Extracted from:

Remote Pairing

Collaborative Tools for Distributed Development

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Press

Remote Pairing

Collaborative Tools for Distributed Development

Joe Kutner

Edited by Brian P. Hogan

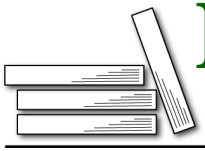
Remote Pairing

Collaborative Tools for Distributed Development

Joe Kutner

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-74-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—January 2014

There is an old saying that two heads are better than one. It may not be universally true, but modern psychology research has provided evidence of its validity in many situations.[1] For programmers, an extra set of eyes can prevent errors and inject new ideas when working to solve tough problems. Writing code in conjunction with another programmer might help us identify edge cases or create better code designs that reduce coupling and improve cohesion—making our programs easier to maintain down the road. But there's no need to speculate about the benefits of writing code with another programmer. Evidence from academia and industry shows that pair programming leads to better code.[2]

Pair programming is a technique in which two programmers jointly produce one artifact, such as a design, an algorithm, or some code. Experiments have demonstrated that pairing improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications, and is considered more enjoyable at statistically significant levels.[3]

Traditionally, a pairing team would sit physically side-by-side, but improvements to screen-sharing tools, terminal-based editors, and virtualization have made it easy for pairing teams to work from different locations. Even better, studies have concluded that the quality and time benefits of remote pairing are the same as when pairing traditionally. But not all pair programming is equal—you've got to follow the rules.

Laying the Ground Rules

No matter where you're pairing from or what tools you're using, you must follow these rules if you want to benefit from the technique.

Share Everything If you are using a tool to debug some code, inspect a runtime, or anything else, your partner must be able to see it. In traditional pairing this usually means sharing the same physical computer. But in remote pairing it's more nuanced, and it's the primary problem we'll address in this book.

Share Equally Your tools should not give one party a control advantage over the other. The best example of violating this in remote pairing is the use of view-only screen sharing.

1. *Optimally Interacting Minds [BOLR10]*
2. *Strengthening the case for pair programming [WKCJ00]*
3. *The costs and benefits of pair programming [CW00]*

Be Comfortable In traditional pairing, comfort is usually a function of your physical surroundings. But in remote pairing it often depends on the quality of audio and video, or the general ability to communicate. For example, if you find it difficult to express that you need a bathroom break, you won't be comfortable.

Stop When You're Tired Many programmers hate pair programming, and understandably so. It's exhausting. But you'll learn about tools that help reduce fatigue and make it easy to pair for longer periods of time. Even with these tools, however, it's important to stop when you become disengaged.

Debate with Your Partner (But Keep It Short) Your pairing environment should be democratic, and you should be comfortable expressing your opinions. But you may be wasting time if the debate goes on for too long. Jeff Langr and Tim Ottinger, the authors of *Agile in a Flash: Speed-Learning Agile Software Development [LO11]*, recommend debating for no more than ten minutes without producing some code.

These are the rules, and they work. We know this because we have scientific evidence to prove it.

Examining the Evidence

In the mid 1990s, a number of software-engineering experts began to observe the growing trend of pairing in development teams. They reported that programmers were producing code faster and freer of bugs than ever before.[4]

Around the same time, a group of Smalltalk programmers and software-engineering consultants began to incorporate this practice into a methodology they called extreme programming (XP).

As XP gained traction in the industry, software-engineering researchers began publishing the results of controlled experiments that compared the work products of paired and individual programmers. A 1998 study from Temple University found paired teams completed their tasks 40% faster than individuals doing the same work, and they produced better algorithms and code.[5] Subsequent studies confirmed these results.[6] Most of the experiments were conducted in a controlled academic environment, but the software industry at large was finding similar results.

4. *Pattern Languages of Program Design 2 [VCK96]*
5. *The case for collaborative programming [Nos98]*
6. *Strengthening the case for pair programming [WKCJ00]*

In one example of real-world success, Ron Jeffries and Kent Beck introduced pair programming to a project for the Chrysler Corporation. Five months later, they found nearly all of the bugs making their way into the production system were written by solo programmers. The project was completed close to schedule and was ultimately deemed a great success.[7]

However, all of these early studies were focused on colocated pair programmers. We cannot just assume that the results will hold true for distributed teams. Fortunately, a number of other studies, which compare distributed pair programming (another name for remote pair programming) to traditional pairing, suggest that the same benefits exist. A 2002 study from North Carolina State University found distributed pair programming teams produced code of the same quality in the same amount of time as colocated teams.[8] More recent studies have confirmed these results.[9]

Not all pair-programming teams are equal, though. Some studies suggest that the expertise of the programmers and the complexity of the tasks may determine the technique's effectiveness. In some cases, it was found that junior programmers require more time when pairing on complex tasks, but still produce higher-quality code. Intermediate-level programmers seem to benefit the most from pairing. One study found that developers in this category experienced a massive 149% increase in correctness over individual programmers.[10] Expert programmers show varying levels of success depending on the complexity of a task. But the worst case is breaking even. There is no evidence to suggest that quality or productivity will be reduced for any level of expertise or task complexity.

