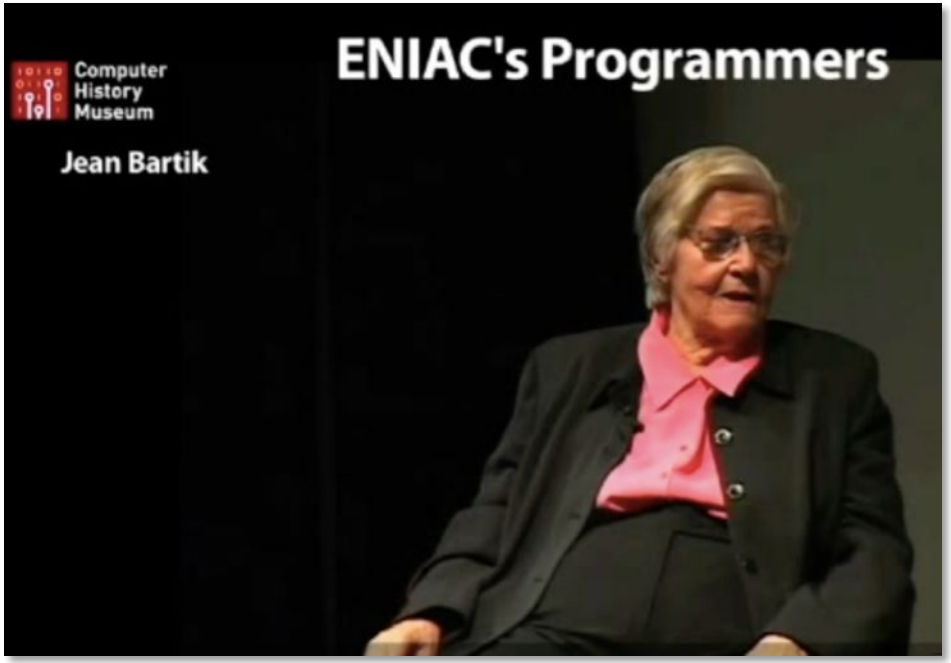

ThoughtWorks®
ON PAIR PROGRAMMING

ThoughtWorks®

WHAT IS IT?

1940s



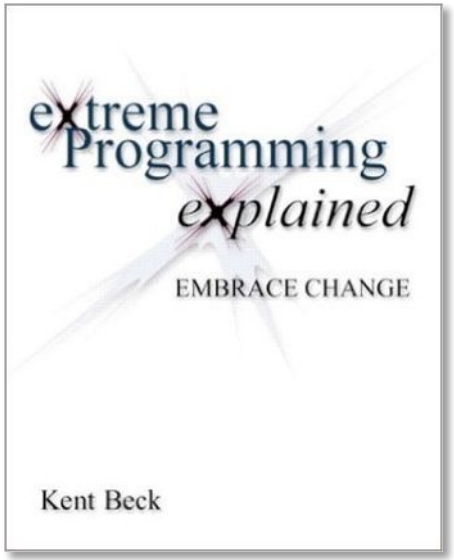
"Betty Snyder and I, from the beginning, were a pair. And I believe that the best programs and designs are done by pairs, because you can criticise each other, and find each others errors, and use the best ideas."

