CHAPTER 21

Does Pair Programming Pay Off?

Franz Zieris, Freie Universität Berlin, Germany
Lutz Prechelt, Freie Universität Berlin, Germany

Introduction: Highly Productive Programming

Immerse yourself in the following software development scenario: You're implementing a new feature in a large, GUI-heavy information system. You found a close match among the existing features and decided to duplicate and tweak the respective code and to eventually refactor it to get rid of unwanted duplications. You already made the copy and are starting to adapt it. You feel most productive, undistracted by your surroundings, deep in the zone, focused, *in the flow*.

You look at the code and read:

`editStrategy.getGeometryType()`
You notice something odd.
That's wrong, no need to call a method here.
You understand why it feels odd.
It's always the same!
You see the parts before your inner eye, see how they fit together.
It's: `Polygon`.
You start typing.
[tap tap]
You read the IDE's auto-completion and have second thoughts.
Or is it `MultiPolygon`*?*

C. Sadowski and T. Zimmermann (eds.), *Rethinking Productivity in Software Engineering*,
https://doi.org/10.1007/978-1-4842-4221-6_21

You consider it. It would be the more general solution.
Could be. That's an open question.
There could be many reasons in favor or against. You make a decision.
`Polygon` is fine for now.
You write the code.
[tap tap]
You are satisfied and did all of this in just 15 seconds; life is great.

If you are a software developer, you know focus phases like this one. It's a great feeling when the ideas appear to be flowing directly from your brain through your fingers to become code. Who would spoil such an experience by adding another developer? At every point there would be endless discussions about which way is the best; and where there is no disagreement, there is misunderstanding because your colleagues often just don't get it.

Well, you are in for a surprise. The previous scenario was not a fictional inner monologue of a single developer. It is in fact an *actual dialogue* of two pair programmers, the two taking turns with the quotes. And it did indeed finish within 15 seconds.

Studying Pair Programming

Pair programming (PP) means that two programmers work together closely on the same programming task on a single computer.

Although super-efficient focus phases like the one described previously do happen during good pair programming sessions, most of the time pair programming evolves in a more pedestrian manner. So, does pair programming pay off overall?

To answer this, researchers have—multiple times—proceeded roughly like this:

- Devise a small task, let some developers (preferably students) solve it alone and some others in pairs, clock their time to completion, and compare the outcomes.
- Make sure the task is isolated and requires little background knowledge to ensure a level playing field for everyone.
- For greater control, assign partners randomly and set up identical workspaces for all of them.

Unfortunately, such settings do not reflect how pair programming happens in industry. The students work on machines they did not configure themselves and may

not even know their partner. Additionally, consider the difference between short-term and long-term effects. In most student PP experiments, *productivity* is reduced to the number of passing (prewritten) test cases per time spent on the task. But that's not what commonly matters in industrial contexts. Here, top priorities might be a short time-to-market or value of implemented features, or they might be long-term goals such as keeping code maintainable and avoiding information silos.

Practitioners have by and large ignored the results of these experiments. You cannot expect to learn much about how PP affects real-world productivity from a setup that so drastically differs from the real world.

In our research, we take a different approach. We talk to tech companies and observe pair programming as it happens in the wild. The pairs are in their normal environment and choose everyday development tasks and programming partners as they always do. The only difference is that we record the interaction of the pair (through webcam and microphones) and their screen content for the duration of their session—typically between one and three hours. Over the years, we have collected more than 60 such session recordings from a dozen different companies.

We analyze this material in great detail by following a qualitative research process based on grounded theory [1]. The following observations are distilled from years of studying pair programming sessions of professional software developers.

Software Development As Knowledge Work

Let's take a step back first, though. What makes programming highly productive? Psychologist Mihaly Csikszentmihalyi described a type of high-productivity mental state, which is much admired (and sometimes achieved) by software developers: flow. He places a *flow* experience in that area between *boredom* and *anxiety* where difficulty (*challenges*) and one's *skills* are on par [2].

In software development, each task is somewhat unique with its own particular challenges. Consequentially, boredom is hardly an issue for software developers. The challenges while developing software, on the other hand, are not just a matter of skill. Many stem from a lack of understanding or *knowledge*. It might take many hours of sifting through modules to finally find the right spot to add that single new if condition required. Or to understand the unfamiliar concepts used by a new library. Or to follow a stacktrace that leads into uncharted territories from the legacy part of the system. The "fluency" of a developer depends on this type of understanding and familiarity

with the software system at hand. The lack thereof is what mostly slows down software developers, more or less independent of their general skill level [3].