One potential drawback of pairing is that it may require additional effort to complete a task. Pairing teams produce faster results because they work in tandem, but the combined effort may lead to as much as a 60% increase in man-hours. Researchers have found, however, that this overhead may begin to subside after awhile. Programmers often go through a phase called "jelling" when they are first introduced to pairing. During this phase, teams may require 60% more man-hours to complete a task, but after the adjustment period it is often reduced to a minimum of 15%.[11] The same research, however, suggests teams can make up for this overhead in the long term because the quality of the code they produce, as measured by bugs, will be better.

7. *Chrysler Goes to 'Extremes' [ABB98]*
8. *Exploring the efficacy of distributed pair programming [BGS02]*
9. *Empirical evaluation of distributed pair programming [Han02]*
10. *The Effectiveness of Pair Programming: A Meta-Analysis [HDAS09]*
11. *Strengthening the case for pair programming [WKCJ00]*

It's believed that once a team jells, it becomes "almost unstoppable, a juggernaut for success," according to IEEE fellow Tom DeMarco.[12] Jelled teams also tend to enjoy tasks that individuals would consider dull. Thus, your goal when pairing is not simply to be in the company of another programmer, but to act as one with that person. It will always take time to jell, but when it comes to *remote* pairing there are additional concerns. To jell with a remote partner, you must take action at the beginning of each session to ensure the process goes smoothly.

Pairing Up

Regardless of who you're partnering with or what you'll be working on, each pairing session should begin with the following three steps.

Step One: Establish a Communication Channel

The communication channel can be as simple as an instant-messaging program, but it's usually preferable to have an audio and even a visual connection. You can use any of the many voice- and video-communication tools available, but we won't explore those in detail. If you don't know where to start, use Skype.[13] Its reputation as a user-friendly tool is weak, but its ubiquity and even the reliability of the voice over IP technology underlying the clunky client interface is unparalleled. We'll discuss Skype and some alternatives in more detail later in the chapter.

Regardless of the voice software you choose, it's also important to have a good microphone. The mic included in your laptop or display will probably reduce the likelihood of your partner understanding your every word—and your partner *should* be able understand every word. If you're going to pair-program often, then get a moderately good microphone such as the Blue Snowball.[14] If you prefer a headset, consider the Logitech H390, which is affordable and has a good reputation.[15]

After getting connected, do a quick mic check. Make sure there is no background noise, static, lag, echo, or anything else that might make the other person difficult to hear. If you find a problem, address it immediately. Once you've established a good communication channel, you'll need to discuss a few things with your partner.

12. *Peopleware: Productive Projects and Teams [DL99]*
13. http://www.skype.com/
14. http://bluemic.com/snowball/
15. http://www.logitech.com/en-us/product/stereo-headset-h390

Step Two: Get Comfortable

If this is the first time you and your partner are pairing, you'll need to define your expectations well. You should discuss your experiences with remote pair programming and be clear about things that might make the process difficult for you. If you need to take frequent bathroom breaks, make it known. If you have low bandwidth or your connection is laggy, then tell your partner. Both you and your partner must be comfortable.

With the basic housekeeping out of the way, the next thing you'll need to establish is your goals. Be specific about what code you want to work on, and what you expect to accomplish. Many programmers like to identify specific tests that need to be fixed. Other times, the goal is to write tests to replicate bugs or define new features.

Step Three: Agree on Your Tools

Before you change a single line of code, you and your partner need to agree on the tools you'll use to do it. This includes operating system, text editors, integrated development environments, testing tools, debugging tools, and more. Choosing the right tools might be the most important part of pair programming, and that's why the majority of this book is dedicated to understanding the pros and cons of each option.

Regardless of your choice of development tools, you should be using a version-control system (VCS) for your code. Most VCSs record each commit, along with the author who made it. That is *author*, not *authors*. The first order of business is telling your VCS that you're pair programming. Some excellent tools make this easier, but most of them work only with Git. Two such Git-based tools are Hitch and Sprout.

Hitch is a Ruby Gem for easily setting and resetting your configuration.[16] You can run commands like `hitch jane john` to get started and `hitch -u` to return your configuration to its defaults. The Sprout project from Pivotal Labs is a set of Chef cookbooks that include a pairing recipe.[17]

Tools like Hitch and Sprout are making it convenient to run these commands:

```
$ git config --global user.name "Jane Doe and John Smith"
$ git config --global user.email janedoe+johnsmith@example.com
```

You can always resort to running them yourself. You'll need to do something similar for Mercurial, svn, or whatever VCS you're using. But even after your

16. https://github.com/therubymug/hitch
17. https://github.com/pivotal-sprout/sprout

VCS is set up, you still aren't ready to code. You need some remote-pairing development tools. Fortunately, that's what the rest of this book is about.