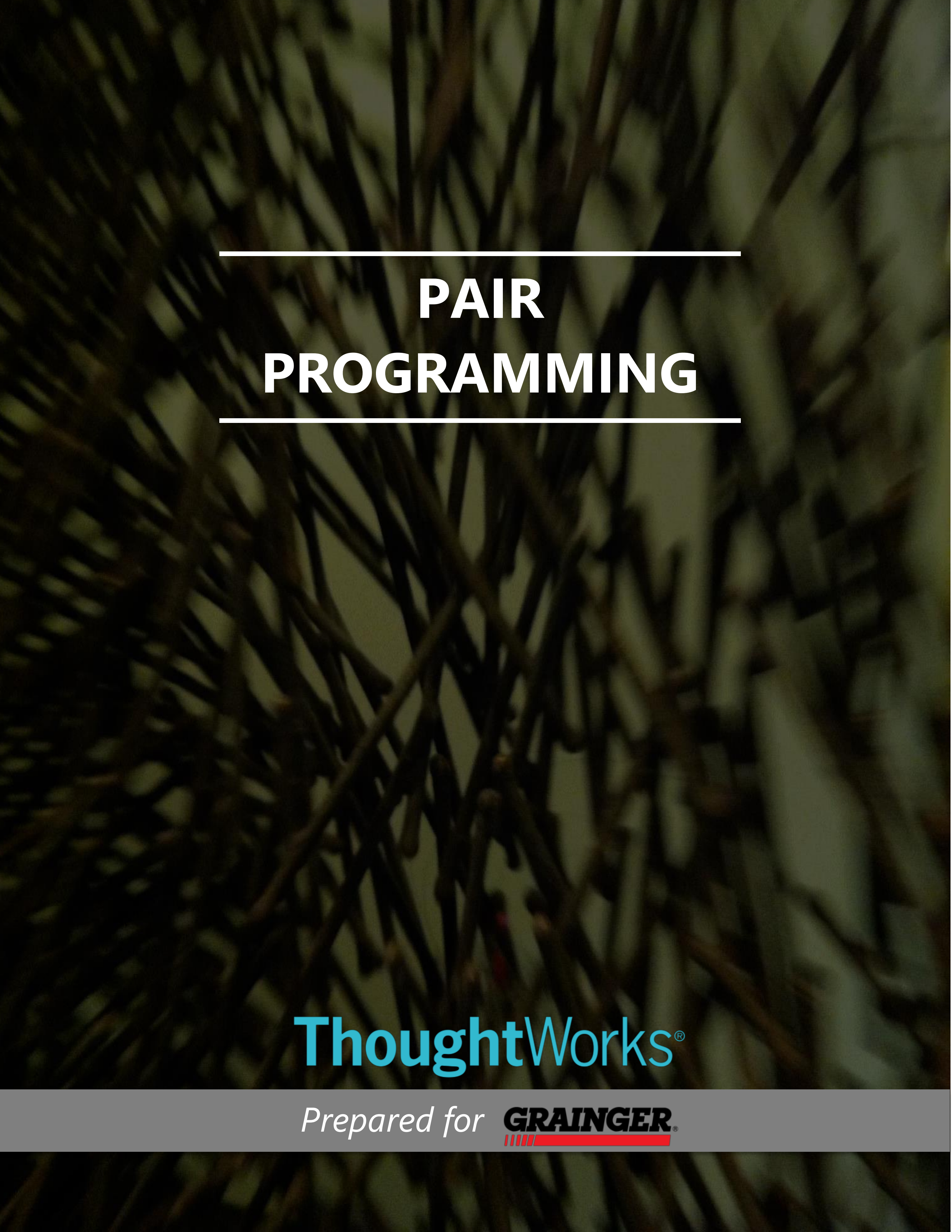


PAIR PROGRAMMING

ThoughtWorks®

Prepared for GRAINGER®

Executive Summary

1. The general goal of software development in a modern digital business is to deliver value-creating features to customers as quickly and assuredly as possible.

2. Viewed as a system, queuing theory shows that optimizing only one component in a development workflow can actually result in waste and delay, as new bottlenecks and non-linearities arise.

3. Defects found in production are exponentially more expensive to fix - up to 100x - than defects found during development. In addition, bugs in production may have expenses related to brand, business outcome targets, and customer experience metrics that bugs in development do not have.

