoke with Jon Burt, group manager in the operations department for Intuit's Financial Management Solutions division. FMS supports the delivery of QuickBooks for Windows, QuickBooks for Macintosh, and QuickBooks Online. Burt manages software configuration management for these three products, plus the operations aspect of QuickBooks Online. He manages a staff that is split between Intuit's Mountain View, California, headquarters and Bangalore, India.

In addition to supporting multiple platforms, Intuit offers different feature sets at different price points. QuickBooks for Windows, for example, is available in Pro, Premier, and Enterprise editions. Specialized editions are available for accounting, contracting, manufacturing and wholesale, nonprofits, professional services, and retail businesses. Additionally, each product may be purchased on CD or via subscription. Subscribers get new features as they are developed, while CD users don't get the new features until they upgrade to a new release of the software.

As if that weren't enough, the software is also offered in localized versions for the U.S., Canada, and the United Kingdom.

A Single Codebase

"You wouldn't guess it," Burt says, "but on each platform, all those products come out of a single codebase. The Windows version is about 80,000 source files, 10+ million lines of C++ code plus a little C# for the .NET parts. Plus help files, tax tables, files defining local accounting rules, tax and other government reporting forms, upgrade offers…a lot of files. Every customer gets the full version. Specific feature sets are turned on and off with the license key."

The license-key approach means Intuit can upgrade customers or license specialized versions of the software with a simple online transaction. It also has big implications for SCM. "The benefit is that we're always extending and refining a single codebase," Burt notes. "We don't have to worry so much about maintaining compatibility among multiple versions of the product. The downside is that a full build rolls out many, many versions of the product. It's a huge build."

As the manager with primary responsibility for build management, Burt has supervised the construction of automated systems that perform continuous builds and continuous integration. The systems incorporate tools for testing, version control, and scheduling.

The decision to use a single codebase for each platform was a strategic one for Intuit. "It's a lot of work," Burt acknowledges. "It means that we needed to put coverage tools in place so we could keep complexity under control as we added features to the product. We needed to perform frequent builds, so we needed to invest in systems and tools to turn the codebase over as quickly as possible."

Managing Multiple Builds

In addition to different combinations of locales, versions, and vertical-market features, Burt specifies that each build includes a debug version and a release version for each locale, a version built for analysis with Coverity, and a build that incorporates HP's Fortify code-analysis tools for security scanning. Plus, Burt says, there are separate builds for the main trunk of the source-code tree and for branches associated with different projects — about four branches per year, on average, each with a full set of builds. That's a lot of specialized builds, and it requires a lot of resources.

"It's not reasonable to ask everyone to do their local builds in both debug and release modes, so we let the continuous-integration system do it"

"The Coverity build alone is a 20-hour run," Burt says. "Coverity does a tremendous amount of analysis, so we threw major resources at it."

Intuit uses Jenkins, the open source continuous-integration tool derived from Oracle's Hudson, to coordinate builds. "We use Jenkins essentially as a job system," Burt says. "It's a scheduling app. A build is triggered when code is checked-in, or if there's a series of check-ins within a 15-minute window. Every 15 minutes, we ask the version-control system if anything has happened along each branch. If the answer is 'Yes,' then we have a set of continuous-integration systems that will go off and do a build."

The continuous builds are deployed to specially prepared virtual PCs (implemented with VMware) where they undergo minimal testing. "We do just enough testing to ensure you can open the application, create an invoice, and close the application," Burt says. "No one does anything with these builds — they're disposable. They're basically to ensure that you didn't forget a semicolon that could break tomorrow's build. They're compiled with zero warnings. If you've been doing your local compiles in debug mode, you may miss or ignore a warning — that's what trips people up. It's not reasonable to ask everyone to do their local builds in both debug and release modes, so we let the continuous-integration system do it. If the test exposes an error, everyone who had a check-in since the last clean run gets an e-mail, and they sort it out. You also get an e-mail when the tests succeed, so you have some assurance that you're not going to have a DOA build in the morning."

