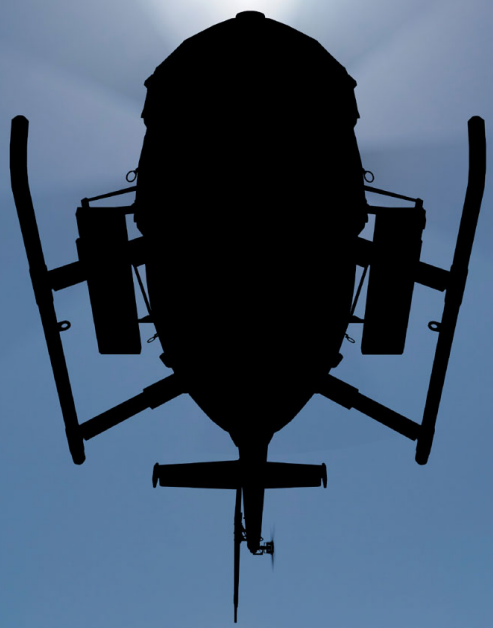
The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE



The Bright Light of Metaphor

Wherein we view software development as trench warfare, but remember that a model is not reality.

Issue #23
May 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

The Bright Light of Metaphor

by Michael Swaine

Welcome to *PragPub* for May, 2011. Pull up a chair. We're polishing metaphors today.

Mark Twain generally gets the credit for that line about the difference between the almost right word and the right word being the difference between the lightning-bug and the lightning. But Twain acknowledged that he got the comparison from Josh Billings. Twain made it his own, though, repeating it often. It's a figure of speech that bears repeating.

And it's when we are using figures of speech, particularly metaphor, that the exact right word really shines. I'd like to think that I selected the right word just then, the word "shine." Because that's what we use metaphor for: to shine light on some aspect of the subject. Anywhere there is complexity, metaphors are a powerful tool for shining light into the darkness. As people who write or talk about software, we need precise metaphors.

As it happens, several of this month's writers employed vivid metaphors in their contributions to the issue. So I thought it might be worthwhile to focus a little extra light on their illuminating figures of speech.

Jared Richardson uses the metaphor of WWI trench warfare to shine some light on certain problems in software shops that lead to stagnation. Brian Tarbox uses a very different metaphor involving a pig to offer some enlightening advice on what to do when everything is falling down around your ears. Jeff Langr and Tim Ottinger reflect on over a decade of agile experience, and along the way have some bright things to say about the use of metaphor in XP and BDD (among other things).

John Shade's essay this month is all about cloud computing, and you can't talk about cloud computing without getting up to your eyebrows in metaphors. John doesn't even try to avoid the metaphors. You can decide for yourself whether or not that's a good thing.

Because metaphors can get you in trouble. "Done badly," Tim says, "it turns the system into a series of odd puns and confusing turns of phrase." And as Andy Hunt points out in this month's "Guru Meditation," "A model is not reality."

Also in this issue, Trevor Burnham shows how CoffeeScript, the hot new language that he calls "JavaScript done right," saves you from some of JavaScript's nastiest traps.

Plus we have another computer history article by Dan Wohlbruck, Choice Bits, and the events calendar. There's some nice feedback in Choice Bits on last month's Arduino issue, and below you'll find the solution to last month's Arduino-themed quiz.

I hope you enjoy the issue.

Agile Reflections

A Dozen Years or so of Agile Development Practices

by Jeff Langr, Tim Ottinger

Jeff and Tim take a break from their recent articles on agile practices to reflect on their personal experiences with agile practices, and specifically extreme programming (XP).

Recently we've been running a series of articles on agile practices by Jeff Langr and Tim Ottinger, the authors of Agile in a Flash [U1]. This month, Jeff and Tim shift gears to reflect on their personal experiences with agile practices and specifically extreme programming (XP). In these short reflections, Jeff and Tim share how XP has changed their professional lives.

Jeff: I found out about extreme programming at a developer's conference in San Jose in 1999, where I attended a talk by Kent Beck, who I'd followed in the Smalltalk world. We had tried spiral [U2] at MCI a few years prior. I'd loved its incremental/iterative cycles, but had struggled with how to keep it from degrading over time. XP immediately felt right to me. My first thought was, "Oh, holy crud, this is how you make it work!" A dozen years later, the most enjoyable projects I've ever been on have been XP efforts.

Tim: I was a Usenet denizen back when Usenet was the "big thing." I was on the C++ and object-oriented groups when Kent and company started talking about this wild and crazy new way of developing software called XP. It sounded daft at first blush, but the more we discussed it the more it spoke to me. It was years later when I finally converted to XP. It remains a brilliant and counter-intuitive system that works.