4. Developing high-quality, extensible, high-performing custom software is a complex task requiring the accurate application of large amounts of technical skills, knowledge, and expertise.

5. Pair programming, or pair development, is one of a set of powerful, commonly used Agile technique to increase the quality of code written and to reduce defects found downstream of development. It is part of ThoughtWorks' "Sensible Defaults" for all our client work.

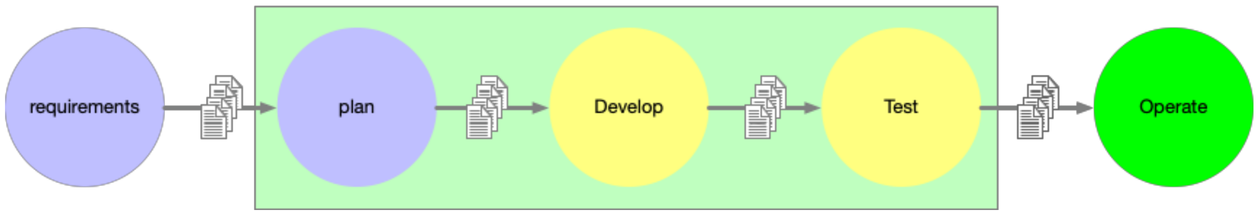
Why Pair Programming?

Software Development as a System

The general goal of software development in a digital business is to **deliver value-creating features to customers as quickly and assuredly as possible**. Shipping bad software, creating security holes, or causing operational problems either destroys value (including brand value), wastes time, or more commonly both. Software that is hard to evolve to meet new requirements restricts the value that can be created in the future or delays it significantly.

Therefore, the goal is to balance increasing the flow of software to production while increasing the quality and security of solutions, which produces value delivered to the customer.

No matter the timeframe involved, generally software development follows the same path:



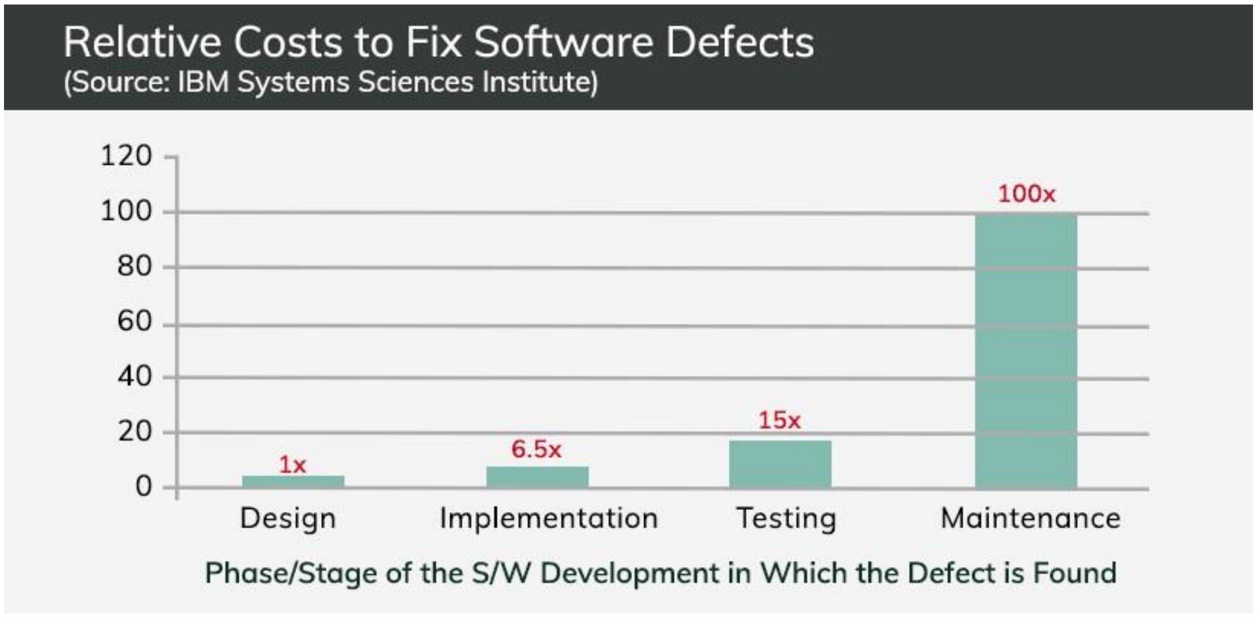
Each step in the process creates a backlog of incomplete work that the next phase works on. Based on the work of Lean practitioners[1], there are three key observations:

- Bottlenecks form when backlogs exceed the capacity of the next step to process them. The bottleneck dictates an upper limit for the throughput of the entire system. Bottlenecks are created by large design documents, more detailed requirements, and large codebases to test at the end..
- Since the goal is to deliver features to customer, the entire system flow should be optimized.
- Optimizing one step often decreases the flow of value across the whole, as it exposes non-linearities in processing at each step. For example, adding more developers actually will slow down production as coordination costs increase; it may also create a larger backlog than is manageable in the testing phase, and result in even longer cycle times.

Since the goal of the software development system is to get valuable features into production, **the goal should be to optimize the flow of work product across the *entire* system**.

[1] Poppendieck, Mary, Tom Poppendieck, and Thomas David Poppendieck. 2003. *Lean Software Development: An Agile Toolkit,* Addison-Wesley Professional. This book draws on the Theory of Constraints (Goldratt, "The Goal") and work in Queuing Theory

The second key set of observations is that the **cost of remediating defects only climbs as you move to the right**. An IBM study[2] found that, in traditionally structured projects, bugs in production can cost 1000x as much to fix as bugs found early.



Another systemic effect that can be observed is that developers must stop any work on new features and switch to fixing the bug, impacting the flow of value into production and reduces developer effectiveness.

Therefore, another goal of a software production system should be to **find bugs as early as possible**.

What is Programming?

> ***Myth: 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."* [3]

Software development is the process of turning written and drawn descriptions and models of features and requirements into working software in production.

It is an intensely knowledge-based work; it requires knowing languages and idioms, using complex toolsets, a knowledge of best patterns and practices from a rapidly evolving industry, and the accurate construction of working production software.

[2] https://www.isixsigma.com/industries/software-it/defect-prevention-reducing-costs-and-enhancing-quality/ contains a discussion of this. There is a more recent study (arXiv:1609.04886v1 [cs.SE] 2016) that shows that the *delayed issue effect* may no longer be consistently observable in modern software projects but posits that is **because** of the widespread adoption of agile practices and devops tooling we're discussing here.

[3] Martin Fowler is the "Chief Scientist" of ThoughtWorks, a title he thinks is "exceedingly inappropriate". He has written extensively, is the author of nine books on software development, and is an original signatory of the Agile Manifesto.

Developing software in a complex environment of regulatory issues, data privacy and security needs, and legacy systems integration takes another layer of careful thought.

Finally, for custom software that is innovative and business re-defining, the act of development is often adapting existing patterns and breaking new ground. We're writing new, custom software because no package solution meets our current and future needs, and Grainger wants full control over its future, in terms of both features and speed-to-value.

Given a feature story, developers commonly do most of the following:

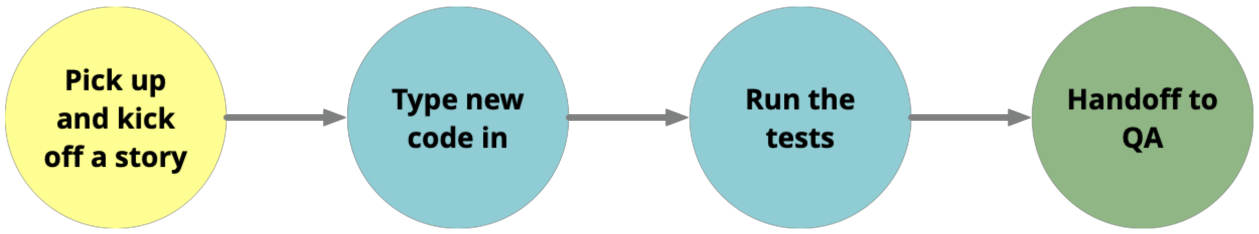
- Analyze the story and define the technical problems necessary to create the functionality described by the story. Figure out the performance, security needs.
- Research different approaches to the problem. Does Grainger already have a solution, and API, or a coding pattern that's been approved? Does that solution cover all the use case requirements? Is there a better or newer approach out in the industry? Didn't I do something like this in my last project? Will this new approach work?
- A developer might do a POC just to be sure the approach will work, or spike on this algorithm, or ask other developers or architects in Architecture Communities of Practice.
- Research the existing code: quite often, there is existing code to wrap around, integrate with, or add new functionality to. Are the existing tests complete enough so that I feel confident changing code? What does that odd function actually do? If we double the number of calls to that API, are they ready for the increase in traffic? How do I make the minimal change to get the story working? Is the person who wrote it here to talk to?
- Tests are also code that needs to be written. Developers enter a cycle of writing the tests, writing the code, testing the code, fixing the code (and maybe the tests), writing more tests, writing more code. Repeat as necessary.
- Potentially have a formal code review by the lead developers to ensure quality goals are met.
- Release the code to other developers on your team and deploy to QA for them to run final functional and integration tests. Fix any bugs found.

