Get Better Together with Mob and Pair Programming

Unlocking Collaborative Excellence

Kevin Denver | August 2023 | London

What is Pair/Mob Programming?

Pair programming is two people collaborating on the same piece of code at the same time using shared controls (i.e., not Teams).

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. (Garber, 2020)

Mob Programming is when three or more people collaborate to solve a problem together.

"Think about pair/mob programming as a tool in your tool belt."

"It's just people working on code together! There are no special rules!"

"You do not have to pair 8 hours a day!"

What does it look like?

In person

Zühlke and HSBC built the next version of HSBC's retail mobile banking app together. The app will eventually receive a 4.8-star rating and serve 5 million customers in the United Kingdom.

The team paired and mobbed every day to design & implement features, write user stories and release the product.

Pairing was practiced at every level of the delivery. Pairs could be a combination of people from many disciplines, including technical, product, security, compliance, legal, and quality assurance.

What does it look like?

Remotely

Zühlke and DHSC Test & Trace worked together to build the NHS COVID-19 app. The app was used by 20 million people and prevented thousands of excess deaths.

The project was staffed with Zühlke colleagues from almost all locations - Austria, Germany, Serbia, Switzerland, Singapore and the UK.

The engineering team mobbed and paired **remotely** throughout the pandemic. They were able to respond very quickly to complex and rapidly evolving policy requirements.

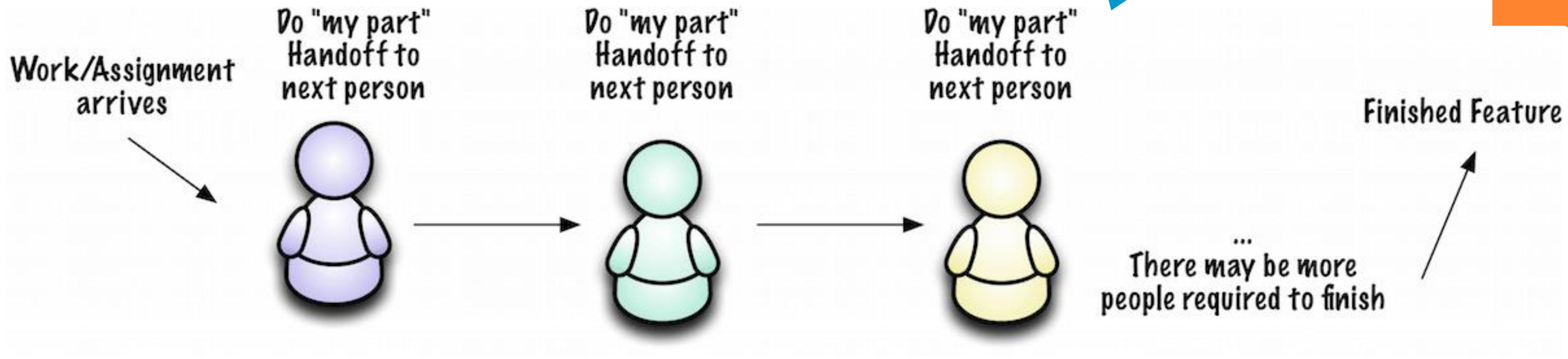
The team used tools such as Tuple and Miro to facilitate remote pair programming and whiteboarding.

Resource Efficiency vs. Flow Efficiency

Only Fred can work on that. He's the only one who knows that code!

You can't take a vacation. That's just before we want to ship and you're the only one who knows that part of the product.

Many features are partly done and too few are complete. (The work in progress is quite high).


Work/Assignment arrives
Do "my part" Handoff to next person
Do "my part" Handoff to next person
Do "my part" Handoff to next person
... There may be more people required to finish
Finished Feature

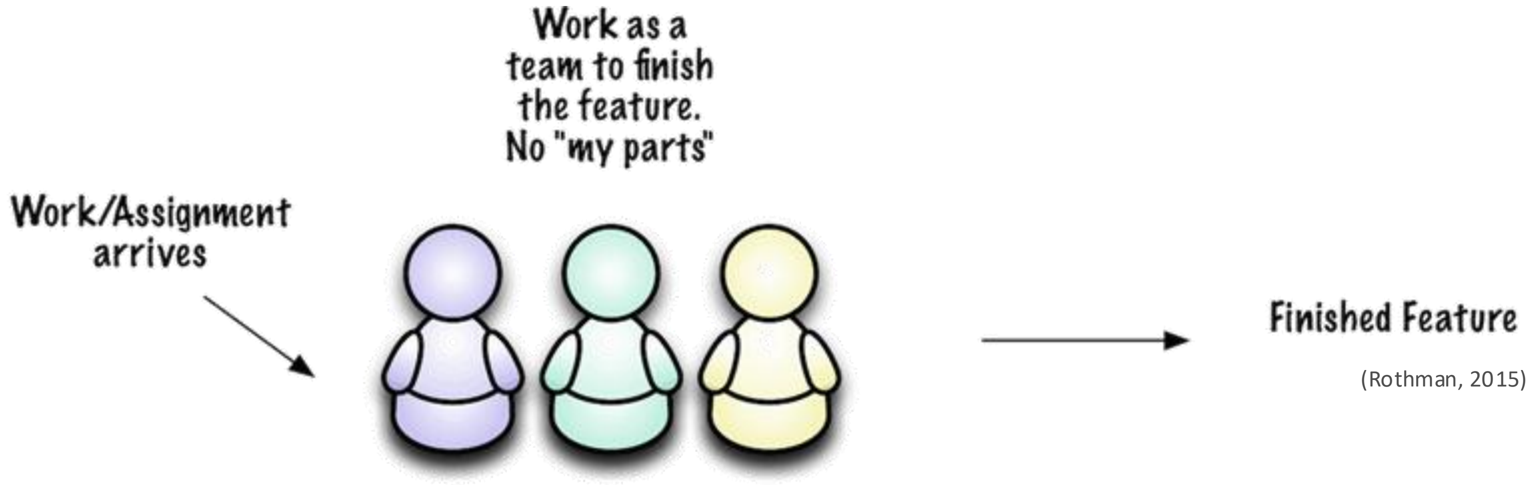
(Rothman, 2015)

Resource Efficiency vs. Flow Efficiency

At its core, Mob and Pair programming are flow centric - you optimise things to get features finished quicker instead of getting them done cheaper.

