On Pair Programming

Many people who work in software development today have heard of the practice of pair programming, yet it still only has patchy adoption in the industry. One reason for its varying acceptance is that its benefits are not immediately obvious, it pays off more in the medium- and long-term. And it's also not as simple as "two people working at a single computer", so many dismiss it quickly when it feels uncomfortable. However, in our experience, pair programming is vital for collaborative teamwork and high quality software.

15 January 2020

Birgitta Böckeler

Birgitta Böckeler is a developer and consultant with ThoughtWorks in Germany. As a Technical Lead on custom software delivery teams, she is splitting her days between coding, coaching, consulting, and keeping the work fun.

Nina Siessegger

Nina Siessegger is a software developer, tech lead and consultant from Hamburg, Germany, and a former ThoughtWorker. Besides the joy of writing code, she is especially interested in the human side of software development. She strongly believes that high-quality software is developed by teams that value

communication, collaboration and trust.

EXTREME PROGRAMMING

COLLABORATION

CONTENTS

> *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 very first programmers*

> *Write all production programs with two people sitting at one machine.*
>
> -- *Kent Beck*

Jean Bartik was one of the ENIAC women, who are considered by many to be the very first programmers. They took on the task of programming when the word "program" was not even used yet, and there were no role models or books to tell them how to do this - and they decided that it would be a good idea to work in a pair. It took about 50 more years for pair programming to become a widespread term, when Kent Beck described the term in his book "Extreme Programming" in the late 1990s. The book introduced agile software development practices to a wider audience, pairing being one of them.

Pair programming essentially means that two people write code together on one machine. It is a very collaborative way of working and involves a lot of communication. While a pair of developers work on a task together, they do not

only write code, they also plan and discuss their work. They clarify ideas on the way, discuss approaches and come to better solutions.

The first part of this article, "How to pair", gives an overview of different practical approaches to pair programming. It's for readers who are looking to get started with pairing, or looking to get better at it.

The second and third parts, "Benefits" and "Challenges", dive deeper into what the goals of pair programming are, and how to deal with the challenges that can keep us from those goals. These parts are for you if you want to understand better why pair programming is good for your software and your team, or if you want some ideas what to improve.

Part four and five, "To pair or not to pair?", and "But really, why bother?", will conclude with our thoughts on pairing in the grand scheme of team flow and collaboration.

How to pair

Styles

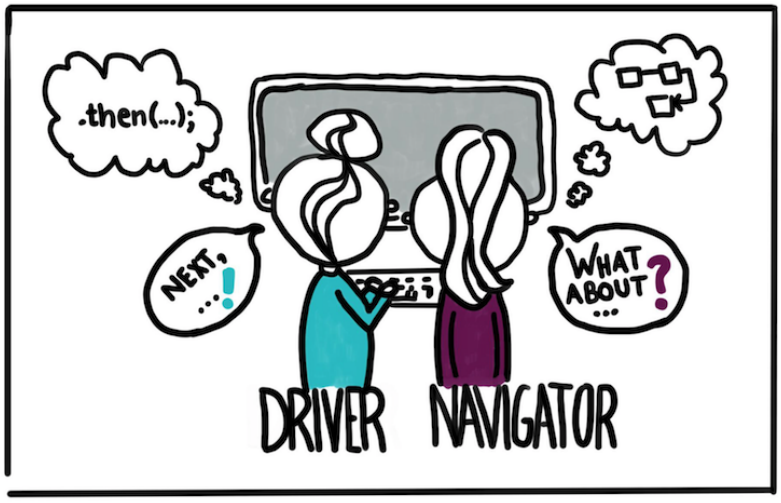
Driver and Navigator

These classic pair programming role definitions can be applied in some way or other to many of the approaches to pairing.

The **Driver** is the person at the wheel, i.e. the keyboard. She is focussed on completing the tiny goal at hand, ignoring larger issues for the moment. A driver

should always talk through what she is doing while doing it.

The **Navigator** is in the observer position, while the driver is typing. She reviews the code on-the-go, gives directions and shares thoughts. The navigator also has an eye on the larger issues, bugs, and makes notes of potential next steps or obstacles.



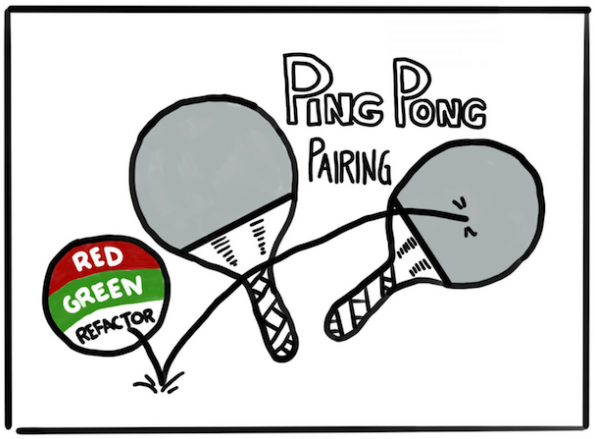
The idea of this role division is to have two different perspectives on the code. The driver's thinking is supposed to be more tactical, thinking about the details, the lines of code at hand. The navigator can think more strategically in their observing role. They have the big picture in mind.

A common flow goes like this:

- Start with a reasonably well-defined task
- Agree on one tiny goal at a time. This can be defined by a unit test, or by a commit message, or written on a sticky note.
- Switch keyboard and roles regularly. Shared active participation keeps the energy level up and we learn and understand things better.
- 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. Park next steps,

potential obstacles and ideas on sticky notes and discuss them after the tiny goal is done, so as not to interrupt the driver's flow.

Ping Pong



This technique embraces Test-Driven Development (TDD) 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 with the minimum necessary means (green), and then refactor.

For some great examples on the red-green-refactor workflow take a look into the 99 bottles of OOP book by Sandy Metz and Katrina Owen.

Strong-Style Pairing

This is a technique particularly useful for knowledge transfer, described in much more detail by Llewellyn Falco here.