In addition, developers are doing all the other tasks necessary for the organization to keep track of project process, like updating Jira tickets, attending standups, status reporting, doing necessary architecture and security reviews, etc. Recent industry surveys find that most developers spend somewhere between 50-75% of the workday actually writing code[4].

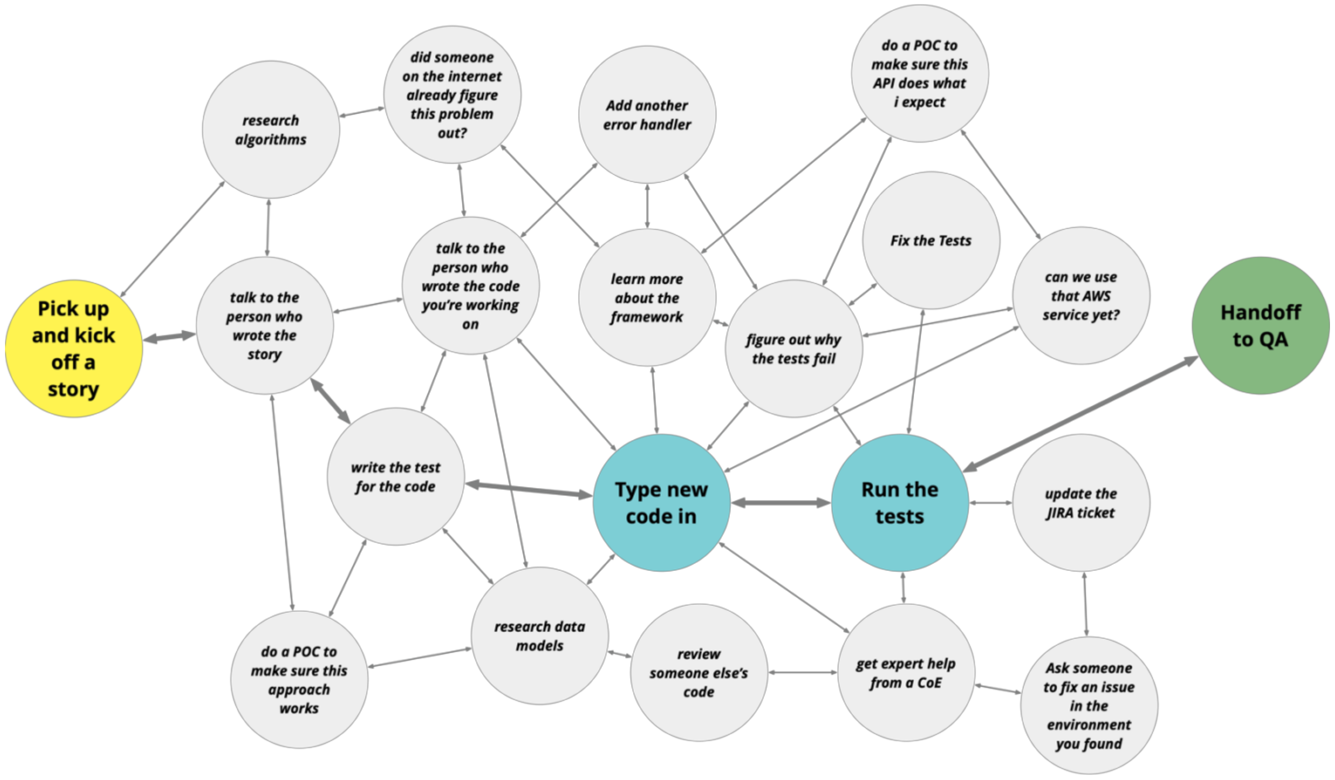
These are all necessary tasks for building quality software that is maintainable, flexible, resilient, integrated into deep systems correctly, and more adaptable to new or improved requirements.