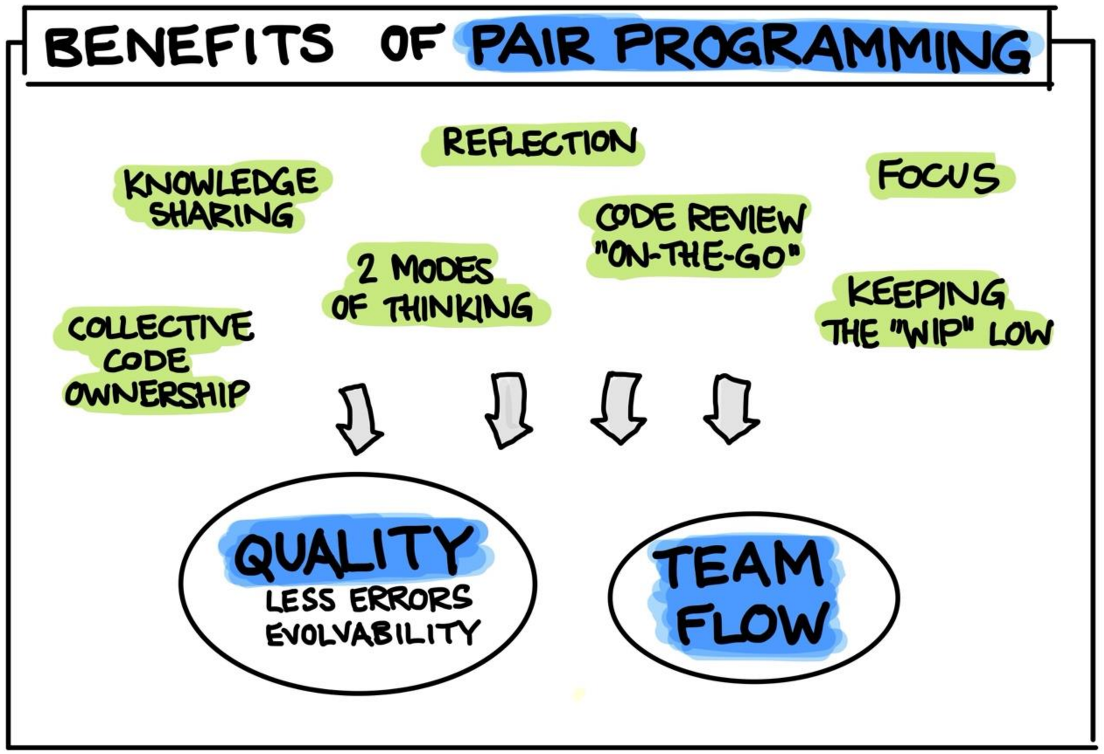
To put it another way, you're optimising for flow efficiency, not resource efficiency. When you work in a resource-efficient way, you get the most skilled person for the specific tasks; this creates experts and bottlenecks.

Optimising for flow efficiency makes sense when the return on getting something to market faster outweighs the cost of getting it developed. As it turns out, a lot of software falls into this category, making Mob and Pair programming a cost-effective solution. (Pearl, 2018)



(Rothman, 2015)

The Case for Pair Programming



Böckeler, B and Siessegger N. (2020)

- **Reflection.** Pair programming forces us to discuss approaches and solutions, instead of only thinking them through in our own head.
- **Code Review.** When we pair, we have 4 eyes on the little and the bigger things as we go, more errors will get caught on the way instead of after we're finished.
- **Collective Code Ownership.** Consistent pairing makes sure that every line of code was touched or seen by at least 2 people. This increases the chances that anyone on the team feels comfortable changing the code almost anywhere.
- **Low "WIP".** Limiting work in progress is one of the core principles of Kanban to improve team flow. Having a Work in Progress (WIP) limit helps your team focus on the most important tasks. Overall team productivity often increases if the team has a WIP limit in place, because multi-tasking is not just inefficient on an individual, but also on the team level.

The Case for Pair Programming

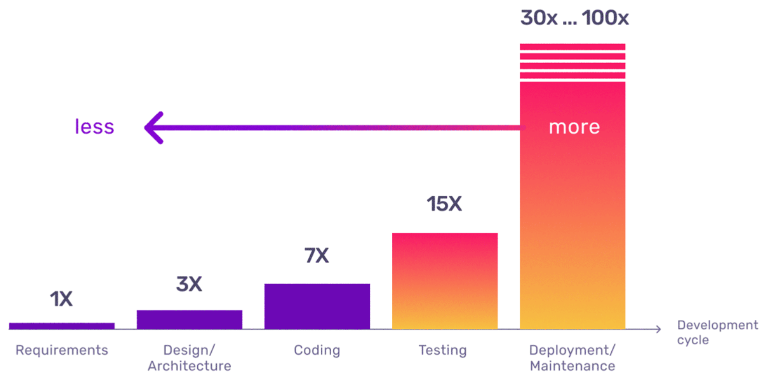
Pairing helps a software team go faster in the long run because it avoids common hindrances:

- **Coordination costs.** Two people working separately 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.

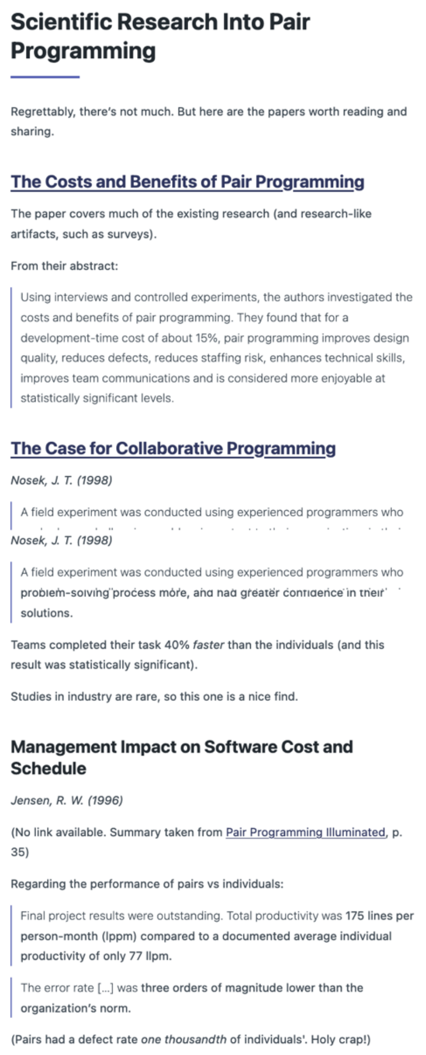
- **Thinking through a problem twice**. A solo engineer who needs their code reviewed will have to revisit the problem, explain their approach, and rationalise the solution to the other members of the team. Some of them may revisit the paths that they already eliminated. Why have two sets of eyeballs 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. (Garber, 2020)

The cost of defects. The cost of defects in software development can be substantial, encompassing both financial and reputational consequences. Defects, whether in the form of bugs, security vulnerabilities, or usability issues, often lead to costly rework, delays in project timelines, and increased support and maintenance expenses.



Scientific Research Into Pair Programming



Scientific Research Into Pair Programming

Regrettably, there's not much. But here are the papers worth reading and sharing.

The Costs and Benefits of Pair Programming

The paper covers much of the existing research (and research-like artifacts, such as surveys).

From their abstract:

> Using interviews and controlled experiments, the authors investigated the costs and benefits of pair programming. They found that for a development-time cost of about 15%, pair programming improves design quality, reduces defects, reduces staffing risk, enhances technical skills, improves team communications and is considered more enjoyable at statistically significant levels.

The Case for Collaborative Programming

Nosek, J. T. (1998)

> A field experiment was conducted using experienced programmers who

Nosek, J. T. (1998)

> A field experiment was conducted using experienced programmers who

> problem-solving process more, and had greater confidence in their solutions.

