O'REILLY®

Software Engineering at Google

Lessons Learned from Programming Over Time

Curated by Titus Winters, Tom Manshreck & Hyrum Wright

Software Engineering at Google

Lessons Learned from Programming Over Time

Titus Winters, Tom Manshreck, and Hyrum Wright

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Software Engineering at Google
by Titus Winters, Tom Manshreck, and Hyrum Wright

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Ryan Shaw
Development Editors: Alicia Young
Production Editor: Christopher Faucher
Copyeditor: Octal Publishing, LLC
Proofreader: Holly Bauer Forsyth

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2020: First Edition

Revision History for the First Edition
2020-02-28: First Release
2020-09-04: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492082798* for release details.

978-1-492-08279-8

[LSI]

Table of Contents

Part II. Culture

Part III. Processes

Preface

This book is titled *Software Engineering at Google*. What precisely do we mean by software engineering? What distinguishes "software engineering" from "programming" or "computer science"? And why would Google have a unique perspective to add to the corpus of previous software engineering literature written over the past 50 years?

The terms "programming" and "software engineering" have been used interchangeably for quite some time in our industry, although each term has a different emphasis and different implications. University students tend to study computer science and get jobs writing code as "programmers."

"Software engineering," however, sounds more serious, as if it implies the application of some theoretical knowledge to build something real and precise. Mechanical engineers, civil engineers, aeronautical engineers, and those in other engineering disciplines all practice engineering. They all work in the real world and use the application of their theoretical knowledge to create something real. Software engineers also create "something real," though it is less tangible than the things other engineers create.

Unlike those more established engineering professions, current software engineering theory or practice is not nearly as rigorous. Aeronautical engineers must follow rigid guidelines and practices, because errors in their calculations can cause real damage; programming, on the whole, has traditionally not followed such rigorous practices. But, as software becomes more integrated into our lives, we must adopt and rely on more rigorous engineering methods. We hope this book helps others see a path toward more reliable software practices.

Programming Over Time

We propose that "software engineering" encompasses not just the act of writing code, but all of the tools and processes an organization uses to build and maintain that code over time. What practices can a software organization introduce that will best keep its

code valuable over the long term? How can engineers make a codebase more sustainable and the software engineering discipline itself more rigorous? We don't have fundamental answers to these questions, but we hope that Google's collective experience over the past two decades illuminates possible paths toward finding those answers.

One key insight we share in this book is that software engineering can be thought of as "programming integrated over time." What practices can we introduce to our code to make it *sustainable*—able to react to necessary change—over its life cycle, from conception to introduction to maintenance to deprecation?

The book emphasizes three fundamental principles that we feel software organizations should keep in mind when designing, architecting, and writing their code:

Time and Change
: How code will need to adapt over the length of its life

Scale and Growth
: How an organization will need to adapt as it evolves

Trade-offs and Costs
: How an organization makes decisions, based on the lessons of Time and Change and Scale and Growth

Throughout the chapters, we have tried to tie back to these themes and point out ways in which such principles affect engineering practices and allow them to be sustainable. (See Chapter 1 for a full discussion.)

Google's Perspective

Google has a unique perspective on the growth and evolution of a sustainable software ecosystem, stemming from our scale and longevity. We hope that the lessons we have learned will be useful as your organization evolves and embraces more sustainable practices.

We've divided the topics in this book into three main aspects of Google's software engineering landscape:

- Culture
- Processes
- Tools

Google's culture is unique, but the lessons we have learned in developing our engineering culture are widely applicable. Our chapters on Culture (Part II) emphasize the collective nature of a software development enterprise, that the development of software is a team effort, and that proper cultural principles are essential for an organization to grow and remain healthy.

The techniques outlined in our Processes chapters (Part III) are familiar to most software engineers, but Google's large size and long-lived codebase provides a more complete stress test for developing best practices. Within those chapters, we have tried to emphasize what we have found to work over time and at scale as well as identify areas where we don't yet have satisfying answers.

Finally, our Tools chapters (Part IV) illustrate how we leverage our investments in tooling infrastructure to provide benefits to our codebase as it both grows and ages. In some cases, these tools are specific to Google, though we point out open source or third-party alternatives where applicable. We expect that these basic insights apply to most engineering organizations.

The culture, processes, and tools outlined in this book describe the lessons that a typical software engineer hopefully learns on the job. Google certainly doesn't have a monopoly on good advice, and our experiences presented here are not intended to dictate what your organization should do. This book is our perspective, but we hope you will find it useful, either by adopting these lessons directly or by using them as a starting point when considering your own practices, specialized for your own problem domain.

Neither is this book intended to be a sermon. Google itself still imperfectly applies many of the concepts within these pages. The lessons that we have learned, we learned through our failures: we still make mistakes, implement imperfect solutions, and need to iterate toward improvement. Yet the sheer size of Google's engineering organization ensures that there is a diversity of solutions for every problem. We hope that this book contains the best of that group.

What This Book Isn't

This book is not meant to cover software design, a discipline that requires its own book (and for which much content already exists). Although there is some code in this book for illustrative purposes, the principles are language neutral, and there is little actual "programming" advice within these chapters. As a result, this text doesn't cover many important issues in software development: project management, API design, security hardening, internationalization, user interface frameworks, or other language-specific concerns. Their omission in this book does not imply their lack of importance. Instead, we choose not to cover them here knowing that we could not provide the treatment they deserve. We have tried to make the discussions in this book more about engineering and less about programming.

Parting Remarks

This text has been a labor of love on behalf of all who have contributed, and we hope that you receive it as it is given: as a window into how a large software engineering organization builds its products. We also hope that it is one of many voices that helps move our industry to adopt more forward-thinking and sustainable practices. Most important, we further hope that you enjoy reading it and can adopt some of its lessons to your own concerns.

— Tom Manshreck

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/software-engineering-at-google*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and more information about our books and courses, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

A book like this would not be possible without the work of countless others. All of the knowledge within this book has come to all of us through the experience of so many others at Google throughout our careers. We are the messengers; others came before us, at Google and elsewhere, and taught us what we now present to you. We cannot list all of you here, but we do wish to acknowledge you.

We'd also like to thank Melody Meckfessel for supporting this project in its infancy as well as Daniel Jasper and Danny Berlin for supporting it through its completion.

This book would not have been possible without the massive collaborative effort of our curators, authors, and editors. Although the authors and editors are specifically acknowledged in each chapter or callout, we'd like to take time to recognize those who contributed to each chapter by providing thoughtful input, discussion, and review.

- **What Is Software Engineering?:** Sanjay Ghemawat, Andrew Hyatt
- **Working Well on Teams:** Sibley Bacon, Joshua Morton
- **Knowledge Sharing:** Dimitri Glazkov, Kyle Lemons, John Reese, David Symonds, Andrew Trenk, James Tucker, David Kohlbrenner, Rodrigo Damazio Bovendorp
- **Engineering for Equity:** Kamau Bobb, Bruce Lee
- **How to Lead a Team:** Jon Wiley, Laurent Le Brun
- **Leading at Scale:** Bryan O'Sullivan, Bharat Mediratta, Daniel Jasper, Shaindel Schwartz
- **Measuring Engineering Productivity:** Andrea Knight, Collin Green, Caitlin Sadowski, Max-Kanat Alexander, Yilei Yang
- **Style Guides and Rules:** Max Kanat-Alexander, Titus Winters, Matt Austern, James Dennett
- **Code Review:** Max Kanat-Alexander, Brian Ledger, Mark Barolak
- **Documentation:** Jonas Wagner, Smit Hinsu, Geoffrey Romer
- **Testing Overview:** Erik Kuefler, Andrew Trenk, Dillon Bly, Joseph Graves, Neal Norwitz, Jay Corbett, Mark Striebeck, Brad Green, Miško Hevery, Antoine Picard, Sarah Storck
- **Unit Testing:** Andrew Trenk, Adam Bender, Dillon Bly, Joseph Graves, Titus Winters, Hyrum Wright, Augie Fackler
- **Testing Doubles:** Joseph Graves, Gennadiy Civil

- **Larger Testing:** Adam Bender, Andrew Trenk, Erik Kuefler, Matthew Beaumont-Gay
- **Deprecation:** Greg Miller, Andy Shulman
- **Version Control and Branch Management:** Rachel Potvin, Victoria Clarke
- **Code Search:** Jenny Wang
- **Build Systems and Build Philosophy:** Hyrum Wright, Titus Winters, Adam Bender, Jeff Cox, Jacques Pienaar
- **Critique: Google's Code Review Tool:** Mikołaj Dądela, Hermann Loose, Eva May, Alice Kober-Sotzek, Edwin Kempin, Patrick Hiesel, Ole Rehmsen, Jan Macek
- **Static Analysis:** Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, Edward Aftandilian, Collin Winter, Eric Haugh
- **Dependency Management:** Russ Cox, Nicholas Dunn
- **Large-Scale Changes:** Matthew Fowles Kulukundis, Adam Zarek
- **Continuous Integration:** Jeff Listfield, John Penix, Kaushik Sridharan, Sanjeev Dhanda
- **Continuous Delivery:** Dave Owens, Sheri Shipe, Bobbi Jones, Matt Duftler, Brian Szuter
- **Compute Services:** Tim Hockin, Collin Winter, Jarek Kuśmierek

Additionally, we'd like to thank Betsy Beyer for sharing her insight and experience in having published the original *Site Reliability Engineering* book, which made our experience much smoother. Christopher Guzikowski and Alicia Young at O'Reilly did an awesome job launching and guiding this project to publication.

