PRACTICAL PAIR PROGRAMMING

JASON GARBER

FOREWORD BY
KELLY VAUGHN

Publisher: Jeffrey Zeldman
Designer: Jason Santa Maria
Executive director: Katel LeDû
Managing editor: Lisa Maria Marquis
Editors: Sally Kerrigan, Danielle Small
Technical editor: David Khourshid
Book producer: Ron Bilodeau

ISBN: 979-8-8691-7513-7

TABLE OF CONTENTS

Dedicated to the memory of Michael J. Sharp
(my college pair prankster),
and his pair UN investigator, Zaida Catalán.
May peace and justice prevail where they have blazed a trail.

FOREWORD

There's a common misconception that software engineering is an individual's job: one person at one computer writing code. While companies can successfully operate under this practice, they're missing out on all of the benefits pair programming has to offer.

Two heads are better than one. Collaboration leads to more creative solutions. Fewer mistakes are found in the code. If these are all benefits to pair programming, why don't we see more companies take advantage of it? Maybe because it's a new way of thinking about writing code—and it takes practice to get it right.

Let *Practical Pair Programming* be your guide to integrating this methodology into your everyday process. Jason clearly lays out the fundamentals: what is and isn't considered pair programming, the benefits the practice can bring to your work and team, and how to successfully configure a pair programming environment for both in-office and remote teams.

Jason also provides actionable advice on how to improve the pair programming experience for both yourself and your partner, from the perspective of someone who's spent hours at it himself. The guidance he offers here will provide you with the tools you need to make pair programming in your company both an effective and enjoyable experience. Whether you're a seasoned manager or new to your team, *Practical Pair Programming* will set you up for success.

— Kelly Vaughn

INTRODUCTION

In this book you'll find practical advice on pair programming for the beginning programmer, experienced software engineer, team lead, or engineering manager. Everything I've written comes from my own pairing experience, having worked as a developer on teams that were resistant to Agile and Extreme Programming (XP) practices, up through becoming a manager of a successful software development firm.

Bigger and more mission-critical software projects require larger, more diverse teams. Counter-intuitively, the greater the number of contributors on any one project, the harder it is to coordinate and finish software on deadline. Pair programming perhaps strikes the ideal balance between solo work and software development processes that need a giant flowchart to be understood: not too rogue and not too rigid; nimble but without the risk of having just one pilot in the cockpit.

Can successful software be made by one super-smart solo programmer? Sure! But in my experience, pairing is safer and more enjoyable than working alone. In fact, as I think back to my big mistakes as a programmer or the times my company's projects went off the rails (and we had to write off hundreds of thousands of dollars), it's almost universally attributable to solo programming.

What pair programming isn't

Pairing doesn't mean doubling the cost to get the same output. I like to say it's the same cost for a product that's twice as good! So why do people say pair programming isn't efficient? Developer Sean Killeen posits that they're not thinking of the long-term velocity and overall team effectiveness:

> I think it's because lots of people think short-term when they say "efficient." I program faster alone, and more effectively when not alone because I'm optimizing for the system and team rather than myself. (http://bkaprt.com/ppp/00-01/)

Pair programming also isn't "backseat typing." An unfair balance of control at the terminal will be more annoying than a backseat driver in your car. In Chapter 2, we'll go over ways to ensure you and your pair get equal time driving and keep your commentary constructive.

Finally, pairing isn't about group code review. You may think it's more efficient to have people work individually, then revise with a buddy, but this isn't a creative writing class. Two people need to be involved in the creative process together, from inception through development, to capture all the benefits of pairing.

What pair programming is

Pair programming is two developers working on the same code using shared controls. In Chapter 1, we'll cover the many benefits pair programming offers, including higher-quality code, improved team communication and cohesion, and overall work satisfaction.

When two people are at the controls writing code, twice the empathy is employed for your users, teammates, and future selves. With two different sets of experience, you can train one another. If one of you needs to step away, the other can keep things moving briefly. And when something goes wrong, two brains are better able to solve the problem calmly and professionally.

How this book is structured

I'd recommend you skim the whole book to get the lay of the land. Then feel free to jump around and dig into whichever parts are most salient for you and your team.

Chapter 1 discusses why you should pair program: the benefits to yourself, your team, and your work product.

Chapter 2 lays down some fundamentals of getting started with pair programming—how you make the radical change from solitary,

silent work to collaborating as an engaged, dynamic, and conversational duo, and avoiding pitfalls.

Chapter 3 helps you be the best pairing partner you can be and bring out the best in others so pairing is a sustainable source of productivity and joy. The approaches I present will help you become a better collaborator, even if you don't end up pairing much.

Chapter 4 deals with the nuts and bolts of getting set up for pairing to mitigate fatigue as best you can. A common complaint is that pair programming is exhausting, due in no small part to a physical or virtual workspace that's not set up with pairing in mind.

Chapter 5 looks at pairing in the context of a team and making it a part of your team culture without triggering a backlash. We then go on to explore ways that pairing on the team can get stagnant and how pairing feeds into a larger concept of the Community of Practice.

By the end of the book, I hope you'll be sold on the value of pair programming, and even more so, internalize the values that make pair programming so enjoyable. I want you to know what it feels like to work with colleagues who are eager to exchange knowledge, care more about quality work than their egos, are reflective and insightful, and want to see everyone's contributions valued. If these colleagues don't sound familiar, this book will help you be an example to your team, help move them in the right direction, or perhaps find a different team of well-adjusted humans that deserve your presence.

Pair programming changes you (in my opinion, for the better). Read on, and I'll show why that is.

1

WHY PAIR PROGRAM?

Together—one of the most inspiring words in the English language. Coming together is a beginning; keeping together is progress; working together is success.

—ATTRIBUTED TO EDWARD EVERETT HALE, NINETEENTH CENTURY AUTHOR, HISTORIAN, AND UNITARIAN MINISTER (http://bkaprt.com/ppp/01-01/)

Computer programming has historically been seen as an individual effort. Time on the old mainframes was parceled out individually, and workers with more standing got higher priority. For decades, the stereotypical office has been made of cubicle walls and PCs—*personal* computers. Not sharing a computer has been an unwritten rule for nearly fifty years.

For some reason, the myth persists that computer people just aren't cut out for working together. Software architect and Agile transformation consultant Allen Holub remarked on Twitter:

> It's interesting that, even though there is hard research showing that most people find pair programming...MORE enjoyable than working alone, a very vocal minority on twitter says that programmers can never work effectively with other people. (http://bkaprt.com/ppp/01-02/)

The prevailing sentiment in the business world seems to be that technologists don't *want* to collaborate. Meetings waste time that could be spent pounding out lines of code. It's true that many programmers treat work like a maximization exercise: get as much compensation for as little risk and effort as possible. The problem I see with this approach is it reduces programmers to little more than machines themselves; even brilliant programmers are likely to plateau under these conditions.

Pair programming as a practice emerged from a subset of Agile methodology called Extreme Programming (XP), a set of human-centered software development practices that offers validation for bringing communication, simplicity, feedback, courage, and respect to your work. In my experience, it's been far more fulfilling than slogging through code on my own; I want to bring my whole self to work and learn from active collaboration with smart people. Beyond personal growth and fulfillment, the shift to pairing also

brings forth improvements in team cohesion, clarity around business goals, and even produces better code. Here's how.

BETTER CODE, BETTER OUTCOMES

If more computing power or faster typing was all it took to ship software faster, you can bet we'd have a supercomputer on every desk and programmers would be hired by typing test, but in reality, a programmer who types 20 percent faster isn't going to be 20 percent more productive. Critical thinking, problem solving, and quick recall make up the majority of our effort. Producing lines of code isn't the objective—it's just the means for attaining the real objectives: correct business logic and intuitive user interfaces.

A second pair of eyes can catch a missing semicolon or find that single-character difference that is failing your test, but much more important than catching typos are the decisions that can't be caught by a linter or compiler. A second brain helps you give good names to classes, methods, and variables and write clearer documentation. It gives the pair enough headroom to think about architecture, apply helpful design patterns, and sometimes even come at a problem from an entirely new angle. It means you'll think through more edge cases in your business logic and have twice the empathy for users as you design how they'll interact with your software. Having twice the brain power is much more important than twice the fingers.

At Promptworks, we've observed that having a pair is not noticeably faster or slower than two individuals working independently over a short period of time (like an hour or four). In a 1999 controlled experiment in a classroom setting, Dr. Laurie Williams found pair programming took a team about 15 percent more developer hours than working as individuals. However, she found defects to be 15 percent lower, design quality higher, and the project completed in 45 percent less calendar time (http://bkaprt.com/ppp/01-03/, PDF).

Pairing helps a software team go faster in the long run because it avoids common hindrances:

- **Coordination costs.** Two programmers working on two copies of the same software have to plan what each will work on, resolve conflicts when they both touch the same part of the codebase, and read the changes that others have made so their future work

takes it into account. With pairing, you're working on the same copy of the software, and you're both aware of how it's changing in real time. There are no merge conflicts between the two of you, nor catching up on what the other did.

- **Thinking through a problem twice.** A solo developer who needs her code reviewed will have to revisit the problem, explain her approach, and rationalize the solution to the other developers. Some of them may revisit the paths that she already eliminated. Unilateral decision-making is a genuine risk, but the need for code review strongly points to pair programming as a more efficient and less antagonistic alternative. Why have two sets of eyeballs looking at the code serially when they could be in parallel? By working together from the outset, you can avoid thinking it through twice and move on to the rest of your long list of features.
- **Throwing away work.** The better designed and built the software is, the longer its life and the more adaptable to changing business needs. Building the components of a sophisticated asset in silos and then expecting the resulting amalgamation to work optimally is a recipe for having to throw it away and build a new solution much too soon. Worse than the cost of replacement is the lost revenue when software can't change fast enough, holds back the workforce using it, or creates a costly security breach.