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 usually 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 mostly stays in the navigator role and guides the novice.

An important aspect of this is the idea that the driver totally trusts the navigator and should be "comfortable with incomplete understanding". Questions of "why", and challenges to the solution should be discussed after the implementation session. In a setting where one person is a total novice, this can make the pairing much more effective.

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 initial knowledge transfer, but shouldn't be overused. Keep in mind that the goal is to be able to easily switch roles after some time, and ease out of the micro management mode. That will be a sign that the knowledge transfer worked.

Pair Development

"Pair Development" is not so much a specific technique to pair, but more of a mindset to have about pairing. (We first came across the term in this thread on Sarah Mei's Twitter account.) The development of a user story or a feature usually requires not just coding, but many other tasks. As a pair, you're responsible for all of those things.

To help get you into the mindset, the following are a few examples of the non-coding activities in a story life cycle that benefit from pairing.

Planning - what's our goal?

When you first start working on something together, don't jump immediately into the coding. This early stage of a feature's life cycle is a great opportunity to avoid waste. With four eyes on the problem this early on, catching misunderstandings or missing prerequisites can save you a lot of time later.

- **Understand the problem:** Read through the story and play back to each other how you understand it. Clear up open questions or potential misunderstandings with the Product Owner. If you have a Definition of Ready in your team, go through that again and make sure you have everything to get started.
- **Come up with a solution:** Brainstorm what potential solutions for the problem are. You can either do this together, or split up and then present your ideas to each other. This depends on how well-defined the solution already is, but also on your individual styles. Some people like some time to think by themselves, others like talking things through out loud while they are thinking. If one of you is less familiar with the domain or tech, take some time to share the necessary context with each other.
- **Plan your approach:** For the solution you chose, what are the steps you need to take to get there? Is there a specific order of tasks to keep in mind? How will you test this? Ideally, write these steps down, in a shared document or on sticky notes. That will help you keep track of your progress, or when you need to onboard somebody else to help work on the task. Writing this down also simply helps remember what needs to be done - in the moment, we too often underestimate how many things we will have forgotten even as quickly as the next day...

Research and explore

When implementing a feature that requires you to use a technology you are both unfamiliar with, you'll have to do some research and exploration first. This work does not fit into the clean-cut "driver-navigator" or "ping-pong" approaches. E.g., browsing search engine results together on the same screen is usually not very effective.

Here is one way to approach this in pair development mode:

- Define a list of questions that you need to answer in order to come up with a suitable solution.

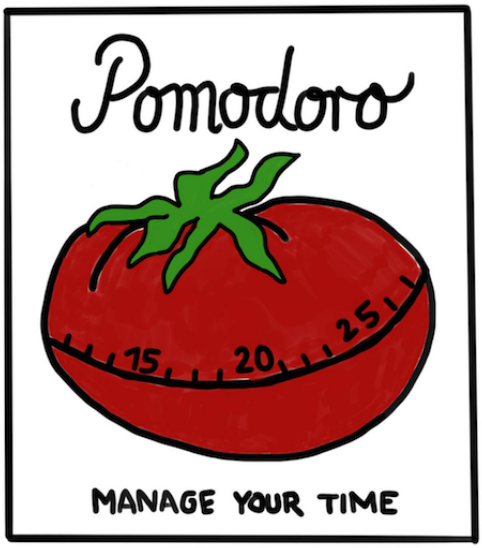
- Split up - either divide the questions among you, or try to find answers for the same questions separately. Search the internet or other resources within your organisation to answer a question, or read up on a concept that is new to both of you.
- Get back together after a previously agreed upon timebox and discuss and share what you have found.

Documentation

Another thing to work on together beyond the code is documentation. Reflect together if there is any documentation necessary for what you've done. Again, depending on the case at hand and your individual preferences, you can either create the documentation together, or have one of you create it, then the other review and word-smith.

Documentation is a great example of a task where a pair can keep each other disciplined. It's often a task left for last, and when it's the last thing keeping us from the great feeling of putting our story into "Done", then more often than not, we skip it, or "wing it". Working in a pair keeps us honest about some of the valuable, but annoying things that we'll be very thankful for in the future.

Time management



In addition to the general styles for pairing, there are other little tools and techniques to make it easier.

The pomodoro technique is ones of those tools. It is a time management method that breaks work down into chunks of time - traditionally 25 minutes - that are separated by short breaks. The technique can be applied to almost all of the pairing approaches described and will keep you focused. Pairing can be an exhausting practice, so it is helpful to get a reminder to take breaks and to switch the keyboard regularly.

Here is an example of how using the pomodoro technique looks like in practice.

- Decide on what to work on next
- Set a timer for 25 minutes, e.g. with the help of the many pomodoro browser extensions - or even a real life tomato shaped kitchen timer...
- Do some work without interruptions
- Pause work when the timer rings - start with short breaks (5-10 minutes)
- After 3 or 4 of these "pomodoros", take a longer break (15–30 minutes)
- Use the short breaks to *really* take a break and tank energy, get some water or coffee, use the bathroom, get some fresh air. Avoid using these short breaks for other work, like writing emails.

Pair Rotations

Rotating pairs means that after working together for some time, one half of the pair leaves the story, while the other person onboards somebody new to continue. The person who stays is often called the "anchor" of a story.

One category of reasons why to rotate is logistics. Your pairing partner could be sick or going on holiday. Or one of you is working remotely for a day, and the work requires physical presence on site, e.g. because there is a hardware setup involved.

Another group of reasons why to rotate is to mix things up. Either the two of you have been working together for a while and are starting to show signs of "cabin fever" because you are spending too much time together. Or you're working on something very tedious and energy-draining - a rotation will give one of you a break, and a new person can bring in some fresh perspectives and energy.

Finally, the most given reason for pair rotations is to avoid knowledge silos, increase collective code ownership, and get some more code review on-the-go. Pairing itself is already helping with those things, but rotations can further increase the average number of eyes on each line of code that goes to production.

