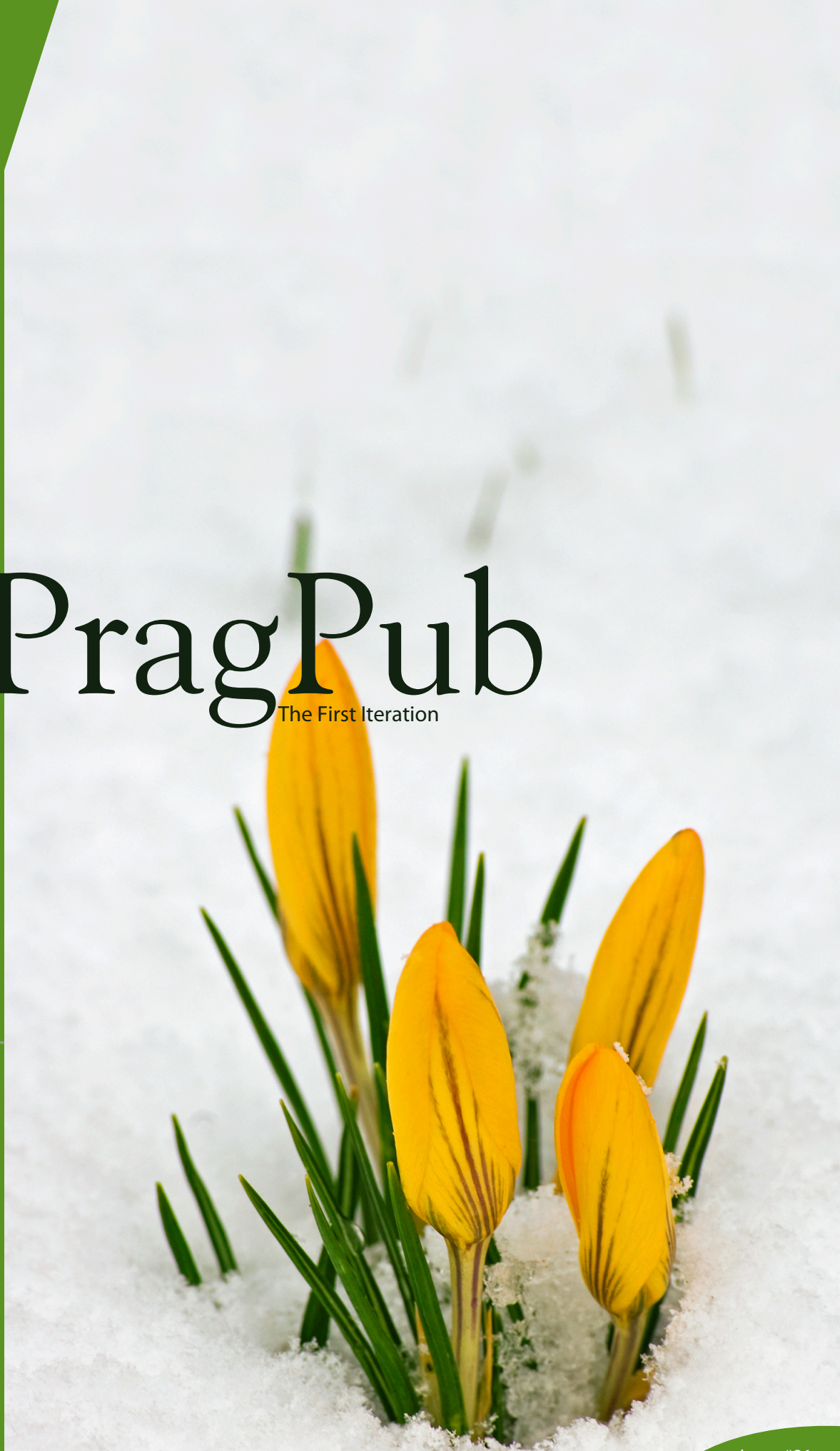
The Pragmatic Bookshelf

PragPub

The First Iteration



IN THIS ISSUE

Testing…

In this issue we tackle testing, Turing, tweets, and trust—not to mention punk programmers, volatile software, and train wrecks.