The curators would also like to personally thank the following people:

Tom Manshreck: To my mom and dad for making me believe in myself—and working with me at the kitchen table to do my homework.

Titus Winters: To Dad, for my path. To Mom, for my voice. To Victoria, for my heart. To Raf, for having my back. Also, to Mr. Snyder, Ranwa, Z, Mike, Zach, Tom (and all the Paynes), mec, Toby, cgd, and Melody for lessons, mentorship, and trust.

Hyrum Wright: To Mom and Dad for their encouragement. To Bryan and the denizens of Bakerland, for my first foray into software. To Dewayne, for continuing that journey. To Hannah, Jonathan, Charlotte, Spencer, and Ben for their love and interest. To Heather for being there through it all.

PART I
Thesis

CHAPTER 1
What Is Software Engineering?

Written by Titus Winters
Edited by Tom Manshreck

> *Nothing is built on stone; all is built on sand, but we must build as if the sand were stone.*
> —Jorge Luis Borges

We see three critical differences between programming and software engineering: time, scale, and the trade-offs at play. On a software engineering project, engineers need to be more concerned with the passage of time and the eventual need for change. In a software engineering organization, we need to be more concerned about scale and efficiency, both for the software we produce as well as for the organization that is producing it. Finally, as software engineers, we are asked to make more complex decisions with higher-stakes outcomes, often based on imprecise estimates of time and growth.

Within Google, we sometimes say, "Software engineering is programming integrated over time." Programming is certainly a significant part of software engineering: after all, programming is how you generate new software in the first place. If you accept this distinction, it also becomes clear that we might need to delineate between programming tasks (development) and software engineering tasks (development, modification, maintenance). The addition of time adds an important new dimension to programming. Cubes aren't squares, distance isn't velocity. Software engineering isn't programming.

One way to see the impact of time on a program is to think about the question, "What is the expected life span[1] of your code?" Reasonable answers to this question

1 We don't mean "execution lifetime," we mean "maintenance lifetime"—how long will the code continue to be built, executed, and maintained? How long will this software provide value?

vary by roughly a factor of 100,000. It is just as reasonable to think of code that needs to last for a few minutes as it is to imagine code that will live for decades. Generally, code on the short end of that spectrum is unaffected by time. It is unlikely that you need to adapt to a new version of your underlying libraries, operating system (OS), hardware, or language version for a program whose utility spans only an hour. These short-lived systems are effectively "just" a programming problem, in the same way that a cube compressed far enough in one dimension is a square. As we expand that time to allow for longer life spans, change becomes more important. Over a span of a decade or more, most program dependencies, whether implicit or explicit, will likely change. This recognition is at the root of our distinction between software engineering and programming.

This distinction is at the core of what we call *sustainability* for software. Your project is *sustainable* if, for the expected life span of your software, you are capable of reacting to whatever valuable change comes along, for either technical or business reasons. Importantly, we are looking only for capability—you might choose not to perform a given upgrade, either for lack of value or other priorities.[2] When you are fundamentally incapable of reacting to a change in underlying technology or product direction, you're placing a high-risk bet on the hope that such a change never becomes critical. For short-term projects, that might be a safe bet. Over multiple decades, it probably isn't.[3]

Another way to look at software engineering is to consider scale. How many people are involved? What part do they play in the development and maintenance over time? A programming task is often an act of individual creation, but a software engineering task is a team effort. An early attempt to define software engineering produced a good definition for this viewpoint: "The multiperson development of multiversion programs."[4] This suggests the difference between software engineering and programming is one of both time and people. Team collaboration presents new problems, but also provides more potential to produce valuable systems than any single programmer could.

Team organization, project composition, and the policies and practices of a software project all dominate this aspect of software engineering complexity. These problems are inherent to scale: as the organization grows and its projects expand, does it become more efficient at producing software? Does our development workflow

2 This is perhaps a reasonable hand-wavy definition of technical debt: things that "should" be done, but aren't yet—the delta between our code and what we wish it was.

3 Also consider the issue of whether we know ahead of time that a project is going to be long lived.

4 There is some question as to the original attribution of this quote; consensus seems to be that it was originally phrased by Brian Randell or Margaret Hamilton, but it might have been wholly made up by Dave Parnas. The common citation for it is "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee," Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO.

become more efficient as we grow, or do our version control policies and testing strategies cost us proportionally more? Scale issues around communication and human scaling have been discussed since the early days of software engineering, going all the way back to the *Mythical Man Month*.[5] Such scale issues are often matters of policy and are fundamental to the question of software sustainability: how much will it cost to do the things that we need to do repeatedly?

We can also say that software engineering is different from programming in terms of the complexity of decisions that need to be made and their stakes. In software engineering, we are regularly forced to evaluate the trade-offs between several paths forward, sometimes with high stakes and often with imperfect value metrics. The job of a software engineer, or a software engineering leader, is to aim for sustainability and management of the scaling costs for the organization, the product, and the development workflow. With those inputs in mind, evaluate your trade-offs and make rational decisions. We might sometimes defer maintenance changes, or even embrace policies that don't scale well, with the knowledge that we'll need to revisit those decisions. Those choices should be explicit and clear about the deferred costs.

Rarely is there a one-size-fits-all solution in software engineering, and the same applies to this book. Given a factor of 100,000 for reasonable answers on "How long will this software live," a range of perhaps a factor of 10,000 for "How many engineers are in your organization," and who-knows-how-much for "How many compute resources are available for your project," Google's experience will probably not match yours. In this book, we aim to present what we've found that works for us in the construction and maintenance of software that we expect to last for decades, with tens of thousands of engineers, and world-spanning compute resources. Most of the practices that we find are necessary at that scale will also work well for smaller endeavors: consider this a report on one engineering ecosystem that we think could be good as you scale up. In a few places, super-large scale comes with its own costs, and we'd be happier to not be paying extra overhead. We call those out as a warning. Hopefully if your organization grows large enough to be worried about those costs, you can find a better answer.

Before we get to specifics about teamwork, culture, policies, and tools, let's first elaborate on these primary themes of time, scale, and trade-offs.

5 Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering* (Boston: Addison-Wesley, 1995).

Time and Change

When a novice is learning to program, the life span of the resulting code is usually measured in hours or days. Programming assignments and exercises tend to be write-once, with little to no refactoring and certainly no long-term maintenance. These programs are often not rebuilt or executed ever again after their initial production. This isn't surprising in a pedagogical setting. Perhaps in secondary or post-secondary education, we may find a team project course or hands-on thesis. If so, such projects are likely the only time student code will live longer than a month or so. Those developers might need to refactor some code, perhaps as a response to changing requirements, but it is unlikely they are being asked to deal with broader changes to their environment.

We also find developers of short-lived code in common industry settings. Mobile apps often have a fairly short life span,[6] and for better or worse, full rewrites are relatively common. Engineers at an early-stage startup might rightly choose to focus on immediate goals over long-term investments: the company might not live long enough to reap the benefits of an infrastructure investment that pays off slowly. A serial startup developer could very reasonably have 10 years of development experience and little or no experience maintaining any piece of software expected to exist for longer than a year or two.

On the other end of the spectrum, some successful projects have an effectively unbounded life span: we can't reasonably predict an endpoint for Google Search, the Linux kernel, or the Apache HTTP Server project. For most Google projects, we must assume that they will live indefinitely—we cannot predict when we won't need to upgrade our dependencies, language versions, and so on. As their lifetimes grow, these long-lived projects *eventually* have a different feel to them than programming assignments or startup development.