PERSONAL ACCOUNTABILITY

When pairing, you have a partner who holds you accountable and doesn't let you get away with lazy thinking, uncertainty, or failing to recognize the merit of your accomplishments. You hold yourself to your own professional expectations just by having someone hear your thoughts. It helps you live up to your potential, as software engineer Sarah Mei observed after pairing full-time for two months at Pivotal Labs:

> Fundamentally, what I love about software development is writing code that people actually use. I love to finish things. So anything that makes me feel like I'm really [getting stuff done] makes me incredibly happy.

> *When people talk about pairing, you hear a lot about how it "amplifies" their productivity. I am going to go on record with the truth, however. Pairing does not amplify my productivity. Instead, it erases all the bad habits I have that keep me from being a superstar on my own.*
>
> *When I'm pairing, I can really get shit done. (http://bkaprt.com/ppp/01-04/)*

Pairing is synchronous, so a person—not a deadline—is your reason to show up on time and not zone out as the mid-afternoon doldrums set in. You can help each other remember to take breaks, eat lunch, and time-box your code spikes. Even the most scrupulous developer is sometimes tempted to skip writing tests under pressure or move on before refactoring code.

"Pair pressure"—the best kind of peer pressure!—can help you avoid the bad habits that keep you from being a superstar programmer:

- **Acting on first impulses.** In your zeal to solve a problem, you are likely to try the first approach that comes to mind. By having to jointly negotiate how to approach the problem, you'll evaluate more alternatives within a pair than you would individually.
- **Rabbit holes.** Sometimes you spend too much time digging into a problem that doesn't really need to be solved right away. Your pair can help you see when you're getting in the weeds and excavating pointlessly.
- **Distractions.** Focus and a state of flow are critical to doing your best work. There's always a temptation to scroll through Instagram while the computer does the work, but pairing encourages sustained attention to the work at hand. While waiting for tests to run, you can talk about the problem, discuss what you're planning to do next, or look at the backlog together.
- **Cutting corners.** When your estimates are too optimistic or you feel guilty for having gone down a particularly deep rabbit hole, you may be tempted to make up for lost time by skipping the writing of tests, implementing only the happy path, or not giving attention to security. Your pair can be the angel on your shoulder, reminding you to do what's right.

- **Fear and shame.** Solo programmers lose time worrying whether they're right. They're afraid of looking foolish, making the wrong choice, or overlooking something that might seem obvious. A pair can piece together enough knowledge to feel confident that they're right—or admit when they can't figure it out! The rest of the team is more likely to trust the decisions a pair made because there was already some level of discussion, deliberation, and agreement.
- **Getting too clever.** Pairing can provide a shared commitment to pragmatism and levelheadedness that those who have to deal with your code, tests, and comments later will appreciate. Don't get me wrong—we use funny fake names all over our test code and documentation. But hopefully, your pair will stop you before you name a class `AbstractInterceptorDrivenBeanDefinitionDecorator` (classic Java Spring framework nonsense!) or write your own encryption function (utterly irresponsible foolishness!).

TEAM COHESION

Better code and more productivity should be an easy sell to any boss who needs at least two developers, particularly one who feels anxiety about the team's "bus factor"—a macabre but memorable metric for how concentrated the team's skills and project-specific knowledge are. What happens if the dev who hoarded all database work gets hit by a bus, or bitten by a badger? (My team is trying to make "badger factor" a thing.)

Pair programming reduces the team's reliance on one or two superstars, and even very similar developers who pair will discover strengths and knowledge they didn't know the other had. Other team-strengthening benefits accumulate with each project:

- **Learning new tools.** Over many years, we develop our own code manipulation habits and style. Good pairing can teach you better, more efficient ways to use your tools. If you've used the modal text editor Vim, for example, you can imagine how pairing with another Vim user could make you more efficient (or if you wanted to try Vim, how pairing would make it less daunting).

- **Providing hands-on training.** Like learning to speak a foreign language, you haven't really learned any new software language or pattern until you're able to produce it yourself. Seeing it done isn't enough, but working it out by yourself is often difficult and inefficient. Pairing is the perfect in-between.
- **Developing your own best practices.** You naturally converge on a set of conventions and good habits without needing to write a style guide or have a manager dictate best practices (a.k.a. "the way we did things when I was a developer").

With the expanded knowledge base that comes as a side effect of pair programming, every member of a team can become comfortable working in any part of the app, or at least be intentional about learning what is needed to become so.

PAIRING KEEPS YOU ENGAGED AND FULFILLED

It's great that pairing makes for better code, air-tight business logic, more intuitive user interfaces, less time being sidetracked while simultaneously building up your team—but even if none of that were true, I would still want to pair because it's fun!

Maybe it's just my work style, but I like having someone with whom I can process thoughts. I like piggybacking on my partner's ideas to come up with the best solution. The synergy gives me energy and pairing feels like an activity worthy of my time. I can stick with hard problems longer when I have a partner and accomplish big things with what feels like little effort.

Pairing deepens your appreciation for the work itself. Clients or superiors can appreciate the result but probably won't celebrate the little victories with you along the way. Your partner gives you an incentive to do the little things right and maybe even show off a little bit.

You have more courage when pairing. Software engineer Nadia Odunayo tweeted,

> You need to play around with stuff and do silly things to learn. There's a whole class of actions you just don't take when pair programming either because you consciously or subconsciously censor them out…I think it should be the main way all production development takes place. (http://bkaprt.com/ppp/01-05/)

Humans are meant to work together. Being given one task at a time to perform in a solitary bubble goes against how humans have been productive for millennia, so why should we accept this approach in software development—especially considering the high cost of mistakes?

Hopefully, at least a few of these reasons are compelling enough that you want to give pair programming a try. Now, let's see how it's done.

2

GETTING STARTED

You don't take on the 10-meter dive at your first visit to the pool, and neither do you need to plunge into spending all your working time as part of a pair. Pairing is a very different style of working; wade in gradually, timebox your pairing sessions around specific goals, and give yourself (and your teammates) time to adapt.

Nothing about pairing is particularly hard, but an open mind is crucial. In this chapter, we'll walk through some of the tips and tricks that I've seen from successful pairs. Be ready to bring your whole self to the task as you and your partner adjust.

WHEN TO PAIR

It makes the most sense to work as a pair when you're making decisions and dealing with complexity. That encompasses most of what we do as programmers, even when we're not writing actual code, so it might surprise you to hear that full-time pairing isn't the goal. As a practical matter, at my company we aim for 50 percent pairing. Everyone gets exhausted by pairing to some degree or another. Yes, the more experience you have pairing, the stronger those muscles become and the more endurance you'll have. The more of an extrovert you are, the more naturally it comes. But it doesn't always make sense to pair and even when it does, it might not be practical to get a pair's undivided attention.

Not-quite-pairing

The work quality is always better with a pair, but if you can't get a pair or you're looking to start with a few baby steps, these are some alternatives that are better than nothing.

- Having someone talk through a problem with you for fifteen minutes before diving into a solo task can be enormously helpful. Sometimes a brief conversation will save you days of building something the wrong way.
- "Rubber ducking," or talking something through to understand it better, is another thing we sometimes do in our office. Even if the listener has little knowledge of the problem domain (or

happens to be a yellow plastic bath toy), just articulating it can help solve the problem.
- You might try "passive pairing," where both people are working separately but look at the shared screens often enough to keep up with what's happening on the other side. You don't get nearly as much benefit as active pairing because you're not engaged in the same creative act, but at least you can't get as far down a bad path as you might on your own.

Passive pairing is a weak form of pairing and is best suited to programming tasks that aren't super critical. It runs dangerously close to what software engineering and process coach Alex Harms calls "side-by-side pairing," where the advanced developer works on something else and the person who is identified as "junior" does all the work and gets to ask questions once in a while. "We call things like this pair programming," says Alex, "and then people say, 'Pair programming sucks and I don't want to do it.' Duh! Pair programming takes work!" (http://bkaprt.com/ppp/02-01/, video)

These alternatives can be helpful when you can't pair. Just don't mistake any of them for the real deal—if it doesn't require two people's full attention, it's not pairing!

Splitting up: "It's not you, it's the task"

This brings us to tasks that are a poor fit for pairing: Research, reading, chores like manually converting templates between languages, and lots of copying-and-pasting. These kinds of tasks should be few and far between—we are programmers, after all, and automating mundane tasks is what we do—but when they do come up, it's usually obvious that overall throughput will be higher if the job is split up.

The cost of error should be low as well. If you make a mistake while soloing, it should be the sort of tactical defect that's caught by your test suite, compiler, or linter. If you're producing *strategic* defects (going down the wrong path), then you should get back to pairing.

When you transition from active pairing to another form of work, clearly communicate your intentions. It's easy to say, "How about we split up for five minutes to search for answers?" If you

wordlessly start typing on your laptop, how is your pairing partner supposed to know whether you're abandoning them or doing something useful?

When we split up for a few minutes while pairing at Promptworks, we tend to mumble what we're seeing and reading, or at least make expressive noises. By this, you keep tabs on how close each of you feel to an answer and tacitly communicate when it's time to share what you've found. Keep a chat window open to paste links back to the shared screen.

SHARING TIME AND SPACE

When you're on top of your pairing game, you're both so focused on the code that you hardly notice who's typing. It's like you're sharing a brain among four hands, sometimes trading control back and forth several times in a handful of seconds, yet rarely typing over each other. It's powerful and magical in a way that begs for intense gesturing from Jony Ive (http://bkaprt.com/ppp/02-02/, video).

This level of "flow" might seem pretty far-fetched, but I can attest that I often feel it when I get an hour or two into a pairing session with a colleague. To get there, equality and fairness are key—and there are some basic starting points that will help any pair get off on the right foot.