To work on a given task, developers (solos and pairs alike) need to understand the *system* (not *all* of it, but at least the parts relevant for the task at hand). And last week's understanding of some of these parts may already be outdated! High system understanding, let's call it *system knowledge*, is necessary to fix bugs and to implement new features.

Of course, general software development skills and expertise (we will call them *general knowledge*) are also relevant. General knowledge is about language idioms, design patterns and principles, libraries, technology stacks and frameworks, testing and debugging procedures, how to best use the editor or IDE, and the like. In contrast to the mostly product-oriented and relatively short-lived system knowledge, general knowledge is also process-oriented and more long-lived. (There is not necessarily a clear-cut separation between system and general knowledge—some pieces of knowledge may belong to both types.)

Developers build up system and general knowledge through experience, but it's not the mere number of years under their belt that matters but whether they possess applicable system and general knowledge for the task at hand.

What Actually Matters in Industrial Pair Programming

There are different PP use cases that developers regularly employ.

- *Getting help from a colleague*: One developer has been working on some task for some time and either finds it hard or needs to hand over the results, so another joins.
- *Tackling an issue together*: Two developers sit down to work on a problem together from the start.
- *Ramping up newbies*: A senior developer pairs with a new team member to bring her up to speed.

We found that it's not so much the particular PP use case that characterizes the dynamics of a session but what the two developers know and don't know—more precisely, their respective level of system knowledge and general knowledge ***concerning today's specific task***. That's because most of the work in programming consists of steps to get your system knowledge to what is needed to solve the task (general knowledge may be helpful along the way). Once you have that, actually solving the task is usually

a piece of cake—the kind of thing we described in the initial scene at the beginning. Therefore, it is the relevant *knowledge gaps* that count in programming.

Framing PP situations in terms of the involved system and general knowledge gaps helps to understand why some constellations are more beneficial than others and where pair programming actually pays off. There are three particularly interesting pair constellations we will discuss here. All of the examples in this chapter are real cases we saw in our data; we just left out some details and changed the developers' names.

Constellation A: System Knowledge Advantage

In this setting, one developer has a more complete or more up-to-date understanding of the task-relevant system parts. This is normal for the "getting help" use case but can occur in the other two as well.

Consider the scenario of developer Hannah who has been working on some task and is at one point joined by Norman. Hannah already looked at the code relevant for the current issue and performed some changes. Norman might have a better understanding of the system in general, but this does not cover all the details relevant for this task and of course not Hannah's recent code changes. Overall, Hannah has a *system knowledge advantage*.

If developers want to work as a pair, they need to address their relative system knowledge gap. Only if Norman understands what Hannah already found out and which changes she performed can they properly discuss ideas and agree on how to proceed.

But some of the pairs we observed, including this one, did not address the system knowledge advantage. Norman takes great pride in his programming skills and assumes he understands everything Hannah did. Hannah tries to explain an intricate matter she encountered, but Norman doesn't pay attention. It takes almost half an hour until Norman realizes his misconception of the status quo, lets Hannah explain it, and, at last, the pair becomes productive.

A pair situation where one partner has a system knowledge advantage (for whatever reason) is challenging because the relative system knowledge gap might be hardly visible but still needs to be addressed before the pair can move together at any speed. Better pairs therefore address the matter proactively at the beginning of their session. If your co-developer already worked on the issue, appreciate her system knowledge advantage, regardless of your own (perceived) seniority, and let her explain what she already has done and learned. We have heard that some developers with high system knowledge may also be reluctant to share what they know, but we did not observe such behavior in our pairs.

Constellation B: Collective System Knowledge Gap

When two developers start on a new task together (but not only then), they also usually both begin with an incomplete system understanding. The pair has a *collective system knowledge gap*.

Consider Paula and Peter who picked a new story card to work on. Both know their way around the system, so it doesn't take long until they find a place where to put the new feature. There are still some dependencies that need to be understood, so they navigate through the source code to complete their mental model. One time it's Paula who sees an important detail or relationship first, and the next time it's Peter. They are not deliberately taking turns here; one of them just happens to have a particular relevant idea first and will then explain it to the other. Sometimes Paula sees no need to dig deeper into the class inheritance graph, but Peter isn't as familiar with the current subsystem so he prefers to keep reading. Paula cuts him some slack and lets him take his time. In any case, both make sure their partner always stays on the same page so they can reach a high system understanding together.

