Solo Programming vs Pair Programming vs Mob Programming

Arvid Gotthard (gotthard@kth.se)

Introduction

One of the core principles of DevOps is to promote collaboration in every facet of the software development process by bringing development and operations under the same roof. As a single team is responsible for the quality of the software at every step, multi-disciplinary teams are required as the tasks are "full-stack" per definition [1].

The goal of a DevOps team is to be autonomous and share the responsibility of the software within the team. To achieve this the team must collaborate effectively during both the development and operation phases of the project [2].

This essay will explore three paradigms of collaboration in programming; *Solo Programming*, *Pair Programming* and *Mob Programming*.

A brief introduction of the three different paradigms will be provided followed by a discussion and comparison of the paradigms and how these practices could be integrated into DevOps teams.

Solo programming

The term *Solo Programming* refers to the process of having individual developers work on separate problems in parallel. This style of development feels very natural and is widely used.

Teams that solo program does still collaborate, but not actively during the development phase. The main form of knowledge sharing between members is through initial design meetings and a process called *code review*, where the developer will submit the changes for manual review [3].

Code review is a quality assurance measure as well as a vehicle for knowledge sharing within teams. At its inception, the process of code review was known as a *Fagan Inspection*, named after its creator Micheal Fagan. A *Fagan Inspection* is highly formalized and covers the entire programming process from the initial design meetings, iteration on the software to final testing of the software. During the production of the software, there would be code inspection where the reviewer would produce a detailed written report documenting how many issues were found and their severity.[4]

This highly formalized code review process is no longer used to the same extent. Many organizations have opted for a more lightweight approach consisting of three steps [5]

- Submission of the change.
- Discussion among developers about the change. Potential fixes are suggested and the change can be re-submitted multiple times.
- Approval from one or more reviewers.

Organizations usually rely on tools to facilitate the review process [3], such as the *pull request* feature on GitHub [6], and popular branching strategies such as *gitflow* [7] promote code review.

Advantages and Criticisms of Solo Programming

By employing solo programming you are using all the team members to full capacity and conversely, you are minimizing the personnel hours and overall costs[8]. And by employing code review you can still share knowledge while ensuring the quality of the software. This method also allows a developer to deeply focus on the task at hand.

The common criticisms of solo programming are related to the context switching that is required for code review. Lacking knowledge of the code to review, and the context switch from development to reviewing was listed as the main challenges associated with code review in a survey study of Mozilla developers [8].

The time it takes from the initial code review submission until it is accepted is also a problem of the review process in solo programming. At Google, the median review time is 4 hours [3], 17.5 hours at AMD, 15.7 hours for Chrome OS, and 14.7, 19.8, and 18.9 hours respectively for Bing, MS SQL, and MS Office [5]. With such huge latencies, it is hard to provide a fast feedback loop in development.

Pair programming

In *Pair Programming*, two developers are simultaneously working on the same problem on a single computer. The most traditional style of pair programming is called *Driver and Navigator*. The driver will sit at the keyboard typing, while the navigator will provide the instruction to the driver. The driver can have a narrow focus on solving small problems related to coding, while the navigator can have a more holistic perspective on the system while providing real-time feedback on the code that the driver writes [9]. At certain intervals, the two developers switch roles.

This practice is meant to increase communication and collaboration between the developers. However, there is a risk that the navigator will disengage and resolve to just watch the driver code. A countermeasure for this is to employ *strong pairing* which is a practice that is predicated on the following rule,

> "For an idea to go from your head into the computer it MUST go through someone else's hands" [10]

In this scenario, the driver fully trusts the navigator to steer him in the right direction, while the navigator can not disengage as he/she is in charge [10].

Another style of pair programming that works well in *Test Driven Development*(TDD) is *Ping Pong.* In this style, the process starts with one of the developers writing a failing test for the feature that is to be implemented. The other developer will then write an implementation that passes this test and then passes back the control to the first developer to write the next failing test [11].

Advantages and Criticisms of Pair Programming

The promise of pair programming is that it produces higher quality software compared to solo development [9], which could potentially eliminate slow quality assurance measures such as code review. A multitude of studies have been conducted to test this statement, but the evidence is somewhat inconclusive[12]. A meta-study of the effectiveness of pair programming showed that there was a small significant effect on code quality and a medium positive effect on development time. However, the study also concluded that the beneficial properties of pair programming are not universally emergent [13].