[4] "Developer Survey 2019 - Open Source Runtime Pains." ActiveState, May 7, 2019. https://www.activestate.com/resources/white-papers/developer-survey-2019-open-source-runtime-pains/. Admittedly focused on developers using open source, but that is most of the developer population. The productivity section is more broadly focused. Also see: https://insights.stackoverflow.com/survey/2019#work-_-weighting-productivity-challenges-in-the-us for a breakdown of productivity challenges. Spoiler: it's meetings and being interrupted.

So, while development may look like this from the outside:



It's really more like this on the inside.


research algorithms
did someone on the internet already figure this problem out?
Add another error handler
do a POC to make sure this API does what i expect
Fix the Tests
can we use that AWS service yet?
Pick up and kick off a story
talk to the person who wrote the story
talk to the person who wrote the code you're working on
learn more about the framework
figure out why the tests fail
Handoff to QA
write the test for the code
Type new code in
Run the tests
update the JIRA ticket
do a POC to make sure this approach works
research data models
review someone else's code
get expert help from a CoE
Ask someone to fix an issue in the environment you found

What is Pair Programming?

Pair programming was more recently popularized by the Extreme Programming methodology but has been practiced since the early days of software engineering. Famously, critical parts of Google's core software were written by a pair of programmers over a weekend[5]. Pairing is in use, in one form or another, at many high-tech companies and consultancies[6].

There are several different styles of pairing - ping-pong, pomodoro, tourist. The most common style, and the most common style of pairing in use at ThoughtWorks, is two programmers who physically share a workstation, with a single view of the desktop (either in one monitor or with two mirrored screens), and only one active keyboard. There are two roles in a pair: the **navigator**, and the **driver** - a good metaphor is rally racing, where one person concentrates tactically on driving the car, and the other looks strategically ahead to what's next on the GPS, spots upcoming trouble, and generally guides the driver to success:

- The **driver** is the person who owns the keyboard; they are writing the code, runs tests, focusing on one problem at a time. The driver commonly verbalizes their thought process with the navigator, so the navigator knows what the driver is thinking and can effectively help.

- The **navigator** is their thinking partner, providing advice and guidance; helps the driver explain their approach, researches potential new approaches, makes suggestions and helps avoid dead ends, reviews the code being written in real-time, looks ahead to the next steps, and generally helps the driver think, code, and test.

The roles can switch during a pairing session as the pair sees fit, and the pair can take breaks when needed to refresh their attention (and grab coffee, etc.).

Why Program in Pairs?

So how does pair development help software development?

1. **Parallel effort:** The pair effectively do many of the activities of programming that we noted above in parallel. For example, the navigator can focus on doing research, seeking external advice, checking API documentation, all while the developer is writing tests and code.

   In addition, the navigator is continually reviewing the code the driver is writing and checking the approach the driver is taking against the requirements of the story. This helps build quality into the code that the pair releases.

2. **Reduce waste**: having two minds work together on solving problems brings a wider perspective, and a wider set of experiences. It also means stories will be more accurately delivered, since misunderstandings are exposed early.

   Two people also means the navigator can be researching the next steps in the development, the framework being used, how other people have solved the same problem, and getting advice and

[5] Albeit most amazing programmers: https://www.newyorker.com/magazine/2018/12/10/the-friendship-that-made-google-huge

[6] Accenture, TCS, Cognizant, among others list "pair programming" as an expected skill in job postings.

guidance from SMEs while productive coding is going on. Missteps in development are caught early, lessening rework.

This improves both the quality of the task at hand and reduces the amount of rework.