Teams completed their task 40% *faster* than the individuals (and this result was statistically significant).

Studies in industry are rare, so this one is a nice find.

Management Impact on Software Cost and Schedule

Jensen, R. W. (1996)

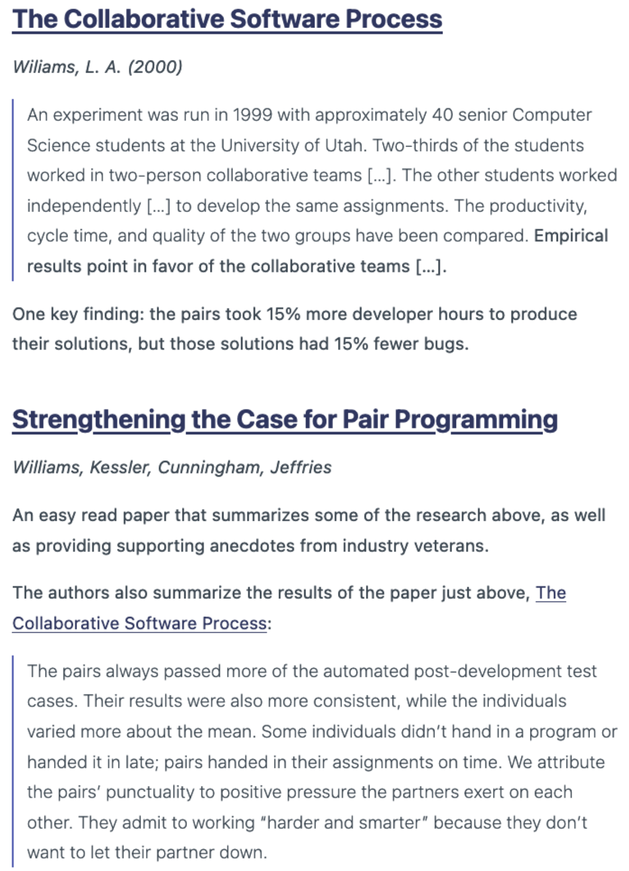
(No link available. Summary taken from Pair Programming Illuminated, p. 35)

Regarding the performance of pairs vs individuals:

> Final project results were outstanding. Total productivity was **175 lines per person-month (lppm) compared to a documented average individual productivity of only 77 llpm.**

> The error rate [...] was **three orders of magnitude lower than the organization's norm.**

(Pairs had a defect rate *one thousandth* of individuals'. Holy crap!)



The Collaborative Software Process

Wiliams, L. A. (2000)

> An experiment was run in 1999 with approximately 40 senior Computer Science students at the University of Utah. Two-thirds of the students worked in two-person collaborative teams [...]. The other students worked independently [...] to develop the same assignments. The productivity, cycle time, and quality of the two groups have been compared. **Empirical results point in favor of the collaborative teams [...].**

One key finding: the pairs took 15% more developer hours to produce their solutions, but those solutions had 15% fewer bugs.

Strengthening the Case for Pair Programming

Williams, Kessler, Cunningham, Jeffries

An easy read paper that summarizes some of the research above, as well as providing supporting anecdotes from industry veterans.

The authors also summarize the results of the paper just above, The Collaborative Software Process:

> The pairs always passed more of the automated post-development test cases. Their results were also more consistent, while the individuals varied more about the mean. Some individuals didn't hand in a program or handed it in late; pairs handed in their assignments on time. We attribute the pairs' punctuality to positive pressure the partners exert on each other. They admit to working "harder and smarter" because they don't want to let their partner down.

When to Pair?

Pairing can be beneficial for tasks involving:

- **Code Reviews, Refactoring and Debugging.** Two people can review code together, catch errors, and discuss potential improvements in real-time. This leads to better code quality and fewer bugs.
- **Complex Problem Solving.** Tackling challenging problems with another person can lead to innovative solutions.
- **Design and Architecture.** Collaborating on designing software architecture and making high-level decisions can result in more robust and scalable solutions.
- **Learning and Knowledge Sharing.**
- **Onboarding New Team Members.**
- **Brainstorming and Ideation.** When you're stuck on a problem or need to produce creative ideas, working with a partner can help generate new perspectives and potential solutions.
- **Time-Intensive Tasks.** Some tasks, such as data analysis or optimisation, can benefit from having two people working together to speed up the process.
- **User Interface Design and Implementation**. Collaborating on user interface design and implementation ensures that the interface is user-friendly and functional.
- **Pairing in Workshops or Training.**
- **Repetitive, Monotonous and Copy & Paste Tasks.** Two people working on a monotonous task can give them the confidence and creativity to automate the job to eliminate it altogether.

Some tasks are a poor fit for pairing such as reading and research.

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?”

Cross Functional Collaboration

Working better with colleagues with other specialisations

Pairing with a Tester. There are several flavours of pairing with a tester. For example, when you have a defect, and the tester shows you how to replicate the defect. It's interesting to see the steps, the journey, and the reasoning behind a flow showing the defect.

Another frequent situation is when you pair with a tester with the purpose of automating checks. You will learn about the test plan, how to think about which checks needed to be automated, how to automate those checks, and many other things. The tester learned how to write better, more expressive automation, and how to structure the test plan so that it was maintainable.

The most exciting flavour is when you have a tester who does exploratory testing through pair programming.

If the tester can review your code as it's being written, they can tell you in an instant if the code you just wrote might generate an issue. The tester explores the code while it's being produced. You cannot get faster feedback than that! The tester learns more about writing code, some design intricacies, and how a programmer thinks while writing code.

Even if it sounds weird in the beginning, I do recommend that you take the first opportunity to pair with a tester who has such skills for exploratory testing. (Bolboaca, 2021)

Cross Functional Collaboration

Working better with colleagues with other specialisations

Pairing with a Business Analyst. You can learn a lot of interesting details about a complex business domain when pairing with a business analyst. Also, it's interesting to learn how to deconstruct a difficult feature into small parts, investigate all the details, and ask questions about all the implications. Usually, pairing with a business analyst improves the analysis skills of any engineer.

Often, while pairing with a BA or other product person, the engineer can get the fastest possible feedback. In a few seconds, you can find out if your logic is good and whether the names of your variables, classes, methods, functions, or namespaces are using a shared, common vocabulary. (Bolboaca, 2021)

At the heart of Domain Driven Design (DDD) is the concept of the ”*domain*”, which refers to the problem space that the software aims to address.

One of the key concepts of DDD is the use of a “*Ubiquitous Language*”. The use of a common, shared vocabulary between developers and domain experts which will ensure that everyone understands and uses the same terms to describe concepts and processes within the domain.

Pairing is a great tool to define and shape the common vocabulary.

Common Arguments against Pair Programming

I need time to think for myself.

I never get to type!

We only need to pair on complex tasks

Pairing is only for engineers

As an introvert, I find extended mix of communication and thinking to be a lot more exhausting than doing those two things separately.

Senior engineers do not need to pair!

Pairing is inefficient. One dev could do this in less time and cost.