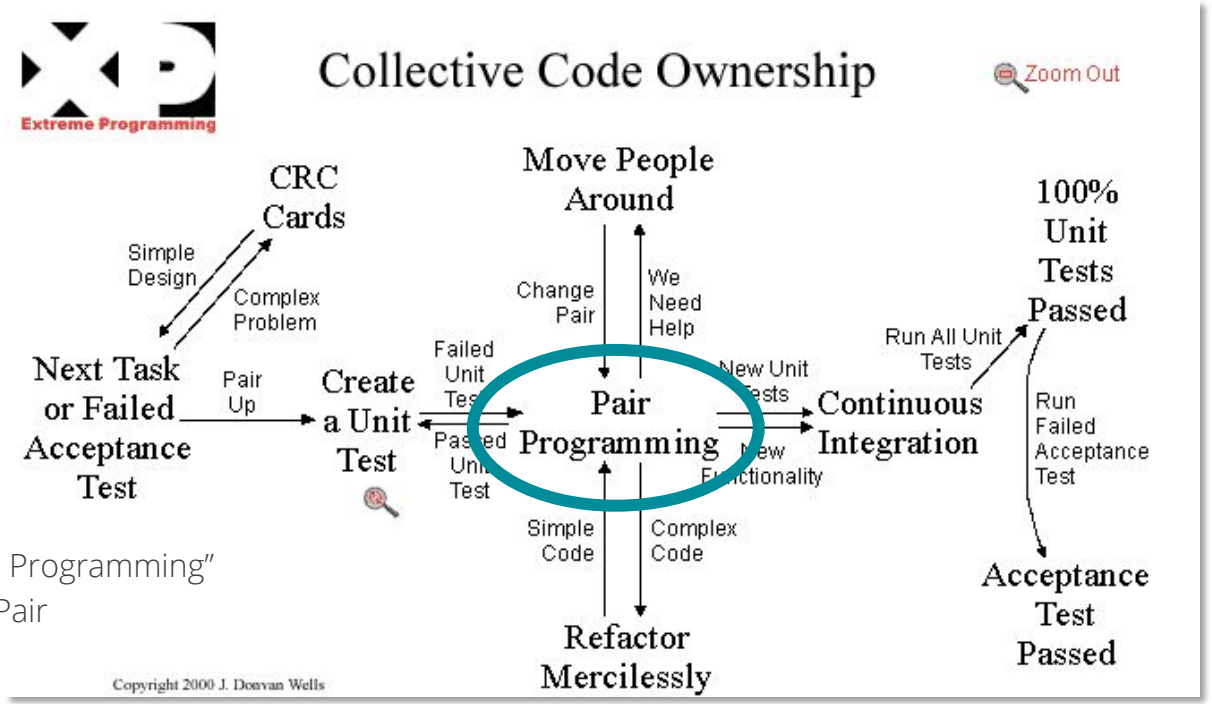
Jean Bartik

One of the ENIAC women, who are considered to be the first programmers

http://www.computerhistory.org/revolution/birth-of-the-computer/4/78/2258

“FAST” FORWARD TO THE LATE 90s…





Kent Beck’s book about the “Extreme Programming” agile practices introduced the term “Pair Programming” more widely.
http://www.extremeprogramming.org/

WHAT IS PAIR PROGRAMMING?

"All code to be sent into production is created by **two people working together** at a single computer. Pair programming increases software quality without impacting time to deliver. It is counter intuitive, but 2 people working at a single computer will add as much functionality as two working separately except that it will be much higher in quality. With **increased quality** comes **big savings later** in the project."

http://www.extremeprogramming.org/rules/pair.html

IT'S A SKILL

Pair programming is a lot more than just "programming with another person". This slide shows a few of the other skills that are important when practicing pair programming.

It's important to approach pair programming with the mindset that all of these skills can be improved and honed by practice, reflection, and adaptation. If it's hard for you today, it does not have to stay that way - hopefully this deck can give some inspiration to try new things and keep improving.

ThoughtWorks®

BENEFITS

Why do pair programming? Awareness of all its benefits is important to decide when you do it, how to do it well, and to motivate yourself to do it in challenging times.

BENEFITS OF PAIRING

Knowledge Sharing	Pairing helps the team share knowledge on technology and domain on a daily basis and prevents silos of knowledge. Recommendations: → Pair on stories where you have no idea about tech or the domain → If you notice that a team member tends to work on the same topics all the time, ask them to mix it up → Create a skill matrix (tech & business) to visualise each person's strengths and gaps
Combine 2 modes of thinking: Tactical and Strategic	Pairing allows you to have different perspectives on the code. The driver's brain is usually more in tactical mode, thinking about the details, the lines of code at hand. The navigator's brain can think more strategically, about the big picture, park next steps and ideas on sticky notes. → Switch the keyboard regularly → As navigator, avoid the "tactical" mode of thinking, leave the details of the coding to the driver - your job is to take a step back and complement your pair's more tactical mode with medium-term thinking.