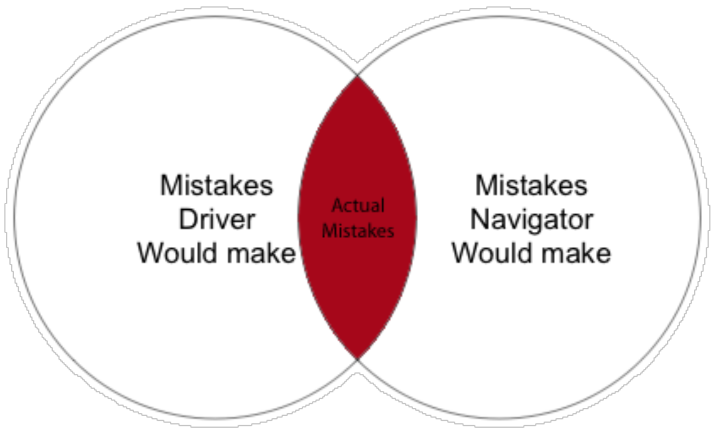
3. **Share knowledge**: By pairing, developers understand more of the codebase than one narrow area. This means they become more flexible in the stories they can pick up and start work on, since they know more of the codebase. Additionally, this makes the team more resilient to someone being sick, on vacation, or winning the lottery.

   This increases overall system flow by reducing wait states.

   In addition, pair programming can drive engineering culture change by providing a high-touch avenue to drive behavior change.

4. **Builds quality in**: During pair programming, code reviews happen constantly, and in real-time, so that both test coverage and remediation of an entire class of errors happens early on - and on smaller batches of code. Coverage of tests increases, as well.

   Two minds with different experiences and skills, also will cover a broader range of possible problems and edge cases gotchas in the code, and make less mistakes[7]



   Secondly, more formal QA steps can be focused on higher value activities, and in general are reduced. This can result in less full cycle time for getting ideas into production.

   This not only brings more expertise to bear on tricky problems, it means errors can be caught earlier, which reduces costs since bugs are exponentially more expensive to fix the closer you get to production.

Does it Work?

One well regarded posterior study has shown that pair development can take up to 15% longer to complete

[7] Harris, Sam. 2017. "The Benefits And Pitfalls Of Pair Programming In The Workplace". Blog. Freecode Camp. https://www.freecodecamp.org/news/the-benefits-and-pitfalls-of-pair-programming-in-the-workplace-e68c3ed3c81f/ and Gartner note "Master Shift-Left Testing and Increase Feedback to Improve Code Quality "

stories but result in 15% less defects[7]. Other studies have shown a 48% increase in program correctness without an increase in time[8]. As defects in production can be 100% more expensive, the economics are in favor of pairing[9].

Recent surveys of industry professionals showed "...respondents with pair-programming experience viewed pair programming more positively than those without it. Finally, the more pair-programming experience the respondents had, the more favorably they viewed pair programming.[10]"

So pair development is both effective in terms of handling complex software tasks, improving quality, and, in general, is preferable. Once you try it, you want to use it more - in the appropriate circumstances.

When Does it Not Work?

Pairing is not for every situation, and developers can and do program alone when appropriate. Here's a few common situations when pairing is not effective:

- When both developers need to do research, go investigate the suitability of various libraries, or do small POCs and spikes that try to find the best approach to a problem. It's often effective to go off on their own and come back to the pair with new information.
- When both pairs are novice programmers, which can lead to stuck pairs that make little or poor-quality progress. Pairing expert-novice or two experts seems to work best[11].
- When the task is - or has become - well-known, straightforward, or when the code being written is rote[12].

There are also times where a pair may not be available (during coffee breaks, or maybe if the team has an odd number of developers due to vacation or illness) and coding should not stop.

[8] (Arisholm et al. 2007)

[9] (Cockburn, Williams 2001). pp 223-243.

[10] (Sun, Marakas 2015)

[11] (Lui et al. 2006)

[12] If you're spending a lot of programming time and money writing basic, rote code, you have a different problem. Also, a pair would probably find an existing library or starter kit/pattern at Grainger to replace it, or would find a better and more necessary abstraction of the problem.

Pairing at Grainger

Based on our experience helping clients through digital transformations, ThoughtWorks "Sensible Default" for all projects is to pair develop as much as possible. We have been pair developing on the Delivery Platform, the KeepStock replatform project, and Grainger 2.0 capability development for PIM, CIM and PUB. On the whole, we find we are pair-developing about 50% of the working time here at Grainger. Feedback about pair programming from Grainger devs is, on the whole, very positive.