This minimal testing has a major impact on workflow, Burt says. "It's really the problem I came to Intuit to solve. Every morning we'd come in to a broken build. We'd spend from 9 a.m. to 1 p.m. unblocking QA so they could test. We were always behind."

Burt relies on Perforce Software's Perforce revision-control system, with good results: "Perforce works well for us. It's rock-solid, it has the features we need, and it's pretty fast. The main server is here in Mountain View, and we have proxies at the remote sites."

In addition to the check-in-by-check-in builds, Burt's team schedules nightly builds of the entire codebase, plus additional runs. There's a Fortify build every night, for instance. A weekly schedule governs which branches get Coverity builds on which nights.

Five years ago, builds were performed on a cluster consisting of 70 rack-mounted 1U nodes. Last year, Burt says, Intuit replaced that hardware with four Dell 2950 2U nodes. "We've got twice the performance

now, after moving from two full racks to just four 2U units," he says. To take full advantage of the hardware, Intuit uses Electric Cloud's ElectricAccelerator to distribute builds across multiple virtual machines. ElectricAccelerator lets Burt's team launch compiles in parallel across the cluster. "We can build the entire product — 8 to 10 million lines of code — in 45 minutes," Burt says. "With ElectricAccelerator, we can do 7 builds in parallel and get 45- to 60-minute turnaround time. Of course, it's a diminishing-returns dynamic. The happy point is about 10 virtual boxes. If you do more, you hit the asymptote — it's not worth it — 10 seems the sweet spot for this codebase."

"It doesn't matter what tools you use, if your team doesn't support it, if they're focused on adding features, you're wasting your time"

Burt says Intuit has been shipping product built in parallel for the past four years. "I'm not sure management knows we're not building sequentially," he says with a laugh. "But there's no reason for them to worry. It's solid. If we have a broken build, it's never because of Accelerator."

Automating QA

The end result of the build is a set of CD-ROM packages and downloadable packages for the Web. At that point, a custom-built app called AutoLab takes control. AutoLab sets up virtual machines and installs the various releases. First it preps the machines across multiple versions of Windows. Then it runs the installation, opens a company, and closes a company. That confirms that the build resulted in code that installs and runs. That's the basic acceptance test.

Once the code is loaded, the team runs feature scenarios and regression tests. For example, they may exercise all the options of the invoice system. This is all done with Micro Focus SilkTest. Intuit relies upon an outside team to maintain these tests. "You don't put your top talent on maintaining test scripts," Burt notes, "but you must have someone who is responsible for maintaining them. That's a natural thing to outsource."

The results of functional testing are delivered to developers every morning. Assuming the code works properly, developers sync their systems to the new codebase.

Unit testing isn't a big part of QuickBooks for Windows — the bulk of the codebase was written before unit testing was acknowledged as a best practice. QuickBooks Online, which is written in Java, incorporates more unit tests. In fact, Burt says, unit tests dominate the build cycle.

Burt is devoting significant resources to code quality this year. "We've sent multiple teams of five developers into the Coverity logs," he says. "They track down the warnings and fix hundreds and hundreds of minor coding issues. We're cleaning up little things that have been there forever. The goal is to ship a release with zero defects." This isn't just a matter of tools, Burt acknowledges. Team and management support are essential because they must remain committed to QA every day. "It doesn't matter what tools you use," Burt says. "If your team doesn't support it, if they're focused on adding features, you're wasting your time. If you don't want to use a tool like Coverity to track down defects,

that's fine — don't use it. But if you say you want it, then you've got to commit. The payoff in code quality is tremendous."

Developers can be resistant to detailed analysis of shipping code at first, Burt says. But Coverity can surprise even senior architects sometimes. "They'll say, 'That's not a bug,'" Burt says. "And then they're like, 'Oh…I didn't think about that path.'"

Burt has worked with the development group to ensure that neither stigma nor penalties apply to developers who find code bugs during detailed analysis. "No one tracks the number of bugs attributed to you," he says. "That's something Intuit has never done. You break the build enough times — well, that gets people's attention. But not bugs. We're focused on eliminating defects, not apportioning blame."