As to the ideal frequency of rotations, this is where opinions diverge. Some people believe that rotations every 2-3 days are crucial to ensure a sufficient knowledge spread and quality. Every rotation comes with some costs though. There's the time to onboard a new person, and the cost of a context switch for one of the two. If there is no constant anchor for continuity, the risk increases that tacit knowledge about the problem and solution space gets lost and triggers rework. For more junior developers it's sometimes more beneficial to stay on something for longer, so they have sufficient time to immerse themselves in a topic and give new knowledge time to settle.

> The term "show and tell" is used in different ways on agile teams. What we are referring to here is a regularly scheduled developer huddle, a time of the week where developers get together and discuss tech debt, learnings, share a significant piece of new code with each other, etc.

Think about the trade off between these costs and the benefits. For example, let's say you have high quality knowledge sharing already, with team "show and tells", readable code and good documentation. In that case, maybe an insistence on frequent rotations only marginally improves your collective code ownership, while creating high amounts of friction and overhead.

Plan the Day

Pairing requires a certain level of scheduling and calendar coordination. If you don't take time to acknowledge and accommodate this, it will come back to haunt you later in the day.

Start the day with a calendar check - agree with your pairing partner on how many hours you are going to pair, and see if you need to plan around meetings or time needed to work on other things outside of the pairing task. If it turns out that one of you will have to work by themselves for a while, then make sure to prepare for things to continue without the other person, e.g. by not using that person's computer to code.

If you have meetings or other commitments during the day, make sure you have a reminder in place that you will notice, especially when working on your pairing partner's machine. If your team pairs by default, consider agreeing on regular "core coding hours" for everyone. This makes scheduling much easier.

Physical Setup

Pair programming means you need to work very closely together in the physical space of one shared desk. This is quite different from having your own table to spread out on. Being that close to one another requires a certain level of respect and attention for each other's needs. That is why it is worth spending some time figuring out a comfortable setup for both of you.

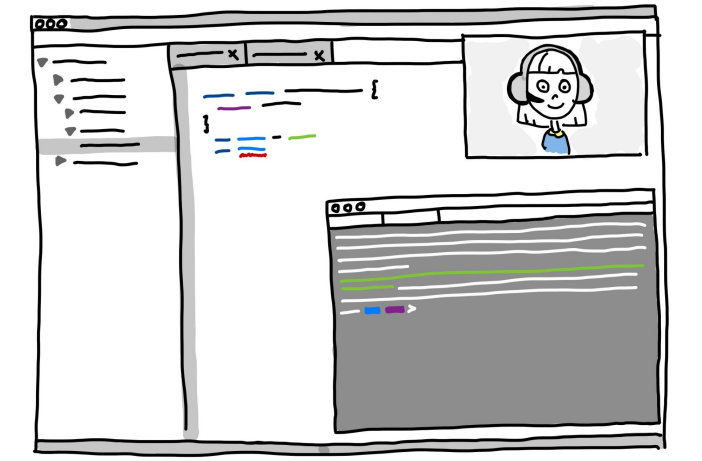
- Make sure both of you have enough space, clear up the desk if necessary.
- Is there enough space for both chairs in front of the desk? Get waste bins and backpacks out of the way.
- Do you want to use two keyboards or one? Same for the mouse, one or two? There's no clear rule that always works, we recommend you try out what works best for each situation. Some of the factors that play into this are hygiene, how good you are at sharing keyboard time, or how much space you have available.

- Do you have an external monitor available, or maybe even two? If not, you can also consider setting up some kind of screen sharing, as if you were remote pairing. In that setup, each of you would use their own laptop keyboards.
- Check with your partner if they have any particular preferences or needs (e.g. larger font size, higher contrast, ...)
- If you have an unusual keyboard/IDE setup check with your partner if they are okay with it. See if you can have a simple mechanism to switch your settings back to a more standard configuration for these situations.

It is beneficial if your team can agree on a default setup, so that you don't have to discuss these things again and again.

Remote Pairing

Are you part of a distributed team, or some team members occasionally work from home? You can still practice pair programming, as long as both of you have reasonably stable internet access.



The Setup

For remote pairing, you need a screen-sharing solution that allows you to not only see, but also control the other person's machine, so that you are able to switch the keyboard. Many video conferencing tools today already support this,

so if you're working at a company who has a license for a commercial VC tool, try that first. There are also open source tools for video calls with remote control, e.g. jitsi. For solutions that work at lower bandwidths, try things like ssh with tmux or the Live Share extension for Visual Studio Code.

Tips

- **Use video:** Since people communicate a lot through gestures and facial expressions it is nice to see the shared screen and your pairing partner's video at the same time. Some video conference solutions come with this feature; if yours doesn't, consider opening up an additional call in order to see each other.
- **Planning and designing:** Use collaborative online visualization tools, to reproduce the experience of sketching out things on paper or a whiteboard.
- **Audio experience:** Look for a quiet area and use a good headset, maybe even with a directional microphone. If you can't get away from the noise, "push to speak" functionality can also help. To avoid distractions on your side, noise-cancelling headphones are your friend.

  > **More on remote pairing**
  >
  > Chelsea Troy has put together a blog post series about advanced pair programming, including a post on remote pairing.

- Dealing with network lag: It can be exhausting to work on a remote computer for a longer period of time when there is a network lag. So make sure to switch computers regularly, so that each of you has a chance to work on their own machine without lag. A network lag can also be annoying when you scroll through files because it can be hard to follow. It helps to avoid scrolling in long files, try to use keyboard shortcuts to open different parts of the file or use the collapse/uncollapse functionality instead.

The Human Part

If you work in a setup where not the whole team is distributed and just one or a few of you are remote, try to include the remote partner in all discussions that are happening in the office. We tend to forget how much we share incidentally just by sitting in the same room.