BENEFITS OF PAIRING

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. It also makes the codebase more consistent than it would be with single coders only. Recommendations: → Pair programming alone does not guarantee this, it needs to be combined with rotations (see “Knowledge Sharing”)
Reflection	Pairing helps reflect more, not just on the code, but also on the user story, the value, the potential solutions, … Recommendations: → Again: talk, talk, talk!

BENEFITS OF PAIRING

Keep team's "WIP" low	Having a Work in Progress (WIP) limit helps your team focus on the most important tasks. We have found that overall team productivity often increases if the team has a WIP limit in place. Especially in larger teams, pairing limits the number of things a team can work on in parallel, and therefore increases the overall focus. Recommendations: → Have your team's WIP limit visible on the wall and have an eye on it in standup - it might naturally force you into a pairing habit
Fast onboarding of new team members	Since pairing facilitates knowledge sharing it can help with the onboarding of new team members. *However* - keep in mind that it is not enough to just put new joiners into a pair, and then they are "magically" included and onboarded. Make sure to also provide the big picture. Recommendations: → Have an onboarding plan with a list of topics to cover → Make sure new joiners had an introduction to the broader context, to make it easier to follow along in their first pairing sessions → Use new joiners' machines when pairing, to make sure that they are set up to work by themselves as well.

BENEFITS OF PAIRING

Helps to keep focus	It's a lot easier to prevent "rabbit holes" and have a structured approach when there is two of you: Each of you has to explicitly communicate why you are doing something and where you are heading. You can help each other to stay on track. Recommendations: → Demand explanations from each other (talk) → Make plans together: Think about each step you need to make to reach your goal, e.g. make a sticky for each step, bring them in order and go one by one → Use the pomodoro technique (described in the "Approaches" chapter)
Code Review "on-the-go"	Doing code reviews after the fact has some downsides. For example, the reviewer has to switch context to get into the unknown code. Or they might dismiss little things they see and think "ah, that's not TOO bad though, not worth changing a working piece of code". When we pair, we have 4 eyes on the little and the bigger things as we go. Recommendations: → Refactoring is part of pairing → Ask each other questions! Questions are the most powerful tool to understand what you are doing and to come to better solutions → If you feel the need to have more code review on pair programmed code, reflect if you can improve your pairing

MISCONCEPTIONS

You have to do pair programming if you're doing an agile process.	"This is utterly false. 'Agile' is a very broad term defined only in terms of values and principles... commonly observed practice, not a prescription."
Pair-Programming halves the productivity of developers.	"My flippant answer to this one is: 'that would be true if the hardest part of programming was typing'."
It's only worth pairing on complex code, rote* code yields no advantage.	I think there is a point to this (...), yields little opportunities for pairing to make a difference. Except this: writing boring rote code is usually a sign that I've missed an important abstraction (...) Pairing will help you find that abstraction.

** rote = routine*

https://martinfowler.com/bliki/PairProgrammingMisconceptions.html

ThoughtWorks®

TO PAIR OR NOT TO PAIR?

#dogmatism

WHAT TO WATCH OUT FOR WHEN NOT PAIRING

Pair programming has so many benefits. How can you mitigate the risk of losing them in the situations when you cannot pair? Note how part of all of these recommendations include variations of "Ask for a pair if…" :-)

Knowledge sharing	→ Put a topic into the Dev Huddle* backlog and share what you have done and learnt with your team → Ask yourself even more often than when pairing: Do I need to create some kind of documentation? → Be particularly mindful of what you could document as you go (for things that have a reasonable frequency to change and cannot be inferred from the code) → Ask yourself "What would happen if I was out the whole next week?" Did you share enough with the team, so that someone else could pick up the task? (However: Taking over work from somebody who is missing unexpectedly will always cause a bit of friction and extra work, you don't have to optimise for this 100%. It's just a good thought experiment to think about how much you have shared, and if you left enough breadcrumbs.) → Insist on pairing, when working on a large and important story. * *aka "Show & Tell", a practice used by many teams to get together regularly and share technical decisions/new implementations/… with each other*

