The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

Clojure Is Hot!

Welcome to our Clojure issue.

Issue #25
July 2011

Contents

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

Clojure Is Hot

by Michael Swaine

In this Clojure issue, four authors take on the Clojure language from four perspectives. And for balance, we also have non-Clojure articles by Jeff Langr, Tim Ottinger, Dan Wohlbruck, and John Shade.

Welcome to our first-ever Clojure issue.

Like most good inventions, this JVM-friendly Lisp dialect draws on the work of many but is the brainchild of one person—in this case, Rich Hickey. So although we've covered it before (in November 2010 [U1] and July 2009 [U2]), we decided that it was time to devote a whole issue to Clojure. And we came to this conclusion on thermal grounds: Clojure is hot.

According to Chas Emerick, commenting on his 2010 State of Clojure Survey [U3], "It seems clear that the Clojure community is growing, and growing fast." Also, "very few people have come directly from Common Lisp and Scheme," suggesting that the growth of Clojure is not just movement within the Lisp community. The metablog Planet Clojure now lists more than 250 blogs. Is Clojure the new Ruby? The last programming language? The future? It's been called all these things.

Here are some resources so you can test the temperature and educate yourself: clojure.org [U4], Clojure/core [U5], Rich Hickey's Clojure Reading List [U6], Planet Clojure metablog [U7], Clojure.conj [U8], disclojure: public disclosure of all things clojure [U9], clojure google group [U10], and Clojure links on Twitter [U11]

A Lively Community

One bit of evidence of vitality in a language is lively action in user groups. And Clojure has a lot of user group activity. So when I thought of putting together a Clojure issue, it seemed the logical thing to tap the user groups for articles. Not just because the lively and technically sophisticated discussions in user group forums led me to think they would be a good source of articles, which proved to be true, but also because I thought an open call to user group members might be a good way to get a feeling for the kind of discussion going on in the Clojure community.

But I wanted to hedge my bet by drawing on tested talent, so I solicited an article from Aaron Bedra of Relevance, who is working with Stuart Halloway on the second edition of Programming Clojure [U12].

So the calls went out, the articles came in, and here's the line-up:

Jeff Héon from the Montreal Clojure User Group leads off with an introduction to Clojure that highlights its capabilities for data manipulation. It's a gentle intro that should let the reader new to Clojure get to know enough about the language to decide if it is worth pursuing further.

Then we swing to the other extreme. Steven Reynolds of the Clojure Houston User Group follows up with a deep dive into the internal representation of some Clojure collections. He illustrates the backing data for objects like a physician using an MRI to see the internals of their patient.

Aaron Bedra finds the pragmatic way between these extremes, walking you through the development of some Unix services in Clojure, with the knowledge and clarity that he's putting into the next edition of Programming Clojure[U13].

Then Ambrose Bonnaire-Sergeant of the Seattle Clojure User Group walks you through the creation of a small Clojure DSL, starting with common building blocks like conditionals and motivating more advanced mechanisms that Clojure uniquely provides, like Multimethods and ad-hoc hierarchies.

But Wait, There's More...

Because there is more to life, even the coding life, than Lisp dialects, we've included a few other goodies. Jeff Langr and Tim Ottinger follow up last issue's article on pair programming with a detailed list of benefits of pairing—benefits to the individual programmers, to the team, and to the management or the project. And Dan Wohlbruck takes us back in time to the birth of the Unix operating system, which celebrates its 37th birthday this month.

John Shade weighs in on a different birthday celebration, with some pointed comments on the industry's most celebrated centenarian. Of course there's the latest Events Calendar, telling you about where our authors will be appearing and other notable events, and Choice Bits, where you'll learn that it's good to be Branson.

And last but not least—no, actually it is least—we've added a page at the end of the issue in which we hint at things to come. It's called "But Wait, There's More..." In an open-ended way, it give the issue a sense of, uh, closure.