Agreeing on your objectives

You should agree on the objective of your pairing session at the outset. If one of you thinks the goal is to fix bugs as fast as possible and the other thinks it's to learn Python, you might both be disappointed by the lack of progress you make in either direction.

In many pairing sessions, the goal is simply to complete stories quickly and elegantly, but there may be some secondary goals you want to talk about. Perhaps you want to also get better at test-driven development or to clean up some smelly code as you go. Once you have these objectives in mind, take a moment to mention them as you're starting and ask if your partner agrees.

Setting a schedule

Working together with shared controls to accomplish a task means you have to be working at the same time. This can be a challenge when pairing across time zones, when you have different obligations outside of work, or when you simply have different preferences for work time. Be intentional about your pairing sessions: coordinate starting and ending times with your pair so you can work effectively and negotiate schedules with family and extracurriculars in mind.

A *laissez-faire* attitude to scheduling can result in missed expectations and lots of soloing by default. When less-attached team members work whenever they want, the team members who are also parents and caregivers may feel marginalized if they miss out on the best collaboration moments.

Taking turns

Dynamic, effective pairs feel empowered to call out disparity: "You've been driving (typing) for a while. Mind if I have a turn?" When my partner says this, I don't hear, "Gosh, you're such a keyboard hog!" but more like, "I feel bad that you've been typing for so long. You must be ready for a break." Speaking for myself, I'm always relieved to pass the baton.

While you're still getting comfortable with pair programming, you might use a timer to take turns driving; this ensures equal time at the controls and helps break any habits of being a keyboard hog or a comfortable spectator.

You might also trade control by tying each turn to a cycle like that of test-driven development (TDD). Under TDD, you first write a test for the tiniest change in behavior, watch it fail, then write the minimum code that makes the test pass. You next take the opportunity to refactor the production code and the test code, and only then move on to the next small change in behavior. "Ping pong" pairing, as Ward Cunningham describes it, simply alternates who writes the test and who makes it pass. After perhaps many iterations, you will have satisfied all the acceptance criteria for the story and can consider the feature done (http://bkaprt.com/ppp/02-03/).

By working in this fashion, you ensure that both of you not only have equal time driving, but you get to challenge each other to only

make small changes to the codebase, have the test suite passing after every change, ensure high test coverage, and collaborate on refactoring at each step.

Checking in and taking breaks

Spending multiple hours in a car with only one traveling companion tests anyone's patience; the same is true of pair programming. An important part of not getting fed up with a pairing partner is taking breaks, having a snack, getting some air, and generally giving each other space to be quirky, flawed human beings.

Take some of your breaks together to check in on how the work is going. If you're growing into the pairing relationship, reflecting on the pairing process is important. Do you type noticeably more (or less) than your partner, and do they mind the discrepancy? If you're pairing remotely, are you able to hear them clearly enough to communicate, or are they drowned out by background noises?

It's tricky to not get fixated on annoying noises and habits, and trickier still to bring them up in a helpful and sensitive way. Psychologists and counselors have made careers out of this stuff. Don't let it fester; speak to pairing partners about it with empathy and non-judgment. Be aware of subtle hints coming in your direction, too.

Taking a walk around the block is a great way to spend a five-minute break. I cannot overstate the value of looking away from a screen and into the distance. It's not just good self-care, it's pair care!

SHARING POWER

For those of us who recognize the value in working alongside colleagues as diverse as the users we're serving, pair programming can be a fantastic vector for leveling the playing field—if its participants are mindful of the power dynamics between them.

Software engineer Sarah Mei identified the problem with unexamined pair programming in heterogeneous groups:

> Pairing has nothing to say about how to structure an interaction to avoid taking unfair advantage of power dynam-

> ics. One of its basic assumptions is that everyone feels empowered to contribute.
>
> *When this is true, pairing is amazing. When it's not, it's a nightmare. (http://bkaprt.com/ppp/02-04/)*

Tech is a predominantly White cis male industry, and beyond the obstacles underrepresented groups face just getting a job, there are dozens of ways White men have privilege that they take for granted at work every day (http://bkaprt.com/ppp/02-05/).

When you feel comfortable in your pairing environment, check in with your partner rather than assume that your feelings are shared. If you're part of a dominant group, be it according to gender, race, or another characteristic, remember that people in underrepresented groups often find it risky to voice concerns that fall outside of the lived experience of their colleagues.

When you're the old pro

Aside from sociocultural factors, discrepancies in professional experience are a fundamental source of power imbalances. The experience and practical knowledge that comes from more years as a programmer or more time on the codebase can be valuable; drawing on a solution from ten years ago sometimes saves the day. But the less experienced person may also feel behind in skill, not valued, and at a disadvantage in every disagreement; they may not feel comfortable voicing a contradiction. As Malcolm Gladwell pointed out in *Outliers*:

> In commercial airlines, captains and first officers split the flying duties equally. But historically, crashes have been far more likely to happen when the captain is in the "flying seat." At first this seems to make no sense, since the captain is almost always the pilot with the most experience. But…planes are safer when the least experienced pilot is flying, because it means the second pilot isn't going to be afraid to speak up.

The privilege of having more experience can shut down valuable lines of inquiry and stifle innovation with old habits, so make sure

you're deferring to a person with less power or status, even if you have to literally sit on your hands. Give them more time driving and make sure you're using interrogatives, not imperatives. Far better to allow yourselves to explore the wrong path together than to split up in frustration and head off through the wilderness in different directions.

When the "pro" leaves you in the dust

If you find yourself being undermined by your partner, an effective intervention is to paraphrase what's going on: "I feel like you think I'm wrong more often than I am," or "We always seem to try your way first." Describing the imbalance makes them aware without ascribing a motive.

Impact statements can help redirect their attention to how their actions made you feel: "I felt like you didn't think what I had to offer was worth considering." You can also take an inquiring approach to explore where they're coming from or use humor to defuse a tense situation. These are just a few techniques experts suggest to interrupt verbal microaggressions, and with a little adaptation, they can be used to correct pair power imbalances that diminish the contributions of one partner (http://bkaprt.com/ppp/02-06/, PDF).

BRING YOUR WHOLE SELF

I know pairing for the first time can be daunting. You have to decide when to pair, how to pair, agree on objectives, coordinate a schedule, share the keyboard, keep your sniffles in check, take healthy breaks, and make sure everyone feels empowered to contribute. With all this to juggle, wouldn't it be easier to keep working alone and not risk disappointment?

Don't worry; you've got this! It all just boils down to bringing your whole self to the programming task and expanding your definition of success to more than lines of code produced. Remember, it's also about team resiliency, better code choices, and ultimately better business outcomes.

With a few dozen pairing experiences under your belt, it becomes routine enough that you start to see ways it could be even better

with a few tweaks. You'll want to get more out of the experience and give more to your partners. You might be facing your first pair that really tries your patience or having difficulties and wondering if you're the problem. That's perfectly normal! Our next chapter looks at ways you can be an even better pairing partner.

3

BEING A BETTER PARTNER

My grandfather once told me, "Marriage is not about finding the right person but rather about *becoming* the right person." In other words, a lot of a partnership's success is within your control.

Pair programming may not have the permanency and legal implications that marriage does, but it's a pretty intimate thing to let someone else see exactly how you work. Complicating the arrangement is the fact that you might have little choice in who you're partnered with. If you don't trust and respect them, it can be a real struggle. "I only like pairing when my partner is a pleasure to work with," software engineer Ryan Kulla tweeted. "So I try to be too" (http://bkaprt.com/ppp/03-01/).

Working with difficult people is a fact of life, and the earlier you develop the skills to receive their energy and turn it into something positive, the quicker you'll succeed in your objectives and advance in your career. Don't write someone off just because you're having difficulty with them in the moment—people and circumstances can change!

Let's dive into some good pairing habits you can develop that add value to the pairing relationship and leave everyone feeling better for having engaged in it.

THINK OUT LOUD

If you were to listen in on a pair, you'd often hear such things as:

- "What do you think? Is this right?"
- "Do you mind if I drive?"
- "Let's see if that worked."
- "I'm going to grab the mouse."
- "Can I look at something?"
- "I'm thinking the problem is… What do you think?"
- "Go ahead."
- "Sounds like we have different opinions. Let's try yours first."

This level of chatter is in fact a fundamental habit of pairing: There should be a dialogue running almost continuously. The driver (the one controlling the keyboard at the moment) is constantly thinking out loud in a practice called "reflective articulation,"

which helps the navigator (the one not typing) understand what they're doing and keep up with what's going on (http://bkaprt.com/ppp/03-02/, PDF). The navigator should be acknowledging the driver's narrative, filling in the gaps, questioning their choices, noting potential problems, helping remember what that thing over there was called, and helping direct where to go next.

It's hard to remember to think aloud when we've spent so much of our lives in school and solitary work environments keeping our thoughts inside our heads, but with practice, it can become second nature. Remember: Silence is selfish. Go ahead and share what you're thinking, whether you're the driver or the navigator. If your partner goes silent while continuing to work, "Hey, catch me up with what you're thinking" is a friendly prompt to come back to the pair and contribute to the shared experience.

It won't take long to realize that you and your partner may have completely different styles of thinking through a problem. Because reflective articulation is so important, asking the right questions when you're not following (or agreeing with) your partner is critical.

I've learned from business leaders and psychotherapists how powerful curious, genuine inquisitiveness can be. Often the best solution isn't one either I or my partner thought of initially, but rather a hybrid we developed by piggybacking off each other's ideas and gently guiding each other along with a series of thoughtful questions.

Good questions are open-ended, asked with genuine curiosity, and non-judgmental.

- **Inquire about their reasoning.** If you ask, "Why did you make that choice?" you acknowledge that there were multiple approaches and they selected one, whether they realize it or not. You get to hear what they rejected and why. If they didn't consider anything else, hopefully they'll ask you what other ways *you* see.
- **Consider alternatives.** "How else might we accomplish the same thing?" invites you to come up with alternate solutions. "Is there another name that would be more semantic?" gently questions whether their choice conveys the right meaning (and we know naming things is hard!). "Is there something we could abstract from this?" invites reflection on function/class size and the single responsibility principle.