Consider Figure 1-1, which demonstrates two software projects on opposite ends of this "expected life span" spectrum. For a programmer working on a task with an expected life span of hours, what types of maintenance are reasonable to expect? That is, if a new version of your OS comes out while you're working on a Python script that will be executed one time, should you drop what you're doing and upgrade? Of course not: the upgrade is not critical. But on the opposite end of the spectrum, Google Search being stuck on a version of our OS from the 1990s would be a clear problem.

6 Appcelerator, "Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan," Axway Developer blog, December 6, 2012.

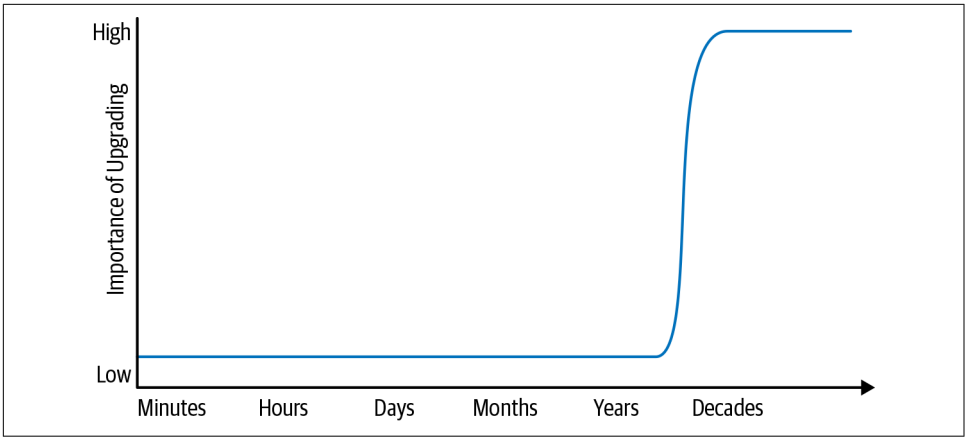

Figure 1-1. Life span and the importance of upgrades

The low and high points on the expected life span spectrum suggest that there's a transition somewhere. Somewhere along the line between a one-off program and a project that lasts for decades, a transition happens: a project must begin to react to changing externalities.[7] For any project that didn't plan for upgrades from the start, that transition is likely very painful for three reasons, each of which compounds the others:

- You're performing a task that hasn't yet been done for this project; more hidden assumptions have been baked-in.
- The engineers trying to do the upgrade are less likely to have experience in this sort of task.
- The size of the upgrade is often larger than usual, doing several years' worth of upgrades at once instead of a more incremental upgrade.

And thus, after actually going through such an upgrade once (or giving up part way through), it's pretty reasonable to overestimate the cost of doing a subsequent upgrade and decide "Never again." Companies that come to this conclusion end up committing to just throwing things out and rewriting their code, or deciding to never upgrade again. Rather than take the natural approach by avoiding a painful task, sometimes the more responsible answer is to invest in making it less painful. It all depends on the cost of your upgrade, the value it provides, and the expected life span of the project in question.

7 Your own priorities and tastes will inform where exactly that transition happens. We've found that most projects seem to be willing to upgrade within five years. Somewhere between 5 and 10 years seems like a conservative estimate for this transition in general.

Getting through not only that first big upgrade, but getting to the point at which you can reliably stay current going forward, is the essence of long-term sustainability for your project. Sustainability requires planning and managing the impact of required change. For many projects at Google, we believe we have achieved this sort of sustainability, largely through trial and error.

So, concretely, how does short-term programming differ from producing code with a much longer expected life span? Over time, we need to be much more aware of the difference between "happens to work" and "is maintainable." There is no perfect solution for identifying these issues. That is unfortunate, because keeping software maintainable for the long-term is a constant battle.

Hyrum's Law

If you are maintaining a project that is used by other engineers, the most important lesson about "it works" versus "it is maintainable" is what we've come to call *Hyrum's Law*:

> With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.

In our experience, this axiom is a dominant factor in any discussion of changing software over time. It is conceptually akin to entropy: discussions of change and maintenance over time must be aware of Hyrum's Law[8] just as discussions of efficiency or thermodynamics must be mindful of entropy. Just because entropy never decreases doesn't mean we shouldn't try to be efficient. Just because Hyrum's Law will apply when maintaining software doesn't mean we can't plan for it or try to better understand it. We can mitigate it, but we know that it can never be eradicated.

Hyrum's Law represents the practical knowledge that—even with the best of intentions, the best engineers, and solid practices for code review—we cannot assume perfect adherence to published contracts or best practices. As an API owner, you will gain *some* flexibility and freedom by being clear about interface promises, but in practice, the complexity and difficulty of a given change also depends on how useful a user finds some observable behavior of your API. If users cannot depend on such things, your API will be easy to change. Given enough time and enough users, even the most innocuous change *will* break something;[9] your analysis of the value of that change must incorporate the difficulty in investigating, identifying, and resolving those breakages.

8 To his credit, Hyrum tried really hard to humbly call this "The Law of Implicit Dependencies," but "Hyrum's Law" is the shorthand that most people at Google have settled on.

9 See "Workflow," an *xkcd* comic.

Example: Hash Ordering

Consider the example of hash iteration ordering. If we insert five elements into a hash-based set, in what order do we get them out?

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

Most programmers know that hash tables are non-obviously ordered. Few know the specifics of whether the particular hash table they are using is *intending* to provide that particular ordering forever. This might seem unremarkable, but over the past decade or two, the computing industry's experience using such types has evolved:

- *Hash flooding*[10] attacks provide an increased incentive for nondeterministic hash iteration.
- Potential efficiency gains from research into improved hash algorithms or hash containers require changes to hash iteration order.
- Per Hyrum's Law, programmers will write programs that depend on the order in which a hash table is traversed, if they have the ability to do so.

As a result, if you ask any expert "Can I assume a particular output sequence for my hash container?" that expert will presumably say "No." By and large that is correct, but perhaps simplistic. A more nuanced answer is, "If your code is short-lived, with no changes to your hardware, language runtime, or choice of data structure, such an assumption is fine. If you don't know how long your code will live, or you cannot promise that nothing you depend upon will ever change, such an assumption is incorrect." Moreover, even if your own implementation does not depend on hash container order, it might be used by other code that implicitly creates such a dependency. For example, if your library serializes values into a Remote Procedure Call (RPC) response, the RPC caller might wind up depending on the order of those values.

This is a very basic example of the difference between "it works" and "it is correct." For a short-lived program, depending on the iteration order of your containers will not cause any technical problems. For a software engineering project, on the other hand, such reliance on a defined order is a risk—given enough time, something will

10 A type of Denial-of-Service (DoS) attack in which an untrusted user knows the structure of a hash table and the hash function and provides data in such a way as to degrade the algorithmic performance of operations on the table.

make it valuable to change that iteration order. That value can manifest in a number of ways, be it efficiency, security, or merely future-proofing the data structure to allow for future changes. When that value becomes clear, you will need to weigh the trade-offs between that value and the pain of breaking your developers or customers.

Some languages specifically randomize hash ordering between library versions or even between execution of the same program in an attempt to prevent dependencies. But even this still allows for some Hyrum's Law surprises: there is code that uses hash iteration ordering as an inefficient random-number generator. Removing such randomness now would break those users. Just as entropy increases in every thermodynamic system, Hyrum's Law applies to every observable behavior.

Thinking over the differences between code written with a "works now" and a "works indefinitely" mentality, we can extract some clear relationships. Looking at code as an artifact with a (highly) variable lifetime requirement, we can begin to categorize programming styles: code that depends on brittle and unpublished features of its dependencies is likely to be described as "hacky" or "clever," whereas code that follows best practices and has planned for the future is more likely to be described as "clean" and "maintainable." Both have their purposes, but which one you select depends crucially on the expected life span of the code in question. We've taken to saying, "It's *programming* if 'clever' is a compliment, but it's *software engineering* if 'clever' is an accusation."

Why Not Just Aim for "Nothing Changes"?

Implicit in all of this discussion of time and the need to react to change is the assumption that change might be necessary. Is it?

As with effectively everything else in this book, it depends. We'll readily commit to "For most projects, over a long enough time period, everything underneath them might need to be changed." If you have a project written in pure C with no external dependencies (or only external dependencies that promise great long-term stability, like POSIX), you might well be able to avoid any form of refactoring or difficult upgrade. C does a great job of providing stability—in many respects, that is its primary purpose.