Do I need to pair 8 hours a day? 40 hours a week?

Common Arguments against Pair Programming

I need time to think for myself.

Stop pairing and reconvene later

I never get to type!

Try a more structured pairing approach

We only need to pair on complex tasks

Even seemingly simple tasks benefit from pairing

Pairing is only for engineers

NO – You can pair with users, QA's, business analysts and others

Senior engineers do not need to pair!

It's not all about you! Think about growing more inexperienced colleagues

Pairing is inefficient. One dev could do this in less time and cost.

Only if we think typing the code is the "hard" part

Do I need to pair 8 hours a day? 40 hours a week?

NO – Focus on outcome, not output

As an introvert, I find extended mix of communication and thinking to be a lot more exhausting than doing those two things separately.

Try mobbing and see whether this improves your experience

Not Quite Pairing

… some alternatives

Even though the quality of work is almost always better with a pair, if you can't find a partner or you're looking to start with a few baby steps, these are some alternatives that are also worth trying:

- 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. (See Story Kick-Offs)

- "*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 out loud 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. (Garber, 2020)

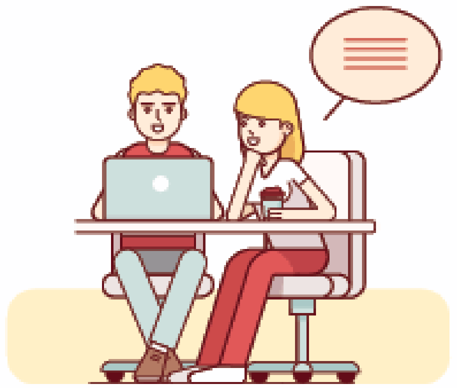
Restricted

Pairing Styles

Unstructured, Driver/Navigator, Ping-Pong

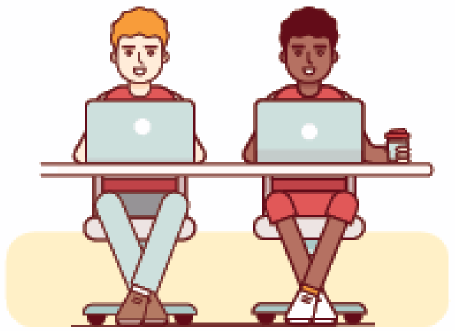
Unstructured

In unstructured pair programming, the developers can trade off who takes the lead, and should discuss decisions about the code.



Driver/Navigator

In the driver/navigator approach to pair programming, one developer sets the architectural or strategic direction, and the other implements these decisions as code.



Ping-pong

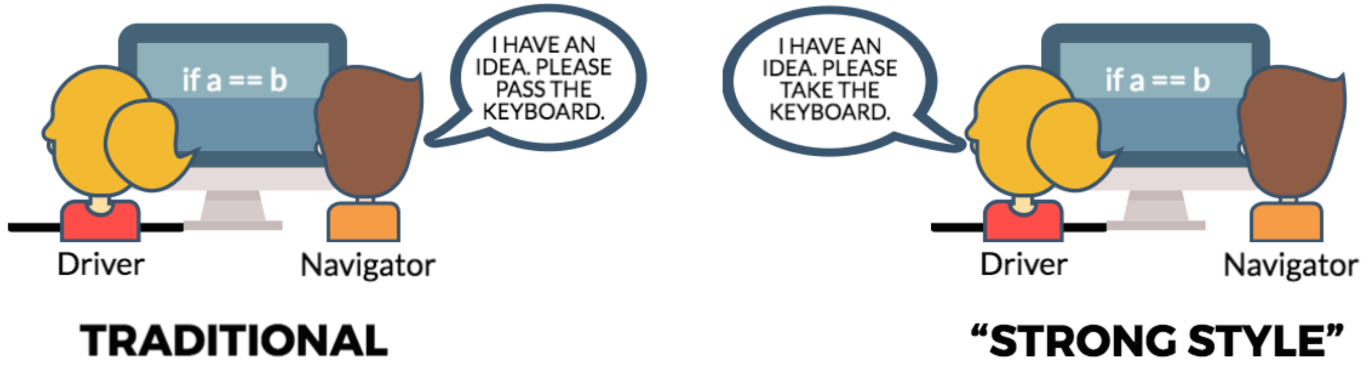
Ping-pong pair programming shifts rapidly back-and-forth between the two developers, like a game of ping pong, where the software is the ball.

Pairing Styles

Strong Style

For an idea to go from your head into the computer it **must** go through someone else's hands.

It keeps the other pairing partner (the Typist) engaged. Instead of being relegated to observing, they are actively participating in writing the code.


if a == b
I HAVE AN IDEA. PLEASE PASS THE KEYBOARD.
Driver
Navigator
TRADITIONAL
I HAVE AN IDEA. PLEASE TAKE THE KEYBOARD.
if a == b
Driver
Navigator
"STRONG STYLE"

It constrains the Navigator to having to articulate their idea coherently and efficiently, so that the Typist can implement it effectively. This develops technical communication skills that lead to a stronger shared understanding on a team. (Sobocinski, 2023)

Pair Programming vs Mob Programming

I've never paired before, should I do that first? While there are benefits to being comfortable with both practices, there is no prerequisite that states you must first master, or even participate in Pair Programming.

I already pair program, why switch to mobbing? Mobs generally have more continuity than pairs. When you're pairing with someone and they get interrupted with a phone call, or a meeting, or anything that requires them to step out, pairing stops.

I hate pairing, will I hate mobbing too? Not everyone likes Pair Programming. With mobbing, the personal interactions are different, the intensity is different, and the way you set things up is different, to name just a few things. So, yes, it is quite possible to hate pairing and love mobbing. (Pearl, 2018)

While it's true that Mob Programming is not the cheapest way to make software, it is cost effective. In fact, there are many benefits of having a group of people work together on one thing.

- See **Resource Efficiency vs. Flow Efficiency**
- **Fewer Key-Person Dependencies.** When you work as a mob, you build redundancy into the group. Having more than one person know how to do things means the impact on the organisation when someone is no longer around is significantly reduced.

Mob Programming

… some practical tips

Being Around to Mob. Collaboration hours consist of establishing a set or core hours where everyone in the team is around to work together. During collaboration time, it's appropriate to mob; outside of collaboration time, it's acceptable to do other things. Having agreed hours for collaboration also removes the pressure from people to mob all the time.

Mob Rustling. Mob rustling means being intentional about starting a mob. Often, all that a mob rustler needs to do is let others know they're starting to mob on something. The number one rule of mob rustling is never force someone to join the mob - encourage, invite, and inspire, but never force.

Mob Fatigue. Sometimes you just need to work on something else. If you're experiencing fatigue because what you're working on is extremely draining, there's a good chance others in the mob are feeling the same way. If this is the case, it may be worth changing what you're working on for a period to give the mob an opportunity to recharge.