Bug-tracking is handled with Serena Software's TeamTrack. "It's really overkill," Burt observes. "Most of Intuit's other teams have moved to lighter-weight tools, and we may do the same at some point."

Lessons Learned

Dealing with such a large codebase has been an education, Burt acknowledges. "The size of the codebase has spawned a' 10 Commandments of Things Thou Shalt Do Before Checking in Code,'" he says. "It's common sense stuff, but it's important. You're going to do a local build. You're going to run positive and negative tests. You're going to find someone to buddy-check it. If it's a bug fix, the comments are going to refer to the bug. Stuff like that."

Branching, Burt says, is a particular danger. "We have gone branch-crazy in the past," he says. "Everybody wanted his own branch for specific projects. That's great until you want to integrate the various branches back into the main system. If you wait too long and try to roll up all at once, you are in for an ugly month." In the past, Burt says, when a build took 4 hours instead of 45 minutes, it took a half-day to see if a branch broke the code. ElectricAccelerator was a game-changer in that regard. But still, Burt says, more than 5 or 6 branches is probably too many.

One surprising lesson is that small teams work, even for very large codebases — especially, Burt says, in sustaining an entrepreneurial, creative culture. "We use teams of five and six, working together," Burt says. "Last year, we created a lot of small projects. At the end of the month, we delivered them to customers and asked, 'What do you think about this?' Sometimes we threw the experimental features away, and sometimes we found they had real value. Just because you have a 10-million-line codebase, that doesn't mean you can't be nimble and quick. You can use Scrum and other Agile techniques with a large codebase — you just need to adjust your thinking a bit. Create a branch, have a small team work on it, see if the idea has merit. There's no substitute for experimenting with the codebase. I think that's changed the way people think about their work. You can have a code jam. And you can, in one day, accomplish something significant in the QuickBooks codebase. And if it only takes 45 minutes to build — why not?"

Most of all, the key to managing a large project was automation. "We automate everything that can be automated," says Burt. "The tools make a huge difference. We maintain all the different versions of QuickBooks, on all our supported platforms, with about 60 code-writing developers. We couldn't do that without automation."

—J.D. Hildebrand is a* Dr. Dobb's *associate editor and Andrew Binstock is editor in chief.

Comment

From the Vault

Survey Shows Unethical Behavior Rampant Inside IT Development Teams

A controversial survey exposed ethically challenged and dysfunctional behavior in the industry.

— *DDJ*

By Scott W. Ambler

One of the challenges faced by IT professionals around the world is that we're often not trusted by the people we're meant to serve. There are several reasons why we're not trusted, but the primary one is that we're often perceived as liars — and rightfully so considering our software development track record. I explored this problem two years ago in my July 2009 Agile Update: "Lies, Great Lies, and Software Development Plans" (http://is.gd/see9q0). The article summarized results from a survey we ran earlier that month that explored how project teams got themselves out of trouble. We discovered that many teams exhibited ethically questionable behavior. I received interesting feedback about this newsletter, particularly because several of the ethically questionable strategies are promoted as best practices within the project management community. So, I decided to repeat this effort with a new survey. I explain the results herewith.

This survey ran in late December 2010 and early January 2011 and had 235 respondents. I wanted to get the most timely information possible, so the survey asked people to describe their most recently completed project. The previous survey had asked about all projects within their organization, which in turn inflated the response rate. The new survey explored scope, schedule, cost, and process-related issues. Respondents were presented with a collection of practices and strategies

that their teams might have used at some point in their project to get back on track. Some of the options were good ideas, some questionable, and a few downright dishonest.

For example, when it comes to schedule, a good idea would be to negotiate a variable schedule at the beginning of the project, a bad idea would be to commit to a "firm" end date and then negotiate a change to it late in the lifecycle (in some cases, this could border on extortion), and a downright dishonest strategy would be to cook the books at the end of the project to make it look like your original schedule predicted the actual end date even when it hadn't.

"When it comes to budgetary practices, 15% of respondents indicated that their team had motivated changes to the budget in an ethically questionable manner, even though a flexible budget had not been negotiated for the project"

After exploring the behaviors and survey results, I discuss preferred alternatives when digging out of a problem project.