External resources referenced in this article:

[U1]	http://pragprog.com/magazines/download/17.HTML
[U2]	http://pragprog.com/magazines/download/1.HTML
[U3]	http://cemerick.com/2010/06/07/results-from-the-state-of-clojure-summer-2010-survey/
[U4]	http://clojure.org/
[U5]	http://clojure.com/
[U6]	http://www.amazon.com/Clojure-Bookshelf/lm/R3LG3ZBZS4GCTH
[U7]	http://planet.clojure.in/
[U8]	http://clojure-conj.org/
[U9]	http://disclojure.org/
[U10]	http://groups.google.com/group/clojure
[U11]	https://twitter.com/clojurelinks
[U12]	http://pragprog.com/shcloj2
[U13]	http://www.pragprog.com/refer/pragpub25/titles/shcloj2/

Pair Programming Benefits

Two Heads Are Better than One

by Jeff Langr, Tim Ottinger

Two heads are better than one, and four hands are better than two.

Pair programming is touted as a way of building a better system: two heads are better than one, they say, and thus two heads will usually produce a higher-quality system. Follow the rules of pairing (see last month's article, Pair Programming in a Flash[U1]), and you'll have an even better chance of realizing this potential.

Review

A colleague and friend of ours said that he despises pairing, but he does it all the time and teaches others to do it. The reason? "It sure makes the code nicer."

The review element of pairing is essential: Unlike manufactured products, code product not only ends up in the consumer's hands, it also stays beneath the programmers' collective feet. As Uncle Bob Martin says, the primary input to a programmer is yesterday's code. Code can serve as a good foundation, or a constant hindrance.

Good code can make it easier to track down the source of a defect. Bad code obscures important details, and duplication scatters them all over the code base. Bad code leaves you scratching your head when your system has crashed and customers (and VPs) are screaming for you to get it back up.

Is this increase in quality enough of a reason to consider throwing two people at the problem? Laurie William's book *Pair Programming Illuminated* goes into considerable detail on the costs and benefits of pairing. The statistic that is most quoted from this book is that pairs produce higher-quality code in 15% more time than individuals. For that additional cost, what other returns on investment can pairing produce?

In the remainder of this article, we'll present our list of benefits (a few of which are the same as outlined in the Williams book) that we've accrued over the past 10+ years of pairing experience. Nothing comes free, of course; there are most certainly costs and other considerations to take into account when considering pairing.

We pair because it makes the code better, and makes us better.

Team and System Benefits

- The value of increased system quality can't be diminished. Allowing slap-happy programmers to run roughshod over a system will drag down future productivity, compounding costs every minute that it's allowed to continue. What do you really know about the quality of product your team members produce?

- Pairing rotation expands the sphere of knowledge of all developers on a team. This broader knowledge increases the potential for individuals to recognize duplicate logic across the code base. Increased awareness of other parts of the system can also help contribute to a better overall system design.
- We tout the team room concept as one of the best ways to increase collaboration and productivity. However, it's not without trade-offs. A room populated with a whole team can be noisy and distracting at times. Pairing can help: A focused pair can more easily block out distractions than an individual. People are also less likely to interrupt a pair deep in work and conversation than an individual sitting alone.
- A set of programmers each doing their own thing in a private office or cube does not a true team make. A real team collaborates closely, and team members understand each other as individuals. Pairing is a great way to get there.
- At some level, standards are useful beasts (although it's possible to go too far with them). But without appropriate mechanisms in place, standards begin to quickly fall by the wayside until they're no longer valuable. The peer pressure of pairing can help ensure that we continue to adhere to basic team agreements.

Programmer Benefits

- Pairing helps prevent pigeonholing. Not only will you move throughout all responsibilities on your team, but you'll also be more free to move to other teams, as your managers learn that they will not be devastated by your departure.
- As a new hire in a pairing environment, you don't spend week one (or month one) sitting and reading out-of-date documentation or fearing a code base that you can barely begin to understand on your own. Instead, you get to jump right in and wet your feet with live production code. The rest of the team doesn't resent having to take time out from "their" work to answer your endless questions about the system--they can instead work with you directly, because that's how the team has chosen to work.
- We don't know about you, but our experiences with ex post facto reviews in lieu of pairing have usually been far from enjoyable. We find that they take a lot of time and distract us from "our" work, which means we typically give them short shrift. We suspect most other programmers feel the same way. When code "in review" cannot be committed to the main development line, it rots while the version control system marches on. Waiting for a code review may subject a programmer to a very costly merge.
- We love learning new things about software development. We think we're pretty good at programming, yet rarely a day goes by when we don't learn something new and significant--even from the most junior programmers on the team.
- If you're the team's rock star, pairing can give you mentoring and teaching opportunities that you've never had before, plus the respect you deserve.