- **Be open to experiments.** Maybe the alternative your partner just suggested sounds outlandish to you. Don't waste too much time debating hypothetical principles. Save your work and try the alternative on for size. Maybe you'll like it once you see it, and there's little in the computer world that can't be undone or reverted.
- **Ask whether the juice is worth the squeeze.** An important job when you're the navigator is to provide a "subliminal process check" and speak up when you think you're working too long, trying too hard, or going too far on a task. Try questions like, "Is this something we should spend more energy on, or is it good enough? Is there something else could we be doing that's more valuable?" Weighing the value of what you're working on versus the effort required should always be in the back of your mind.

It's important to employ non-confrontational questions to keep yourselves in a critical-thinking space without putting each other in a defensive mode. Always keep in mind the overarching objective and make it explicit when necessary: you're working together to discover the best solution.

CHECK YOUR EGO

When developers fear losing their identity, their personal style, and their sense of control, it presents a real challenge to effective pairing. I've heard many stories of developers intentionally obfuscating their code or adopting an extreme personal style to make themselves more indispensable, perhaps reasoning that if they don't rule their corner of the codebase and defend it from all intruders, they can be replaced easily. Others might go to the other extreme and have a crisis of confidence, thinking they don't have anything to contribute. These are both examples of unmanaged ego, and it's probably the number one threat to successful pair programming.

Pair programming is a practice that helps moderate ego, because we're all simultaneously teachers and students. Learning from a colleague means finding the balance between questioning their assumptions while remaining open to having your own assumptions questioned in turn.

Looking at a language, framework, problem domain, or codebase through the eyes of a newbie can be a real gift; you drop your expectations and preconceived ideas and see things with an open mind, curiosity, and fascination—a state of mind called "beginner's mind" in Buddhist practice. You notice what's going on, try to see the details, and don't take anything for granted. A beginner exposes where the code doesn't adhere to the Principle of Least Astonishment (POLA), which is when a piece of code does exactly what you'd expect it to do from reading over it.

If you suffer from too little confidence, pairing can be an opportunity to see inside someone else's work and realize that they really aren't any smarter, faster, or better than you. "I find that pairing regularly is great for my confidence, getting to see everyone else typo and Google stuff too," programmer Danielle Sucher tweeted (http://bkaprt.com/ppp/03-03/).

Wherever you are on the software engineering growth curve, you're probably the best person to teach the knowledge you recently acquired to the folks just behind you. It's a gift for you as well, because by articulating the thinking, circumstances, or history behind why things are done a certain way, you'll get an even better grasp yourself on the principles you've already learned.

Growing as a person who can pair confidently with anyone requires introspection, acceptance, and commitment to the practice when it would be easier to quit. In the words of IBM CEO Ginni Rometty: "Growth and comfort don't coexist" (http://bkaprt.com/ppp/03-04/).

Of course, the only person over whom you truly have control is yourself; no matter how much humility and non-defensiveness you model, you may still find yourself paired with someone whose ego becomes a real problem. This brings us to another important practice in pairing: getting comfortable with giving and receiving feedback.

GIVE AND RECEIVE FEEDBACK

Each member of the team needs to be open to giving and receiving critical feedback. Being a good pairing partner means staying engaged, wrestling with interpersonal challenges, and not checking

out and walking away when you experience adversity. Yes, it would be easier to hide behind your own screen and ice out a difficult collaborator, but you're just putting off a larger conflict.

In my opinion, honesty and kind-but-fair micro-confrontations are better than repressing resentment. I believe (or at least operate as if) everyone is trying their best and would want to improve if given the right information. I've found success with a few different approaches:

- **Soften the criticism by asking permission first.** "Can I offer some feedback?" you might ask. "Can I bring up something I noticed?"
- **Frame it as an observation**. "So I noticed we've been leaving a lot of comments about what the code does. I wonder, is that something we should be doing?" (Even if you're sure it isn't, make sure you ask sincerely.)
- **Emphasize what you would improve, not what went wrong**. "Next time, I would like to spend more time driving, because I need to get a better feel for where things are in the codebase." Or perhaps, "Next time we should try ping pong pairing, so we each have equal time driving. Would you be open to that?"
- **After a pairing session, conduct a nano-retro** to quickly evaluate it and make improvements. On the count of three, each person scores the session on their fingers from zero to five. Take turns sharing what improvements would have made you give it a five and record the improvement ideas for the next time you pair together.

When you're on the receiving end of critical feedback, it can be hard to take! You may feel misunderstood or even attacked. But try to see it as a gift, because new information is the only way to grow and we all have things we can learn about ourselves and our work. Don't explain or defend; just offer a simple "Thanks for the feedback" so the person offering it feels heard. Review the feedback later and decide what action you want to take.

When you implement a suggestion and find it successful, it's good to share it with the team (if not too personal) so others can benefit from your learnings. As a group, talk about what's going well

with pairing, what's not working, and what you'd like to change. Perhaps you need a better hardware setup, a subscription to a better remote pairing tool, an environment with fewer distractions, a less library-like atmosphere, longer or shorter sessions, or different parameters around rotation.

Becoming a better pairing partner is all about curiosity, experimentation, and reflection—thinking about how things could be better and giving it a shot. Not every idea is a good one, but it costs very little to give something a try. Clear, direct communication will establish trust, which is ultimately what makes our pairing highly efficient and our work product the best it can be.

4

THE PAIRING ENVIRONMENT

The key feature of pair programming is shared control, which usually means shared hardware, but might be virtual in the case of remote pairing. With shared control, you can't turn the plane in two different directions at once. If you can both work on different code at the same time, you're not pairing!

Whether you and your pair are a few feet or a few hundred miles from each other, it's important to take the time to get set up properly so you have parity with your partner and prevent computer-related injury or fatigue. The setup matters a great deal for the long-term sustainability of your pairing practice. Be aware of and advocate for your needs and you can enjoy pairing for a long, long time.

IN-PERSON PAIRING: THE IMPORTANCE OF HARDWARE PARITY

At Promptworks, we prefer having the same desk, chairs, monitors, and power supplies at each station for uniformity and interchangeability. We have unassigned, standardized desks at our company so a team can trade partners as necessary and no one has to worry about shuffling hardware around. As Forrest Gump said, "That's good! One less thing."

We determined early on that we didn't want to maintain dedicated pairing computers separate from our individual machines. Some people say that's pairing heresy—that the pair computer should be neutral ground—but we don't think it's worth the IT hassle. Instead, we bring our own laptops and take turns hosting. It lets us share new tools or configurations we're trying out and get to experience another person's setup. That way, we all gradually converge toward the same configuration.

It's important to have the same equipment on both sides of the pairing workstation, particularly the same monitor, because operating at different resolutions will make it difficult to mirror the screen. We've seen some weird monitor behavior at Promptworks, even with two monitors that the operating system ought to see as the same. Using two completely identical monitors, bought at the same time, seems to work best.

We provide cubbies for storing personal items and peripherals near the pairing area. We discourage people from leaving behind chargers, plants, mugs, fans, photos, or anything else that marks your territory; "nesting" discourages flexibility and makes it less likely that you'll change up pairing partners or have someone drop in to pair for a minute.

Arranging the furniture

For ideal pairing, a normal desk with sides or drawers won't do. What you want is a table—or better yet—two small, identical tables that let you sit on opposite sides. Ideally, the environment is quiet enough that you can hear each other easily at a normal conversation volume, but not library-like silence where you'll feel self-conscious for thinking out loud non-stop.

If you share the same monitor, as you might when first dabbling in pairing, you'll want to sit side-by-side and pass the keyboard and mouse back and forth (Fig 4.1). This works well enough to start and many people do it frequently for short stretches. You'll want to get a second keyboard and mouse as soon as possible, though (**FIG 4.2**). Sliding the keyboard or shuffling your bodies back and forth gets old in a hurry and makes it easier for bad pairing habits (such as keyboard-hogging) to form.

Two keyboards mean you don't have to pass or shuffle, but you're still looking at the monitor from the side when they are designed to be viewed head-on. If you can, get two monitors so each person can look at their own screen (**FIG 4.3**). Your back and neck will thank you!

Once you're each looking at your own screen, you may as well sit on opposite sides of the table and offset yourselves, so you can see each other diagonally (**FIG 4.4**). If you use two separate desks, you can slide the right side forward about eight inches, so you're even closer (**FIG 4.5**). The empty desktop space between you is useful for papers and pens, food, and other shared artifacts.

FIG 4.1: The "Please Pass the Butter!" configuration makes it unambiguous who is driving, but passing the controls gets tiring.

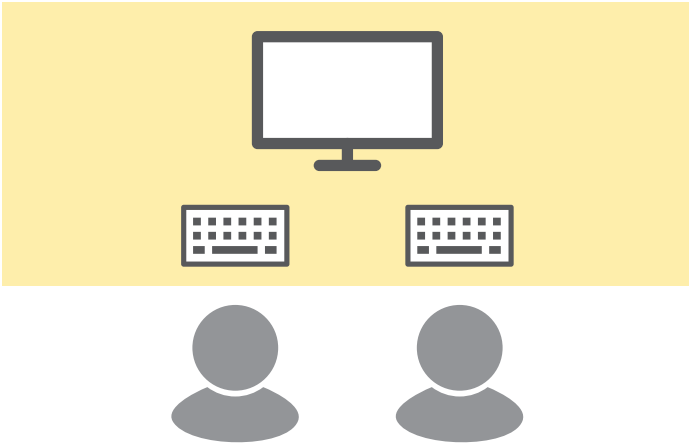

FIG 4.2: The "Tennis Doubles" configuration allows trading control more often, but viewing the screen at an angle can become uncomfortable.