Disrupting the Mob When Leaving and Joining It. When stepping back into the mob, do so with as little disruption as possible. Avoid asking questions like, "What did you guys do while I was gone?" or "What have I missed?" These sorts of questions disrupt the flow of the mob because people feel obliged to cover things they've just dealt with and often feel rude if they ignore your questions (like they should). Instead, quietly re-join the mob, and move in as a typist as quickly as possible. As the typist, you'll gain context as the rest of the mob directs the next steps.

The Law of Two Feet. The principle for being in a mob is an adaptation of the "law of two feet." The law of two feet originated from open spaces. In its original form, it goes like this: "If at any time during our time together you find yourself in any situation where you are neither learning nor contributing, use your two feet, go someplace else." (Pearl, 2020)

Collaborating Over Different Time Zones

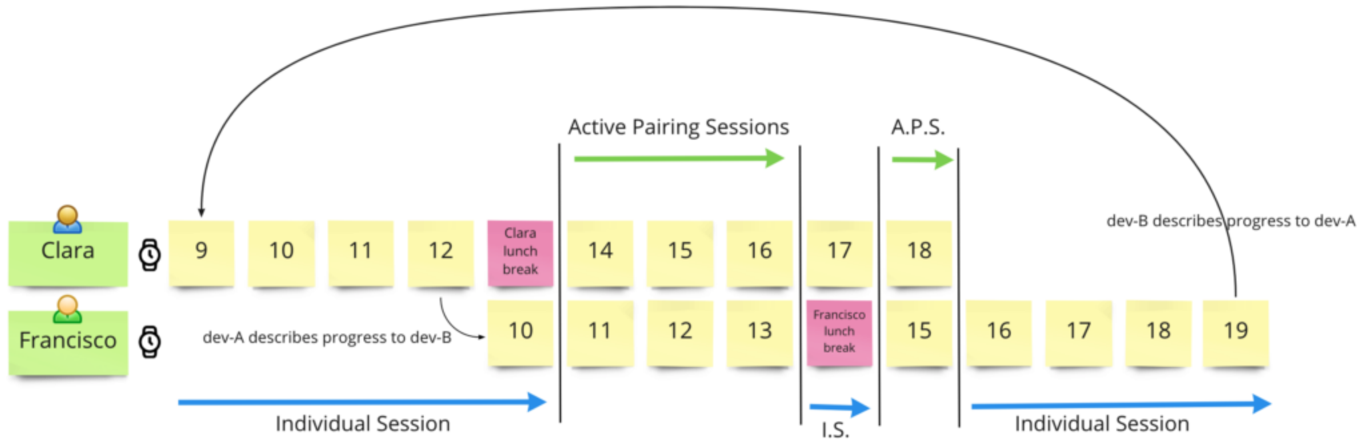
A rolling mob programming session

Collaborating with people in different geographical locations and time zones can be extremely challenging for a team.

A “*rolling mob*” can help to reduce the friction caused by differing time zones.

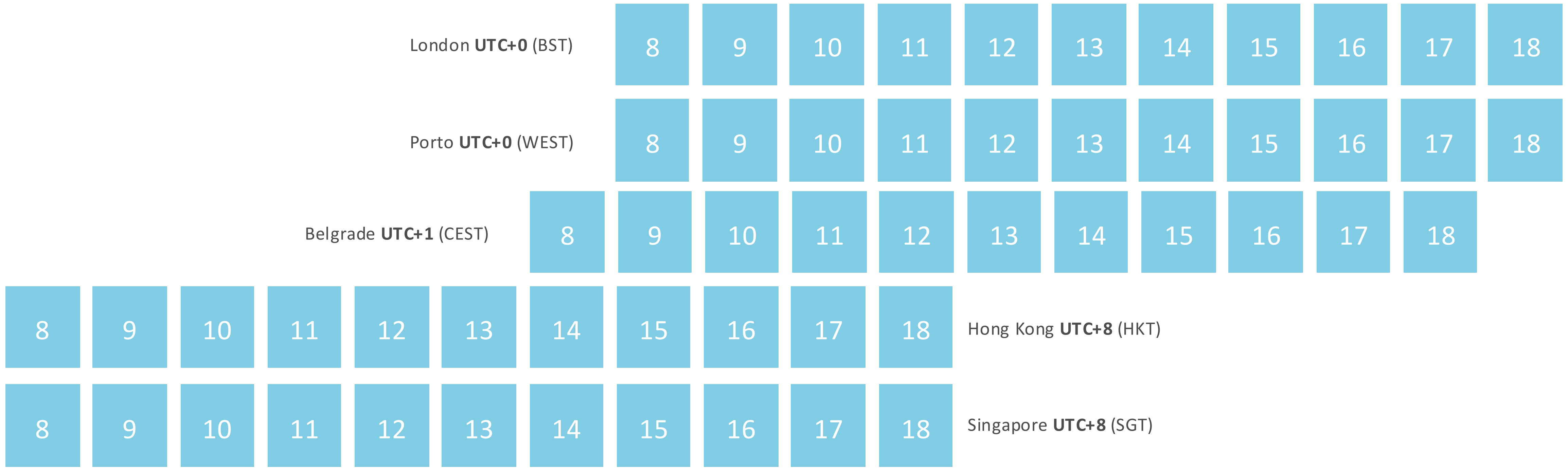
For example, the mob starts collaborating on a coding task with colleagues located in Singapore at 3pm (SGT). After an hour, colleagues from London join the mob at 9am (BST). Team members from Singapore start leaving the mob as their day comes to an end. Colleagues located in London can continue the mob and finish the task.

Alternatively, dedicate overlapping hours as “Active Pairing Sessions”



(LiveRamp, 2021)

Collaborating Over Different Time Zones



London **UTC+0** (BST)								8	9	10	11	12	13	14	15	16	17	18
Porto **UTC+0** (WEST)								8	9	10	11	12	13	14	15	16	17	18
Belgrade **UTC+1** (CEST)							8	9	10	11	12	13	14	15	16	17	18	
Hong Kong **UTC+8** (HKT)	8	9	10	11	12	13	14	15	16	17	18							
Singapore **UTC+8** (SGT)	8	9	10	11	12	13	14	15	16	17	18							