Another draw of pair programming is that it promotes learning for the developers, especially for the junior developers that pair up with senior developers [9]. In another meta-study of pair programming [14] learning was consistently better in pair programming compared to solo programming.

Generally, you would want the developers in the pair to be co-located, which could be limiting logistically for businesses that employ remote engineers. However, it is possible to do pair programming remotely [11] using some video chat software with screen-sharing capabilities like Zoom [15] or Teams [16]. This solution may be a good compromise but is not completely satisfactory, co-locating is still the most effective way of pairing.

Mob programming

Mob Programming is an extension of pair programming where 3 or more developers are working on a single problem on the same computer. Much like in strong pairing the flow of ideas is meant to pass through multiple sets of hands [17]. As in classical pair programming, there is a driver and navigator, but while the navigator is instructing the driver he/she is also broadcasting the ideas to the entire group which facilities a collective understanding of the solution [18].

“Mobbing” (the act of practicing mob programming) poses some logistic problems compared with regular pair programming. As there is a larger set of people collaborating, an isolated workspace is needed. The workspace should have the necessary tools, such as a projector, to allow all the members of the “mob” to partake in the coding process[18]. Another tool that can be utilised during “mobbing” is `Mob.sh` which is a wrapper for git by streamlining the rotation of the driver and navigator in the mob. This tool is especially useful for remote mob programming [19].

Advantages and Criticisms of Mob Programming

Mob programming shares many of the positive aspects with pair programming, but the main motivation for “mobbing” how easy it is to share knowledge within the team. Multiple organizations that have adopted mob programming have attested to an improved environment for learning and knowledge sharing [20] [21]. However, generally these organizations did not adopt mob programming dogmatically and had some hesitations against the practice.

In the case of *Unruly* [20], they observed that there was a tendency for *Group-Think* as teams developed a sort of identity. Over time, this resulted in a skewed perception of the problem. *Unruly* also identified a problem with personalities that dominated the mobbing session, where other team members would simply defer to that person in discussions. In general, all the practitioners agreed to not use mob programming for all sprints, and that the practice was best suited for complex problems. They also observed a small increase in delivery time.

A development team from New Zeeland reported a shared sense of ownership over the code after practicing “mobbing” for 18 months, as well as the onboarding process for new team members was quicker[21].

In an experiment where teams from *Qualogy Solutions* that solely consisted of junior developers practiced mob programming weekly, there was significant accelerated learning according to the agile coach that led the session [22]. Although this result is highly subjective, it is an example of one of the advertised benefits of mob programming.

In all of the above-mentioned cases of mob programming, there were logistical challenges with assembling the team in one place and setting up an environment so that everyone in the group could contribute comfortably.

Discussion

I have mainly worked in environments where I have solo programmed with code review on all pull requests, and from my experience, a lot of my time is spent waiting for a senior developer to review my code. I am very intrigued by the potential benefits of pair programming and mob programming.

If you look at the state of the art when it comes to solo, pair, and mob programming, you can not really draw a definitive conclusion on which of the paradigms that is the most effective.

The quantitative data suggest a potential minor improvement of code quality when pair programming, and a potential speed increase. But the cost is still almost twice as high compared to solo programming. But these results are still deemed to be inconclusive. There is a lack of quantitative data and actual scientific proof of the effectiveness of mob programming. All studies recount personal experiences with “mobbing” and subjective opinions [23].

In the context of a DevOps team, methods such as pair programming and mob programming make sense as it promotes collaboration which is the cornerstone of an effective DevOps team [24]. For instance, imagine a DevOps team that consists of developers, operations, designers and managers that practices mob programming. In such a scenario all the members would at some point have touched code at all points of the development process, which aligns very well with the concept “you build it, you run it” in DevOps culture[2].