Most projects have far more exposure to shifting underlying technology. Most programming languages and runtimes change much more than C does. Even libraries implemented in pure C might change to support new features, which can affect downstream users. Security problems are disclosed in all manner of technology, from processors to networking libraries to application code. *Every* piece of technology upon which your project depends has some (hopefully small) risk of containing critical bugs and security vulnerabilities that might come to light only after you've started relying on it. If you are incapable of deploying a patch for Heartbleed or mitigating

speculative execution problems like Meltdown and Spectre because you've assumed (or promised) that nothing will ever change, that is a significant gamble.

Efficiency improvements further complicate the picture. We want to outfit our datacenters with cost-effective computing equipment, especially enhancing CPU efficiency. However, algorithms and data structures from early-day Google are simply less efficient on modern equipment: a linked-list or a binary search tree will still work fine, but the ever-widening gap between CPU cycles versus memory latency impacts what "efficient" code looks like. Over time, the value in upgrading to newer hardware can be diminished without accompanying design changes to the software. Backward compatibility ensures that older systems still function, but that is no guarantee that old optimizations are still helpful. Being unwilling or unable to take advantage of such opportunities risks incurring large costs. Efficiency concerns like this are particularly subtle: the original design might have been perfectly logical and following reasonable best practices. It's only after an evolution of backward-compatible changes that a new, more efficient option becomes important. No mistakes were made, but the passage of time still made change valuable.

Concerns like those just mentioned are why there are large risks for long-term projects that haven't invested in sustainability. We must be capable of responding to these sorts of issues and taking advantage of these opportunities, regardless of whether they directly affect us or manifest in only the transitive closure of technology we build upon. Change is not inherently good. We shouldn't change just for the sake of change. But we do need to be capable of change. If we allow for that eventual necessity, we should also consider whether to invest in making that capability cheap. As every system administrator knows, it's one thing to know in theory that you can recover from tape, and another to know in practice exactly how to do it and how much it will cost when it becomes necessary. Practice and expertise are great drivers of efficiency and reliability.

Scale and Efficiency

As noted in the *Site Reliability Engineering* (SRE) book,[11] Google's production system as a whole is among the most complex machines created by humankind. The complexity involved in building such a machine and keeping it running smoothly has required countless hours of thought, discussion, and redesign from experts across our organization and around the globe. So, we have already written a book about the complexity of keeping that machine running at that scale.

11 Beyer, B. et al. *Site Reliability Engineering: How Google Runs Production Systems*. (Boston: O'Reilly Media, 2016).

Much of *this* book focuses on the complexity of scale of the organization that produces such a machine, and the processes that we use to keep that machine running over time. Consider again the concept of codebase sustainability: "Your organization's codebase is *sustainable* when you are *able* to change all of the things that you ought to change, safely, and can do so for the life of your codebase." Hidden in the discussion of capability is also one of costs: if changing something comes at inordinate cost, it will likely be deferred. If costs grow superlinearly over time, the operation clearly is not scalable.[12] Eventually, time will take hold and something unexpected will arise that you absolutely must change. When your project doubles in scope and you need to perform that task again, will it be twice as labor intensive? Will you even have the human resources required to address the issue next time?

Human costs are not the only finite resource that needs to scale. Just as software itself needs to scale well with traditional resources such as compute, memory, storage, and bandwidth, the development of that software also needs to scale, both in terms of human time involvement and the compute resources that power your development workflow. If the compute cost for your test cluster grows superlinearly, consuming more compute resources per person each quarter, you're on an unsustainable path and need to make changes soon.

Finally, the most precious asset of a software organization—the codebase itself—also needs to scale. If your build system or version control system scales superlinearly over time, perhaps as a result of growth and increasing changelog history, a point might come at which you simply cannot proceed. Many questions, such as "How long does it take to do a full build?", "How long does it take to pull a fresh copy of the repository?", or "How much will it cost to upgrade to a new language version?" aren't actively monitored and change at a slow pace. They can easily become like the metaphorical boiled frog; it is far too easy for problems to worsen slowly and never manifest as a singular moment of crisis. Only with an organization-wide awareness and commitment to scaling are you likely to keep on top of these issues.

Everything your organization relies upon to produce and maintain code should be scalable in terms of overall cost and resource consumption. In particular, everything your organization must do repeatedly should be scalable in terms of human effort. Many common policies don't seem to be scalable in this sense.

Policies That Don't Scale

With a little practice, it becomes easier to spot policies with bad scaling properties. Most commonly, these can be identified by considering the work imposed on a single

12 Whenever we use "scalable" in an informal context in this chapter, we mean "sublinear scaling with regard to human interactions."

engineer and imagining the organization scaling up by 10 or 100 times. When we are 10 times larger, will we add 10 times more work with which our sample engineer needs to keep up? Does the amount of work our engineer must perform grow as a function of the size of the organization? Does the work scale up with the size of the codebase? If either of these are true, do we have any mechanisms in place to automate or optimize that work? If not, we have scaling problems.

Consider a traditional approach to deprecation. We discuss deprecation much more in Chapter 15, but the common approach to deprecation serves as a great example of scaling problems. A new Widget has been developed. The decision is made that everyone should use the new one and stop using the old one. To motivate this, project leads say "We'll delete the old Widget on August 15th; make sure you've converted to the new Widget."

This type of approach might work in a small software setting but quickly fails as both the depth and breadth of the dependency graph increases. Teams depend on an ever-increasing number of Widgets, and a single build break can affect a growing percentage of the company. Solving these problems in a scalable way means changing the way we do deprecation: instead of pushing migration work to customers, teams can internalize it themselves, with all the economies of scale that provides.

In 2012, we tried to put a stop to this with rules mitigating churn: infrastructure teams must do the work to move their internal users to new versions themselves or do the update in place, in backward-compatible fashion. This policy, which we've called the "Churn Rule," scales better: dependent projects are no longer spending progressively greater effort just to keep up. We've also learned that having a dedicated group of experts execute the change scales better than asking for more maintenance effort from every user: experts spend some time learning the whole problem in depth and then apply that expertise to every subproblem. Forcing users to respond to churn means that every affected team does a worse job ramping up, solves their immediate problem, and then throws away that now-useless knowledge. Expertise scales better.

The traditional use of development branches is another example of policy that has built-in scaling problems. An organization might identify that merging large features into trunk has destabilized the product and conclude, "We need tighter controls on when things merge. We should merge less frequently." This leads quickly to every team or every feature having separate dev branches. Whenever any branch is decided to be "complete," it is tested and merged into trunk, triggering some potentially expensive work for other engineers still working on their dev branch, in the form of resyncing and testing. Such branch management can be made to work for a small organization juggling 5 to 10 such branches. As the size of an organization (and the number of branches) increases, it quickly becomes apparent that we're paying an ever-increasing amount of overhead to do the same task. We'll need a different approach as we scale up, and we discuss that in Chapter 16.

Policies That Scale Well

What sorts of policies result in better costs as the organization grows? Or, better still, what sorts of policies can we put in place that provide superlinear value as the organization grows?

One of our favorite internal policies is a great enabler of infrastructure teams, protecting their ability to make infrastructure changes safely. "If a product experiences outages or other problems as a result of infrastructure changes, but the issue wasn't surfaced by tests in our Continuous Integration (CI) system, it is not the fault of the infrastructure change." More colloquially, this is phrased as "If you liked it, you should have put a CI test on it," which we call "The Beyoncé Rule."[13] From a scaling perspective, the Beyoncé Rule implies that complicated, one-off bespoke tests that aren't triggered by our common CI system do not count. Without this, an engineer on an infrastructure team could conceivably need to track down every team with any affected code and ask them how to run their tests. We could do that when there were a hundred engineers. We definitely cannot afford to do that anymore.

We've found that expertise and shared communication forums offer great value as an organization scales. As engineers discuss and answer questions in shared forums, knowledge tends to spread. New experts grow. If you have a hundred engineers writing Java, a single friendly and helpful Java expert willing to answer questions will soon produce a hundred engineers writing better Java code. Knowledge is viral, experts are carriers, and there's a lot to be said for the value of clearing away the common stumbling blocks for your engineers. We cover this in greater detail in Chapter 3.

Example: Compiler Upgrade

Consider the daunting task of upgrading your compiler. Theoretically, a compiler upgrade should be cheap given how much effort languages take to be backward compatible, but how cheap of an operation is it in practice? If you've never done such an upgrade before, how would you evaluate whether your codebase is compatible with that change?