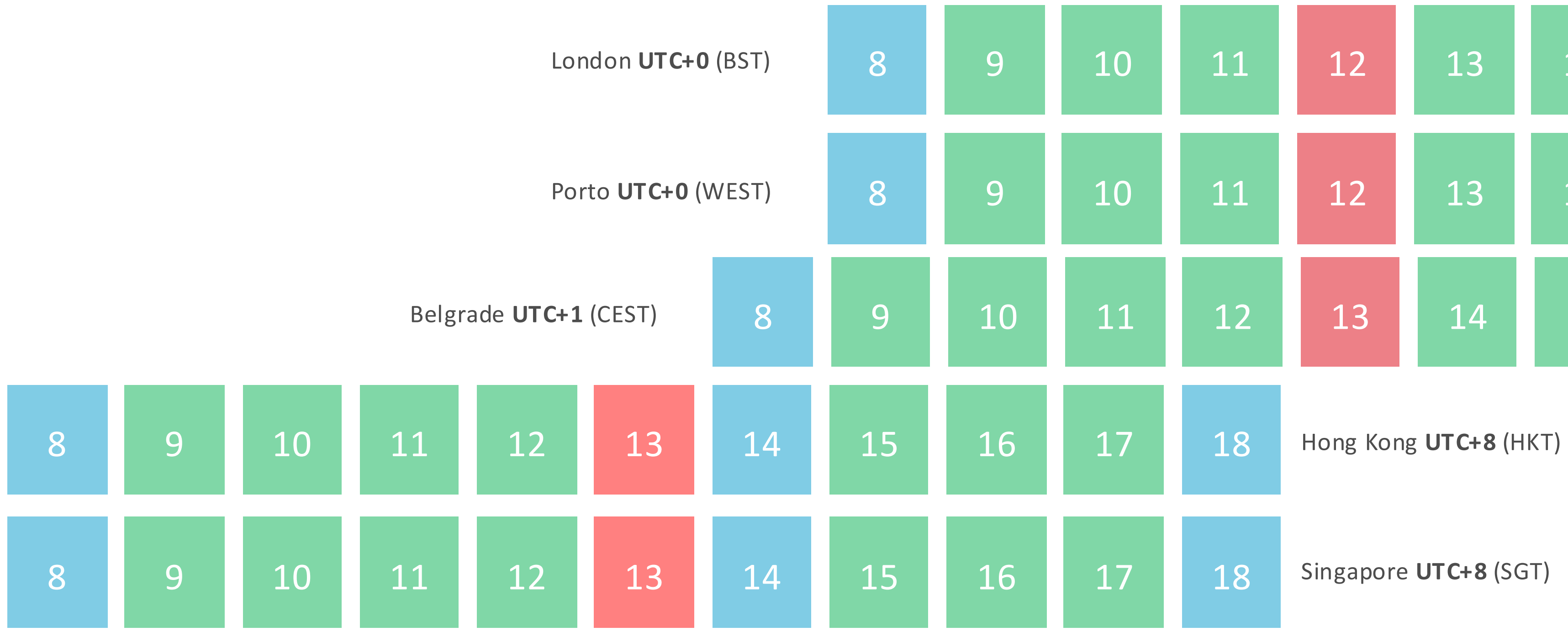
Collaborating Over Different Time Zones

Lunch Breaks

London **UTC+0** (BST)								8	9	10	11	🟥 12	13	14	15	16	17	18
Porto **UTC+0** (WEST)								8	9	10	11	🟥 12	13	14	15	16	17	18
Belgrade **UTC+1** (CEST)							8	9	10	11	12	🟥 13	14	15	16	17	18	
Hong Kong **UTC+8** (HKT)	8	9	10	11	12	🟥 13	14	15	16	17	18							
Singapore **UTC+8** (SGT)	8	9	10	11	12	🟥 13	14	15	16	17	18							