Working remotely with someone you haven't met and do not know creates an additional challenge. On the one hand, pairing is a chance to get closer to each other on a remote team. On the other hand, it's sometimes easy to forget that part of the collaboration. If there is no chance that you meet in person, think about other ways to get to know each other a bit better, e.g. try to have a remote coffee together.

Finally, while remote pairing can have its challenges, it can also make it easier to focus than when pairing on site, because it is easier to blend out distractions with headphones on.

Have a Donut Together

Celebrate when you have accomplished a task together! High-fiving each other might seem corny, but it's actually a little "power pose" you can do together that can energize and get you ready for the next task. Or maybe you create your own way of celebrating success, like Lara Hogan, who celebrates career achievements with a donut.

Things to Avoid

The different approaches and techniques help you to have a better pairing experience. Here are a few common pitfalls to avoid:

Drifting apart

When you pair, avoid to read emails or to use your phone. These distractions might come across as direspectful to your pair, and they distract you from the

task you are working on. If you really need to check something, make it transparent what you are doing, and why. Make sure that everyone has enough time to read their emails by taking enough breaks and reserving some individual time each day.

Micro-Management Mode

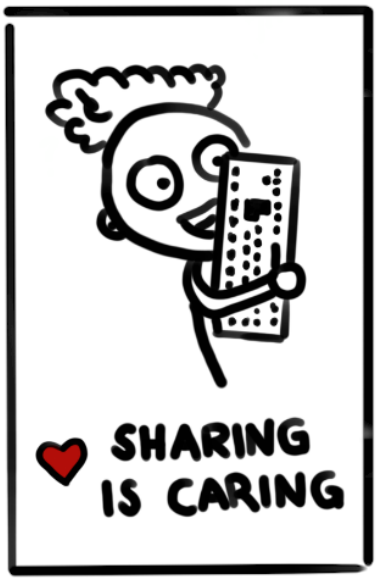
Watch out for micro-management mode: It doesn't leave room for the other person to think and is a frustrating experience, if someone keeps giving you instructions like:

- "Now type 'System, dot, print, "...
- "Now we need to create a new class called..."
- "Press command shift O..."

Impatience

Apply the "5 seconds rule": When the navigator sees the driver do something "wrong" and wants to comment, wait at least 5 seconds before you say something - the driver might already have it in mind, then you are needlessly interrupting their flow.

As Navigator, don't immediately point out any error or upcoming obstacle: Wait a bit for the driver to correct or write a sticky note to remember later. If you intervene immediately, this can be disruptive to the driver's thinking process.



Keyboard Hogging

Watch out if you're "hogging the keyboard": Are you controlling it all the time, not letting your pairing partner do some typing as well?

This can be a really annoying experience for your pair and might cause them having a hard time focussing because of limited "active participation". Try one of the approaches described earlier to make sure that you switch the keyboard frequently.

Pairing 8 Hours per Day

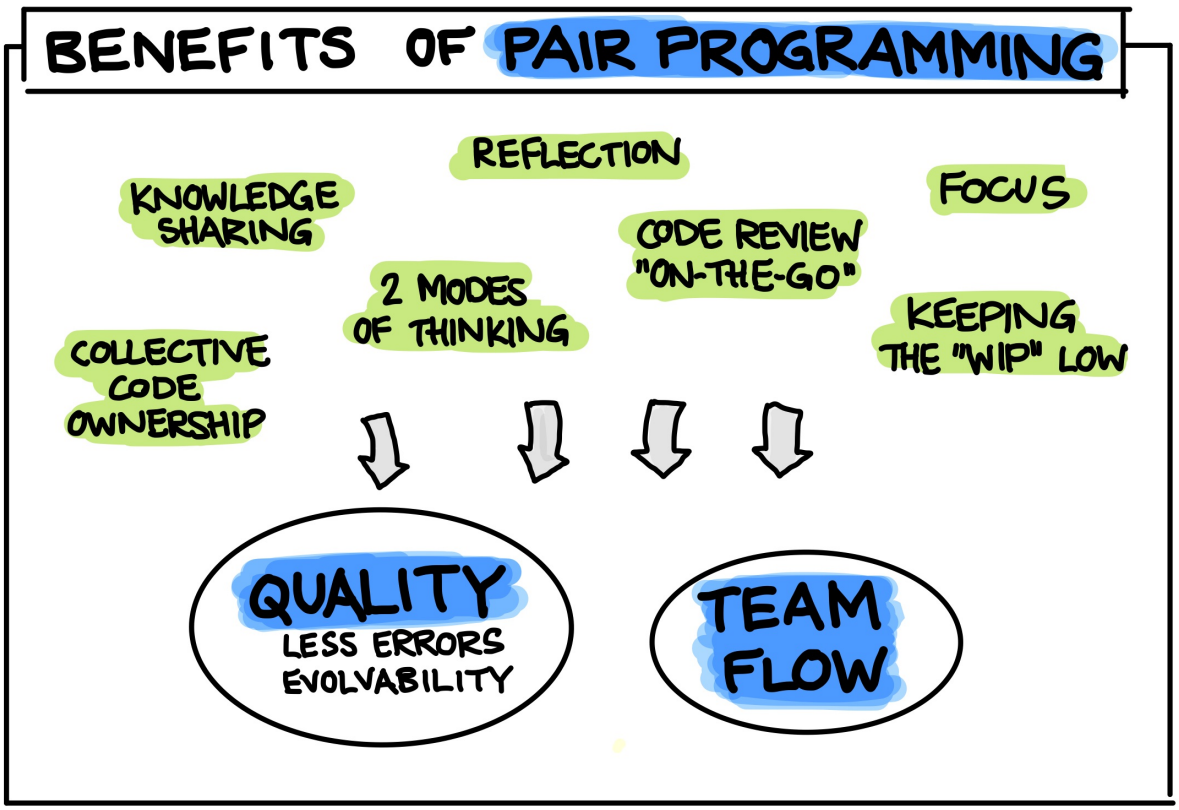
Teams that are really committed to making pair programming work sometimes end up pairing for 8 hours a day. In our experience, that is not sustainable. First of all it is just too exhausting. And secondly, it does not even work in practice because there are so many other things you do other than coding, e.g. checking emails, having 1:1s, going to meetings, researching/learning. So keep that in mind when planning your day and don't assume it will be 100% coding together.