Issue #21
March 2011

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

Punk Rock Languages

by Michael Swaine

This month Chris Adamson delivers a sprited polemic on what he calls punk rock languages. You may agree or disagree with his arguments, but you can't deny that they cut to the heart of how we do our work as programmers. If you find the subject interesting, you can follow further punk rock programming thoughts at @punkrockcode [U1].

Like most software disciplines, testing gets tricky at the boundaries. It's great when the code is all under your control, but in the modern programming environment of Web services and the cloud, you have to deal with some extra issues. In this issue, Noel Rappin takes on testing for Web services and Adam Goucher looks at what it takes to test for the cloud.

Tim Ottinger and Jeff Langr are back with another in their series on big ideas in software, this time looking at software volatility. Theorizing that the more often a file changes, the more likely it is to result in a code defect, they consider how to reduce and deal with software volatility.

Jonathan Rasmusson's "Way of the Agile Warrior" deals with a fundamental issue this month: trust, how to earn it, and the freedom and power it buys you.

Dan Wohlbruck's back with another history article this month, the Quiz returns with a tricky puzzle about relationships, and John Shade ponders train wrecks and burning platforms.

le:

Software Volatility

Do Most Changes to Your Code Base Occur in Just a Few Files?

by Tim Ottinger, Jeff Langr

In the fourth in their four-part series of Big Ideas in software development, Tim and Jeff tackle software volatility.

What is Volatility?

As the final of our "big ideas in software development," we look at volatility. We provide a very simple definition of volatility for purposes of this discussion: Volatility is a measure of how often a file actually changes—never mind the extent or reason for the change. Using this definition, we theorize that volatility directly correlates to the risk of making a change: the more volatile a file is, the more likely it is that changes to that file will result in a code defect.

Measuring Volatility

Michael Feathers recently posted a graph [U1] showing that commits across a code base are not evenly distributed. Some files change frequently, even continually, while the majority seem never to be touched at all. His blog report of these findings was widely tweeted and referenced.

Noticing the same trend, Tim presented the idea of a Heat Map [U2] last year, in a blog post in which he counted Jira tickets associated with source files in the source control system for a sizable production web application. Some changes were due to feature enhancements, whereas others were due to bug fixes and others were merely side-effects of a refactoring (usually Rename or Move Method).

In Tim's analysis, a small number of files were shown to be involved in the vast majority of changes made in the system. When the five most volatile files were identified, they were shown to be poorly covered with tests, if at all, and less than optimally structured. (Why does this not surprise us?) Developers were hesitant to touch these files because of their high implementation complexity, yet they were still visited frequently.

Michael Feathers suggested that counting commits was a reasonable approximation for the volatility of a code module, and we agree. Here are a few projects you can check for yourselves:

1. In the *Gwibber* project, the most changed files are client with 302, gwui with 149, and dispatcher with 95 commits. All 247 other files have less than 70 updates and the vast majority of them have less than 30.
2. In the *Mongo* project, most files have fewer than 100 modifications, but the most-modified file has 663 (caveat: an Scons file). The top nine have from over 300 to over 500 changes. This is another data point suggesting that changes clump, rather than spreading over the code base.

3. Source files in the *rails* project tend to have fewer than 150 changes per file, the vast majority having fewer than 20 changes. However, the top five have 953, 568, 481, 309, and 307 changes respectively.

Files can change frequently because:

1. they are "god classes" (poor cohesion),
2. they have poor dependency structure (poor coupling),
3. they depend on many implementation details (poor abstraction), or
4. the code is poorly written and has many flaws.

Sometimes all of the above are true!

