Editor: **Ciera Jaspan**
Google
ciera@google.com

Editor: **Collin Green**
Google
colling@google.com

# Developer Productivity for Humans, Part 7: Software Quality

Collin Green, Ciera Jaspan, Maggie Hodges, and Jessica Lin

**AT GOOGLE, WE** regularly get asked to help teams measure how different developer tools and processes impact productivity. A common form of this request is a team that has built a new developer tool and wants to show that this tool increases developer velocity.

## Speed, Ease, and Quality

Yet developer velocity is clearly not the only goal; we also want to build high-quality products. After all, we can easily increase velocity…by removing code review or test suites. Doing so might make our developer velocity appear faster, but it would clearly not be a good strategy for the company. So while we do want higher velocity, we don't want it to come at the expense of our software quality. We also don't want it to come at the expense of our engineers; we could increase velocity by asking everyone to work overtime, but that's also going to be trading off short-term positive gains for long-term negative impact.

*Digital Object Identifier 10.1109/MS.2023.3324830*
*Date of current version: 20 December 2023*

Due to these tradeoffs, we measure three components of developer productivity: speed, ease, and quality. Even if we are expecting to influence only one of those three, it's important to measure all three components to ensure that we aren't making an unexpected tradeoff. This is not an unheard of idea; Microsoft uses the SPACE framework,[1] which has several overlapping concepts, and both frameworks have their origins from a Dagstuhl seminar in March 2017 where 27 researchers from across academia and industry gathered to discuss developer productivity.[2] At the end of that seminar, we had a similar set of components and published the discussions in a book.[3] In the end, it doesn't really matter which specific components one uses; what matters is the recognition that developer productivity is a complex topic with several interwoven factors, and we need to measure each of them to ensure that we get a complete picture of productivity.

In this installment of our column, we're going to dive in on just one of these components: quality. Of the three components, quality is the most difficult to measure because it is also the most difficult to define. What *is* software quality anyway? Software quality means different things to different people. To the vice president concerned about their business, high software quality means having a product that people want to use, pay for, and recommend to others. To the developer, high software quality means that the code itself is maintainable and easy to work with. To operations, high software quality means a site that is reliable, fault tolerant, and resilient to security threats. These are all valuable perspectives on software quality, but if these three people enter into a conversation about "how will we increase software quality," they're likely to find disagreement in how they approach the problem and how they measure success. We've seen these conversations play out even at Google, so it became imperative for us to provide everyone with a shared understanding of software quality that would encompass these viewpoints and how they interact.

## Four Types of Software Quality

To better understand what "quality" means to a software developer, we

conducted two series of interviews with developers at Google. In the first series, we asked eight engineers about *code quality*, and in the second series, we asked a different set of nine engineers about *product quality*. (Notice that we asked only engineers for their takes on these topics; we did not specifically ask product managers, executives, or other roles.)

Before the code quality interviews, we did an extensive literature review to understand how the research treats code quality. We looked for closely related terms and identified the goals of the research and the approach to understand what underlying theory the researchers had about code quality. For example, in "The Influence of Organizational Structure on Software Quality," Nagappan et al.[4] explore whether metrics about code ownership are predictive of failures in released binaries; this indicates that they are presuming that the defect rate is a component of software quality. Meanwhile, in "Program Complexity Metrics and Programmer Opinions," Katzmarski and Koschke[5] explore whether complexity metrics are correlated with developer perceptions about their ease of modifying the software; this indicates that they are presuming that maintenance is the target. We found that there were seven items that regularly appeared in the research literature related to code quality

- defect rate
- reliability
- maintainability
- testability
- complexity
- comprehensibility (clarity of overall purpose/structure)
- readability (clarity at line/method level).

In our interviews, we first asked the engineers how they would define code quality. We also asked how they would describe the impacts and consequences of code quality on their own productivity, their projects, their dependent projects, and the organization as a whole. Finally, we asked the engineers which of the seven items noted earlier influences their satisfaction with code quality in their projects.

We ran a similar series of interviews about product quality with nine engineers. As in the first set, we asked the engineers to define product quality. We also provided the following list of attributes and asked engineers about their relation with product quality:

- ability to meet user's needs
- performance and reliability
- product complexity
- privacy and security
- innovativeness.

Finally, we asked the engineers the extent to which code quality impacts product quality.

Based on these interviews and reading the prior literature on software quality, we've created a "theory of quality" that posits that there are four types of quality that influence each other. Figure 1 includes a nonexhaustive list of indicators of each type of quality. While there are other major influencing factors as well, and while these types of quality also influence other aspects of the development process, we theorize that they have the relationship shown in Figure 1.