Invariably, a great programmer on any team (whether outgoing or quiet) becomes revered by the team. If you have the skills alone, you have the skills paired too.

- If you are the weakest player on the team, you will find that pairing gives you an opportunity to learn from your teammates. In addition, as the partner shares the keyboard and ensures that you're doing test-first work, you will find that it's harder to make a mistake that gets through to integration (let alone release). You have a safer working/learning environment.
- Pairing is enjoyable and sustainable. Lest you think we only consult with teams, not drinking our own Kool-Aid, both of us have paired daily for extended periods as part of software development teams. We appreciate the social and personal growth aspects of pairing immensely.
- When you are tired, frustrated, less well, hungover, underslept, low on biorhythms or feeling unlucky, you are far more likely to stay engaged and productive if you are pairing. Partners look out for you. Your worse days pairing won't look like your worse days as a solo programmer.
- Accomplishment is the ultimate motivator. Working in pairs allows you to participate in successes more often than solo work does.

Management/Project Management Benefits

- We promote expertise, not specialty. The increased team member knowledge gained from pair rotation reduces your risk of depending on team specialists. Most team members will end up with competency in most areas of your system. Loss of an expert does not devastate your team's productivity while you secure a replacement.
- New hires usually represent a drain on productivity. We've been in shops where new hires weren't trusted to work alone for months (and in one place, years). With pairing in place, however, a new hire almost immediately becomes a productive team contributor.
- Not only do you need not worry about losing team members, you can use pairing as part of a larger "cross-pollination" strategy. If you manage multiple pairing teams, you can swap team members with negligible negative impact to the teams involved (see previous bullet). Temporarily swapping team members can reinvigorate both teams by introducing new perspectives or techniques.
- Individual capabilities are usually all over the map in a typical team. Planning and estimation is tougher because of these disparities. Pairing instead begins to produce a more-leveled team: Under-performers are pulled up by their more-capable team members, producing a team that has a better long-term chance for success. The leveling produces a more predictable rate of development, which in turn can improve the quality of estimates.
- No one can hide in a team that's pairing. It's tough for team members to go off and surf the net when their peers are depending on them to

contribute via pairing. An engaged team is status quo when frequent pair-swapping is common.

- Your technologies of choice become far less important as new hire criteria. It can be tough to find a qualified Clojure developer, for example, but if you already have a team who is well-versed in Clojure and pairing, it's a non-issue. Instead of technologies, you concern yourself with primarily two things: attitude and aptitude. Can this candidate work well with my team (and does he or she want to work in this manner), and does he or she have the chops to quickly learn the technologies and contribute?
- Interviews themselves become simpler. A few minutes of relaxed pairing with team members is often all it takes to determine if a candidate is up to it. No dumb puzzles or whiteboard programming sessions required!

Ultimately, what is the value of a true team that works well together, collaborates, continuously improves the code base, and encourages each member to improve? That's the kind of team that you can foster with healthy pairing. The bean counters might not get it, but the benefits to all involved—be they programmer, manager, customer, or business—warrants serious consideration.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash[U2] with Tim, he's written over 100 articles on software development and a couple books, *Agile Java* and *Essential Java Style*, and contributed to Uncle Bob's *Clean Code*. 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[U3], 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[U4] or discuss the article in the magazine forum[U5].

External resources referenced in this article:

[U1]	http://pragprog.com/magazines/2011-06/pair-programming-in-a-flash
[U2]	http://www.pragprog.com/refer/pragpub25/titles/olag/Agile-in-a-flash
[U3]	http://www.pragprog.com/refer/pragpub25/titles/olag/Agile-in-a-flash
[U4]	mailto:michael@pragprog.com?subject=agile-reflections
[U5]	http://forums.pragprog.com/forums/134