13 This is a reference to the popular song "Single Ladies," which includes the refrain "If you liked it then you shoulda put a ring on it."

In our experience, language and compiler upgrades are subtle and difficult tasks even when they are broadly expected to be backward compatible. A compiler upgrade will almost always result in minor changes to behavior: fixing miscompilations, tweaking optimizations, or potentially changing the results of anything that was previously undefined. How would you evaluate the correctness of your entire codebase against all of these potential outcomes?

The most storied compiler upgrade in Google's history took place all the way back in 2006. At that point, we had been operating for a few years and had several thousand engineers on staff. We hadn't updated compilers in about five years. Most of our engineers had no experience with a compiler change. Most of our code had been exposed to only a single compiler version. It was a difficult and painful task for a team of (mostly) volunteers, which eventually became a matter of finding shortcuts and simplifications in order to work around upstream compiler and language changes that we didn't know how to adopt.[14] In the end, the 2006 compiler upgrade was extremely painful. Many Hyrum's Law problems, big and small, had crept into the codebase and served to deepen our dependency on a particular compiler version. Breaking those implicit dependencies was painful. The engineers in question were taking a risk: we didn't have the Beyoncé Rule yet, nor did we have a pervasive CI system, so it was difficult to know the impact of the change ahead of time or be sure they wouldn't be blamed for regressions.

This story isn't at all unusual. Engineers at many companies can tell a similar story about a painful upgrade. What is unusual is that we recognized after the fact that the task had been painful and began focusing on technology and organizational changes to overcome the scaling problems and turn scale to our advantage: automation (so that a single human can do more), consolidation/consistency (so that low-level changes have a limited problem scope), and expertise (so that a few humans can do more).

The more frequently you change your infrastructure, the easier it becomes to do so. We have found that most of the time, when code is updated as part of something like a compiler upgrade, it becomes less brittle and easier to upgrade in the future. In an ecosystem in which most code has gone through several upgrades, it stops depending on the nuances of the underlying implementation; instead, it depends on the actual abstraction guaranteed by the language or OS. Regardless of what exactly you are upgrading, expect the first upgrade for a codebase to be significantly more expensive than later upgrades, even controlling for other factors.

14 Specifically, interfaces from the C++ standard library needed to be referred to in namespace std, and an optimization change for `std::string` turned out to be a significant pessimization for our usage, thus requiring some additional workarounds.

Through this and other experiences, we've discovered many factors that affect the flexibility of a codebase:

Expertise
: We know how to do this; for some languages, we've now done hundreds of compiler upgrades across many platforms.

Stability
: There is less change between releases because we adopt releases more regularly; for some languages, we're now deploying compiler upgrades every week or two.

Conformity
: There is less code that hasn't been through an upgrade already, again because we are upgrading regularly.

Familiarity
: Because we do this regularly enough, we can spot redundancies in the process of performing an upgrade and attempt to automate. This overlaps significantly with SRE views on toil.[15]

Policy
: We have processes and policies like the Beyoncé Rule. The net effect of these processes is that upgrades remain feasible because infrastructure teams do not need to worry about every unknown usage, only the ones that are visible in our CI systems.

The underlying lesson is not about the frequency or difficulty of compiler upgrades, but that as soon as we became aware that compiler upgrade tasks were necessary, we found ways to make sure to perform those tasks with a constant number of engineers, even as the codebase grew.[16] If we had instead decided that the task was too expensive and should be avoided in the future, we might still be using a decade-old compiler version. We would be paying perhaps 25% extra for computational resources as a result of missed optimization opportunities. Our central infrastructure could be vulnerable to significant security risks given that a 2006-era compiler is certainly not helping to mitigate speculative execution vulnerabilities. Stagnation is an option, but often not a wise one.

15 Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*, Chapter 5, "Eliminating Toil."

16 In our experience, an average software engineer (SWE) produces a pretty constant number of lines of code per unit time. For a fixed SWE population, a codebase grows linearly—proportional to the count of SWE-months over time. If your tasks require effort that scales with lines of code, that's concerning.

Shifting Left

One of the broad truths we've seen to be true is the idea that finding problems earlier in the developer workflow usually reduces costs. Consider a timeline of the developer workflow for a feature that progresses from left to right, starting from conception and design, progressing through implementation, review, testing, commit, canary, and eventual production deployment. Shifting problem detection to the "left" earlier on this timeline makes it cheaper to fix than waiting longer, as shown in Figure 1-2.

This term seems to have originated from arguments that security mustn't be deferred until the end of the development process, with requisite calls to "shift left on security." The argument in this case is relatively simple: if a security problem is discovered only after your product has gone to production, you have a very expensive problem. If it is caught before deploying to production, it may still take a lot of work to identify and remedy the problem, but it's cheaper. If you can catch it before the original developer commits the flaw to version control, it's even cheaper: they already have an understanding of the feature; revising according to new security constraints is cheaper than committing and forcing someone else to triage it and fix it.

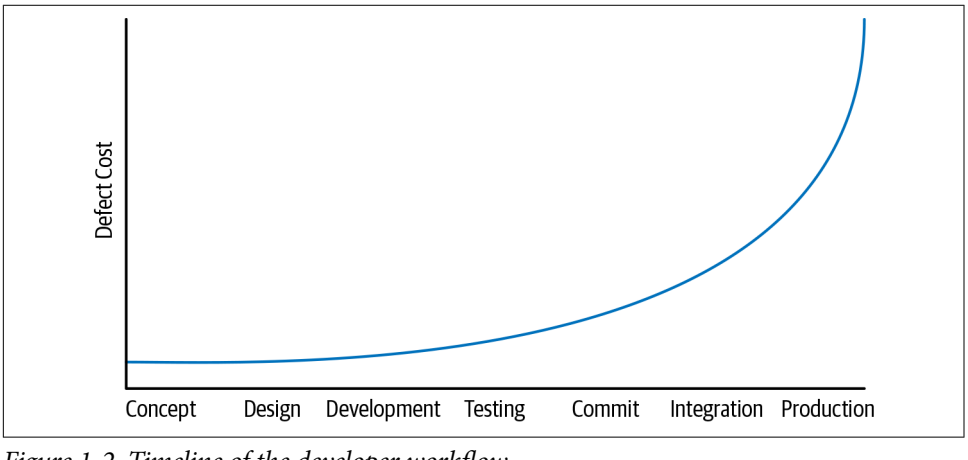

Figure 1-2. Timeline of the developer workflow

The same basic pattern emerges many times in this book. Bugs that are caught by static analysis and code review before they are committed are much cheaper than bugs that make it to production. Providing tools and practices that highlight quality, reliability, and security early in the development process is a common goal for many of our infrastructure teams. No single process or tool needs to be perfect, so we can assume a defense-in-depth approach, hopefully catching as many defects on the left side of the graph as possible.

Trade-offs and Costs

If we understand how to program, understand the lifetime of the software we're maintaining, and understand how to maintain it as we scale up with more engineers producing and maintaining new features, all that is left is to make good decisions. This seems obvious: in software engineering, as in life, good choices lead to good outcomes. However, the ramifications of this observation are easily overlooked. Within Google, there is a strong distaste for "because I said so." It is important for there to be a decider for any topic and clear escalation paths when decisions seem to be wrong, but the goal is consensus, not unanimity. It's fine and expected to see some instances of "I don't agree with your metrics/valuation, but I see how you can come to that conclusion." Inherent in all of this is the idea that there needs to be a reason for everything; "just because," "because I said so," or "because everyone else does it this way" are places where bad decisions lurk. Whenever it is efficient to do so, we should be able to explain our work when deciding between the general costs for two engineering options.

What do we mean by cost? We are not only talking about dollars here. "Cost" roughly translates to effort and can involve any or all of these factors:

- Financial costs (e.g., money)
- Resource costs (e.g., CPU time)
- Personnel costs (e.g., engineering effort)
- Transaction costs (e.g., what does it cost to take action?)
- Opportunity costs (e.g., what does it cost to not take action?)
- Societal costs (e.g., what impact will this choice have on society at large?)

Historically, it's been particularly easy to ignore the question of societal costs. However, Google and other large tech companies can now credibly deploy products with billions of users. In many cases, these products are a clear net benefit, but when we're operating at such a scale, even small discrepancies in usability, accessibility, fairness, or potential for abuse are magnified, often to the detriment of groups that are already marginalized. Software pervades so many aspects of society and culture; therefore, it is wise for us to be aware of both the good and the bad that we enable when making product and technical decisions. We discuss this much more in Chapter 4.