I believe most of the benefits of pair and mob programming are for the developers, and in general, it seems hard to motivate a business to pair or “mob” as the main way of working as the costs are so much higher than solo programming. However, I believe that if you as a business have the resources to allow some of these techniques as a complement to regular solo programming with code reviews it can provide benefits when comes to stimulating learning of your developers. I also believe that there is no point in dogmatically choosing one of the paradigms for an organization, rather they should be seen as tools that can be employed in different situations.

In my eyes in the best case scenario, you would do solo programming as the default way of working. Then you would encourage developers within the team to pair programming, especially for complex problems. You could also occasionally do “mobbing”, especially when a new team is being assembled as a team building exercise.

References

[1] “DevOps principles | atlassian.” https://www.atlassian.com/devops/what-is-devops.

[2] “DevOps culture | atlassian.” https://www.atlassian.com/devops/what-is-devops/devops-culture.

[3] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: A case study at google,” in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, p. 181190. doi: 10.1145/3183519.3183525.

[4] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Syst. J.*, vol. 15, no. 3, p. 182211, Sep. 1976, doi: 10.1147/sj.153.0182.

[5] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, p. 202212. doi: 10.1145/2491411.2491444.

[6] “About pull requests - GitHub docs.” https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

[7] “Gitflow workflow | atlassian git tutorial.” https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

[8] E. Arisholm, H. Gallis, T. Dyba, and D. I. Sjoberg, “Evaluating pair programming with respect to system complexity and programmer expertise,” *IEEE Transactions on Software Engineering*, vol. 33, no. 2, pp. 65–86, 2007.

[9] K. Beck, *Extreme programming explained: Embrace change*. addison-wesley professional, 2000.

[10] “The way things work in llewellyn’s world: Llewellyn’s strong-style pairing.” http://llewellynfalco.blogspot.com/2014/06/llewellyns-strong-style-pairing.html.

[11] “On pair programming.” https://martinfowler.com/articles/on-pair-programming.html.

[12] “The benefits of pair programming: Will it work on your team? · raygun blog.” https://raygun.com/blog/pair-programming-benefits/.

[13] J. E. Hannay, T. Dybå, E. Arisholm, and D. I. K. Sjøberg, “The effectiveness of pair programming: A meta-analysis,” *Information and Software Technology*, vol. 51, no. 7, pp. 1110–1122, 2009, doi: https://doi.org/10.1016/j.infsof.2009.02.001.

[14] C. Alves De Lima Salge and N. Berente, “Pair programming vs. Solo programming: What do we know after 15 years of research?” in *2016 49th hawaii international conference on system sciences (HICSS)*, 2016, pp. 5398–5406. doi: 10.1109/HICSS.2016.667.

[15] “Video conferencing, cloud phone, webinars, chat, virtual events | zoom.” https://zoom.us/.

[16] “Logga in i microsoft teams.” https://www.microsoft.com/sv-se/microsoft-teams/log-in.

[17] D. Ståhl and T. Mårtensson, “Mob programming: From avant-garde experimentation to established practice,” *Journal of Systems and Software*, vol. 180, p. 111017, 2021, doi: https://doi.org/10.1016/j.jss.2021.111017.

[18] W. Zuill and K. Meadows, “Mob programming: A whole team approach,” in *Agile 2014 conference, orlando, florida*, 2016, vol. 3.

[19] "Fast git handover with mob | tool for smooth git handover." https://mob.sh/.

[20] A. Wilson, "Mob programming - what works, what doesn't," in *Agile processes in software engineering and extreme programming*, 2015, pp. 319–325.

[21] J. Buchan and M. Pearl, "Leveraging the mob mentality: An experience report on mob programming," in *Proceedings of the 22nd international conference on evaluation and assessment in software engineering 2018*, 2018, p. 199204. doi: 10.1145/3210459.3210482.

[22] K. Boekhout, "Mob programming: Find fun faster," in *Agile processes, in software engineering, and extreme programming*, 2016, pp. 185–192.

[23] M. Shiraishi, H. Washizaki, Y. Fukazawa, and J. Yoder, "Mob programming: A systematic literature review," in *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, 2019, vol. 2, pp. 616–621. doi: 10.1109/COMPSAC.2019.10276.

[24] "The importance of team structure in DevOps | atlassian." https://www.atlassian.com/devops/frameworks/team-structure.