## Process Quality

Our theory is that everything begins with a high-quality development process. Signals of a high-quality process

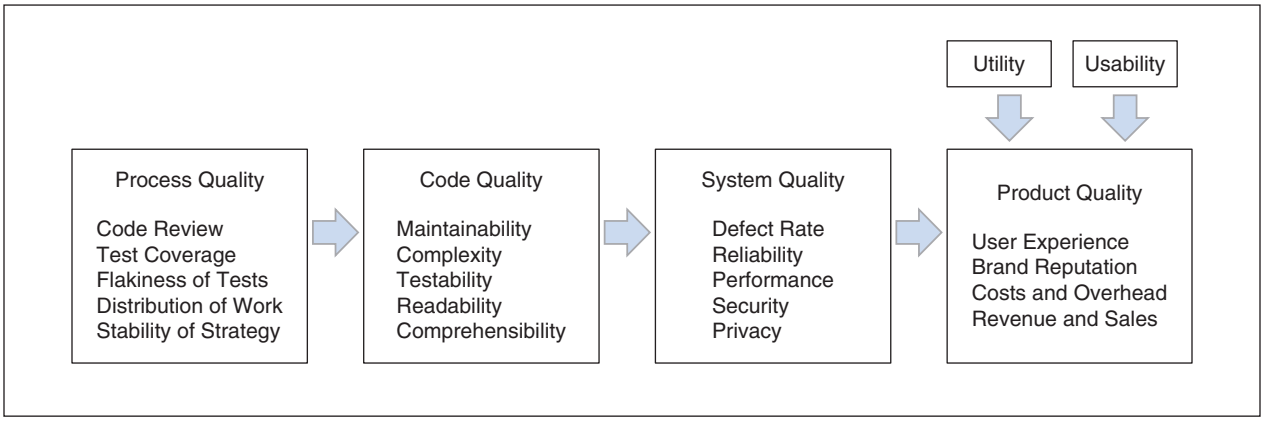



**FIGURE 1.** A theory for how "software quality" is broken into four component types. The arrows represent the direction of influence: process quality is believed to influence code quality.

include things like having comprehensive and deterministic testing, thorough code reviews, organizational consistency, and an effective planning process. There is good evidence that these measures can predict the overall software quality; multiple studies have shown that process-based metrics are *more* predictive of postrelease defects than existing code quality metrics.[4,6,7,8,9] Our theory is that when an organization has higher process quality, it does lead to higher code quality, but perhaps existing "code quality" metrics are not capturing the underlying phenomenon of code quality. Therefore, the research literature is capturing the influence of process quality on system quality instead. 😛

### Code Quality

The entire point of achieving a higher process quality is to have a higher code quality. But what is code quality? All eight of our participants in the code quality interviews defined code quality as primarily relating to maintainability, and they viewed testability, comprehensibility, complexity, and readability as subcategories of maintainability. This is consistent with a recently published study by Börstler et al.[10] about developer perceptions of code quality, where their interviews of 34 developers found nearly identical results.

Overall, our developers described code quality as the ease of working with and understanding the code so that they can easily make changes to it. Developers noted that, with high-quality code files, "you know immediately what it's trying to do" and that "it's neatly organized into files, each with its own piece of logic." When developers see high-quality code, they are seeing code that has a clear purpose; the code provides a single and coherent mental model for the developer. This clarity is what makes it easy to comprehend and easy to modify later.

The impact of code quality on reliability and fewer defects was more tenuous; only half of our participants stated that these are connected. The other half noted that other factors could impact reliability and defect rates ("Code can still break even if there were no bugs due to external factors") or that reliability may not even fully depend on code quality at all ("I've seen lots of reliable code that's poor quality.") They indicated that while these are related, they are not the same concept as "code quality."

Developers noted that the impact of code quality was twofold, though; while it improves the quality of the system by reducing defects and increasing reliability, high code quality also leads to higher velocity for themselves. Maintainability in particular is important because, as one developer put it, "Code is written once and read many many times—everyone within the system has to understand and be able to make changes to it." We've seen this connection before though; in prior research, we found that developers' perceptions of code quality were early indicators of their later perceptions of developer velocity.[11] This is interesting because it highlights that our three components of productivity (speed, ease, and quality) are not always in strict tradeoffs with each other; in some cases, they can also amplify each other.

### System Quality

System quality is where we shift from "quality as the developers see it" to "quality as the business sees it." Most developers hear "software quality" and think about their code and process quality, but when you talk with executives and product managers, they're more interested in product quality. (This insight comes from casual discussions with executives and product managers, not the interview studies.) These two viewpoints come together at system quality, and indeed, we've seen that most discussions between these groups at Google result in a focused discussion about system quality. However, these two viewpoints can cause a disconnect; it can result in an engineering executive asking for higher product quality (because they want to increase customer satisfaction) and then being surprised when the software developers respond by improving their code's modularity. While we hypothesize that these are connected through system quality, the connection isn't obvious to both parties; each is only thinking about their half of the work.