Circumstances do exist where heavy change is *not* due to poor design—files can contain key abstractions in the solution domain or the problem domain. New features in a banking system might touch upon Account or User. In a retail app, it might be Sale or Receipt or Product.

No matter the reason, we can confirm that certain source files change much more often than others, and then we can form strategies to deal constructively with code that shows high volatility.

Risk of Breakage

Programming is more an art of continual tailoring than invention. Systems gather changes over time: functional alterations, adjustments to user experience, performance and scalability changes, upgrades to the supporting platform or platforms, integration with other systems, and adjustments or corrections to internals (both functional and structural). It has been said that a system spends 80% of its life in maintenance. A successful system should spend far more years in extension and improvement than it did in original authoring.

A sad fact of software development is that every change to existing code has a non-zero chance of causing breakage. Some changes are simple errors in function name or type, some are simple miscalculations, some are naive changes that damage scalability or performance, and some are subtle effects that only show up when seemingly unrelated parts of the code base fail. Many defects are coded due to a simple misunderstanding of how the code currently operates.

In our previous article on coupling, we mentioned that errors tend to fan out along dependency lines, including implicit dependencies. In our articles on cohesion and abstraction, we mentioned how duplication makes maintenance a "sometime thing," since a programmer is unlikely to find all the occurrences of a duplicated passage of code and correct them all. A bug may return several times—there exists potentially one "emergency bug fix" for every point of duplication in the code base.

In general, the sooner a program error is detected, the less damage it does to the project's schedule and to the product's reputation. If the programmer realizes an error while typing, he may backspace and rewrite the code immediately with no ill effect at all. If the error escapes the programmer, it may be spotted by another programmer in the same development cycle or by a QA team member.

An otherwise undetected error escapes "into the wild," the real world where customers call technical support and programmers are pulled from feature work to diagnose and fix the code. The defect may bring servers to their knees, denying users access. It may cause data to be corrupt, requiring painstaking data repair. It may have consequences well beyond those reported by the users. Of course, the bug could also be cosmetic, or a trivial problem with an easy work-around.

Simple odds suggest that the likelihood of breakage is greater in more volatile modules. We also have observed that messy, ill-structured, badly written modules are easy to misunderstand and hard to repair. Where there is duplication, a lack of abstraction, and needless couplings, the breakage will be most likely to fan out across the code base.

To re-emphasize: *Where we have messy, volatile code we are at the greatest risk of failure*. What are we going to do about it?

The Review Alternative

"Given enough eyeballs," says Linus' Law [U3], "all bugs are shallow." Various forms of review—inspection, code walk-through, hierarchical review, and pair programming—exist to provide structure for the eyeballs.

Many companies and open source projects use *hierarchical review*. Someone submits a patch, which is reviewed at the first tier. If the patch has no obvious defect it is passed up to the next layer, otherwise it is returned or rejected. This is an effective strategy, but many companies do not or cannot adopt the strategy because it requires layers of skilled programmers to do little but monitor changes. It can be hard to keep up with those writing the code. The reviewer/acceptors need to be among the most skilled developers and those most familiar with the code base, and while they are spending time reviewing, their skills are *not* being put to use producing product.

Other companies, especially those employing agile techniques, perform constant review via pair programming. They switch partners frequently, always having two heads involved in the decision-making, and having multiple sets of eyes on the code. All of the programmers are actively involved in code-writing most of the time.

Our most volatile code demands the most eyeballs. *Even if your team has reviews for no other code and won't consider pair programming, consider either when entering into its risky, volatile areas*.

The Testing Alternative

Breakage often occurs somewhere else in the code base, far from a change. "Never thought *that* would break!"

It is foolhardy to release a product on faith alone. Most companies supplement faith with a QA department full of people knowledgeable about the product and skilled at operating it. They may have little knowledge of the code that comprises the product, but are brilliant at sniffing out defects and their causes. These QA folks fulfill the innately human needs in testing—testing scenarios that require reasoning about the software, predicting usage errors, noting

inconsistencies in data handling or presentation, and so on. Such exploratory testing is labor-intensive but invaluable.