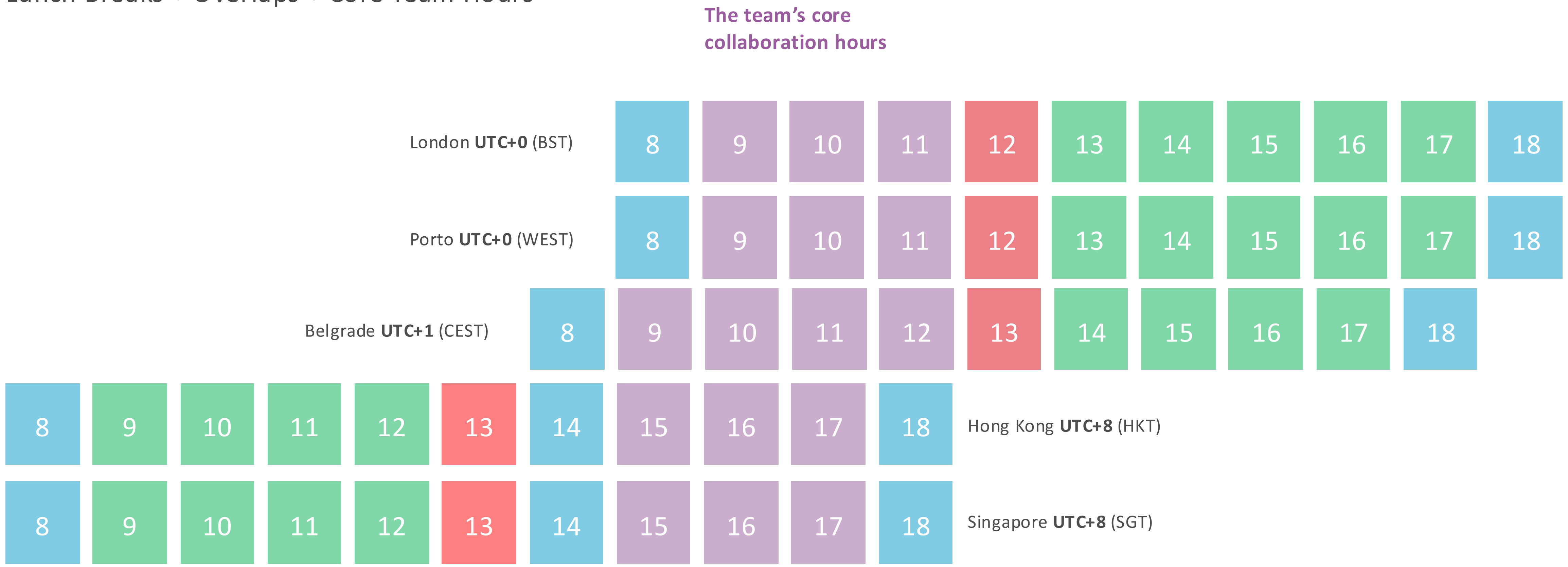
Collaborating Over Different Time Zones

Lunch Breaks + Overlaps



Collaborating Over Different Time Zones

Lunch Breaks + Overlaps + Core Team Hours

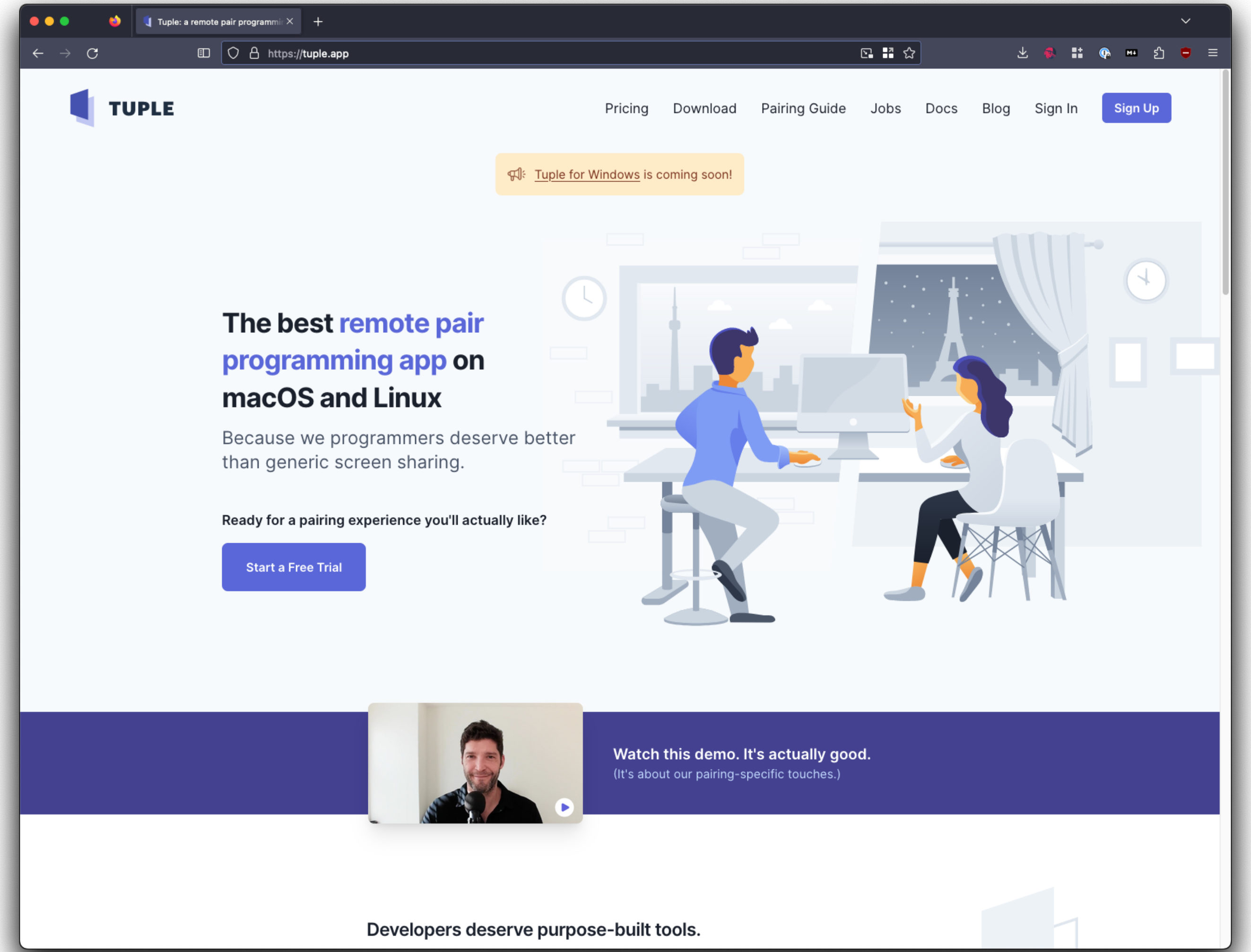


It's important emphasize the importance of a **"One Team"** mindset when working together over different time zones. The core hours are reserved for the team to pair/mob, write user stories, discuss architectures, hold white-boarding sessions, etc.

Pair Programming Etiquette

- **It's ok to not pair 100% of the time.**
- **Focus on outcome, not output.**
- Pairing is a skill and needs training and experimentation.
- Ask for a break if you feel tired.
- Suggest a break if you notice your partner is tired.
- Schedule regular checkpoints for possible breaks/micro retros.
- Stop if you're both lost. Reconvene when one or both have gathered enough information to proceed.
- Consider moving the discussion to the whiteboard.
- Stop if one party is not engaged. If possible, tell or ask why.
- Give your partner a chance to spot their mistakes before correcting them.
- Pay attention to non-verbal cues.
- Not every small decision needs to be discussed.
- Be mindful of only one side making all decisions.
- Use pairing with a less experienced person as a chance to revisit your assumptions.
- Use pairing to incrementally define what good looks like for the whole team.
- Embrace pairing as a learning tool for less experienced developers (make delivery a secondary goal).

Tooling



Tuple for Windows is coming soon!

The best remote pair programming app on macOS and Linux

Because we programmers deserve better than generic screen sharing.

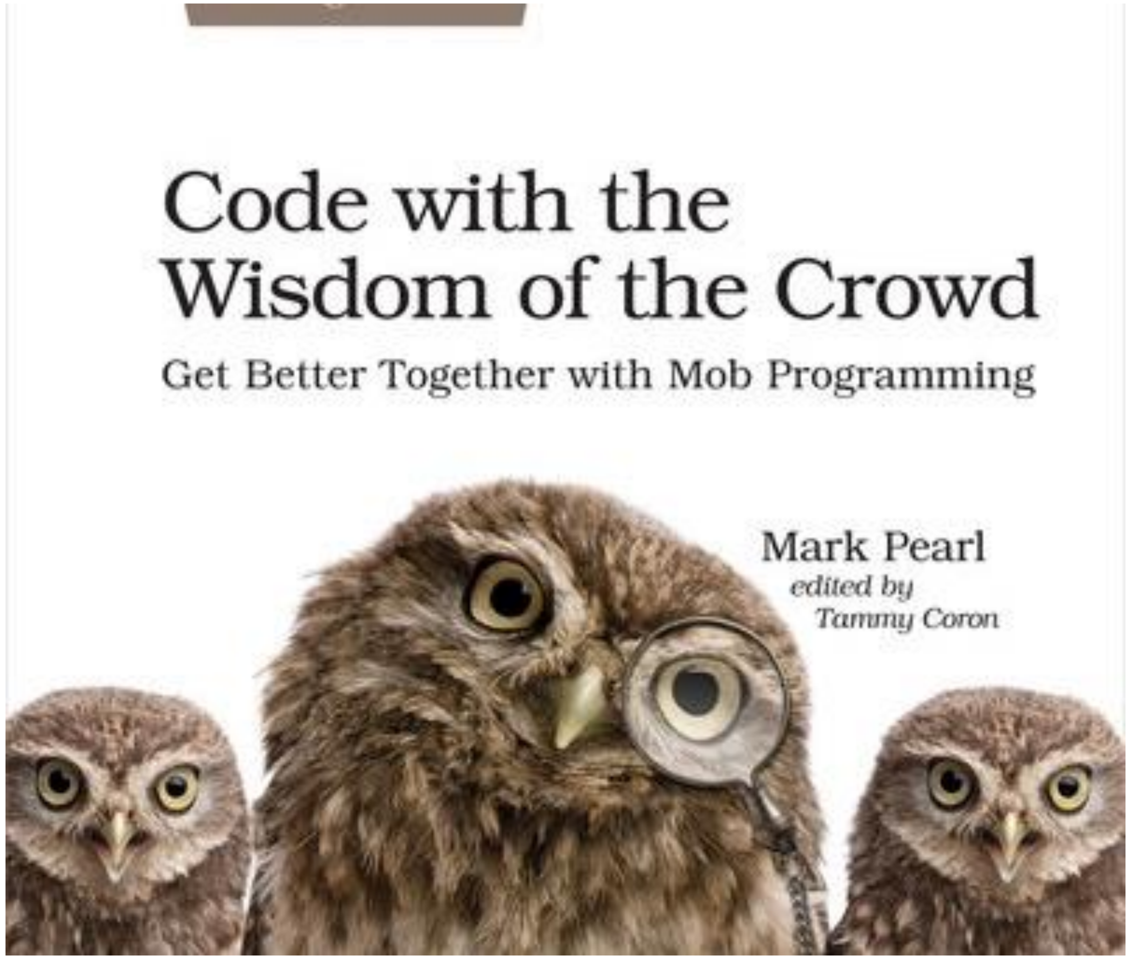
Ready for a pairing experience you'll actually like?