A high-quality system has high reliability, high performance, and low defect rates. Having high code quality is a necessary requirement for having high system quality, but factors such as security and privacy can really be measured only at the system level, and they also play a part in overall system quality. Similarly, high system quality is a necessary, but not sufficient, requirement for having high product quality; there are other factors that come into play at the product quality level as well.

In our experience, one of the largest difficulties with measuring system quality is the sparsity of data. Outages are (and should be!) a very rare event. This means that if a team had only two small outages in a year, and then they had no outages in the next year, we can't really tell for certain whether system quality improved. It might have, or it might be that we were measuring statistical noise, and they just got lucky. Similarly, security

threats and privacy incidents are very impactful but also very rare. In all these metrics, we're aiming for a metric value of "zero," but it's hard to tell whether we've actually improved on a single project.

Process and code quality metrics enable tracking of the indicators that determine system quality. Whether they are validated metrics derived from logs or are based on self-report data from engineer surveys, such measures allow engineers to communicate areas in need of attention as well as the impact of code health investments. Without these intermediary metrics, stakeholders may think that a year with no outages is evidence that the system is high quality because they lack visibility into how developers are experiencing the code base and the ways in which it may be slowing them down. If stakeholders interpret a low defect rate as a guarantee of high-quality code, they may encourage a more exclusive focus on launching features to improve product quality without allocating sufficient resources to improve a potentially high-risk system.

In a year with no outages or incidents, two realities could be at play. In the worst-case scenario, developers may have been effective at inefficiently working around various weaknesses to prevent bugs in a poorly operating system. In the best-case scenario, developers were working in a system with a low risk of defects and were free to focus on enhancements and iterating on new features. Without code quality metrics, leadership can't be sure, and they won't know how to direct engineering efforts to ensure developer velocity and product quality over the long term.

### Product Quality

Product quality is primarily experienced by the customers, but we did also ask developers about how they thought of product quality. In these interviews, developers identified three key factors of product quality: utility, usability, and reliability. Interestingly, engineers identified "innovativeness" as a distinct concept that is not part of product quality. One engineer explained it like this: "I think quality is how well does it do what it says it's doing, and innovativeness is like whatever it says it's doing—Is that interesting or not? Is that complicated or not? Is that… exciting or not?"

Engineers noted that they primarily have influence over reliability but that they worked with product managers and with user experience designers and researchers to contribute to the utility and usability of their product. Engineers made the connection back to code quality in two ways. They again noted that low code quality can slow the engineering velocity and consequently delay product improvements or even render them infeasible. The participants also made the connection to lower code quality increasing the risk of defects (and thus system quality), which would influence product quality.

## Connecting the Types of Quality

These four types of quality are not wholly independent of each other. We do theorize that there's a connection between these—that process quality affects code quality, which affects system quality, which affects product quality. Research has already shown that some process quality metrics can be used to predict defect rates (system quality). The connections are tenuous, though, and other research has shown that the predictive power is not consistent across projects[6,12,13,15,15] and not reliable over time.[6,14,16] For example, Nagappan et al.[12] attempted to predict postrelease defects using code quality metrics, but each project had a different set of metrics that predicted defects. Most concerning was the work by Ekanayake et al., which found that, even for a single project, the predictive value of metrics decreased significantly over time. Our field needs more research to understand *why* we are seeing such results when our intuition would say that the metrics of quality should be the same across products and should be stable across time for a given product.

We hypothesize that part of the problem here is that the existing code quality metrics are not actually measuring the underlying concepts of code quality, as engineers see them, which is possibly what is leading to those metrics not being predictive of system quality. Plenty of prior research, for example, has found that cyclomatic complexity is effectively the same as measuring lines of code,[17] to the point that controlling for this effect through some means is now standard practice in the research community. This makes cyclomatic complexity a poor proxy of code quality, which is likely why so much research has found that it doesn't predict defect rates or represent developers' views of complexity.[10] That's only a single example, but given the depth of this space, there are many further opportunities for improved metrics across all four types of quality.

## What Does an Engineering Lead Do With This?

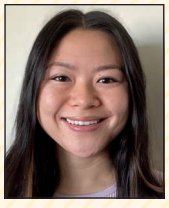
Our theory provides a more nuanced view of quality, which we hope leads to better outcomes when in discussions about how to improve software quality by ensuring that all participants are referring to the same thing. If the team is trying to improve product quality, this might indeed require improving process quality and

## ABOUT THE AUTHORS

**COLLIN GREEN** is the user experience research lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact him at https://research.google/people/107023/ or colling@google.com.