On the other hand, often a testing department involves a small army of testers who run explicit test scripts more or less by rote. They repeat the same steps month after month and release after release. Sometimes they find errors, but ideally and usually they run all of their tests without seeing any errors. Their job is to provide assurance that important procedures still work just as they always did. Maintaining and running manual regression scripts is expensive and slow, but it is the last chance to spot defects before releasing code to customers.

If an error was detected immediately after it was created, it would be corrected without delay. It makes sense that developers test as they work, but if a developer manually ran all of the test scripts after every line of code, productivity would be dismal. We might see a feature developed in the course of a decade.

An undetected error in a very volatile source file might have many layers of additional change heaped on it by the time it reaches release, so early detection is key. Errors indirectly related to a change in volatile code must have an automated test produced to reproduce it.

In order to develop code quickly and well, our most volatile code requires the most extensive automated testing. The tests need to run frequently and quickly so that developers can run them many times per hour. This makes solving the problem easier, but also protects against future breakage being undetected or detected late.

Agile teams use extensive testing at the requirement level (acceptance tests), at the level of individual effects of functions or classes (unit testing), and often at the UI level as well (though UI tests tend to be too slow to be run by human beings and are offloaded to a continual testing environment). Many agile teams tackle test-first programming (*aka* test-driven development, or TDD) as well, to both build up a significant body of tests and guide an improved design.

On the surface, it may appear that the investment in automating such test scripts is too high, since they seem to report only rarely on defects in the system. In reality, the tests *prevent* gobs of bugs—since these tests are written first, they gate the defects from ever making their way past the programmer and into the code base (in the case of TDD) or past the QA team and into the release.

Automated tests protect you against the exploding minefield that is volatile code.

The Wrap-up

This trip through Cohesion, Coupling, Abstraction, and Volatility—the four most important ideas in software development—brings us to the premise of the agile development process. Software development should be fluid and productive, but problems that develop in the code base can slow releases and frustrate the authoring of new features. For a project to be successful, it must have constant attention to quality, especially in the code that is touched most often.

Hardcore agile teams use constant testing, constant teamwork, continual integration, and constant refactoring to manage the code base. The testing practices actively shorten the period between defect creation and detection. The refactoring practices improve cohesion, reduce coupling, and add abstraction in order to keep the code workable. Keeping the code workable leads to more productive teams and higher quality month over month.

Even if your team does not wish to identify as an agile team, we suggest that it might be prudent to at least adopt these practices when it comes to the most volatile (and therefore most risky) code in your code base.

If you are working in an agile team, the proof of your growing agility is in examining the quality and test coverage of your most volatile code. If it is being touched frequently, it should be continually improving in structure, readability, and code coverage. If it is not improving, then you have some technical practices to work on, and some debt to pay down.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U4] with Tim, he's written another couple books, *Agile Java* and *Essential Java Style*, contributed to Uncle Bob's *Clean Code*, and written over 90 articles on software development. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

About Tim

Tim Ottinger is the other author of Agile in a Flash [U5], another contributor to *Clean Code*, a 30-year (plus) software developer, agile coach, trainer, consultant, incessant blogger, and incorrigible punster. He writes code. He likes it.

Send the authors your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

[U1] http://michaelfeathers.typepad.com/michael_feathers_blog/2011/01/measuring-the-closure-of-code.html

[U2] http://agileotter.blogspot.com/2010/10/heatmap-new-hotness.html

[U3] http://en.wikipedia.org/wiki/Linus%27_Law

[U4] http://www.pragprog.com/refer/pragpub21/titles/olag/Agile-in-a-flash

[U5] http://www.pragprog.com/refer/pragpub21/titles/olag/Agile-in-a-flash

[U6] mailto:michael@pragprog.com?subject=Agile-cards

[U7] http://forums.pragprog.com/forums/134