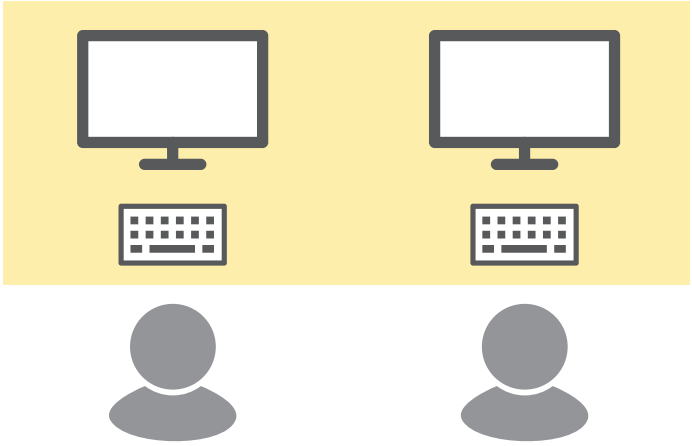

FIG 4.3: The Mission Control setup gives each partner all the peripherals to which they are accustomed.

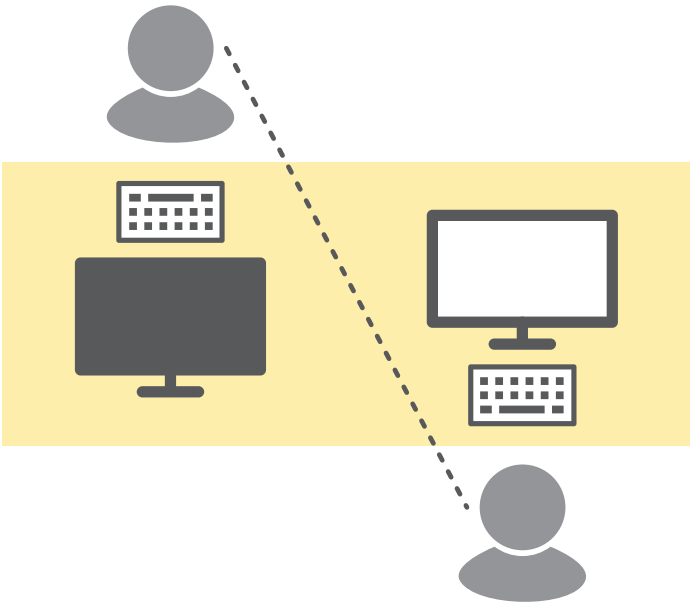

FIG 4.4: The "Mr. Darcy" improves upon Mission Control by allowing eye contact…albeit at a distance.

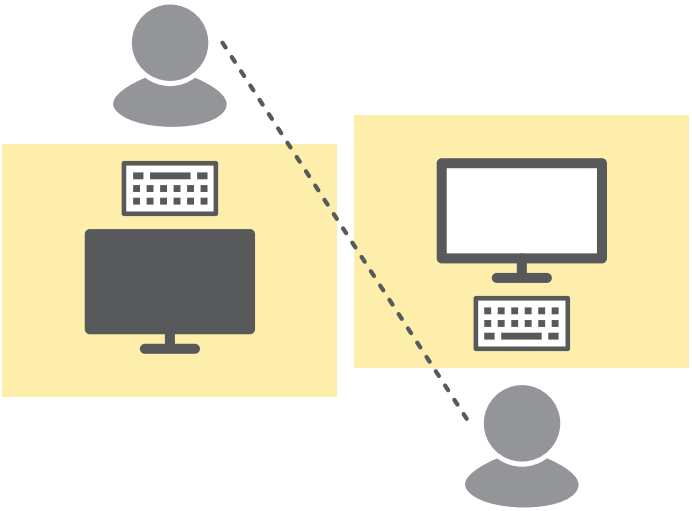

FIG 4.5: The "Tête-à-tête" maximizes human-computer and human-human ergonomics. It also works great in an open office environment with a row of offset pairing workstations.

This is a great arrangement—you don't have to speak loudly, your partner is just off the left side of your monitor, easily in your peripheral vision, and it takes up less space since your partner's desk is partially to the left of your chair. If you mount the monitor on an arm, make sure there's room to position it perfectly at eye height directly in front of your body.

REMOTE PAIRING

Remote pairing can be very effective and an absolute necessity—a few of my friends have done it every day for many years! But in full disclosure, I've found it to be orders of magnitude more difficult than in-person pairing because of the technological hiccups. All sorts of things can get in your way, from connection troubles to software shortcomings. Technical issues can easily result in burnout.

Mid-pairing session is not the ideal time to figure out why your audio drops out or why your mouse is lagging on a shared screen. Here are some things to check out before you decide to conduct your first remote pair programming session:

Network connection

Pair programming requires a high degree of synchronous communication and instantaneous feedback, which demands a good internet connection. Most household internet service is designed for downloading a lot of content, but when pairing, your upstream connection gets a workout as you share video, audio, and your computer screen or control signals. Latency and dropped packets can also be a problem, even if you're paying for a lot of bandwidth. See the Resources section for useful troubleshooting tools.

Get familiar with your router's admin interface and see if you're maxing out your bandwidth in either direction. If upstream bandwidth seems to be the barrier you can't overcome, a shared server or co-editing integrated development environment (IDE) might be better options, eliminating screen sharing from the upstream channel.

- **Enable QoS or WMM,** so your video and audio streams aren't hampered by a software update in the background or a backup on another computer.
- **For the fastest Wi-Fi, set your WLAN to WPA2 and AES encryption only** (the latest standards; try to avoid WPA and TKIP, which are older and less secure). Put your laptop on a 5 GHz channel, and scan for a channel with the least interference.
- **Clear a path to your Wi-Fi.** Avoid Wi-Fi repeaters and radio-frequency-opaque obstacles. When placing access points, experts recommend finding a central location for the Wi-Fi access point placed above head height with no more than two rooms and two walls between the access point and devices (http://bkaprt.com/ppp/04-01/).
- **Radio waves will always lag behind wires, no matter how advanced the technology gets.** If Wi-Fi is your source of latency and lost packets, try connecting via wired ethernet, even

if that means running some Cat 6 cable through the attic or under a baseboard. A voice and data installer (not an electrician) can get it done right for a few hundred bucks and can certify their work by using a fancy network tester.

- **If you have the option, switch to a fiber internet connection** for the very best performance and symmetric connectivity—i.e. as much upstream bandwidth as downstream (http://bkaprt.com/ppp/04-02/). These providers offer aggressive introductory pricing to gain subscribers and recoup their last-mile infrastructure investment, but don't be fooled into buying "High Speed internet" (a.k.a. DSL) when fiber isn't available. They're tricky like that.

Video

Your first thought when considering remote pairing is probably how to share the computer, but don't forget how much of pairing is about human interaction! You need to see and hear your partner almost as well as you do in person for effective interpersonal communication and to catch subtle visual cues like a furrowed brow or a faraway look. Switching windows to look at your pair's face gets old in a hurry; the most ideal way is to have a dedicated screen for the video chat, usually placed slightly to the side of your monitor. An iPad works well.

In Joe Kutner's *Remote Pairing: Collaborative Tools for Distributed Development*, Joe Moore, a developer with Pivotal who has been remotely pairing full-time since 2010, pointed out that being self-aware and disciplined is even more important when you're remote pairing—and video can help widen that channel:

> I find that [video] is invaluable. I want to be able to see people's faces. I want to know if they're confused, if they're laughing, or if they're looking down at their phone.

If you can, opt for one of the premium video conferencing platforms, not a free service like Google Hangouts or Skype. Our company tested many in 2016 and preferred the reliability and quality of Zoom—and still do, as of this writing.

Audio

MacBook Pro speakers have gotten quite good in recent years, but the more isolation your microphone has from your speakers, the less noise cancellation work software like Zoom has to do. Headsets with a boom microphone are most remote workers' preference, though some people prefer a high-quality directional microphone like the Blue Snowball rather than wearing a headset.

Screen sharing

Your video conferencing software might have screen sharing built-in, so it's easy to request or give control over the software you're already used to. In our case, that's Zoom. If screen control is your primary consideration, you can try software like Screen.so, which is optimized for responsive pair programming. Screen gives each person their own mouse pointer, lets you sketch on the screen, and has impressively low latency.

Sharing your screen with your pairing partner is super easy… until it's not. It can completely consume bandwidth and the lag can increase as the hours go by. If you don't already use a text-mode editor and the shell, this might be an excellent time to consider them because the more graphical your tools are, the more difficult remote pairing can be. Sharing a terminal and programs running from the command line is more reliable and takes just a fraction of the bandwidth.

What you need in order to share computing resources somewhat depends on the kind of development environment you have. When we're remote pairing at Promptworks, we generally use SSH, tmux, and MacVim in text mode for editing code, but we still need screen sharing for the browser and sometimes a graphical git client. This way, both partners can see everything that goes on—copying and editing files, running code, and seeing test output—without having to share an entire screen. Vim stays nice and responsive for both of us, and if the browser lags a bit through screen sharing, it's not a very big deal.

When we're developing mobile apps, we generally use command-line runners to build the apps, but have to use screen sharing

for the mobile device simulator and the occasional visit to XCode or Android Studio.

Collaborative editors

Your code editor or IDE can facilitate collaborative editing by connecting to your pair's editor over the internet. Once linked, they work together to share the same editing environment—either partner can edit and run code as if they had a physical pairing setup or screen sharing, except that each person can have their own editor configuration, color scheme, shortcut keys, and keyboard layout.

The most prevalent editor is VS Live Share, available for Visual Studio and VS Code. It comes with a shared terminal, co-debugging, and localhost tunnels out of the box. You'll need to install additional extensions for sharing a browser, git, audio chat, and so forth, but it's a lot more responsive that screen sharing and could replace most of it. We've sometimes experienced some weirdness when it gets out of sync, so your mileage may vary.