Watch this demo. It's actually good.
(It's about our pairing-specific touches.)

Developers deserve purpose-built tools.

Restricted

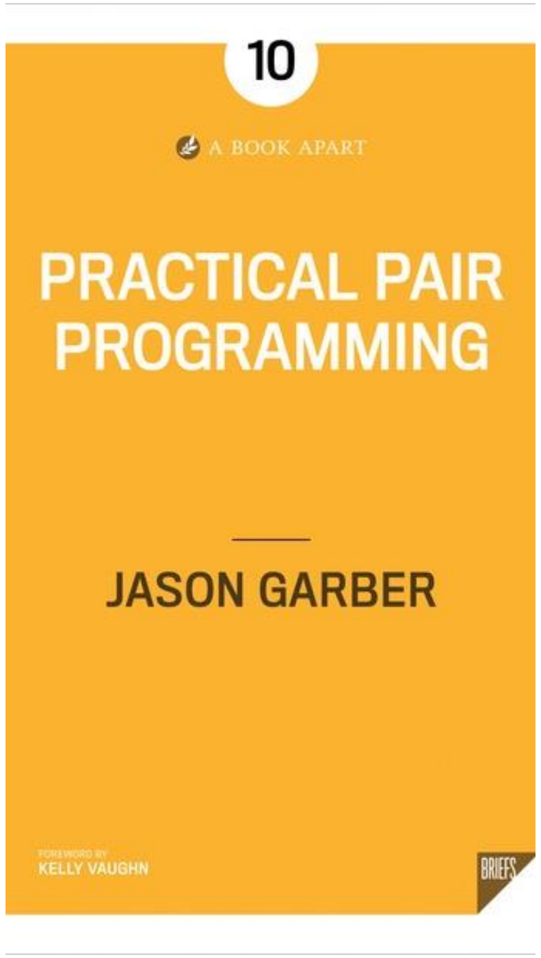
Further Reading


Code with the Wisdom of the Crowd

Get Better Together with Mob Programming

Mark Pearl

edited by
Tammy Coron


10

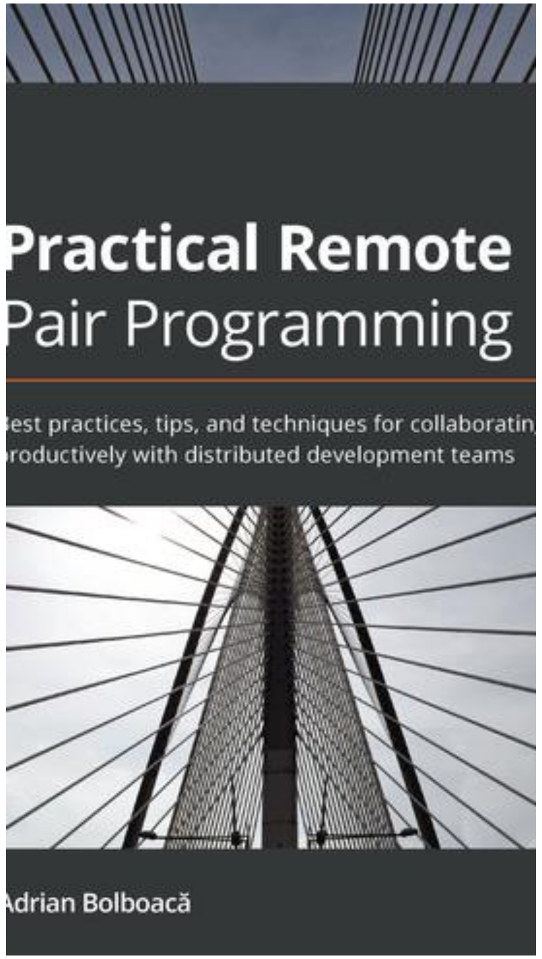
A BOOK APART

PRACTICAL PAIR PROGRAMMING

JASON GARBER

FOREWORD BY
KELLY VAUGHN

BRIEFS


Practical Remote Pair Programming

est practices, tips, and techniques for collaboratin
roductively with distributed development teams

drian Bolboacă

References

- Garber, J. (2020) Practical Pair Programming, New York: A Book Apart.
- Beck, K. (2004). Extreme programming explained : embrace change. Boston, Mass.: Addison-Wesley.
- Böckeler, B and Siessegger N. (2020) On pair programming. Available at: https://martinfowler.com/articles/on-pair-programming.html.
- Pearl, M. (2018). Code with the Wisdom of the Crowd. Pragmatic Bookshelf.
- Rothman J. (2015). Resource Efficiency vs. Flow Efficiency, Part 1: Seeing Your System. [online] Johanna Rothman, Management Consultant. Available at: https://www.jrothman.com/mpd/agile/2015/09/resource-efficiency-vs-flow-efficiency-part-1-seeing-your-system/ [Accessed 28 Aug. 2023].
- Cellan-Jones, R. (2023). Yes, the contact tracing app did work. [online] Rory's Always On Newsletter. Available at: https://rorycellanjones.substack.com/p/yes-the-contact-tracing-app-did-work?utm_source=post-email-title&publication_id=352999&post_id=104634832&isFreemail=true&utm_medium=email [Accessed 28 Aug. 2023].
- Bolboaca, A. (2021). Practical remote pair programming : best practices,tips, and techniques for collaborating productively with distributed development teams. London, England: Packt Publishing, Limited.
- Sobocinski, P. (n.d.). Strong-Style Pairing. [online] spikes.sobes.co. Available at: https://spikes.sobes.co/p/strong-style-pairing [Accessed 31 Aug. 2023].
- LiveRamp. (2021). Is Pair Programming NOT for Geographically Distributed Teams? Think Again! [online] Available at: https://liveramp.com/developers/blog/pair-programming-for-geographically-distributed-teams/ [Accessed 1 Sep. 2023].
- martinfowler.com. (n.d.). bliki: CannotMeasureProductivity. [online] Available at: https://www.martinfowler.com/bliki/CannotMeasureProductivity.html.
- www.functionize.com. (n.d.). The Cost of Finding Bugs Later in the SDLC. [online] Available at: https:// www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc#:~:text=The%20cost%20to%20fix%20an [Accessed 11 Sep. 2023].

Version History

Version	Date	Author	Change
1.0	01.09.2023	Kevin Denver	Initial Version
1.1	11.09.2023	Kevin Denver	Added a slide on scientific research into pair programming. Added an info-graphic on the cost of defects. Added a slide on how to collaborate across time-zones
1.2	13.09.2023	Kevin Denver	Removed pop.com as an alternative remote pairing software because it is not vetted by IT Security
1.3	20.09.2023	Kevin Denver	Added a slide with some practical mobbing tips

zühlke
empowering ideas