The Pragmatic Bookshelf

PragPub

The Second Iteration

IN THIS ISSUE

Issue #70
April 2015

Contents

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail webmaster@swaine.com. 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

On Tap

Feed Your Head

You can download this issue at any time and as often as you like in any or all of our three formats: pdf [U1], mobi [U2], epub [U3].

Down the Rabbit Hole

You're thinking, like Lewis Carroll's White Rabbit, that you don't want to be late to the party. You're checking your watch and imagining the apps you'll develop for it. But before you follow the fuzzy fellow down the rabbit hole, maybe you should pause to think about this new Apple Watch programming opportunity.

Like the move from desktop applications to mobile apps, this Watch thing is a paradigm shift. You won't be merely porting your phone apps to the watch, or even your phone app concepts. It's a whole new game.

Jeff Kelley's upcoming *Developing for Apple Watch* is a great way to get introduced to Apple Watchkit development. But Jeff's article in this month's *PragPub* is something different. In it, Jeff writes about how to *think* about Watch apps, what to consider before you start coding — or even before you start imagining — your first Watch app. It's an exercise in thinking in this new paradigm.

Also in this issue, we have a mind-altering article by Adam Tornhill on getting to know the social side of your codebase. Adam's book Your Code as a Crime Scene [U4], inspired by forensic psychology methods, teaches strategies to predict the future of your codebase, assess refactoring direction, and understand how your team influences the design. In this article, he shows you how the structure of your organization can be read out of the history of your codebase — and how that can help shape your thinking about your projects.

Jeff Langr, no stranger to the pages of *PragPub*, revisits his own coding history and finds it necessary to revise one of his basic principles. Along the way he shares some hard-won insights on how to make distributed development teams work.

And there's more. Expert iOS developers Chris Eidhof, Wouter Swierstra, and Florian Kugler are back again with another bit of functional Swift code. Johanna Rothman and Andy Lester offer sage advice on dealing with recruiters. Marcus Blankenship has some encouraging words if you've just made the transition to manager and are having some misgivings. Anthony Cangiano has again tracked down all the new tech books, John Shade worries about the diet of 3D printers, and we have a selection of tasty tweets, a Pub Crawl of sites to see, and our regular Pub Quiz — which is really more of a sudoku-anagram mashup, but Pub Sudoku-Anagram Mashup just sounds odd.

Who Did What

photo credits: Cover: “Alice Meets the White Rabbit” [U5] by Margaret Tarrant is licensed under Creative Commons [U6]. • p. 14: “Nice! Dick Tracy went for the Pink Apple Watch” [U7] by Alan Levine is licensed under Creative Commons [U8]. • p. 18: chimney swift [U9] by Ed Schipul is licensed under Creatuve Commons [U10]. • p. 19: “System, Network, News” [U11] by geralt is licensed under Creative Commons [U12]. • p. 26: “Computer/Communication” [U13] by geralt is licensed under Creative Commons [U14].

editing and production: Michael Swaine

editing and research: Nancy Groth

customer support, subscriptions, submissions: webmaster@swaine.com

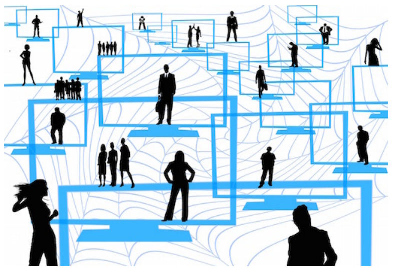
staff curmudgeon: John Shade (john.shade@swaine.com)

Rule #1 for Distributed Teams

The 2015 Edition

by Jeff Langr

Distributed development teams face some daunting challenges. Jeff has been there and knows how to solve them.



The first rule for distributed teams is: *Don't* [U1].

Had my co-author Tim Ottinger crafted the second rule to be again *Don't*, à la Fight Club [U2] ("The first rule of Fight Club is: you do not talk about Fight Club. The second rule of Fight Club is: you DO NOT talk about Fight Club!"), I might not have found myself in the awkward position of promoting distributed teams in this article.