If you prefer editing code in Atom, Sublime Text, IntelliJ, Emacs, or Neovim, check out Floobits or one of the other options in the Resources section.

Shared servers

Some teams like to have a server that pairs log into to pair program remotely. The environment is neutral, already set up with the team's agreed-upon tools. It can be cloned or regenerated at a moment's notice and treating it like a disposable server means you can use DevOps tools to set it up quickly and repeatedly. This helps the team converge on a uniform environment with room for one-off customization that a pair finds necessary for a given project.

With a remote server, you don't have to worry about firewalls, NAT, or that you're working on your partner's computer while they stepped away. Both partners have equal access to the webserver if your app requires one, though you'll still have to figure out how you both interact with it remotely.

Most pairs that use a shared server are using text-based editors like Vim or Emacs and multiplexing their terminals with tmux. This setup uses very little bandwidth, and theoretically, you can still

pair from a coffee shop or airplane. More likely, you're jumping on when your partner isn't available (we don't pair 100 percent of the time!) and want to pick up right where the two of you left off from wherever you happen to be.

If you're using a graphical editor, you might need remote desktop capabilities from your server. The industry term is virtual desktop infrastructure (VDI), and since you'll be screen sharing with remote desktop protocol (RDP) or virtual network computing (VNC), the location and reliability of the provider matters a lot. See more specific hints in the Resources section.

ABOVE ALL: PARITY

However your setup looks, the important thing is that you're ready to focus on getting stuff done when you sit down. Creating the right environment enables successful pair programming, but it doesn't assure success on its own. Alex Harms surveyed seventy-six people to figure out the best hardware setup for pairing and was surprised to find that whether the respondents loved or hated pairing had nothing to do with the furniture or the equipment (http://bkaprt.com/ppp/04-03/, video). It was all about both people being focused on the code.

Above all, make sure your pairing environment puts you on an equal footing so you can focus on the advantages it enables: being confident you're making the right decisions, avoiding distractions, and being encouraged to engage hard problems longer and do the right thing more often.

PAIRING IN THE ORGANIZATION

So far, we've talked about pairing as if it's just you and a partner with total agency in how you do your programming work. In reality, you're probably part of an organization or at least a team that has an opinion on how, when, and with whom you program. You might think you won't be able to pair program in your situation, or that your boss would never get on board with it (though there's a good chance they would love the results). In this chapter, we'll navigate how to incrementally add pair programming to your organization at the right pace and level of subtlety to give it a fair shot.

SITUATING PAIRING IN YOUR TEAM

Before you eagerly dive into pairing, you need to be circumspect: What is the team's attitude toward pairing, and how will it affect your ability to find a partner and successfully work together, even for short stretches? Will it attract resentment from your peers or a skeptical look from anyone in management?

Talking to management

I'll be honest: I'm a manager myself, and I'm most inspired by employees who come to one-on-ones and team retrospectives prepared with questions, ideas, or topics they've been thinking about. It shows that the employee values my leadership and cares about my managerial agenda.

If no one on your team has paired before and you want buy-in from the top before you begin, consider: what are your manager's objectives and priorities when it comes to pairing? Are there larger company values (knowledge-sharing or cross-training) or objectives (reducing churn, defects, helpdesk calls, or missed deadlines) that would support a pairing initiative?

As you identify opportunities around big-picture goals, team improvement, and career development, you can open a conversation about pairing to test the waters.

- **Start from shared values.** You can lay some groundwork just by being a good listener and staying curious about the problems management deals with on a regular basis. Learn their thoughts

on coordination costs, time wasted on red herrings, reducing training costs, or the consequences of a programmer's over- or under-confidence. What are their ideas to reduce wasted effort and improve output quality? Don't get too evangelical or push any agenda—but do pay attention to opportunities where pairing might offer advantages.

- **Tap into their experiences.** Your manager has probably been building software for a long time. What have been their experiences with doing code reviews, talking through problems with a coworker, or sharing creative control? How have they seen cross-training or mentorship work best?
- **Take interest in sharing knowledge.** A manager who is responsible for personnel development should be looking for ways to spread the knowledge and experience that more senior team members have. You can suggest pairing part-time—either with someone more junior than you or with someone you'd like to learn from—as a career development opportunity. You'll either look generous with your knowledge and time or hungry to learn, both of which are attitudes most managers want to encourage if they're even remotely serious about developing a robust and effective team.
- **Emphasize the value of risk mitigation.** Single points of failure are dangerous, as we discussed with the bus factor. Surely your manager has been left in the lurch when someone quit or took a sudden leave of absence. If you demonstrate attention to the business risks that pair programming mitigates, not only will you support a case for pairing, you'll also build affinity with your manager and show them support. You might also point to research showing improvements in the code as a result of pairing; what are the consequences if your team allows a big mistake to slip through?
- Annual reviews, check-ins, and even one-on-ones are great times to discuss pair programming. With a little prior reflection, you can align it with personal and team goals. Don't let these opportunities to talk to your manager sneak up on you!

Building ground-level support

It's great to work with your manager's blessing if you can get it, but even if you can't (or, more likely, your manager is too busy to think about implementing a top-down, team-wide pairing effort), you might find just as much success by adopting some of the practices of pairing with your colleagues gradually and non-disruptively. The goal is to approach it like an experiment, testing the hypothesis that pair programming produces better software faster than two people would individually.

1. **Find your co-conspirator(s).** Who is most tuned-in to XP and continuous improvement? For teammates who seem open, propose that you experiment together with pair programming. If they're more reluctant, ask if they would be open to helping each other with stories from time to time.
2. **Try a few short sessions with a lightweight setup** (screen sharing or collaborative editor; act like you're remote even if you're close by) to see if you work well together.
3. **Keep the momentum going.** If you're not explicitly pairing, you can start by simply saying, "Wow, I really like working together on stories. Can we do this more often? How about for a half-hour every Friday?" A month later: "This is great! Can we increase our time to an hour?"
4. **Identify some KPIs** (key performance indicators, in manager-speak) that pairing is helping. Productivity is highly subjective, so think about what matters to your boss and your team. Tracking the number of story points completed or defects produced can be meaningful. What matters might be subjective, like familiarity with the codebase or the amount of confidence you feel, but you can still assign it points or a rating scale to measure it.
5. **Start pairing slowly at first**—short, occasional sessions without a lot of pressure. You want it to work, and jumping in too quickly might jeopardize the experiment. Don't overestimate how adaptable you are. It takes time to get used to a new way of working.

Take some notes after each pairing session so you can look back over your progress later. Whatever metrics you're tracking, the more diligently you measure them and track how these KPIs improve as you pair, the better you can defend your choice and get buy-in to make it an official practice on your team. What did you work on? What did you learn? What domain knowledge, tools, or parts of the system did you become acquainted with because of your partner? You might be enjoying pairing, but if it's not leading to quality and productivity gains, you'll have difficulty getting the rest of the team or your manager on board.

GROWTH AND ADVANCEMENT

If pairing does take hold on your team, you might have some new problems to manage: getting stuck in a rut or having overly prescriptive pairing.

Pair rotation

We've talked about how pairing eliminates silos, but if two members of the team get in a rut of always pairing with each other, they can create a silo of their own. No matter how well-intentioned the rationale may be, any imbalance in pairing frequency is leaving some knowledge on the table.

To combat this, many pairing teams draw a pairing matrix on a whiteboard or keep a spreadsheet that tallies how many times any two people have paired. This way, you can correct for imbalances without having to explicitly make a pairing schedule. You can see who pairs together a lot and steer toward the less common pairings when you have the freedom to do so.

Pair rotation sounds simple enough in theory, but in practice, it's a lot harder than you'd think. Do you break up a pair in the middle of a long-running story? There are pros and cons. Do you have an anchor that sees the story through to the end, or does that create a power imbalance because the anchor becomes the de-facto owner and will probably end up driving a lot? These are things to work out in your retrospectives. Remember, the team should be willing to try anything once, as long as it's a small change that doesn't kill

your productivity. Once you've tried something, be honest about how it compares and whether it's worth continuing.

Avoiding institutionalization and calcification

You may have fought a long hard battle to get pairing accepted among your team, but now you have a new challenge: resisting central planning and rigid institution of pair programming. Among managers, a common reaction to a new way of working is to mandate, manage, and control it. It's pairing by fiat: Somebody in charge makes a plan that assigns who is pairing with whom and on what story. They define exactly how pair programming works, so everyone will do it right. A pairing schedule goes on the wall, and procurement gets a purchase order for ten chess clocks.

In some organizations, this is the best you can hope for, but the ideal is to self-organize based on what needs to be done. It's a bit like the difference between project planning by Gantt chart versus an Agile or Scrum board. The former doesn't take in all the information and can't keep up with changes on the ground as they happen. It gives a false sense of security that all the uncertainty is being managed when really, it's better to define the objectives clearly and let smart people figure out how best to get there.

If your boss is pushing rigid pairing practices, the best way to deal with these mandates is to approach them with an inquisitive mindset. Ask what they're hoping to get out of the pairing schedule. A few weeks later, ask them how it's working. A few weeks after that, ask if they considered a pairing matrix to accomplish the same goal.

If you are the manager who wants your team to pair, be intentional about where you want to see your team headed and the steps to get there, don't push it too fast, and remember that these practices serve the outcomes; we don't serve the practices for their own sake. If there's another way to get there, so be it!

Wherever you fall on the org chart, remember to keep an open mindset of experimentation and "trying anything once." That's how organizations keep up with changing best practices and create a community of continuous learning and self-reinforcement.

BUILDING A COMMUNITY OF PRACTICE

There's an old story of a newly married couple that bakes their first brisket together. The meat turns out delicious, but the husband asks the wife, "Why did you cut off the ends? That's the best part!" She answers, "I don't know; that's the way my mother always made it." They ask the mother why, and she gives the same response. Finally, they ask the grandmother, and she says, "Because my pan wasn't big enough!"