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, experience, 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 this is not the case, try out something else, reflect, discuss and adjust to get them.

Benefits

What is pair programming good for? Awareness of all its potential benefits is important to decide when you do it, how to do it well, and to motivate yourself to do it in challenging times. The main goals pairing can support you with are software quality and team flow.



How can pairing help you achieve those goals then?

Knowledge Sharing

Let's start with the most obvious and least disputed benefit of pairing: Knowledge sharing. Having two people work on a piece of the code helps the team spread knowledge on technology and domain on a daily basis and prevents silos of knowledge. Moreover, when two minds understand and discuss a problem, we improve the chances of finding a good solution. Different experiences and perspectives will lead to the consideration of more alternatives.

Practical Tips

Don't shy away from pairing on tasks when you have no idea about the technology involved, or the domain. If you keep working in the area that you feel most comfortable in, you will miss out on learning new things, and ultimately spreading knowledge in your team.

If you notice that a team member tends to work on the same topics all the time, ask them to mix it up to spread their expertise. It can also help to create a skill matrix with the team's tech & business topics and each person's strengths and gaps in each area. If you put that on a wall in your team area, you can work together on getting a good spread of knowledge.

Reflection

Pair programming forces us to discuss approaches and solutions, instead of only thinking them through in our own head. Saying and explaining things out loud pushes us to reflect if we really have the right understanding, or if we really have a good solution. This not only applies to the code and the technical design, but also to the user story and to the value a story brings.

Practical Tips

It requires trust between the two of you to create an atmosphere in which both of you feel free to ask questions and also speak openly about things you don't understand. That's why building relationships within the team becomes even more important when you pair. Take time for regular 1:1 and feedback sessions.

Keeping focus

It's a lot easier to have a structured approach when there are two of you: Each of you has to explicitly communicate why you are doing something and where you are heading. When working solo, you can get distracted a lot easier, e.g. by "just quickly" trying a different approach without thinking it through, and then coming back out of the rabbit hole hours later. Your pairing partner can prevent you from going down those rabbit holes and focus on what is important to finish your task or story. You can help each other stay on track.

Practical Tips

Make plans together! Discuss your task at hand and think about which steps you need to make to reach your goal. Put each of the steps on sticky notes (or if remotely, subtasks in your ticket management system), bring them in order and go one by one. Try this in combination with the Pomodoro technique and try to finish one of the goals in one pomodoro.

Never forget that communication is key. Talk about what you are doing and demand explanations from each other.

Code review on-the-go

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.

The refactoring of code is always part of coding, and therefore of pair programming. It's easier to improve code when you have someone beside you because you can discuss approaches or the naming of things for example.

Doing code reviews after the fact has some downsides. We will dive more into this later, in "To pair or not to pair?".

Practical Tips

Ask each other questions! Questions are the most powerful tool to understand what you are doing and to come to better solutions. If code is not easy to read and understand for one of you, try a different way that is easier to understand.

If you feel the need to have more code review on pair programmed code, reflect if you can improve your pairing. Weren't you able to raise all questions and concerns during your pairing session? Why is that? What do you need to change?

Two modes of thinking combined

As mentioned when we described the classic driver/navigator style earlier, 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 current line of code. Meanwhile, the navigator's brain can think more strategically, consider the big picture, park next steps and ideas on sticky notes. Could one person combine these two modes of thinking? Probably not! Having a tactical and strategic view combined will increase your code quality because it will allow you to pay attention to the details while also having the bigger picture in mind.

Practical Tips

Remember to switch the keyboard and thus the roles regularly. This helps you to refresh, to not get bored, and to practice both ways of thinking.

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.

Collective Code Ownership

> *Collective code ownership abandons any notion of individual ownership of modules. The code base is owned by the entire team and anyone may make changes anywhere.*
>
> -- *Martin Fowler*

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.

Practical Tips

Pair programming alone does not guarantee you achieve collective code ownership. You need to make sure that you also rotate people through different pairs and areas of the code, to prevent knowledge silos.

Keeps the team's WIP low

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. Especially in larger teams, pair programming limits the number of things a team can work on in parallel, and therefore increases the overall focus. This will ensure that work constantly flows, and that blockers are addressed immediately.

Practical Tips

Limit your team's WIP to the number of developer pairs on your team and make it visible in your team space (or, if you work remotely, in your online project management tool). Have an eye on the limit before picking up new tasks. WIP limit discipline 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. New joiners can get to know the project, the business and the organisation with the help of their pair. Changes in a team have an impact on the team flow. People just need some time to get to know each other. Pair programming can help to minimize that impact, because it forces people to communicate a lot more than they need when working solo.

Practical Tips

It is not enough to just put new joiners into a pair, and then they are "magically" included and onboarded. Make sure to provide the big picture and broader context before their first pairing session, and reserve some extra time for the onboarding. This will make it easier for them to follow along and contribute during the pairing, and get the most out of it. Always use the new joiners' machine when pairing, to make sure that they are set up to work by themselves as well.

Have an onboarding plan with a list of topics to cover. For some topics you might want to schedule dedicated sessions, for other topics the new team member can take the onboarding plan with her from pair to pair. If something is covered in the pairing session, you can check it off the list. This way the onboarding progress is visible to everyone on the team.

Challenges

While pair programming has a lot of benefits, it also requires practice and might not work smoothly from the start. The following is a list of some of the common challenges teams experience, and some suggestions on how to cope with them. When you come across these challenges, keep the benefits in mind and remember why you pair. It is important to know what you want to achieve with a practice, so that you can adjust the way you do it.