When we compiled the rules for distributed teams for the Agile in a Flash blog back in June of 2011, Tim and I had various joint and disjoint experiences working with and in distributed teams. We were both remote developers for a now defunct company named GeoLearning for almost all of 2010. We poured our mixed experiences into the rules.

Nearly four years later, I've added almost two more years' experience as a remote developer for Outpace, where we've made distributed teams a successful reality — and even a competitive advantage in some ways. We've also managed to validate the rest of the rules.

At Outpace, we follow the Agile in a Flash Rules [U3] for Distributed Teams:

- Almost all of us are remote, so we don't leave "satellite" workers in the dark.
- We don't prevent co-location where it makes sense — a number of our business folks work together daily.
- We understand that "longitude kills," so we restrict our hiring to folks who can pair during the same core hours (roughly 8 a.m. through 5 p.m. Mountain time, or 10 a.m. to 7 p.m. Eastern time).
- We're not always remote — we get together as a whole team a couple of times a year to better understand and appreciate each other as real people, not just faces on a computer screen. People in some of our busier cities (such as San Francisco, Chicago, and Calgary) get together after hours at times for drinks, and fourteen recently met up for a skiing weekend in Jackson Hole.

Beyond having a solid business model and product vision, other core factors that have contributed to our success include:

- *A flat organizational structure*. For the most part, there are business folk, and there are developers. No one's full-time job is management. From the development side, we've designated team leads whose primary responsibility is to be on point for communications with the business and other teams. From the business side, most everyone is a domain expert who works closely with the development team to impart and clarify the business needs. Our simple structure has increased transparency, simplified communication channels, and allowed us to pay more for people who know how to get the job done.

- *A broad set of high-tech collaboration tools*. We talk to each other face-to-face via Zoom video chats. We text each other privately and publicly via Slack channels. We plan collaboratively using Trello and TargetProcess. We document collaboratively using Google Docs and Google Sites. We even resort to email from time to time!
- *Paired development*. The bulk of my day involves collaborating with other developers online. Pairing keeps us on track and helps ensure a higher quality solution. It also brings back the necessary social interaction that remote work removes. You don't have the opportunity to feel lonely or isolated when pairing is the norm.
- *Hiring sharp team members*. Rather than focus on technologies, we seek experienced developers who are good at solving problems as part of a pair and team. We've passed on sharp candidates who showed disinterest in extensive pairing or who were unable to verbalize their thought processes as they worked through solutions.
- *A focus on high-quality code*. We write lots of unit tests for our code, and have rarely shipped logic defects. We've been increasing the amount of acceptance-level testing steadily as well.

An Outpace Day

(Note: The following synopsis is an amalgamation of real day-to-day events, not a single actual day — I'm way too lazy to keep such a detailed log. It's also a slightly out-of-date one at that — things can move fast at Outpace, and I've moved into a different team recently.)

- *7:47 a.m.* I finish reading the news. I meander back upstairs, brush my teeth, and slip on a pair of comfy boots — I'll be standing for a good part of my day.
- *7:51:00 a.m* My commute begins.
- *7:51:09 a.m.* I arrive safely at my desk in my office, located off the family room. I push my chair aside, hit the #2 preset on my Jarvis sit/stand desk, and wait until it rises to 42.2".
- *7:52 a.m.* I spend a few minutes perusing Slack chat and email. I check the status of the deployed system. All is well.
- *7:58 a.m.* I take a look at the stand-up document, shared in Google Drive. As usual, a few people are adding short sentences to today's bulleted list. I note that Paul, our team's primary business person, is in the act of bolding one of the bulleted items, indicating that he wants to talk about it. I read the ones not bolded — we won't discuss these, they are more "FYI."
- *8:00 a.m.* I hit the link for our stand-up's Zoom video chat. Less than twenty seconds past 8 a.m., eight people have joined. I kick things off by asking Paul to talk about the item he bolded. The customer has asked for a change in the way we process their external events feed. We discuss a

solution for a minute, and end with a brief plan of what we'll tackle today. We talk through the other two bolded items similarly, then ask if anyone plans on deploying to production today. Tanya, from the core product team, responds that she and her pair will be finishing testing their persistence changes this morning, and plan on deploying around mid-day. Nate, also from the core product team, adds that he and his pair would like us to review a pull request for code changes in the customer repository. We agree and let him know we'll take a look soon.