In programming, we also tend to do things just because we've always done them that way, or we followed someone else without really examining if they need to be done or if there's a better way. Pairing gives us the opportunity to see how someone else works, the mental space to ask "why?", the courage to try completely new things with a partner, and the opportunity to share what you learned with the broader team.

Pair programming doesn't only transfer technical knowledge—like how to use a framework, how a codebase was architected, or tools and techniques to make your programming more efficient. It also transfers implicit knowledge—the experiences, routines, strategies, and intuition that members of a team accumulate over the arc of their careers. When this knowledge is shared among a team, an informal Community of Practice (CoP) can form, where individuals lean on each other to share insights, gain knowledge, and advance their practice.

At its core, a CoP is about a self-motivated commitment to making software the best way possible through a set of shared practices and values. It's a critical mass of continuous learners who naturally share their knowledge and humbly open themselves to the wide world of things they don't know.

Pair programming is an important tactic (but not the only one by any means) to develop a strategic CoP among software engineers. Pairing helps develop implicitly shared practices that transcend two perspectives through pair rotation, cross-pollination by proximity, and collective reflection on the pairing process.

CONCLUSION

Until you've had some really great pair programming experiences, you may not believe just how enjoyable and empowering it can feel. You feel like you're on fire—a dynamic duo that is more than the sum of its parts. When you have to go back to programming alone, you feel like you're missing half of yourself.

Like any powerful tool, it matters who wields it and how it's used. Pairing can join people together and make new technologists feel like they have a place in the programming world, and it can also exaggerate power differences and make a person feel like even more of an outsider. Be mindful of the power you have and of those who come to the pairing table with a history of having their contributions ignored and devalued. Be sure to keep a running dialogue, give up control easily, be open to feedback, and reflect honestly on what's working well and what's not.

Even great pairing experiences can still leave you feeling like you just did double the work—perhaps because you did! The intense focus and lack of distractions are tiring, so an ergonomic pairing setup and good pair-care are key to make the practice sustainable. Make sure you have parity in a dozen ways between you and your partner, so it's not too easy for one to dominate—and if possible, get comfortable pairing IRL before you try it remotely.

I hope you find pairing as fulfilling as I have and that you develop your own insights into what makes pair programming work well on your team. My wish is that every team is able to lower their stress and produce better software through broader adoption of these collaborative coding techniques.

In the end, we're all looking to improve ourselves, get more done, and work more happily with others. When you're deeply immersed in the joy of pair programming, you'll start looking for opportunities to pair on non-programming tasks, from unloading the dishwasher to reconciling the checkbook. There's little work that isn't better when shared with a partner!

ACKNOWLEDGMENTS

I'd like to thank Mat Schaffer for turning me on to pair programming and for devising our original tête-à-tête desk setup. Thanks to Dan Shipper for the idea to collect our learnings into a book and to Jon Long for spotting the manuscript's potential and connecting me to the perfect publisher.

The torchbearers of pair programming who were defining and refining the practice long before me, deserve special thanks: Laurie Williams, Ward Cunningham, Kent Beck, Alastair Cockburn, Martin Fowler, Ron Jeffries, Bil Kleb, Bill Wood, Robert Kessler, and Joe Moore—to name a few. I'm grateful that they not only had the vision to practice pairing but also took the time to refine it with their peers and write down what they learned. I owe much of my career happiness to them.

To my editors, Lisa Maria Marquis, Sally Kerrigan, and Danielle Small: thank you for cutting many a silly joke and pointless peregrination to find the few good words that made me sound passably smart. Every first-time author should have such an amazing team. ABA CEO Katel LeDû, thank you for believing I had something important to say and knowing how to turn it into a viable book.

Finally, to my wife Karena: thank you for putting up with this project for the last year. I know how astonishing it must be that I started something without appreciating how much work it would take or the tradeoffs I'd need to make. Your steadfast encouragement and insistence that I close the computer and look at a tree every once in a while, brought me through.

RESOURCES

Agile and XP

In this book, I've assumed you're at least a little familiar with Agile and XP. If you're not or you want to dig in deeper, I recommend the following resources:

- In Rachel Davies's XP talk at NewCrafts 2017, "What Ever Happened to Being eXtreme?", she discussed what modern Extreme Programming includes, where it came from, and its most important aspects. She also revisited old-school XP and what it has to offer even though it's been forgotten by the mainstream (http://bkaprt.com/ppp/06-01/, video).
- Though things like continuous integration (CI), small feature branches, and test-driven development (TDD) are the modern norm of effective engineering teams, Extreme Programming Explained: Embrace Change by Kent Beck and Cynthia Andres remains a great reference and an important read if you're coming from a background where XP principles aren't practiced.
- ExtremeProgramming.org has an interactive map that puts pair programming in the context of an Extreme Programming project. You can zoom out to see its relationship to testing, refactoring, standups, and Agile processes (http://bkaprt.com/ppp/06-02/).
- Build confidence in your test-driven development (TDD) skills and practice ping pong pairing with some TDD katas (http://bkaprt.com/ppp/06-03/).

Pair programming under a microscope

- The book Pair Programming Illuminated by Laurie Williams and Robert Kessler devotes a whole section to various combinations of expertise, introversion, gender, culture, ego, and so on. If you're struggling to make a particular pairing relationship work and the basic habits of humility, confidence, receptivity, communication, and compromise aren't working for you, perhaps consult Chapters 12–23 for advice!

- If you really want to get into the economics of pair programming, the same book has an Appendix B that presents a detailed economic analysis. For a broader look at academic pair programming studies, see this meta-analysis (http://bkaprt.com/ppp/06-04/, PDF).

Managing resistance and anti-patterns

- "Does Pair Programming Have to Suck?", Alex Harms' talk at Ruby Midwest 2011, is an accessible and frank look at how pairing can either go really well or really badly, depending on how it's practiced. (http://bkaprt.com/ppp/06-05/, video).
- "10 Reasons Pair Programming Is Not For The Masses" by Obie Fernandez looks at why pair programming, though highly effective at Hashrocket, isn't more widely practiced (http://bkaprt.com/ppp/06-06/).
- Framed as "etiquette," this article by training site Techtown lists sixteen behaviors that help pair programming go well (http://bkaprt.com/ppp/06-07/).
- The Recurse Center's social rules (not to be confused with their Code of Conduct) help create a "friendly, intellectual environment where you can spend as much of your energy as possible on programming" (http://bkaprt.com/ppp/06-08/).

Code quality

These books will help you be a better navigator as you continuously review the code you're writing.

- Practical Object-Oriented Design by Sandi Metz (http://bkaprt.com/ppp/06-09/).
- Clean Code by Robert C. Martin (http://bkaprt.com/ppp/06-10/).
- Agile Technical Practices Distilled: A Journey Toward Mastering Software Design by Pedro Moreira Santos, Marco Consolaro, and Alessandro Di Gioia (http://bkaprt.com/ppp/06-11/).

Pair rotation

- This pair programming matrix Google Sheet created by Pivotal Labs can be used to highlight hot spots if two people pair disproportionately often. Copy the sheet and replace the cartoon characters with your faces or names (http://bkaprt.com/ppp/06-12/).
- Git Pair Trix is one example of an automated matrix that reads the git authors from commit messages in your repository (assuming you're using the pivotal git-pair script) and produces text output showing the matrix (http://bkaprt.com/ppp/06-13/).

Pair code authorship

Git version control is designed for collaboration but assumes that a single person is making the commits. Since that isn't the case when pairing, we might want to give the credit—or blame—to the right people. Here are a couple tools that will let you temporarily make commits as a pair.

- **git-duet:** Both your names will go on the commit messages—one as author and one as committer. You can set it up to alternate who is who if you wish. You can use git duet-install-hook pre-commit to set a hook that reminds you if you haven't specified who is pairing for a while. Note that you have to use git duet-commit, git duet-revert, and git duet-merge to get both names on the commit unless you set `GIT_DUET_SET_GIT_USER_CONFIG` to `1` (http://bkaprt.com/ppp/06-14/).
- **pairwith:** This simple utility just adds a Co-Authored-By line to the body of your commit messages (http://bkaprt.com/ppp/06-15/).

Collaborative editors

- Microsoft Live Share for Visual Studio, VS Code, and Visual Studio Codespaces (which can be used directly in GitHub) provides the most advanced collaborative editing experience, including collaborative debugging and terminal sharing (http://bkaprt.com/ppp/06-16/).

- Floobits enables collaborative editing for Atom, Sublime Text, IntelliJ, Emacs, and Neovim (http://bkaprt.com/ppp/06-17/).
- CodePen (http://bkaprt.com/ppp/06-18/) or CodeSandbox (http://bkaprt.com/ppp/06-19/) allow you to collaboratively edit front-end code in a browser.

Terminal sharing

- tmux is a terminal multiplexer that lets you have multiple windows and panes in one terminal session. If this interests you beyond checking out a primer online, pick up the Pragmatic Programmers book tmux: Productive Mouse-Free Development by Brian P. Hogan, because tmux and the customization thereof is a deep, deep subject (http://bkaprt.com/ppp/06-20/).
- Many people prefer iTerm over the Apple-provided Terminal app. It has more configuration options and integrates nicely with tmux. With this integration in place, you can almost forget tmux is running because you don't have to use tmux commands to manage your windows. Just click, scroll, highlight text, split, and resize windows like you're used to and your partner's will, too (http://bkaprt.com/ppp/06-21/9)!
- tmate is a fork of tmux that makes sharing a breeze. When you start the session, it gives you an SSH connection string you can share with your pair. They just paste it in their terminal and they're connected to your session without any VPNs, SSH tunnels, or fixed IPs (http://bkaprt.com/ppp/06-22/).