Pairing can be exhausting

When working alone, you can take breaks whenever you want, and your mind can drift off or shut down for a bit when it needs to. Pairing forces you to keep focus for potentially longer stretches of time, and find common ground with the other person's rhythm and ways of thinking. The increased focus is one of the benefits of pairing, but can also make it quite intense and exhausting.

Ways to tackle

Taking enough breaks is key to face this challenge. If you notice you are forgetting to take regular breaks, try scheduling them with an alarm clock, for example 10 minutes per hour. Or use a time management technique like Pomodoro. Don't skip your lunch break: Get away from the monitor and take a real break. Pairing or not, taking breaks is important and increases productivity.

Another important thing to prevent exhaustion is to not pair 8 hours per day. Limit it to a maximum of 6 hours per day. Regularly switching roles from driver to navigator can also help to keep the energy level up.

Intense collaboration can be hard

Working so closely with another person for long stretches of time is intense. You need to communicate constantly and it requires empathy and interpersonal skills.

You might have differences in techniques, knowledge, skills, extraversion, personalities, or approaches to problem-solving. Some combinations of those might not match well and give you a rocky start. In that case, you need to invest some time to improve collaboration, and make it a mutual learning experience instead of a struggle.

Ways to tackle

A conversation at the beginning of your pairing session can help you to understand differences between your styles, and plan to adapt to that. Start your first session with questions like "How do we want to work together?", "How do you prefer to pair?". Be aware of how you like to work and how you are efficient, but also don't be closed off to other approaches - maybe you'll discover something new.

At the end of a day of pairing, do a round of feedback for each other. If the idea of giving feedback seems daunting to you, think about it more as a mini retrospective. Reflect on how you both felt during the pairing session. Were you

alert? Were you tired? What made you feel comfortable, what not? Did you switch the keyboard often enough? Did you achieve your goals? Is there anything you would like to try next time? It's good to make this a routine early on, so you have practice in giving feedback when something goes wrong.

There are excellent trainings and books that can help you deal with interpersonal conflicts and difficulties, for example on difficult conversations.

Face the challenges as a team and don't leave conflicts to individuals. You can do this for example by organising a session on pairing in which you discuss how to deal with difficulties together. Start the session by collecting the benefits of pairing, so that you know what you all want to get out of it. Afterwards collect the challenges each individual feels when pairing. Now the group can think about which actions might help to improve. You could also collect the hot button triggers of the team members: What makes you immediately feel uncomfortable when pairing?

Interruptions by meetings

Have you ever had days full of meetings, and you get the impression you are not getting anything done? This probably happens in every delivery team. Meetings are necessary to discuss, plan and agree on things you are going to build, but on the other hand they interrupt the flow. When a team practices pair programming the effect of too many meetings can get even worse. If each of the persons pairing has meetings at different times, the interruptions are multiplied.

Ways to tackle

One approach is to limit the time slots in which meetings happen, for example by defining core pairing hours without meetings, or by blocking out no-meeting-times with rules like "no meetings after noon".

It is also worth thinking about meeting length and overall amount. Which meetings do you really need? What goals do they have and how can you improve

their quality, for example with proper preparation, facilitation, and a clear agenda.

But one thing is for sure: There will always be meetings. So how to deal with that as a pair? Check your calendars together at the beginning of your pairing session, make sure you have enough time to start pairing at all. If you have any meetings consider attending them as a pair. Rely on your Product Owner, or other non-pairing team members, to keep interruptions away from the team in the core pairing hours.

Different skill levels

When two people with different experience levels pair on a topic, this often leads to false assumptions on how much each of them can contribute, or frustrations because of difference in pace.

Ways to tackle

If your pair has more experience on the topic: Don't assume they know best. Maybe the need to explain why they are doing things the way they are will bring them new insights. Asking questions on the how and why can lead to fruitful discussions and better solutions.

If your pair has less experience on the topic: Don't assume they cannot contribute much to the solution. You might be stuck wearing blinders and a different viewpoint can help you to come to a better solution. Also, remember that having to explain a concept is a great opportunity to test if you've really understood it and thought it all the way through.

It also helps to be aware of different learning stages to understand how the learning process from novice to expert works. Dan North has described this very nicely in his talk Patterns of Effective Teams. He introduces the Dreyfus Model of Skills Acquisition as a way to understand the different stages of learning, and what combining them means in the context of pairing.

Power Dynamics

Dealing with power dynamics is probably one of the hardest challenges in this list. Pair programming does not happen in a space without hierarchies. There are formal hierarchies, for example between a manager and their report, and informal ones. Examples for informal hierarchies are:

- junior - senior
- non-men - men
- career changers - folks with a CS degree
- People of color - white folks

And these are just a few. Power dynamics are fluid and intersectional. When two people pair, multiple of those dynamics can be in play and overlap. To get an idea of how power imbalance can impact pairing, here are a few examples.

- One person is dominating the pairing session by hogging the keyboard and not giving room to their pairing partner.
- One person stays in a teaching position and attitude all the time.
- One person is not listening to the other one, and dismissing their suggestions.

It sometimes can be subtle to tie these situations back to hierarchies, you often just think that you don't get along with each other. But the underlying issue is often times influenced by an imbalance between the two folks pairing.

Sarah Mei has written an excellent series of tweets on the topic and has also given a talk that covers power dynamics in agile in a more general way.

Ways to tackle

The first step to tackle this is for the person on the upward side of the power dynamic to acknowledge and admit to themselves their position. Only then can you honestly reflect on interactions you have with your pairing partner, and how power dynamics impact them. Try to think about your own positionality and situatedness. What can you actively do to neutralize power imbalance?

Recognizing these types of differences and adapting our behaviour to improve collaboration can be hard. It requires a lot of self reflection. There are trainings that can help individuals or teams with this, for example "anti-bias" or ally skills trainings.

Pairing with lots of Unknowns

When you work on a large topic where both of you don't have an idea how to solve a problem, the usual pairing styles often don't work as well. Let's say you need to use a technology for the first time, or try out a new approach or pattern. Researching and experimenting together works in some constellations, but it can also be frustrating because we all have different approaches to figuring out how things work, and we read and learn at different paces.