In addition to the aforementioned costs (or our estimate of them), there are biases: status quo bias, loss aversion, and others. When we evaluate cost, we need to keep all of the previously listed costs in mind: the health of an organization isn't just whether there is money in the bank, it's also whether its members are feeling valued and productive. In highly creative and lucrative fields like software engineering, financial cost is usually not the limiting factor—personnel cost usually is. Efficiency gains from

keeping engineers happy, focused, and engaged can easily dominate other factors, simply because focus and productivity are so variable, and a 10-to-20% difference is easy to imagine.

Example: Markers

In many organizations, whiteboard markers are treated as precious goods. They are tightly controlled and always in short supply. Invariably, half of the markers at any given whiteboard are dry and unusable. How often have you been in a meeting that was disrupted by lack of a working marker? How often have you had your train of thought derailed by a marker running out? How often have all the markers just gone missing, presumably because some other team ran out of markers and had to abscond with yours? All for a product that costs less than a dollar.

Google tends to have unlocked closets full of office supplies, including whiteboard markers, in most work areas. With a moment's notice it is easy to grab dozens of markers in a variety of colors. Somewhere along the line we made an explicit trade-off: it is far more important to optimize for obstacle-free brainstorming than to protect against someone wandering off with a bunch of markers.

We aim to have the same level of eyes-open and explicit weighing of the cost/benefit trade-offs involved for everything we do, from office supplies and employee perks through day-to-day experience for developers to how to provision and run global-scale services. We often say, "Google is a data-driven culture." In fact, that's a simplification: even when there isn't *data*, there might still be *evidence*, *precedent*, and *argument*. Making good engineering decisions is all about weighing all of the available inputs and making informed decisions about the trade-offs. Sometimes, those decisions are based on instinct or accepted best practice, but only after we have exhausted approaches that try to measure or estimate the true underlying costs.

In the end, decisions in an engineering group should come down to very few things:

- We are doing this because we must (legal requirements, customer requirements).
- We are doing this because it is the best option (as determined by some appropriate decider) we can see at the time, based on current evidence.

Decisions should not be "We are doing this because I said so."[17]

17 This is not to say that decisions need to be made unanimously, or even with broad consensus; in the end, someone must be the decider. This is primarily a statement of how the decision-making process should flow for whoever is actually responsible for the decision.

Inputs to Decision Making

When we are weighing data, we find two common scenarios:

- All of the quantities involved are measurable or can at least be estimated. This usually means that we're evaluating trade-offs between CPU and network, or dollars and RAM, or considering whether to spend two weeks of engineer-time in order to save *N* CPUs across our datacenters.
- Some of the quantities are subtle, or we don't know how to measure them. Sometimes this manifests as "We don't know how much engineer-time this will take." Sometimes it is even more nebulous: how do you measure the engineering cost of a poorly designed API? Or the societal impact of a product choice?

There is little reason to be deficient on the first type of decision. Any software engineering organization can and should track the current cost for compute resources, engineer-hours, and other quantities you interact with regularly. Even if you don't want to publicize to your organization the exact dollar amounts, you can still produce a conversion table: this many CPUs cost the same as this much RAM or this much network bandwidth.

With an agreed-upon conversion table in hand, every engineer can do their own analysis. "If I spend two weeks changing this linked-list into a higher-performance structure, I'm going to use five gibibytes more production RAM but save two thousand CPUs. Should I do it?" Not only does this question depend upon the relative cost of RAM and CPUs, but also on personnel costs (two weeks of support for a software engineer) and opportunity costs (what else could that engineer produce in two weeks?).

For the second type of decision, there is no easy answer. We rely on experience, leadership, and precedent to negotiate these issues. We're investing in research to help us quantify the hard-to-quantify (see Chapter 7). However, the best broad suggestion that we have is to be aware that not everything is measurable or predictable and to attempt to treat such decisions with the same priority and greater care. They are often just as important, but more difficult to manage.

Example: Distributed Builds

Consider your build. According to completely unscientific Twitter polling, something like 60 to 70% of developers build locally, even with today's large, complicated builds. This leads directly to nonjokes as illustrated by this "Compiling" comic—how much productive time in your organization is lost waiting for a build? Compare that to the cost to run something like `distcc` for a small group. Or, how much does it cost to run a small build farm for a large group? How many weeks/months does it take for those costs to be a net win?

Back in the mid-2000s, Google relied purely on a local build system: you checked out code and you compiled it locally. We had massive local machines in some cases (you could build Maps on your desktop!), but compilation times became longer and longer as the codebase grew. Unsurprisingly, we incurred increasing overhead in personnel costs due to lost time, as well as increased resource costs for larger and more powerful local machines, and so on. These resource costs were particularly troublesome: of course we want people to have as fast a build as possible, but most of the time, a high-performance desktop development machine will sit idle. This doesn't feel like the proper way to invest those resources.

Eventually, Google developed its own distributed build system. Development of this system incurred a cost, of course: it took engineers time to develop, it took more engineer time to change everyone's habits and workflow and learn the new system, and of course it cost additional computational resources. But the overall savings were clearly worth it: builds became faster, engineer-time was recouped, and hardware investment could focus on managed shared infrastructure (in actuality, a subset of our production fleet) rather than ever-more-powerful desktop machines. Chapter 18 goes into more of the details on our approach to distributed builds and the relevant trade-offs.

So, we built a new system, deployed it to production, and sped up everyone's build. Is that the happy ending to the story? Not quite: providing a distributed build system made massive improvements to engineer productivity, but as time went on, the distributed builds themselves became bloated. What was constrained in the previous case by individual engineers (because they had a vested interest in keeping their local builds as fast as possible) was unconstrained within a distributed build system. Bloated or unnecessary dependencies in the build graph became all too common. When everyone directly felt the pain of a nonoptimal build and was incentivized to be vigilant, incentives were better aligned. By removing those incentives and hiding bloated dependencies in a parallel distributed build, we created a situation in which consumption could run rampant, and almost nobody was incentivized to keep an eye on build bloat. This is reminiscent of Jevons Paradox: consumption of a resource may *increase* as a response to greater efficiency in its use.

Overall, the saved costs associated with adding a distributed build system far, far outweighed the negative costs associated with its construction and maintenance. But, as we saw with increased consumption, we did not foresee all of these costs. Having blazed ahead, we found ourselves in a situation in which we needed to reconceptualize the goals and constraints of the system and our usage, identify best practices (small dependencies, machine-management of dependencies), and fund the tooling and maintenance for the new ecosystem. Even a relatively simple trade-off of the form "We'll spend $$$s for compute resources to recoup engineer time" had unforeseen downstream effects.

Example: Deciding Between Time and Scale

Much of the time, our major themes of time and scale overlap and work in conjunction. A policy like the Beyoncé Rule scales well and helps us maintain things over time. A change to an OS interface might require many small refactorings to adapt to, but most of those changes will scale well because they are of a similar form: the OS change doesn't manifest differently for every caller and every project.

Occasionally time and scale come into conflict, and nowhere so clearly as in the basic question: should we add a dependency or fork/reimplement it to better suit our local needs?

This question can arise at many levels of the software stack because it is regularly the case that a bespoke solution customized for your narrow problem space may outperform the general utility solution that needs to handle all possibilities. By forking or reimplementing utility code and customizing it for your narrow domain, you can add new features with greater ease, or optimize with greater certainty, regardless of whether we are talking about a microservice, an in-memory cache, a compression routine, or anything else in our software ecosystem. Perhaps more important, the control you gain from such a fork isolates you from changes in your underlying dependencies: those changes aren't dictated by another team or third-party provider. You are in control of how and when to react to the passage of time and necessity to change.

On the other hand, if every developer forks everything used in their software project instead of reusing what exists, scalability suffers alongside sustainability. Reacting to a security issue in an underlying library is no longer a matter of updating a single dependency and its users: it is now a matter of identifying every vulnerable fork of that dependency and the users of those forks.

As with most software engineering decisions, there isn't a one-size-fits-all answer to this situation. If your project life span is short, forks are less risky. If the fork in question is provably limited in scope, that helps, as well—avoid forks for interfaces that could operate across time or project-time boundaries (data structures, serialization formats, networking protocols). Consistency has great value, but generality comes with its own costs, and you can often win by doing your own thing—if you do it carefully.