WHAT TO WATCH OUT FOR WHEN NOT PAIRING

Bouncing ideas off each other, preventing rabbit holes	When pairing you can always discuss your idea and approaches with someone. → When working alone, recognise the need to this early, and see if any dev might have some time to **discuss** some ideas with you, or sanity check your current approach. → You already spent a lot of time on something and you are not sure if you entered the rabbit hole already? Try to **talk** to a fellow developer.
Higher code quality	→ Get code reviews from other devs, walk them through what you did. Be mindful of other people's time and flow when you ask them for reviews. → If it is a larger chunk of work (which hopefully it isn't, because then you should really be pairing…), ask for feedback on smaller changes before finishing the whole thing. Code reviews on large change sets tend to cover not as much details. → Especially watch out for moments when you are taking larger decisions, i.e. decision that will be harder to rework, or will impact other parts of the code. In those cases especially, ask for a pairing partner, get input from other team members, and document your thinking.

CODE REVIEW AFTER THE FACT vs. PAIRING

Factors that lead to sloppy code reviews

"The advantage of pair programming is its gripping immediacy: it is impossible to ignore the reviewer when he or she is sitting right next to you" Jeff Atwood

The cartoon below describes one of the traps of code reviews. There is also a kind of "sunk cost fallacy" at play: We are often reluctant to cause rework for something that the team already invested in and that's already working, so we have a tendency to let more things slide in reviews after the fact. Finally, doing a code review requires a context switch for somebody on the team, so the more often they occur (and they *should* occur frequently, for Continuous Integration), the more disruptive they will be for reviewers. With the added time pressure of frequent integration, this also leads to sloppy reviews.

Cartoon seen on https://devops.com/dark-side-infrastructure-code/

Code reviews & Continuous Integration don't go well together

With Continuous Integration (and Delivery), we want to reduce our risks and minimise our time to deliver to the user by delivering small chunks of change frequently. This ultimately means trunk-based development. When practicing trunk-based development, code reviews are less effective, because they go into trunk immediately anyway. Also, they are harder to do logistically without pull requests, because the commits for different features are intertwined with each other in the commit history (and that's what we want, that is the goal of Continuous Integration).
A compromise would be to use pull requests and code reviews in the exceptional cases when you're not pairing on production code, but have measures in place to make sure pull requests don't live for too long, to make sure you still practice Continuous Integration.

ThoughtWorks®

APPENDIX

Different Approaches
Common Challenges

ThoughtWorks®

APPROACHES

Before we get to the challenges that come in a package with all the benefits, let's talk about a few common approaches to pair programming.

THE ROLES: DRIVER AND NAVIGATOR

These classic pair programming role definitions are the basis for a lot of other approaches as well.

- Start with a reasonably well-defined task
- Agree on one tiny goal at a time - defined by a unit test, or defined by a commit message, or written on a sticky note, or…
- **Driver:**
  Has the **keyboard**; is focussed on completing the current **tiny goal**, ignoring larger issues for the moment; **narrates** what they are doing while doing it
- **Navigator:**
  Code reviewer; giving directions, sharing thoughts; has an eye on the **larger issues**, bugs, potential next steps > brings them up after the task is done

More at: *http://www.wikihow.com/Pair-Program*

“PING PONG”

This technique embraces **Test-Driven Development** and is perfect when you have a clearly defined task that can be implemented in a test-driven way.

- Ping: Developer A writes a failing test
- Pong: Developer B writes the implementation to make it pass.
- Developer B then starts the next Ping, i.e. the next failing test.
- Each Pong can also be followed by refactoring the code together, before you move on to the next failing test. This way you follow the “Red - Green - Refactor” approach: Write a failing test (red), make it pass (green) and refactor.

“STRONG PAIRING”

This technique can be used for **knowledge transfer**.