We continue to work with teams to understand the optimal mix for Grainger of pairing and solo work, but have had tangible results for both KeepStock, and for the Delivery Platform, which is supporting more teams and more complex deployment scenarios than expected for their MVP.

In the end, teams should make decisions on how they can best meet their targets and OKRs, and their expectations of quality;[13] pair development is one powerful tool for not only development teams but business capability teams to achieve their goals.

[13] "The power behind the Toyota Production System (progenitor of Lean thinking) is a company's management commitment to continuously invest in its people and promote a culture of continuous improvement" (Liker, "The Toyota Way", 2004). Wakamatsu and Kondo, TPS experts: "The essence of [the Toyota system] is that each individual employee is given the opportunity to find problems in his own way of working, to solve them, and to make improvements."

References

1. Arisholm, Erik, Hans Gallis, Tore Dyba, and Dag I.K. Sjoberg. 2007. "Evaluating Pair Programming With Respect To System Complexity And Programmer Expertise". *IEEE Transactions On Software Engineering* 33 (2): 65-86. doi:10.1109/tse.2007.17. https://ieeexplore.ieee.org/document/4052584

2. Cockburn, Alistair and Laurie L. Williams. "The costs and benefits of pair programming." (2001). https://www.semanticscholar.org/paper/The-costs-and-benefits-of-pair-programming-Cockburn-Williams/f6be1786de10bf484decb16762fa66819930965e

3. Lo Giudice, Diego. "State of Agile 2017: Agile at Scale". Forrester Research (2017).

4. Lo Giudice, Diego, "Build the Right Software Faster With Agile and DevOps Metrics". Forrester Research (2018).

5. Fowler, Martin. 2016. "Pair Programming Misconceptions". Blog. *Martinfowler.Com*. https://www.martinfowler.com/bliki/PairProgrammingMisconceptions.html.

6. Hannay, Jo E., Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg. 2009. "The Effectiveness Of Pair Programming: A Meta-Analysis". *Information And Software Technology* 51 (7): 1110-1122. doi:10.1016/j.infsof.2009.02.001. https://www.sciencedirect.com/science/article/abs/pii/S0950584909000123

7. Kim Man Lui, K.C.C. Chan, and J.T. Nosek. 2008. "The Effect Of Pairs In Program Design Tasks". *IEEE Transactions On Software Engineering* 34 (2): 197-211. doi:10.1109/tse.2007.70755. https://www.semanticscholar.org/paper/The-Effect-of-Pairs-in-Program-Design-Tasks-Lui-Chan/dec1d0ca051f5c61df19745df599e158d1c441d9

8. Lui, Kim Man, and Keith C.C. Chan. 2006. "Pair Programming Productivity: Novice–Novice Vs. Expert–Expert". *International Journal Of Human-Computer Studies* 64 (9): 915-925. doi:10.1016/j.ijhcs.2006.04.010. https://www.cs.utexas.edu/users/mckinley/305j/pair-hcs-2006.pdf

9. Sun, Wenying, George Marakas, and Miguel Aguirre-Urreta. 2016. "The Effectiveness Of Pair Programming: Software Professionals' Perceptions". *IEEE Software* 33 (4): 72-79. doi:10.1109/ms.2015.106. https://www.computer.org/csdl/magazine/so/2016/04/mso2016040072/13rRUyeTVg8

10. Williams, Laurie L. and Hakan Erdogmus. "On the Economic Feasibility of Pair Programming." (2002). https://pdfs.semanticscholar.org/3918/81acebcf21072364316b812617c06140f67f.pdf

11. Williams, Laurie, and Robert Kessler. 2003. *Pair Programming Illuminated*. Boston: Addison-Wesley. https://www.amazon.com/dp/0201745763

Various ThoughtWorks posts on pair programming:
https://www.thoughtworks.com/insights/blog/effective-navigation-in-pair-programming
https://www.thoughtworks.com/insights/blog/pairing-are-you-doing-it-wrong

An exhaustive list of relevant blog postings and books:
https://insimpleterms.blog/paired-programming-useful-articles-resources-and-research