Ways to tackle

When there are lots of unknowns, e.g. you work with a new technology, think about doing a spike to explore the topic and learn about the technology before you actually start working. Don't forget to share your findings with the team, maybe you have a knowledge exchange session and draw some diagrams you can put up in the team space.

In these situations, remember to take on the mindset of pair development, as opposed to pair *programming*. It's okay to split up to do research - maybe after agreeing on the set of questions you need to answer together.

No time for yourself

We've talked about how being in a constant conversation with each other can be pretty energy draining. Most people also need some time on their own throughout the day. That is especially true for more introverted folks.

When working solo, we quite naturally take time to dig into a topic or learn when we need to. But that can feel like an interruption in pairing. So how can you take that alone and learning time when needed?

Ways to tackle

Again, don't pair 8 hours a day, agree on core coding hours with your team and keep it to a maximum of 6 hours per day. Maybe you also want to allocate a few hours self learning time.

When a pair feels that they don't have the collective knowledge to approach a problem, split up to read up and share back, then continue implementation.

Rotations lead to context switching

Knowledge sharing is one of the benefits of pairing, and we have mentioned how rotations can further increase that effect. On the other hand, too may rotations leat to frequent context switching.

Ways to tackle

Find a balance between frequency of rotations 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, and give it enough time to be effective.

Pairing requires vulnerability

> *To pair requires vulnerability. It means sharing all that you know and all that you don't know. This is hard for us. Programmers are supposed to be smart, really-crazy-smart. Most people look at what we do and say 'I could never do that.' It makes us feel a bit special, gives us a sense of pride and pride creates invulnerability.*

> -- *Tom Howlett*

When you pair, it can be hard to show that you don't know something, or feel insecure about a decision. Especially in an industry where myths like the 10x engineer regularly make their rounds, and where we have a tendency to judge each other by what languages we use, or what design decisions we took 5 years ago.

Vulnerability is often connected with weakness and in most modern cultures the display of strength is the norm. But as the researcher Brené Brown has laid out in several talks and books, vulnerability is actually a very important ingredient for innovation and change.

> *Vulnerability is the birthplace of Innovation, Creativity and Change.*
>
> -- *Brené Brown*

Ways to tackle

Showing vulnerability requires courage and creating an environment where people feel safer to show that their vulnerable. Again, this is all about building teams where people trust each other (regular 1:1s, Feedback, culture where people can ask questions, etc)

Being vulnerable is easier and less risky for people on the team who have more authority, either naturally (e.g. because they are well-respected already), or institutionally (e.g. because they have a title like "Tech Lead"). So it's important that those people start and role model this, making it the norm and therefore safer for others to be vulnerable as well.

Convincing managers and co-workers

Advocates of pair programming often struggle to convince their managers or their co-workers to make pairing part of a team's daily routine.

Ways to tackle

There is not a simple recipe to persuade others of the efficacy of pair programming. However, a key element should always be to take time to talk about it first, and make sure that everybody has the same understanding (e.g. by reading this article :-)). Then find a way to try it out, either by starting with one pair who share their experience with the others, or by proposing a team experiment, like "let's pair by default for the next 2 sprints". Make sure to build in opportunities for feedback and retrospection to share what is going well and what you are struggling with.

Ultimately, you can't force a practice on people, and it does not work for everybody. You might end up pairing with only a part of the team - at least in the beginning. From our experience the best way to convince people is by having a regular exposure to the practice, experiencing the benefits and fun their team members have while pairing.

A question that comes up most frequently in this situation is the economics of the practice: Does pairing just cost double the money, and is it ultimately worth extra cost because of the increased quality and team benefits? There are a few studies on the topic, most notably this one, that are cited to provide evidence that pairing is worth it. We are wary though of attempts to "scientifically prove" pairing effectiveness. Software development is a process full of change and uncertainty, with a lot of outcome beyond lines of code that is hard to compare and measure, like analysis, testing, or quality. Staunch opponents of pairing will always find ways to poke holes into the reproducibility of any "scientific" experiments set up to prove development productivity. In the end, you need to show that it works for YOU - and the only way to do that is to try it in your environment.

To pair or not to pair

Our experience clearly shows that pair programming is a crucial practice to create high quality, maintainable software in a sustainable way (see "Benefits"). However, we also don't believe it is helpful to approach it dogmatically and *always* pair. How exactly pair programming can be effective for you, how much of it, and for which tasks, can vary. We've found it useful to set pair programming as the "sensible default" on teams, and then discuss whenever we want to make an exception.

Let's look at a few examples where it's helpful to balance how and when you pair.

Boring Tasks

Some coding tasks are "boring", e.g. because they are about using some well defined boilerplate approach - so maybe you don't need to pair? The whole team already knows this type of approach, or it's very easy to grasp, so knowledge sharing is not that important? And live code review is less useful because the well-established pattern at hand has been used successfully in the past? So yes, maybe you don't need to pair.

However, always consider that rote tasks might be a smell for bad design: Pairing can help you find the right abstraction for that boring code. It's also more probable to miss things or make cursory errors when your brain goes into "this is easy" autopilot.

"Could I Really Do This By Myself?"

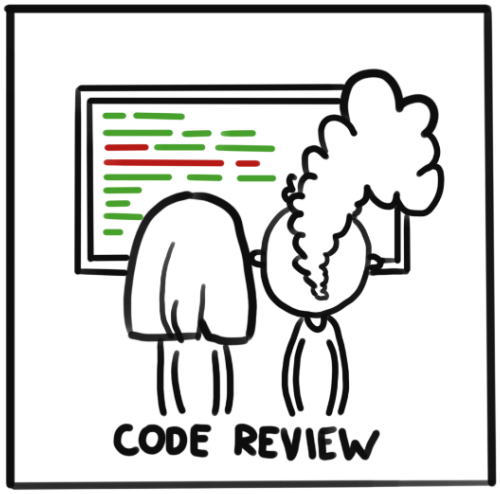
Pairing has a lot of benefits for programmers who are just starting out, because it is an opportunity to learn relatively quickly from a more experienced member of the team. However, junior programmers can also experience a loss of confidence

in their own abilities when pairing. "Could I really do this without somebody looking over my shoulder?". They also miss out on learning how to figure things out by themselves. We all go through moments of frustration and unobserved experimentation with debugging and error analysis that ultimately make us better programmers. Running into a problem ourselves is often a more effective learning experience than somebody telling us that we are going to walk into it.