Revisiting Decisions, Making Mistakes

One of the unsung benefits of committing to a data-driven culture is the combined ability and necessity of admitting to mistakes. A decision will be made at some point, based on the available data—hopefully based on good data and only a few assumptions, but implicitly based on currently available data. As new data comes in, contexts change, or assumptions are dispelled, it might become clear that a decision was in

error or that it made sense at the time but no longer does. This is particularly critical for a long-lived organization: time doesn't only trigger changes in technical dependencies and software systems, but in data used to drive decisions.

We believe strongly in data informing decisions, but we recognize that the data will change over time, and new data may present itself. This means, inherently, that decisions will need to be revisited from time to time over the life span of the system in question. For long-lived projects, it's often critical to have the ability to change directions after an initial decision is made. And, importantly, it means that the deciders need to have the right to admit mistakes. Contrary to some people's instincts, leaders who admit mistakes are more respected, not less.

Be evidence driven, but also realize that things that can't be measured may still have value. If you're a leader, that's what you've been asked to do: exercise judgement, assert that things are important. We'll speak more on leadership in Chapters 5 and 6.

Software Engineering Versus Programming

When presented with our distinction between software engineering and programming, you might ask whether there is an inherent value judgement in play. Is programming somehow worse than software engineering? Is a project that is expected to last a decade with a team of hundreds inherently more valuable than one that is useful for only a month and built by two people?

Of course not. Our point is not that software engineering is superior, merely that these represent two different problem domains with distinct constraints, values, and best practices. Rather, the value in pointing out this difference comes from recognizing that some tools are great in one domain but not in the other. You probably don't need to rely on integration tests (see Chapter 14) and Continuous Deployment (CD) practices (see Chapter 24) for a project that will last only a few days. Similarly, all of our long-term concerns about semantic versioning (SemVer) and dependency management in software engineering projects (see Chapter 21) don't really apply for short-term programming projects: use whatever is available to solve the task at hand.

We believe it is important to differentiate between the related-but-distinct terms "programming" and "software engineering." Much of that difference stems from the management of code over time, the impact of time on scale, and decision making in the face of those ideas. Programming is the immediate act of producing code. Software engineering is the set of policies, practices, and tools that are necessary to make that code useful for as long as it needs to be used and allowing collaboration across a team.

Conclusion

This book discusses all of these topics: policies for an organization and for a single programmer, how to evaluate and refine your best practices, and the tools and technologies that go into maintainable software. Google has worked hard to have a sustainable codebase and culture. We don't necessarily think that our approach is the one true way to do things, but it does provide proof by example that it can be done. We hope it will provide a useful framework for thinking about the general problem: how do you maintain your code for as long as it needs to keep working?

TL;DRs

- "Software engineering" differs from "programming" in dimensionality: programming is about producing code. Software engineering extends that to include the maintenance of that code for its useful life span.
- There is a factor of at least 100,000 times between the life spans of short-lived code and long-lived code. It is silly to assume that the same best practices apply universally on both ends of that spectrum.
- Software is sustainable when, for the expected life span of the code, we are capable of responding to changes in dependencies, technology, or product requirements. We may choose to not change things, but we need to be capable.
- Hyrum's Law: with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.
- Every task your organization has to do repeatedly should be scalable (linear or better) in terms of human input. Policies are a wonderful tool for making process scalable.
- Process inefficiencies and other software-development tasks tend to scale up slowly. Be careful about boiled-frog problems.
- Expertise pays off particularly well when combined with economies of scale.
- "Because I said so" is a terrible reason to do things.
- Being data driven is a good start, but in reality, most decisions are based on a mix of data, assumption, precedent, and argument. It's best when objective data makes up the majority of those inputs, but it can rarely be *all* of them.
- Being data driven over time implies the need to change directions when the data changes (or when assumptions are dispelled). Mistakes or revised plans are inevitable.

PART II
Culture

CHAPTER 2

How to Work Well on Teams

Written by Brian Fitzpatrick
Edited by Riona MacNamara

Because this chapter is about the cultural and social aspects of software engineering at Google, it makes sense to begin by focusing on the one variable over which you definitely have control: you.

People are inherently imperfect—we like to say that humans are mostly a collection of intermittent bugs. But before you can understand the bugs in your coworkers, you need to understand the bugs in yourself. We're going to ask you to think about your own reactions, behaviors, and attitudes—and in return, we hope you gain some real insight into how to become a more efficient and successful software engineer who spends less energy dealing with people-related problems and more time writing great code.

The critical idea in this chapter is that software development is a team endeavor. And to succeed on an engineering team—or in any other creative collaboration—you need to reorganize your behaviors around the core principles of humility, respect, and trust.

Before we get ahead of ourselves, let's begin by observing how software engineers tend to behave in general.

Help Me Hide My Code

For the past 20 years, my colleague Ben[1] and I have spoken at many programming conferences. In 2006, we launched Google's (now deprecated) open source Project

1 Ben Collins-Sussman, also an author within this book.

Hosting service, and at first, we used to get lots of questions and requests about the product. But around mid-2008, we began to notice a trend in the sort of requests we were getting:

> "Can you please give Subversion on Google Code the ability to hide specific branches?"
>
> "Can you make it possible to create open source projects that start out hidden to the world and then are revealed when they're ready?"
>
> "Hi, I want to rewrite all my code from scratch, can you please wipe all the history?"

Can you spot a common theme to these requests?

The answer is *insecurity*. People are afraid of others seeing and judging their work in progress. In one sense, insecurity is just a part of human nature—nobody likes to be criticized, especially for things that aren't finished. Recognizing this theme tipped us off to a more general trend within software development: insecurity is actually a symptom of a larger problem.

The Genius Myth

Many humans have the instinct to find and worship idols. For software engineers, those might be Linus Torvalds, Guido Van Rossum, Bill Gates—all heroes who changed the world with heroic feats. Linus wrote Linux by himself, right?

Actually, what Linus did was write just the beginnings of a proof-of-concept Unix-like kernel and show it to an email list. That was no small accomplishment, and it was definitely an impressive achievement, but it was just the tip of the iceberg. Linux is hundreds of times bigger than that initial kernel and was developed by *thousands* of smart people. Linus' real achievement was to lead these people and coordinate their work; Linux is the shining result not of his original idea, but of the collective labor of the community. (And Unix itself was not entirely written by Ken Thompson and Dennis Ritchie, but by a group of smart people at Bell Labs.)

On that same note, did Guido Van Rossum personally write all of Python? Certainly, he wrote the first version. But hundreds of others were responsible for contributing to subsequent versions, including ideas, features, and bug fixes. Steve Jobs led an entire team that built the Macintosh, and although Bill Gates is known for writing a BASIC interpreter for early home computers, his bigger achievement was building a successful company around MS-DOS. Yet they all became leaders and symbols of the collective achievements of their communities. The Genius Myth is the tendency that we as humans need to ascribe the success of a team to a single person/leader.

And what about Michael Jordan?

It's the same story. We idolized him, but the fact is that he didn't win every basketball game by himself. His true genius was in the way he worked with his team. The team's coach, Phil Jackson, was extremely clever, and his coaching techniques are legendary.

He recognized that one player alone never wins a championship, and so he assembled an entire "dream team" around MJ. This team was a well-oiled machine—at least as impressive as Michael himself.

So, why do we repeatedly idolize the individual in these stories? Why do people buy products endorsed by celebrities? Why do we want to buy Michelle Obama's dress or Michael Jordan's shoes?

Celebrity is a big part of it. Humans have a natural instinct to find leaders and role models, idolize them, and attempt to imitate them. We all need heroes for inspiration, and the programming world has its heroes, too. The phenomenon of "techie-celebrity" has almost spilled over into mythology. We all want to write something world-changing like Linux or design the next brilliant programming language.

Deep down, many engineers secretly wish to be seen as geniuses. This fantasy goes something like this:

- You are struck by an awesome new concept.
- You vanish into your cave for weeks or months, slaving away at a perfect implementation of your idea.
- You then "unleash" your software on the world, shocking everyone with your genius.
- Your peers are astonished by your cleverness.
- People line up to use your software.
- Fame and fortune follow naturally.

But hold on: time for a reality check. You're probably not a genius.

No offense, of course—we're sure that you're a very intelligent person. But do you realize how rare actual geniuses really are? Sure, you write code, and that's a tricky skill. But even if you are a genius, it turns out that that's not enough. Geniuses still make mistakes, and having brilliant ideas and elite programming skills doesn't guarantee that your software will be a hit. Worse, you might find yourself solving only analytical problems and not *human* problems. Being a genius is most definitely not an excuse for being a jerk: anyone—genius or not—with poor social skills tends to be a poor teammate. The vast majority of the work at Google (and at most companies!) doesn't require genius-level intellect, but 100% of the work requires a minimal level of social skills. What will make or break your career, especially at a company like Google, is how well you collaborate with others.