Ethically Challenged Behavior

The ideas in this section are straightforward, but for many readers they may be the exact opposite of what you've been taught over the years. The problem is that the survey explored many strategies that are touted as project management best practices. And to be fair, when they're considered in exclusion, they arguably are. The problem is that these practices aren't exclusive, they're part of an overall toolkit that project teams pick and choose from appropriately.

For example, there is a low probability that any given team will try to extort more money out of their stakeholders near what should be the end of their project, so low that we're tempted to accept this indiscretion as "business as usual" when it does occur. The issue is that these probabilities add up, as you'll soon see, until there is a reasonably high probability that the IT project team will purposely deceive stakeholders about one or more of the cost, schedule, scope, or quality of their work. When we continue to do this from project to project to project, it constitutes unethical behavior. We set expectations at the beginning of the project, often because we're asked to by our stakeholders (a flimsy excuse at best), but know full well that we're likely to do something different when the going gets tough. If you were on the receiving end of this behavior, how would you feel?

When it comes to budgetary practices, 15% of respondents indicated that their team had motivated changes to the budget in an ethically questionable manner, even though a flexible budget had not been negotiated for the project. They had done one or more of the following: padding the initial budget (effectively deceiving stakeholders as to the actual expected costs); requesting extra funds at the end of the project even though they'd committed to a fixed budget; or reworked the initial estimate to align with the actual results at the end (effectively "cooking the books"). Analyzing the results by development paradigm, we find that 9% of agile teams, 20% of traditional teams, 22% of iterative teams, and 12% of ad hoc teams followed these questionable strategies. I'm not completely sure why there are such differences.

Granted, agile teams are more likely to initially negotiate a variable budget than traditional teams because they recognize that the requirements will evolve over time, but the same should be said of iterative teams as well. Expectations for ad hoc teams are lower, so it's less likely that they'd be asked to provide an initial estimate anyways — or, if they were asked to commit to a budget it might motivate them to move away from ad hoc to begin with. The traditional paradigm assumes that it should be possible to define the budget up front, but in practice this does not work out for a significant percentage of the teams.

A similar rate of questionable behavior occurs when it comes to requirements practices, with 15% of respondents indicating that they descoped at the end of the project even though flexible scope wasn't initially negotiated. By paradigm, the survey found that 17% of agile teams, 11% of traditional teams, 17% of iterative teams, and 15% of ad hoc teams took this strategy. The fact that the results were so close and so low is an indication to me that teams are consistent in their desire to deliver what their stakeholders want regardless of paradigm. Scope management was the only instance in which traditional teams acted in a more ethical manner than agile or iterative teams. This is likely a reflection of the underlying rigidity of the traditional approach to requirements.

The rates of ethically dubious schedule practices were significantly worse, with 31% of respondents indicating that they had done at least one of padding the schedule, extending the schedule at the end, or reworking the original schedule at the end to reflect the actual schedule even though a flexible schedule was not initially negotiated. By paradigm, 21% of agile teams, 42% of traditional teams,

35% of iterative teams, and 32% of ad hoc teams exhibited questionable behavior. Once again, agile teams seemed to work in a more ethical manner than teams following other paradigms, although to be fair agile teams are more likely to negotiate a flexible schedule up front and therefore are less likely to run into trouble.

"My experience is that many of these improper IT practices are adopted to protect us from bad business practices, such as the desire of business leaders to have "accurate" cost and schedule estimates early in an IT project"

When it comes to the willingness of teams to compromise quality the good news is that the numbers are fairly low, with 15% of respondents indicating that their teams had done so in order to get themselves "out of trouble." By paradigm, 4% of agile teams, 16% of traditional teams, 17% of iterative, and 24% of ad hoc compromised quality. When it comes to agile I suspect that this is the result of the greater focus on quality that we see in that community, although I'm surprised that iterative and traditional had similar rates because iterative teams tend to test earlier in the lifecycle.