There are a few ways to counteract this. One is to let junior programmers work by themselves from time to time, with a mentor who regularly checks in and does some code review. Another way is letting the more junior programmers on the team pair with each other. They can go through finding solutions together, and still dig themselves out of rabbit holes faster than if they were coding by themselves. Finally, if you are the more experienced coder in a pair, make sure to be in the navigator's seat most of the time. Give the driver space to figure things out - it's sometimes just a matter of waiting a little bit until you hit that next wall together, instead of pointing it out beforehand.

Code Review vs. Pairing

> *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*



Many people see the existence of a code review process as reason enough not to need pair programming. We disagree that code reviews are a good enough alternative to pairing.

Firstly, there are usually a few dynamics at play that can lead to sloppy or superficial code reviews. For example, when coder and reviewer rely too much on each other without making that explicit: The coder might defer a few little decisions and improvements, thinking that problems will be caught in the review. While the reviewer then relies on the diligence of the coder, trusting their work and not looking too closely at the code. Another dynamic at play is that of the sunk cost fallacy: We are usually reluctant to cause rework for something that the team already invested in.

Secondly, a code review process can disrupt the team's flow. Picking up a review task requires a context switch for somebody. So the more often code reviews occur, the more disruptive they will be for reviewers. And they should occur frequently, to ensure continuous integration of small changes. So a reviewer can become a bottleneck to integrate and deploy, which adds time pressure - again, something that leads to potentially less effective reviews.

> Check out the 2017 edition of the "State of Devops Report" for more on the performance of teams who practice trunk-based development: "Last year's results confirmed that the following development practices contribute to higher software delivery performance: Merging code into trunk on a daily basis; Having branches or forks with very short lifetimes (less than a day); Having fewer than three active branches."

With Continuous Integration (and Delivery), we want to reduce risk by delivering small chunks of changes frequently. In its original definition, this means practicing trunk-based development. With trunk-based development, delayed code reviews are even less effective, because the code changes go into the master branch immediately anyway. So pair programming and continuous integration are two practices that go hand in hand.

An approach we've seen teams use effectively is to pair by default, but use pull requests and code reviews for the exceptional cases when somebody has to change production code without pairing. In these setups, you should carefully monitor as a team that your pull requests don't live for too long, to make sure you still practice continuous integration.

But really, why bother?

We talked a lot about the benefits of pair programming, but even more about its challenges. Pairing requires a lot of different skills to get it right, and might even influence other processes in the team. So why bother? Is it really worth the hassle?

For a team to be comfortable with and successful at pair programming, they will have to work on all the skills helpful to overcome its challenges: Concentration and focus, task organisation, time management, communication, giving and receiving feedback, empathy, vulnerability - and these are actually all skills that help immensely to become a well-functioning, collaborative and effective team. Pairing gives everybody on the team a chance to work on these skills together.

> "Pairing unlocks the potential of individuals and teams to learn." Dave Farley

Another factor that is widely talked about today as a success factor for effective teams is diversity. Diversity of perspectives, genders, backgrounds and skills has proven to improve a team's performance - but it often increases friction first. It can even increase some of the challenges with pair programming we talked about. For example, one of the key ingredients we suggested is showing

vulnerability, which is especially hard for team members of underrepresented groups.

Consider this headline from Harvard Business Review: "Diverse Teams Feel Less Comfortable - and That's Why They Perform Better". The authors are making the point that "Homogeneous teams feel easier - but easy is bad for performance. (...) this idea goes against many people's intuitions". To explain, they point out a cognitive bias called the fluency heuristic: "We prefer information that is more easily processed, and judge it to be more true, or more beautiful."

This bias makes us strive for simplicity, which serves us very well in a lot of situations in software development. But we don't think it serves us well in the case of pair programming. Pairing feels hard – but that doesn't necessarily mean it's not good for a team. And most importantly, it does not have to stay hard. In this talk, Pia Nilsson describes measures her team at Spotify took to get over the uncomfortable friction initially caused by introducing practices like pair programming. Among other things, she mentions feedback culture, non-violent communication, psychological safety, humility, and having a sense of purpose.

Pair programming, extreme programming, and agile software development as a whole are all about embracing change. Agile software practitioners acknowledge that change is inevitable, so they want to be prepared for it.

We suggest that another thing we should embrace and prepare for is friction, because it's also inevitable on the way to becoming a highly effective, diverse team. By embracing friction we do NOT mean to say, "let's just have lots of conflicts and we'll get better". What we mean is that teams should equip themselves with the tools necessary to deal with friction, and have them in their toolbox by default, not just when the team is already having problems. Practice feedback, improve team communication, take measures to create a psychologically safe environment.

We believe that pair programming is often avoided because it can create friction, but we would ask you to give it a chance. If you consciously treat it as an

improvable skill, and work on getting better at it, you will end up with a more resilient team.

Acknowledgements

This text started out as a ThoughtWorks-internal slide deck which we curated and crowd-sourced to share knowledge about the practice of pair programming. Throughout the whole process we received so much feedback and great suggestions from so many people that it's hard to name everyone who contributed, we would probably forget a lot names. So we would like to thank everyone - in particular our colleagues at ThoughtWorks - who discussed with us, pointed us to further resources, left comments on the original slide deck, or reviewed this article. This has truly become a collection of many people's experiences, thank you all!

Special thanks to Stefanie Grewenig who created most of the drawings for this article.

▶ Significant Revisions