Screen sharing

- Zoom is a market leader in video meetings for good reason. Zoom's video and audio compression is fantastic, their echo cancellation is second-to-none, and you can actually see your partner and their screen at the same time if you have multiple monitors. We've done side-by-side comparisons with Google Meet and vastly prefer Zoom. It's free for one-to-one meetings of unlimited length. We have Zoom integrated with Slack, so the `/zoom` command pops open a new Zoom meeting in an instant. From there, you can share your screen and give your partner control. Either person can type or run the mouse, but there's just

one mouse cursor. The shared whiteboard is easiest to use when you have a tablet with a stylus (http://bkaprt.com/ppp/06-23/).

- We used to use Screenhero all the time for remote pairing. It was low lag, included audio, and allowed each person to have their own mouse pointer, which was handy for trading control back and forth quickly. Then Slack bought it in 2015 and turned it into their video chat feature, killing off some of the best functions before removing screen sharing entirely in 2019. Fortunately, the original authors have written new software to fill the Screenhero void. It takes some cues from Zoom (like scheduling meetings, using meeting codes, and showing a live thumbnail of your partner's face) but puts collaborative screen sharing at the center of its feature set. Screen is available for Mac, Windows, Linux, iOS and Android (http://bkaprt.com/ppp/06-24/).
- When Screenhero was dissolved, three French devs built USE Together. It's reasonably priced (free for students!) and is available in an on-premises solution if your organization is cloud-averse. The chat is voice-only, though, so you won't be able to see your partner (http://bkaprt.com/ppp/06-25/).
- If you're in a Mac-only environment, check out Tuple, screen sharing software made explicitly for pairing. It tucks away in the menu bar, automatically enables Mac's do-not-disturb mode, and gives you loads of insight into and control over your stream quality (going up to 5k resolution with very low latency). As of this writing, it's a little more expensive than the others, but highly endorsed by respected engineering teams. The developers say Windows and Linux support is coming (http://bkaprt.com/ppp/06-26/).

Pairing servers

- You can use Amazon WorkSpaces (http://bkaprt.com/ppp/06-27/) to spin up Windows or Linux desktops in a nearby datacenter. There are similar offerings on Microsoft Azure (http://bkaprt.com/ppp/06-28/) and Google Cloud (http://bkaprt.com/ppp/06-29/).
- macOS and iOS developers have fewer virtual desktop options, but a few include MacinCloud.com (worldwide datacenters), MacCloud.me in Cleveland, and xcloud.me in Zurich. MacSta-

dium (US and Europe) is doing some cool stuff with Kubernetes to manage macOS VMs (http://bkaprt.com/ppp/06-30/). If you're doing iOS development, it's not a bad way to go.

Keyboard tools

- Karabiner-Elements is useful for keyboard remapping, needed if one of the pair uses a different keyboard layout (e.g. Dvorak) or is used to just a few keys being remapped—such as swapped Control and Caps Lock or the backtick as escape. I'm grateful that Takayama Fumihiko implemented my feature request to allow mappings to only apply to specific keyboards. It's saved many in-person pairs from fighting or struggling with different keyboard preferences (http://bkaprt.com/ppp/06-31/)!

Network tuning

- PacketLossTest.com will test your upload and download speeds, lost or late packets, and jitter (the variation in packet delay) to help you figure which part of your network connection might be to blame for poor performance (http://bkaprt.com/ppp/06-32/).
- iPerf is a sophisticated command-line app. If you're remote pairing, run it in server mode on one end and client mode on the other. Test with UDP packets to measure jitter. You can use a public server, but they're often busy. `iperf3 -u -i 1 -c bouygues.iperf.fr` is an easy way to start (http://bkaprt.com/ppp/06-33/).
- The Zoom desktop client provides network statistics, which are described in this article. The article lists acceptable ranges for latency, jitter, packet loss, and video resolution and framerate that make good guidelines for any remote pairing with audio, video, and screen sharing (http://bkaprt.com/ppp/06-34/).
- If you're toying with different Wi-Fi routers, repeaters, or mesh networking, the tools and techniques Jim Salter has developed at Ars Technica can be useful, especially if your Wi-Fi has other simultaneous users that put pressure on your pairing bandwidth. Access point placement is way more important than the top speed numbers on the product box or your internet plan for minimizing glitches when remote pairing. He has ten rules

for getting the most out of your Wi-Fi (http://bkaprt.com/ppp/06-35/). If you really want to optimize, his testing tools and techniques can help you figure out the best equipment and layout in your own home or office (http://bkaprt.com/ppp/06-36/).

- At a network level, remote pair programming looks a lot like online multiplayer gaming. League of Legends ranks hundreds of internet providers by latency, packet loss, and jitter and provides many suggestions for troubleshooting your connection and reducing lag (http://bkaprt.com/ppp/06-37/, http://bkaprt.com/ppp/06-38/).

REFERENCES

Shortened URLs are numbered sequentially; the related long URLs are listed below for reference.

Introduction

00-01 https://twitter.com/sjkilleen/status/1144684073119571969?s=21

Chapter 1

01-01 https://books.google.com/books?id=Wg5MAQAAIAAJ&lpg=PA80&ots=jfzNfMhikz&dq=Addresses%20of%20the%20President%20of%20the%20U.S.%20and%20the%20Director%20of%20the%20Bureau%20of%20the%20Budget%20edward%20hale&pg=RA10-PA14&ci=109%2C445%2C785%2C93&source=bookclip

01-02 https://twitter.com/allenholub/status/1144631354044272641?s=21

01-03 https://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.PDF

01-04 http://www.sarahmei.com/blog/2010/04/14/thoughts-on-two-months-of-pairing/

01-05 https://twitter.com/nodunayo/status/686295215532015616

Chapter 2

02-01 https://youtu.be/OQXEzwXtzJ8?t=424

02-02 https://www.youtube.com/watch?v=_ynXKHC9Wo4

02-03 http://wiki.c2.com/?PairProgrammingPingPongPattern

02-04 https://twitter.com/sarahmei/status/991028340571103232

02-05 https://geekfeminism.wikia.org/wiki/Male_Programmer_Privilege_Checklist

02-06 https://academicaffairs.ucsc.edu/events/documents/Microaggressions_InterruptHO_2014_11_182v5.pdf

Chapter 3

03-01 https://twitter.com/rkulla/status/440165393823305728

03-02 https://c2.com/doc/episodes.pdf

03-03 https://twitter.com/DanielleSucher/status/440178048403390465

03-04 https://fortune.com/2014/10/07/ibms-rometty-growth-and-comfort-dont-coexist/

Chapter 4

04-01 https://arstechnica.com/gadgets/2020/02/the-ars-technica-semi-scientific-guide-to-wi-fi-access-point-placement/

04-02 https://broadbandnow.com/guides/dsl-vs-cable-vs-fiber

04-03 https://youtu.be/OQXEzwXtzJ8?t=326

Resources

06-01 https://vimeo.com/221024846

06-02 http://www.extremeprogramming.org/map/code.html

06-03 https://kata-log.rocks/tdd

06-04 https://www.idi.ntnu.no/grupper/su/publ/ebse/R11-pairprog-hannay-ist09.pdf

06-05 https://www.youtube.com/watch?v=OQXEzwXtzJ8

06-06 https://blog.obiefernandez.com/content/2009/09/10-reasons-pair-programming-is-not-for-the-masses.html

06-07 http://techtowntraining.com/resources/blog/etiquette-for-pair-programming

06-08 https://www.recurse.com/social-rules

06-09 https://www.poodr.com/

06-10 https://www.oreilly.com/library/view/clean-code/9780136083238/

06-11 https://leanpub.com/agiletechnicalpracticesdistilled

06-12 https://docs.google.com/spreadsheets/d/17qgykS1zviaHDQQ5-dJ49c1X-0sRf3Lss0q9_R9UASN8/edit?usp=sharing

06-13 https://github.com/thiagoghisi/gitpairtrix

06-14 https://github.com/git-duet/git-duet

06-15 https://github.com/patricksmith/pairwith

06-16 https://visualstudio.microsoft.com/services/live-share/

06-17 https://floobits.com/

06-18 https://codepen.io/

06-19 https://codesandbox.io/

06-20 https://pragprog.com/titles/bhtmux2/

06-21 https://www.iterm2.com/

06-22 https://tmate.io/

06-23 https://zoom.us/

06-24 https://screen.so/

06-25 https://www.use-together.com/pair-programming/

06-26 https://tuple.app/

06-27 https://aws.amazon.com/workspaces/

06-28 https://azure.microsoft.com/en-us/services/virtual-desktop/

06-29 https://cloud.google.com/solutions/chrome-desktop-remote-on-compute-engine

06-30 https://www.macstadium.com/

06-31 https://pqrs.org/osx/karabiner/

06-32 https://packetlosstest.com/

06-33 https://iperf.fr/

06-34 https://support.zoom.us/hc/en-us/articles/202920719-Meeting-and-phone-statistics#h_82592927-e937-43cd-a442-7a913b3d4d4d

06-35 https://arstechnica.com/gadgets/2020/02/the-ars-technica-semi-scientific-guide-to-wi-fi-access-point-placement/

06-36 https://arstechnica.com/gadgets/2020/01/how-ars-tests-wi-fi-gear-and-you-can-too/

06-37 https://lagreport.na.leagueoflegends.com/

06-38 https://lagreport.na.leagueoflegends.com/en/steps

INDEX

L

M

N

O

P

R

S

T

V

W

ABOUT THE AUTHOR

Jason Garber is COO and cofounder of the software firm Promptworks, where he leads internal operations and guides client work. He carries the same entrepreneurial mindset and attention to detail that launched his first company back in 1997, when he was a 14-year-old web developer. He is a passionate advocate for Ruby on Rails, clean code, and automated testing. He lives in Philadelphia.