To summarize, based on our reported behaviors, there's a good chance that our stakeholders will perceive that we're lying to them at the start of a project when they're making critical funding decisions. There is a 15% chance that we'll deceive them about the cost of a project, 15% chance of deceiving them about what we're going to deliver, a 31% chance that we're lying about the schedule, and a 15% chance that we'll promise greater quality than we'll actually deliver. The bad news is that there's a 44% chance that we'll do at least one of these things (by paradigm, 32% for agile teams, 53% for traditional teams, 48% for iterative teams, and 50% for ad hoc teams). Looking at it from the point of view of our stakeholders, there's a 44% chance that a given project team will deceive them about one or more important success factors. Doing some simple combinatorics analysis, if your organization is currently running five projects there's a 95% that at least one of the teams has committed an ethically questionable act. Or, if you've been on five projects in your career, there's a 95% chance that you've been involved with such a team. Is it any wonder that business stakeholders consider IT untrustworthy?

Why are these things happening? My experience is that many of these improper IT practices are adopted to protect us from bad business practices, such as the desire of business leaders to have "accurate" cost and schedule estimates early in an IT project. For example, the survey found that we're being asked to give initial estimates on an average range of +/- 12%, which seems reasonable from a business point of view. Yet research by Barry Boehm, based on data from thousands of projects, shows that initial estimates for IT projects should be on a range of -25% to +75%. I believe that this discrepancy between what we're being asked to do and what is actually realistic to do motivates poor practices on the part of IT professionals.

Dysfunctional Behavior

The survey asked about two requirements practices that I believe to be dysfunctional although not ethically improper. Actually, it really depends on the motivation behind following the practices. The first practice is a strict approach to change management that makes it difficult for stakeholders to change their requirements. This approach is employed by 29% of project teams. Strict change management processes are, in effect, change prevention processes: While they enable a team to deliver to the specification, they ensure that what you're delivering isn't what your stakeholders actually want. This is clearly not good for your stakeholders and isn't a very good career move on your part either.

"I believe that we can do better; and that if we're ever to be considered true professionals, we need to change our behavior"

The second dysfunctional practice is writing a detailed requirement specification at the beginning of the project, followed by 27% of respondents. This is something called "big requirements up front (BRUF)" within the agile community and it is problematic because it results in a significant level of waste on IT projects.

Interestingly, the number of respondents doing at least one of these practices was 44% on average — by paradigm Agile (36%), iterative (54%), traditional (53%), and ad hoc (29%). Not surprising that the ad hoc teams hadn't adopted more formalized strategies for requirements management.

We Have Better Options

The good news is that all IT projects aren't the wretched hives of villainy that I've made them out to be. Many teams are in fact following some respectable practices to help keep them on the straight and narrow. These practices include:

- Implementing functionality in priority order (followed by 64% of teams)
- Evolving the schedule throughout the project (51%)
- Evolving requirements throughout the project (50%)
- Taking a flexible approach to scope from the start (38%)
- Regularly produce shippable solutions (35%)
- Adopting a flexible approach to the schedule from the start (30%)
- Having a flexible budget from the start (21%)
- Presenting stakeholders with a ranged schedule (18%)
- Presenting stakeholders with a ranged estimate (18%)
- Updating the budget throughout the project (17%)
- Taking a stage-gate approach to funding (8%)

Getting Off This Treadmill

I believe that we can do better; and that if we're ever to be considered true professionals, we need to change our behavior. This is a choice that we need to make as individuals first, as project teams second, and finally as organizations. To do this, we will need to:

- Recognize that we have a problem. I hope that this article has helped to illuminate our bad behavior.
- Choose to work in a trustworthy manner. We need a trust-based relationship with the business, and we clearly don't have that.

We will never be considered true partners with the rest of our organizations if we're not trustworthy.