Compared to the one-sided scenario of Hannah and Norman, Peter and Paula are better off. There are multiple strategies how they can build up the necessary system understanding as they don't depend on the knowledge flowing in one direction. The developers may stay closely together for a period of time, building up system knowledge in what we call an episode of knowledge "co-production" [4]. Alternatively, one developer may dig deeper in a self-paced manner, while the other is temporarily more passive ("pioneering production"). Either way, the development work done in such constellations can be very effective—*if* the pair takes care of maintaining their collaborative understanding as it grows, e.g., by explaining ("push") or getting asked about ("pull") what one of them just found out during his or her pioneering episode.

Constellation C: Complementary Knowledge

Every time a new developer joins the team, her system knowledge will be very low. But, depending on the partner's background and the nature of the current task, being low on system knowledge can occur in every PP use case. How well a pair performs then is limited by the general knowledge level of the low-system-knowledge developer. At least for the ramping-up use case, one would usually expect a twofold deficit, but this is not necessarily the case. Remember, what matters is the applicable knowledge for the current task, so with the right choice of task, even a fresh team member can score

high on general knowledge, perhaps higher than a given senior. We've seen developers on their first work day teaching their programming partner design patterns and neat tricks in the IDE. Senior developers pair up in complementary constellations as well, since neither system understanding nor generic software development skill is evenly distributed in development teams.

Andy and Marcus, for instance, have quite different competencies. Andy advocates always writing clean, readable, and maintainable code, whereas Marcus has a pragmatic approach of patching things together that get the job done. A particular module that Marcus wrote a year ago needs an update, but since Marcus has trouble figuring out how it actually works, he asks Andy for help. Their session is a complementary one: Andy has a general knowledge advantage but is low on system knowledge, as he knows next to nothing about Marcus's module; Marcus, as the module's author, has a system knowledge advantage but lacks general knowledge to systematically improve its structure. Their session is mutually satisfactory, as they get the job done *and* Marcus learns a lot about code smells and refactorings.

So, Again: Does Pair Programming Pay Off?

You probably now appreciate that "Does pair programming pay off?" is an entirely inappropriate question, because

- It is hard to tell since too many different benefits have to be quantified and added up with respect to code functionality, code and design quality, and learning within the team.
- It depends, because different knowledge and task constellations provide very different opportunities for being efficient as a pair.

The key aspects are the knowledge gaps the developers have to deal with. To succeed with the task, the pair as a whole can benefit from various pieces of pertinent-for-this-task general software development knowledge and absolutely must possess or build the pertinent-for-this-task system knowledge. As system knowledge is more short-lived, it is usually the scarcer resource.

If the task-relevant knowledge of a pair is highly complementary, a pair programming session will probably pay for its cost multiple times. But even if it is not and the pair's visible work output is less than the two could have produced as two solo programmers, the PP session's midterm benefits in terms of learning provide ample

opportunity for time saved in the future and mistakes not made in the future to pay off the higher expense today.

From an industrial perspective, an answer to the question might be this: given the dominant role of system knowledge for productive development, companies may not like to let their top-general-knowledge developer go, but they are *terrified* of losing their single top-system-knowledge developer. And frequent pair programming is an excellent technique to make sure system knowledge spreads continuously across a team.

Key Ideas

The following are the key ideas from this chapter:

- Pair programming will tend to pay off if the pair manages to have high process fluency.
- Pair programming will pay off if the pair members' knowledge is nicely complementary.

References

[1] Stephan Salinger, Laura Plonka, Lutz Prechelt: "A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming," Human Technology: An Interdisciplinary Journal on Humans in ICT Environments, Vol. 4 No. 1, 2008, pp.9–25

[2] Mihaly Csikszentmihalyi: "Flow: The Psychology of Optimal Experience," Harper Perennial Modern Classics, 2008, p.74

[3] Minghui Zhou, Audris Mockus: "Developer Fluency: Achieving True Mastery in Software Projects," Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10), 2010, pp.137–146

[4] Franz Zieris, Lutz Prechelt: "On Knowledge Transfer Skill in Pair Programming," Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2014