- *8:08 a.m.* Stand-up complete. I fill my glass of water in the kitchen.

The use of the shared document has been a great boon to facilitating our stand-up meetings. Useful information ends up in as bullets that people can quickly scan, asking questions if needed. Otherwise the owners of the bolded items speak up when it's their turn.

We had initially tried a number of different flows for stand-ups. Having someone prod everyone during the stand-up seemed annoying, and trying to get people to randomly speak up didn't seem to work well either.

The shared Google document makes things simple and straightforward, and holds up well in larger meetings (we've had some video stand-ups involving 25-30 folks).

- *8:09 a.m.* I start looking at Nate's pull request in GitHub. Mario, my pair for the day, does the same. We send a few text messages to each other in a private Slack channel, and as a result, end up making a few comments about the code on the GitHub pull request page. The comments show up in Slack. Nate sees the comments, which starts a brief Slack exchange that ends with Nate going off to clean up the code.
- *8:20 a.m.* I initiate a Zoom session and invite Mario. We're going to pair on a change to our order service regarding how we filter out offers for unavailability. First, we move the card in our development board in Trello from the Iteration Backlog to the In Progress column.

We discuss the change for a few minutes, looking at some related code and tests to better understand what we're up against. When we're ready to work on the solution, I join a tmate session that Mario has started in a large terminal window on his machine. Since our code is in Clojure, Mario cranks up emacs in the terminal window. He loads the unit tests and the .clj file for the production source. While he's looking at the code, I split the tmate window and start lein-test-refresh (a utility built by Outpacer Jake McCrary that aids in re-running tests; see lein-test-refresh [U4] in a short terminal window at the bottom. All tests pass, so we start test-driving our solution.

Does it sound like we're a dogmatic agile-ish shop? We really don't have a lot of standards. Everyone has the same hardware setup — MacBook Pros with a couple of Thunderbolt monitors for screen sharing — which makes screen sharing a bit more effective. Process-wise, we're expected to pair as much as makes sense, we are expected to be at the stand-up meetings, we write unit tests for our code, and we ensure the code gets tested before going into production. Few formalities exist for these expectations, and there are no further demands. What ends up mattering more is the implied standards that teams create for themselves, and even those can change at the whim of a pair.

Editors/IDEs in play include primarily emacs, vim, and IntelliJ (and we've seen a bit of Eclipse, Sublime, and LightTable); languages in play include primarily Clojure and CoffeeScript or ClojureScript, plus a bit of Ruby and Python. Programming approaches in play include REPL-driven development, TDD, and TAD (test-after development). Our first six months involved a flurry of experimentation and exploration with techniques, tools, and libraries. The flurries have mostly settled, though we're not averse to changing direction as needed.

We're not afraid of new technologies, but we've surpassed the need to play with new toys just for geek gratification. Our stack has become a balance of tools that hold up well to our system's high-volume demands (minus a few things that didn't pan out quite as well that we'll soon jettison).

- *9:15 a.m.* We're mulling over a sticky part of our change. I get up to clear my head for a minute, and refill my glass of water. So does Mario.
- *9:17 a.m.* Mario has hit upon a potential solution. He scratches out our next test. Some days, today included, we feel like ping-ponging. I provide some Clojure code that gets the test to pass, and write another one, watching it fail first. Mario puts up a solution, gets the test to pass, and we each do a small bit of refactoring cleanup.