- Educate the business. I believe many of the bad behaviors stem from ill-informed requests from business stakeholders. They often don't understand the implications of what they're asking for, and that's because we haven't taken the time to educate them as to their choices and the trade-offs associated with them. We need to deal with the root cause of the problem.
- Recognize that we have choices. There are ethical strategies to allow scope, cost, schedule, and quality to vary which we could choose to follow. As this survey shows, many teams choose to adopt them.
- Recognize this will be hard. It's really easy to point out a problem, it's a lot harder to actually deal with it. Many of these ethically challenged practices are touted as project management best practices, indicating that we have a significant cultural challenge to overcome.
- Recognize this is a risky career decision. For many IT departments adopting ethical project management policies will be a significant shift in strategy. The people and teams who are at the forefront of this shift risk being the messengers of bad news who end up being punished for doing so. Make the changes that you are comfortable with making.

I believe that we've gained some significant insights from the analysis of the survey results. I run these surveys to discover what is actually going on out there and the results are always interesting. Whenever you see a request from me to fill out a survey, I hope you will take a few minutes of your valuable time to share your experiences with the rest of the community.

Further Reference

Results from the December 2010 State of the IT Union Survey are posted at http://www.ambysoft.com/surveys/stateOfITUnion201012.html.

The July 2009 State of the IT Union Survey (http://is.gd/SvTd70) explored IT governance and project management issues. It found that many project teams exhibited ethically questionable behavior, behavior which was examined again in the December 2010 survey.

My July 2009 *Dr. Dobb's* Update: "Lies, Great Lies, and Software Development Plans" (http://is.gd/see9q0) discusses results from the July 2009 State of the IT Union survey.

The change prevention process anti-pattern is described at http://www.ambysoft.com/essays/changePrevention.html.

The perils of big requirements up front (BRUF) are explored in detail at http://www.agilemodeling.com/essays/examiningBRUF.htm.

My paper "Scaling Agile: An Executive Guide" [PDF] explores the issues surrounding agility at scale and can be downloaded at http://is.gd/wnkcbN.

The Surveys Exploring the Current State of Information Technology Practices page at http://www.ambysoft.com/surveys/ links to the results of all the *DDJ* surveys that I've run over the years.

—Scott Ambler is a* Dr. Dobb's *contributing editor.

Comment

This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

GAMIFICATION: THE ART OF ATTRACTING AND KEEPING USERS

Game designs and artifacts involve users more deeply in communities. Soon, prizes, trophies, and badges will be a standard part of the user experience of Web apps..

http://is.gd/L4X6Xh

IMPLEMENTING PARTITION WITH PREFIX SCAN

In this post, Clay Breshears presents one implementation of an algorithm using OpenMP for the first level of task parallelism and Intel Threading Building Blocks (TBB) to produce manycore parallel operations.

http://is.gd/sMagav

THE DESIGN PRINCIPLES OF METRO APPS

A Metro-style app uniquely leverages the features of Microsoft Windows 8. It is touch-enabled (but not limited to touch), and it displays in a single, plain, borderless window devoid of any adornment such as resizable edges, caption bar, or icons.

http://is.gd/gOxcpl

WRITING A BI-ENDIAN COMPILER

The techniques for writing a Bi-Endian compiler for porting byte order-sensitive applications. Plus, a look at its application and benchmarks for its performance.

http://is.gd/dwT74H

C++ MYSTERIES

What does C++ add to your code at runtime? One thing to keep in mind is that there is very little difference between a class and a structure. In fact, in C++, they are identical except for the default visibility of members (structure members are public by default and class members are private).

http://is.gd/RypA6n

GOOGLE'S REDEFINITION OF THE BROWSER AS PLATFORM

The company's vision of the browser as a universal platform has driven remarkable technology innovation and has pushed Chrome into first place in the new browser wars.

http://is.gd/Tvw8wb

IN THIS ISSUE

Dr.Dobb's

Dr.Dobb's Business Contacts

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Salvatore Silletti
(212) 600-3327, ssilletti@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Account Executive, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Marketing Director, Angela Lee-Moll
(516) 562-5803, aleemoll@techweb.com

Marketing Manager, Monique Kakegawa
(949) 223-3609, mluttrell@techweb.com

Director, Client Marketing, Michelle Somers
(516) 562-7928, msomers@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Steven Sorhaindo
(212) 600-3092, ssorhaindo@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Account Manager, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000