**MAGGIE HODGES** is a user experience researcher on the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact her at https://research.google/people/108095/ or hodgesm@google.com.

**CIERA JASPAN** is the software engineering lead for the Engineering Productivity Research team at Google, Mountain View, CA 94043 USA. Contact her at https://research.google/people/CieraJaspan/ or ciera@google.com.



**JESSICA LIN** is a user experience researcher on the Engineering Productivity Research team at Google, Mountain View, CA 94943 USA. Contact her at jsclin@google.com.


code quality, but everyone needs to be aware that product quality is the end goal, and the connection between the changes being made and product quality needs to be clear. While increasing test coverage might help product quality a little bit, the connection is farther away. It might be better to focus on (and measure the impact of) changes to system quality. Meanwhile, if the team is concerned with code quality, there is a different set of metrics to consider, and focusing on improved process quality might be in order. The actions taken to improve software quality—and the metrics to measure it—depend on which type of quality we want to improve.

In this article, we explored the components that make up software quality, but that's only one aspect of the entire story around productivity. In future articles, we'll also look at how we've tried to reason about speed and ease as well as the connection between these three components of productivity. The most important part when measuring productivity, though, is recognizing that there are many aspects at play here, and measuring only one of them will result in inadvertent consequences. Even within a single component, such as quality, we quickly find that there are many forms of quality, and it's important to recognize which one we are trying to improve so that we can use the best possible measures of impact.

## References


1. N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The SPACE of developer productivity: There's more to it than you think," *Queue*, vol. 19, no. 1, Jan./Feb. 2021, Art. no. 10, doi: 10.1145/3454122.3454124.
2. C. Sadowski and T. Zimmerman, *Rethinking Productivity in Software Engineering*. Berkeley, CA, USA: Apress, 2019. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4842-4221-6
3. "Rethinking productivity in software engineering (Dagstuhl Seminar 17102)," Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2017. [Online]. Available: https://www.dagstuhl.de/en/seminars/seminar-calendar/seminar-details/17102
4. N. Nagappan et al., "The influence of organizational structure on software quality," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 521–530, doi: 10.1145/1368088.1368160.
5. B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions," in *Proc. 20th IEEE Int. Conf. Program Comprehension (ICPC)*, Jun. 11–13, 2012, pp. 17–26, doi: 10.1109/ICPC.2012.6240486.
6. F. Rahman and P. T. Devanbu, "How, and why, process metrics are better," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, 2013, pp. 432–441, doi: 10.1109/ICSE.2013.6606589.
7. R. Moser et al., "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 181–190, doi: 10.1145/1368088.1368114.

8. M. D'Ambros et al., "On the relationship between change coupling and software defects," in *Proc. 16th Working Conf. Reverse Eng.*, 2009, pp. 135–144, doi: 10.1109/WCRE.2009.19.
9. M. Pinzger et al., "Can developer-module networks predict failures?" in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (SIGSOFT/FSE)*, 2008, pp. 2–12, doi: 10.1145/1453101.1453105.
10. J. Börstler et al., "Developers talking about code quality," *Empirical Softw. Eng.*, vol. 28, no. 6, 2023, Art. no. 128, doi: 10.1007/s10664-023-10381-0.
11. L. Cheng et al., "What improves developer productivity at Google? Code quality," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1302–1313, doi: 10.1145/3540250.3558940.
12. N. Nagappan et al., "Mining metrics to predict component failures," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, 2006, pp. 452–461, doi: 10.1145/1134285.1134349.
13. A. B. Binkley and S. R. Schach, "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures," in *Proc. 20th Int. Conf. Softw. Eng.*, 1998, pp. 452–455, doi: 10.1109/ICSE.1998.671604.
14. S. Kim et al., "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng. (ISEC)*, 2007, pp. 489–498, doi: 10.1109/ICSE.2007.66.
15. A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88, doi: 10.1109/ICSE.2009.5070510.
16. J. Ekanayake et al., "Tracking concept drift of software projects using defect prediction quality," in *Proc. 6th IEEE Int. Working Conf. Mining Softw. Repositories*, 2009, pp. 51–60, doi: 10.1109/MSR.2009.5069480.
17. K. E. Emam et al., "The confounding effect of class size on the validity of object-oriented metrics," *IEEE Trans. Softw. Eng.*, vol. 27, no. 7, pp. 630–650, Jul. 2001, doi: 10.1109/32.935855.




*Digital Object Identifier 10.1109/MS.2023.3334596*