- The rule: "For an idea to go from your head into the computer it MUST go through someone else's hands"
- In this style, the navigator is the person much more experienced with the setup or task at hand, while the driver is a novice (with the language, the tool, the codebase, ...)
- The experienced person stays almost always in the navigator role and guides the novice.
- While this technique borders on micro-management, it can be a useful onboarding tool to favor active “learning by doing” over passive “learning by watching”.
- This style is great for knowledge transfer, but shouldn’t be overused. Keep in mind that your goal should be to easily switch roles after some time.

More at: https://llewellynfalco.blogspot.de/2014/06/llewellyns-strong-style-pairing.html

PAIR DEVELOP-MENT

This approach is particularly great for **exploratory** tasks, when both pairing partners don't have a good idea of possible solutions yet, or both are using the technology at hand for the first time.

- Get to a common understanding of your task.
- Define a list of questions that you need to answer in order to implement a solution.
- Split up to research and explore.
  - Separately search the internet to answer a question
  - Read up on a concept that is new to both of you
- Use your pairing partner as a sparring partner to discuss and share what you have found.
- Make sure you sit next to each other, even when working separately.

This "**split, merge, repeat**" approach is not only good for exploratory tasks, but can be a good change of pace in general. Just make sure that your common checkpoints, context-sharing and discussions give you the benefits of pairing, even if you are doing part of the work by yourselves.

POMODORO

The pomodoro technique is a **time management** method that can be applied to almost all of the pairing approaches described. It can help you to stay focused and to remember to take breaks, and also to regularly switch the keyboard.

1. Decide on the task to be done.
2. Set a timer (traditionally to 25 minutes)
3. Work on the task.
4. End work when the timer rings and put a checkmark on a piece of paper
5. If you have fewer than four check marks, take a short break (3–5 minutes), then go to step 2.
6. After four pomodoros, take a longer break (15–30 minutes), reset your checkmark count to zero, then go to step 1.

Track time & pomodoros online: https://tomato-timer.com/

https://en.wikipedia.org/wiki/Pomodoro_Technique

HOUSE-KEEPING

Scheduling (“Vorausschauendes Fahren”)

- Consider regular, overlapping pairing hours for everyone
- Agree with your pair on how many hours you are going to pair at the start of the day.
- Check if you have meetings during the day and plan for that. If you have any meetings and you need to leave your pair, make sure things can continue without you.

Physical setup

- It’s worth spending some time on figuring out the best setup for your situation — including chairs, table height, …
  - Explore: Try two keyboards, two mice, external monitor, …
  - Make sure everyone has enough space
- Check with your partner if they have any particular preferences (e.g. larger font size)
- Ideally, the team agrees on a standard setup. If you have an unusual keyboard/IDE setup: check with your partner if they are okay with it.

THINGS TO AVOID

- Do not **schedule meetings** during pairing (but if the meeting is for the story, both pairing partners attend)
- Do not **read email or use phone** during pairing (email and calls needed for the story should be paired on)
- Watch out for **micro-management** mode, because it doesn't leave room for the other person to think and is a frustrating experience.
  - "Now type 'System, dot, print, "...
  - "Now we need to create a new class called..."
  - "Press command shift O..."
- Watch out if you're "**hogging the keyboard**" - this might mean that your pairing partner is having a hard time focussing because of limited "active" participation.
- "**5 seconds rule**" - when the navigator sees the driver do something and wants to comment, wait at least 5 seconds before you say something - the driver might already have it in mind, then you are interrupting their flow
- As Navigator, don't immediately point out any error or upcoming obstacle - write sticky notes, wait a bit for driver to correct. If you **intervene immediately** this can be disruptive to the driver's thinking process. It is also absolutely fine to make mistakes and learn from them.