Continuous Integration (CI)

Tim: Before continuous integration, I remember suffering through "hell week"—or "hell month"—when all the developers tried to get their code to work together, and QA tried to find the seams where disparate changes created side-effects and failures. No other single practice improves a software department as much as CI. It smooths the process, adds predictability, and makes continual testing possible. It helps draw a collection of developers together into a team all of the time, not just in the pre-release crunch. It creates the necessary context for the other practices to produce value.

Jeff: From the converse standpoint, CI requires other key practices to make it work. First, it falls flat without the tests—frequent integration causes too many problems without controls to let developers know that their change didn't integrate with the rest of the code. Also, in settings where I've seen teams use a CI server to track a build but had few or no tests, the developers generally ignored it since it told them little about the health of the system. Finally, frequent check-ins create more clashes in poorly designed systems, so we're driven to build better systems with more cohesive classes. CI is a great way to work, but is even more valuable when combined with these other agile development practices.

Test Driven Development (TDD)

Jeff: Agile or no agile, XP or no XP, TDD is a fantastic practice that continues to teach me endless lessons. Equally thrilling is seeing the light bulbs come on in peoples' heads as they take up and embrace the practice. A favorite encounter was revisiting a team who'd shipped a 100,000 line Java application. The stats on the wall told one story—monthly product defect counts were 0, 3, 1, 2, 1, 1, 4, 2, 0, 1, 0, 0. More gratifying was my discussions with team members, all very proud of their work, and emphatic that their use of TDD was what had made them successful. I go faster with TDD in the long run, and it's fun—what's not to like?

Tim: It's funny how much experience exists in the world regarding TDD, and yet how controversial it has remained. It is not intuitive that you'll make better (faster, steadier) progress by writing tests first, but it is so. It is not intuitive that writing tests first is important, yet it produces better tests and more manageable code. TDD is not a substitute for design or algorithm analysis, but it allows us to explore design and algorithms. The act of writing tests first is a powerful boon to developers, nearly equal to the value of having copious tests. Even so, it is one of the points that must be constantly defended, sold, and re-explained—at least to people who have not tried it long enough to get a handle on it. Maybe the biggest lesson I learned from TDD was not to prejudge, but to immerse myself in a practice before judging its value.

Design Improvement

Tim: I have been an object-oriented nutcase for many years. Before I heard Robert Martin explain his first principle, or dug through my first copy of *Structured Programming*, I knew that we weren't doing things as well as we could and that the problem was in our daily work instead of our closed-door architecture meetings. We weren't going wrong from the start, we were steering wrong as we traveled. Just as cruft and crud are emergent, a clean design can be emergent. It's all in the small steps and paying attention to the shape the code is taking.

Jeff: I think continual design improvement through refactoring is one of the hardest technical practices to get right. Out of the hundred-plus systems I've seen where the developers employed TDD, the code was refactored really well in less than a handful. Over time, poor design—lots of duplication, poorly structured and otherwise difficult code—is what will slow you down most from a technical standpoint. It's easy to see why—who hasn't spent countless hours wallowing through convoluted code in an attempt to simply figure out what it's supposed to be doing?

Coding Standard

Jeff: Really, having a standard is agile? It's foundational to me, but there are plenty of shops where self-absorbed developers take pride in bucking standards at every possible turn. I've seen too much pointless waste from lack of standards, coding or otherwise. Wasted time arguing, wasted time understanding each others' code (little bits of discordance add up to significant irritants over time), wasted time reworking other peoples' code, wasted pairing sessions, and no end of increased problems due to the silos that insufficient standards create.

Before you can be a true team, you have to learn how to agree on a few key things.

Tim: My old naming paper ended up in a lot of coding standards, so it's unsurprising that I'm in support of standards. The change that agile brought me was an insistence on minimalism. We wrote style guides and programming tutorials and recipes and pontification into our tomes of style, always hoping our effort was not wasted. It always was. Instead, a more agile way of programming proved possible with a one-page guide and a collective understanding of how code should be. Less is more.

Collective Code Ownership

Tim: Collective code ownership is another of those "in the large" improvements. When authorship matters and individuals "own" modules, you would expect programmers to make changes more quickly. And it may be true "in the small." But a person can work on only one thing at a time. In the large, requests for changes to a given module will pile up and create large-scale delays. Collective ownership also attacks the "culture of blame" because each developer's work is mingled with the work of all the others. This particular value has been close to my heart even before I heard of "agile" or "XP." It's what smart teams have done for ages.