We used the Mac's built-in Screen Share application for a good while. Pairing sessions ended up being more "worker-rester" (see the Agile in a Flash card #37, "Pair Programming Smells" [U5]), where the host — the person sharing their desktop for the session — does most of the driving, and the other party to the pair observes and pipes up as needed, occasionally hitting the keyboard. Unfortunately, the remote programmer is at a significant disadvantage — the lag time between typing a character and seeing it reflected on the host's screen can be a half-second or so, making for a frustrating experience.

To improve the lag time, Mario and I switched full-time to the terminal multiplexer tmate (a fork of tmux that simplifies and secures the connection process), which allows us to share a terminal window. We do the bulk of our work in a terminal window, which becomes an inviting space for the remote party — character-mode transmission means we almost never see any real lag time when keying into a remote tmux session. If we need to show something outside the terminal (e.g., the browser), we simply share the screen using Zoom. (I've even toyed with the idea of bringing the text-based browser Lynx into the mix. :-))

Beyond the initial keyboard and color configuration hurdles (Brian Hogan's book tmux: Productive Mouse-Free Development [U6] provides some great advice here), we've run into a few odd curveballs that tmate throws at us — all nuisances but never show-stoppers.

The main tradeoff, however, is that we must use a character-mode editor. That's not a worry for Mario and me, as we prefer vim and emacs anyway. We've commandeered a good number of keyboard shortcuts that help us control tmate — split windows, new windows, and window navigation are an instant away beneath a keypress.

But, oh, the response time is fantastic! It's almost like being there — remote pairing seems real when using tmate. I no longer demonstrate reluctance toward typing when connected to Mario's machine. Things like ping-pong pairing become feasible.

- *10:25 a.m.* Break.
- *10:35 a.m.* Back to work. I press button #1 on my Jarvis desk controls; the desktop lowers to 30.0" while I roll my chair under my rear and sit. After re-Zooming Mario, we review Nate's changes to his pull request and merge it. We return to working on the filtering code.
- *10:50 a.m.* We merge our code into master, run all tests, and push to GitHub. While waiting for the build to complete, we talk with Nate, who's ready to push his change into production. Nate and his pair partner join our Zoom. We grab the tag created by the CI build, and deploy to our production staging Amazon machine instance. We verify his changes with a quick set of manual tests, and then switch IPs to put our changes into production.
- *11:15 a.m.* Back to the filtering code. We work for another 40 minutes, commit a few changes, and decide to break for lunch.

It's a few minutes before noon, my time (U.S. Mountain). The ever-accommodating Mario is located in the central time zone, an hour ahead, meaning he's eating at 1 p.m. Most of us are pretty flexible; I might take lunch starting at anywhere from 11 a.m. to 1 p.m., depending on what's going on (and yes, sometimes I'll eat at my desk).

- *1:00 p.m.* The afternoon starts with a long cross-team pow-wow that involves the team leads — about ten folks total. Most teams actually have a couple of leads who represent their team at such group meetings. Our meeting, like most, ends on time at 1:30 p.m. We're rarely swamped with meetings, and having representatives allows most developers to get their work done without interruption.

We first used Zoom without paying for it, to ensure the videoconferencing technology worked well for us (we abandoned Google Hangouts after realizing it wasn't meeting our needs). Free Zoom meetings are time-limited, however, and cut off automatically after 40 minutes. We realized that the 40-minute limit was a great feature — if only all live meetings had that hard stop!

Like any organization, we seek to find ways to get together aside from project work to discuss things important to Outpace's culture and also to socialize.

One team has initiated a "stand-down" time toward the end of the day to talk about how things are going and perhaps pop a beer. I've put "4:20 time" on my calendar to carve out time at the end of the day to experiment and learn. (Those not living in Colorado or Washington might refer to this as "42 time.") Like other teams, we have retros, information-sharing sessions ("learn-you-a-things"), and Secret Santas.

- *1:30 p.m.* Back to the work on the filtering story. We code, laugh a bit, finish the programming work, and spend the rest of the afternoon (minus a couple of short breaks) deploying the change to a QA environment where we flesh out and run a few tests. Happy with the change, we let our QA person and Paul know that they can take a look. We agree on a plan to deploy the completed feature to production tomorrow afternoon as long as everything is still looking good.
- *5:11 p.m.* Mario and I wish each other a good evening. I start my commute, this time half the distance, to my kitchen.