REMOTE PAIRING

- Try to offer your remote partner the best **audio experience** you can manage: Look for a quiet area and use a good headset. “Push to speak” can also help.
- To avoid distractions, noise-cancelling headphones are your friend.
- If one of you is with the rest of the team, try to make sure to include the remote partner in discussions you have with colleagues in the office.
- Remember to **take breaks** - it seems even easier to forget...
- Make sure to **switch computers regularly**, so that each of you has a chance to work on their machine. It might be exhausting to work on a remote computer for the whole day, especially if there is a network lag.
- **Avoid scrolling** in a long file, this can be very difficult to follow if there is network lag. Try to use short keys to open different parts of the file or use the collapse/uncollapse functionality instead.
- **Tools**: Zoom, Screenhero, thread, tmux/tmate, and many more threads on swdev and other mailing lists :-)
- Pairing is hard. It is even harder when you have to do it remotely with someone you haven't met and do not know. Try to **spend some facetime** together.
- Use **shared visual tools** when planning and designing together.

More at:
https://chelseatroy.com/2017/04/01/advanced-pair-programming-pairing-remotely

There is not “THE” right way

There are many approaches to pair programming and there is not “THE” right way to do it. It depends on your styles, personalities, the situation, the task and many other factors.

In the end, the most important question is,
do you get the promised benefits out of it?
If not: Reflect, discuss and adjust to get them.

ThoughtWorks®

CHALLENGES

While pair programming has a lot of benefits, it's also not easy. The following pages list some of the common challenges teams experience, and some suggestions how to cope with them.

CHALLENGES OF PAIRING

Exhausting	Pairing requires intense focus for long stretches of time. Options to tackle: → Take scheduled breaks, for example: 10 minutes per hour. Try Pomodoro. → Do not work during lunch, even solo → Limit pairing to fixed, team-agreed hours, max. 6 hours per day → Make sure to switch roles frequently
Intense interpersonal collaboration can be hard ...	Options to tackle: → See this as an opportunity to improve your mentoring and coaching skills. → Schedule regular feedback sessions with your team members and give feedback on pairing → Could a training help you? For example on difficult conversations? → Discuss the challenges with your team, by organising a session on pairing → Team events build group camaraderie

CHALLENGES OF PAIRING

Challenging to schedule around meetings	Options to tackle: → Define core pairing hours without meetings → Check your calendars together at the beginning of your pairing session, make sure you have enough time to start pairing at all. → Attend meetings as a pair, not solo → Block out meeting times as a team, for example: no meetings before noon → Limit your team meetings in length, but also in overall amount. Which meetings do you really need, what goals do they have? → Rely on PO/PM, or non-pairing team members to run interference for the team
Differences between pairers	The two of you might have differences in techniques, knowledge, skills, extroversion, personalities, ... Options to tackle: → Start your pairing session with the question: "How do we want to work together?" → Give regular feedback: Do a feedback session at the end of your session and reflect on how it went for both of you. → Be aware of how you like to work/are efficient (but also don't be closed off to other approaches) → Do a team session to explore your styles and potential differences, so that everyone in the team is aware of them.

CHALLENGES OF PAIRING

Communication "overhead" vs. Delivery Pressure	Yes, stories will take longer to complete - but if you are doing it right, you will also have fewer defects, and all the other many long-term benefits mentioned before, plus it will save time on code review and other checkpoints that can become unnecessary with an efficient pairing approach. Options to tackle: → Take the increased development time into consideration for scheduling → Make the benefits of pairing visible for your team and your stakeholders → Point out a long-term benefit in action when you see it, so you can draw satisfaction from them as much as from short-term successes like a "red-green" test.
Especially hard on big stories, with lots of Unknowns	Options to tackle: → If a story is too big, discuss options to split the story with your team → When there are lots of unknowns, e.g. you work with a new technology, think about doing a spike to explore the topic and don't forget to share your findings with the team. → In these situations, think of it as "pair development", not just "pair programming". It's okay to split up to do research - maybe after agreeing on the set of questions you need to answer together.

CHALLENGES OF PAIRING

No time for yourself	If you need knowledge to build a feature when working solo, you would just take some time to dig into it. How can you take that time while pairing? Also, more introverted people need time to think by themselves, are more drained by constant conversation. Options to tackle: → Don't pair 8 hours a day, agree on pairing hours with your team. → When you feel like as a pair, you don't have the collective knowledge to approach a problem, split up to read up and share back, then continue implementation.
Lots of context switching with pair rotations	Knowledge sharing is one of the benefits of pairing on the other hand frequent context switching through pair rotations can be tough. Options to tackle: → Find a balance between rotation on stories and the possibility for a new pairing partner to get enough context on the story and contribute properly. → Don't rotate for the rotation's sake, think about if and why it is important to share a certain context.