Jeff: Heading out of the 90s, I was coming from an environment where class ownership was commonplace. It had seemed like an OK practice when I was able to control virtually all of a subsystem. But every time I ventured outside of my realm, I got frustrated at having to wait on other developers. And once again, I started recognizing the haphazard results of silo development: some good code, most just passable, and some truly wretched subsystems we "inherited" from long-gone travelers.

Simple Design

Jeff: Most casual readers interpret simple design as "you ain't gonna need it"—in other words, put into the system the simplest possible set of design elements for the features you are working on currently. That's a start, but the full meaning is captured in Kent Beck's four rules, in priority order: code must run all its tests, the system must exhibit minimal duplication, it must express intent clearly, and it must contain a minimum number of classes and methods.

Simple design is the most under-appreciated gem of agile programming. I've built some subsystems where I deliberately focused only on these four rules. The emergent designs, which I modeled after completion, were elegant and flexible, about as good as any other outcomes where I'd brought my full design background to bear.

Tim: Oh, the joys of YAGNI and KISS and DRY (you ain't gonna need it; keep it simple, stupid; and don't repeat yourself)! XP again taught me to do less, better. I used to engage in a lot of speculative design, and it didn't always pay off. I hoped to run a net positive, but was too afraid to measure it. My designs often were intricate and overly abstracted. I was well-intentioned, but things got out of hand. With simple design, I have less to worry about, and the supporting XP practices (testing, etc.) allow me to make changes as I need them. It is like a super power that makes my mistakes small and correctable.

System Metaphor

Tim: Metaphor is funny stuff. Of all of the practices, it's the one that gets the least play. When it's well used, it really can help you to visualize a design and name the various parts of the solution. Done badly, it turns the system into a series of odd puns and confusing turns of phrase. If it's not done at all, the system will be a mess of mixed ideas and names drawn from concrete details leaking into abstractions. What has stood the test of time is the principle that we need a shared understanding of how our systems work, and a high-level vocabulary we can use to discuss it.

Jeff: We all struggled initially with finding the value in XP's metaphor practice. Either you had a convenient metaphor ("shopping cart" being the canonical example), or you were stuck with the "naive metaphor" of naming a rose a Rose. But isn't Eric Evans' concept of domain-driven design (DDD) what Beck was really getting at? Both metaphor and DDD promote having a common, ubiquitous language, getting the customer and development team all on the same terminology page. Today's programmatic embodiment of metaphor and DDD is BDD (behavior-driven development).

Pair Programming

Tim: Pair programming may be the most surprising practice of all. You would expect it to be inefficient, riddled with personality problems, imposing, and expensive. Instead, it is a highly streamlined way to get work done, saving think time, preventing errors, eliminating "stovepipe" specializations and their associated queuing, reinforcing good design practices, and accelerating teams. It is exactly not what you expect it to be. Most of us who started pairing just to improve the code end up loving it and looking forward to our next session. Add pair programming to the other practices, and soon you truly have a proud and competent team.

Jeff: And yet pairing still languishes. It's probably the hardest XP practice to implement and sustain. Why? Because it requires you to learn how to collaborate closely with every other member of your team. That's tough! With pairing, programming moves from a solo artistry—subject to the whim of each individual's capabilities—to a team effort, requiring coordination and concert. It requires strong social spirit and the desire to succeed, but it's extremely rewarding if you can get pairing clicking in your team.

Conclusions

Tim: Whatever comes next will come on the back of agile practices. It will likely leverage what we are learning about managing teams, managing code bases, new open-source frameworks and libraries, new ways to sell software, and possibly new tools for telepresence. Shops that have not adopted XP values may find a different way into the future, but the smart money is not on them.

Jeff: Agile caught on a lot better than that thing called XP from whence these development practices originally came. But I was and remain greatly concerned: the reality is that it's hard enough to build out a solid system, even with never-changing business priorities and without short cycles. Agile turns that upside down and can devastate a system's quality in very short order if you

don't have the controls and disciplines to keep it clean. The teams who really want to be successful are coming back to "XP in all but name."

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U3] 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 [U4], 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 [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://www.pragprog.com/refer/pragpub23/titles/olag/Agile-in-a-flash

[U2] http://en.wikipedia.org/wiki/Spiral_model

[U3] http://www.pragprog.com/refer/pragpub23/titles/olag/Agile-in-a-flash

[U4] http://www.pragprog.com/refer/pragpub23/titles/olag/Agile-in-a-flash

[U5] mailto:michael@pragprog.com?subject=agile-reflections

[U6] http://forums.pragprog.com/forums/134