- *5:11:05 p.m.* Commute home finished.

We learn something new every day about what works well in this distributed world and what doesn't, and we try to adapt accordingly. Mario weighs in on some of the challenges of distributed communication:

"One thing that seems like a downside is that it's not as easy to overhear things as it is when everyone is co-located. It's not impossible, it's just that everyone has to be in the same group meeting. Just being able to overhear conversations is something that is handy from time to time."

"Slack lends itself to overhearing conversations. But there are more private conversations as opposed to public ones. Plus, there's overhead to having a lot of channels — some that I monitor more closely to others. My ears will pick up keywords. It's not as often that I find myself noticing the same thing in Slack."

Indeed, it's easy to get wrapped up in email and meetings "IRL;" Outpace adds to that the distraction of private chat conversations and several dozen Slack channels (such as the #cuteness channel, replete with posted photos of amazingly cute animals). But as in the physical world, people vote with their feet. I've learned to bow out of most of the fluffy channels (but not #cuteness!) and I've added audible notifications on the couple channels that really matter.

My first long-term, full-time experience with distributed development in 2010 (at GeoLearning) was less than ideal. We rarely used a camera and we were often frustrated by the sluggishness of the GUI-shared-screen development. Since the co-located folks represented the bulk of the teams, I always felt like I was an adjunct resource, too often forgotten or left out of important conversations —not a true member of the team.

My second experience has been rewarding in a number of ways:

- The flat organizational structure has allowed me to work directly with our very sharp business folks, saving me from the headaches of the typical corporate bureaucracy.
- The evolved technology set has made pairing and communication in a distributed environment effective and a little closer to the ideal of "being there."
- The distributed nature of the company has provided me with the opportunity to work with an impossible number of highly capable developers and business folk, far more than usually available in any one physical location.
- The pairing has given me the opportunity to learn from these top-notch people, and to ramp up on technologies that most companies would require as prerequisites for the job.
- The insistence on code quality has allowed me to be proud of the work that I ship, and it has also helped us ship more often.

All of this in the comfort of my own home. (I do have to prod myself to leave the house during the week; it's fairly easy to become a permanent homebody.) Spouse and kitchen are moments away.

Our original Rules for Distributed Teams still hold true. In 2011, immediately after saying "don't do it," we wrote that distributed teams can work — just make sure you have a compelling reason and that you *really* commit to it. To fully commit, you must find *all ways possible* to make up for the loss of face-to-face communication with co-located teams, which is extremely significant. That means spending the time and money as Outpace has done, and continually seeking to improve.

Number One Rule for Distributed Teams, 2015 version

1. *Don't …*

unless you really mean it

About the Author

Jeff is a veteran software developer with a quarter century of experience. He's written five books on software development: Pragmatic Unit Testing in Java 8 with JUnit [U7] with Andy Hunt and Dave Thomas, Modern C++ Programming with Test-Driven Development [U8], Agile In a Flash [U9] with Tim Ottinger, *Agile Java*, and *Essential Java Style*. He also contributed two chapters to Uncle Bob's (Robert C. Martin's) book *Clean Code*.

External resources referenced in this article:

[U1]	http://agileinaflash.blogspot.com/2011/04/rules-for-distributed-teams.html
[U2]	http://www.imdb.com/title/tt0137523/quotes
[U3]	https://pragprog.com/book/olag/agile-in-a-flash
[U4]	https://github.com/jakemcc/lein-test-refresh
[U5]	https://pragprog.com/book/olag/agile-in-a-flash
[U6]	https://pragprog.com/book/bhtmux/tmux
[U7]	https://pragprog.com/book/utj2/pragmatic-unit-testing-in-java-8-with-junit
[U8]	http://pragprog.com/book/lotdd/modern-c-programming-with-test-driven-development
[U9]	http://pragprog.com/book/olag/agile-in-a-flash