“GROWTH MINDSET”

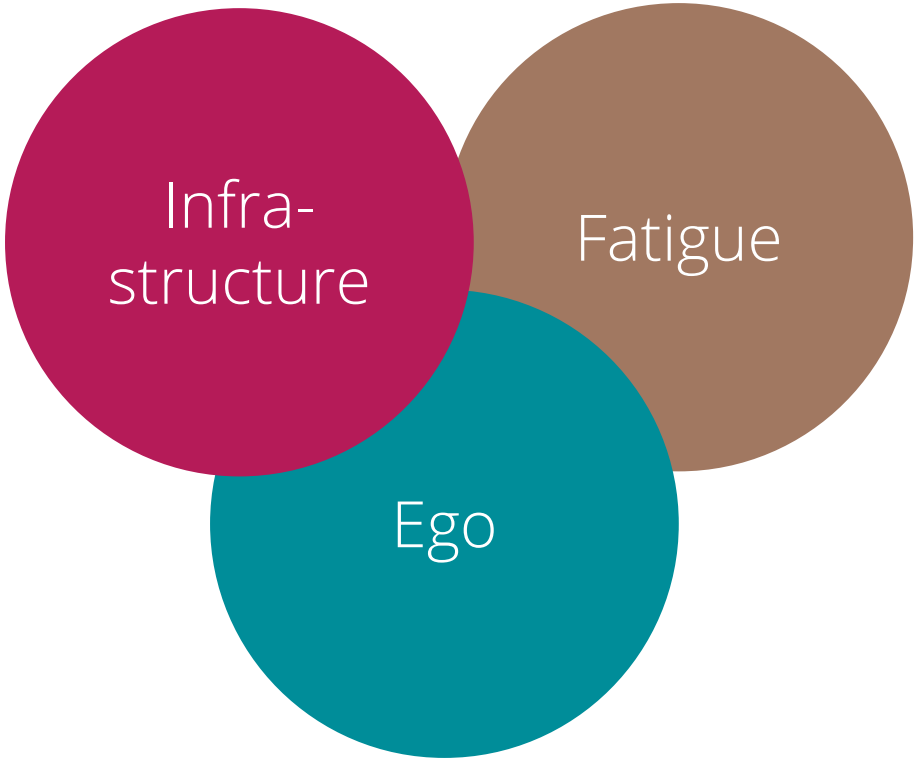
Pairing can be challenging because it requires a lot of focus and interpersonal interactions.

You can get better at that with practice, even if it feels very hard and challenging for you in the beginning, don’t give up or develop a negative attitude to it too quickly.

It’s a skill that requires constant reflection. With the help of the mentioned practices in this document, you can grow and improve your pairing skills.

https://hbr.org/2016/01/what-having-a-growth-mindset-actually-means

3 INHIBITORS



Infrastructure: The most obvious issue in pair programming is the team setup. From things such as editors/IDEs to keyboard layouts and OS, if your team doesn't have a common setup for everyone, it's going to be really hard to get people comfortable with pair programming. Having dedicated workstations can be of a great help, but also agreeing on a common setup or having a way to share the code in a way that anyone can use their own configuration, work just as well

Fatigue: I have heard so many people say that they were exhausted after a day or a session of pair programming. The increased focus doesn't come easily, it takes a lot of energy to focus on the problem, share your thoughts and hear the other person. It also takes a lot of effort to zone out and not check your twitter feed, while someone by your side is depending on you

Ego: Be it when you are pairing or even when you are considering pairing with someone else, it's important to stay humble and hear the other person. I have seen so many instances when people argue, rather than discuss. I find this to be a big issue in pair programming. After a few situations like this, pairing just becomes so painful that no one wants to do it, and will even advocate against it!

https://www.thoughtworks.com/insights/blog/pair-programming-hard-talk-your-team