It turns out that this Genius Myth is just another manifestation of our insecurity. Many programmers are afraid to share work they've only just started because it means peers will see their mistakes and know the author of the code is not a genius.

To quote a friend:

> I know I get SERIOUSLY insecure about people looking before something is done. Like they are going to seriously judge me and think I'm an idiot.

This is an extremely common feeling among programmers, and the natural reaction is to hide in a cave, work, work, work, and then polish, polish, polish, sure that no one will see your goof-ups and that you'll still have a chance to unveil your masterpiece when you're done. Hide away until your code is perfect.

Another common motivation for hiding your work is the fear that another programmer might take your idea and run with it before you get around to working on it. By keeping it secret, you control the idea.

We know what you're probably thinking now: so what? Shouldn't people be allowed to work however they want?

Actually, no. In this case, we assert that you're doing it wrong, and it *is* a big deal. Here's why.

Hiding Considered Harmful

If you spend all of your time working alone, you're increasing the risk of unnecessary failure and cheating your potential for growth. Even though software development is deeply intellectual work that can require deep concentration and alone time, you must play that off against the value (and need!) for collaboration and review.

First of all, how do you even know whether you're on the right track?

Imagine you're a bicycle-design enthusiast, and one day you get a brilliant idea for a completely new way to design a gear shifter. You order parts and proceed to spend weeks holed up in your garage trying to build a prototype. When your neighbor—also a bike advocate—asks you what's up, you decide not to talk about it. You don't want anyone to know about your project until it's absolutely perfect. Another few months go by and you're having trouble making your prototype work correctly. But because you're working in secrecy, it's impossible to solicit advice from your mechanically inclined friends.

Then, one day your neighbor pulls his bike out of his garage with a radical new gear-shifting mechanism. Turns out he's been building something very similar to your invention, but with the help of some friends down at the bike shop. At this point, you're exasperated. You show him your work. He points out that your design had some simple flaws—ones that might have been fixed in the first week if you had shown him. There are a number of lessons to learn here.

Early Detection

If you keep your great idea hidden from the world and refuse to show anyone anything until the implementation is polished, you're taking a huge gamble. It's easy to make fundamental design mistakes early on. You risk reinventing wheels.[2] And you forfeit the benefits of collaboration, too: notice how much faster your neighbor moved by working with others? This is why people dip their toes in the water before jumping in the deep end: you need to make sure that you're working on the right thing, you're doing it correctly, and it hasn't been done before. The chances of an early misstep are high. The more feedback you solicit early on, the more you lower this risk.[3] Remember the tried-and-true mantra of "Fail early, fail fast, fail often."

Early sharing isn't just about preventing personal missteps and getting your ideas vetted. It's also important to strengthen what we call the bus factor of your project.

The Bus Factor

> Bus factor (noun): the number of people that need to get hit by a bus before your project is completely doomed.

How dispersed is the knowledge and know-how in your project? If you're the only person who understands how the prototype code works, you might enjoy good job security—but if you get hit by a bus, the project is toast. If you're working with a colleague, however, you've doubled the bus factor. And if you have a small team designing and prototyping together, things are even better—the project won't be marooned when a team member disappears. Remember: team members might not literally be hit by buses, but other unpredictable life events still happen. Someone might get married, move away, leave the company, or take leave to care for a sick relative. Ensuring that there is *at least* good documentation in addition to a primary and a secondary owner for each area of responsibility helps future-proof your project's success and increases your project's bus factor. Hopefully most engineers recognize that it is better to be one part of a successful project than the critical part of a failed project.

Beyond the bus factor, there's the issue of overall pace of progress. It's easy to forget that working alone is often a tough slog, much slower than people want to admit. How much do you learn when working alone? How fast do you move? Google and Stack Overflow are great sources of opinions and information, but they're no substitute for actual human experience. Working with other people directly increases the collective wisdom behind the effort. When you become stuck on something absurd, how much time do you waste pulling yourself out of the hole? Think about how

2 Literally, if you are, in fact, a bike designer.

3 I should note that sometimes it's dangerous to get too much feedback too early in the process if you're still unsure of your general direction or goal.

different the experience would be if you had a couple of peers to look over your shoulder and tell you—instantly—how you goofed and how to get past the problem. This is exactly why teams sit together (or do pair programming) in software engineering companies. Programming is hard. Software engineering is even harder. You need that second pair of eyes.

Pace of Progress

Here's another analogy. Think about how you work with your compiler. When you sit down to write a large piece of software, do you spend days writing 10,000 lines of code, and then, after writing that final, perfect line, press the "compile" button for the very first time? Of course you don't. Can you imagine what sort of disaster would result? Programmers work best in tight feedback loops: write a new function, compile. Add a test, compile. Refactor some code, compile. This way, we discover and fix typos and bugs as soon as possible after generating code. We want the compiler at our side for every little step; some environments can even compile our code as we type. This is how we keep code quality high and make sure our software is evolving correctly, bit by bit. The current DevOps philosophy toward tech productivity is explicit about these sorts of goals: get feedback as early as possible, test as early as possible, and think about security and production environments as early as possible. This is all bundled into the idea of "shifting left" in the developer workflow; the earlier we find a problem, the cheaper it is to fix it.

The same sort of rapid feedback loop is needed not just at the code level, but at the whole-project level, too. Ambitious projects evolve quickly and must adapt to changing environments as they go. Projects run into unpredictable design obstacles or political hazards, or we simply discover that things aren't working as planned. Requirements morph unexpectedly. How do you get that feedback loop so that you know the instant your plans or designs need to change? Answer: by working in a team. Most engineers know the quote, "Many eyes make all bugs shallow," but a better version might be, "Many eyes make sure your project stays relevant and on track." People working in caves awaken to discover that while their original vision might be complete, the world has changed and their project has become irrelevant.

Case Study: Engineers and Offices

Twenty-five years ago, conventional wisdom stated that for an engineer to be productive, they needed to have their own office with a door that closed. This was supposedly the only way they could have big, uninterrupted slabs of time to deeply concentrate on writing reams of code.

I think that it's not only unnecessary for most engineers[4] to be in a private office, it's downright dangerous. Software today is written by teams, not individuals, and a high-bandwidth, readily available connection to the rest of your team is even more valuable than your internet connection. You can have all the uninterrupted time in the world, but if you're using it to work on the wrong thing, you're wasting your time.

Unfortunately, it seems that modern-day tech companies (including Google, in some cases) have swung the pendulum to the exact opposite extreme. Walk into their offices and you'll often find engineers clustered together in massive rooms—a hundred or more people together—with no walls whatsoever. This "open floor plan" is now a topic of huge debate and, as a result, hostility toward open offices is on the rise. The tiniest conversation becomes public, and people end up not talking for risk of annoying dozens of neighbors. This is just as bad as private offices!

We think the middle ground is really the best solution. Group teams of four to eight people together in small rooms (or large offices) to make it easy (and non-embarrassing) for spontaneous conversation to happen.

Of course, in any situation, individual engineers still need a way to filter out noise and interruptions, which is why most teams I've seen have developed a way to communicate that they're currently busy and that you should limit interruptions. Some of us used to work on a team with a vocal interrupt protocol: if you wanted to talk, you would say "Breakpoint Mary," where Mary was the name of the person you wanted to talk to. If Mary was at a point where she could stop, she would swing her chair around and listen. If Mary was too busy, she'd just say "ack," and you'd go on with other things until she finished with her current head state.

Other teams have tokens or stuffed animals that team members put on their monitor to signify that they should be interrupted only in case of emergency. Still other teams give out noise-canceling headphones to engineers to make it easier to deal with background noise—in fact, in many companies, the very act of wearing headphones is a common signal that means "don't disturb me unless it's really important." Many engineers tend to go into headphones-only mode when coding, which may be useful for short spurts but, if used all the time, can be just as bad for collaboration as walling yourself off in an office.

Don't misunderstand us—we still think engineers need uninterrupted time to focus on writing code, but we think they need a high-bandwidth, low-friction connection to their team just as much. If less-knowledgeable people on your team feel that there's a barrier to asking you a question, it's a problem: finding the right balance is an art.

4 I do, however, acknowledge that serious introverts likely need more peace, quiet, and alone time than most people and might benefit from a quieter environment, if not their own office.

In Short, Don't Hide

So, what "hiding" boils down to is this: working alone is inherently riskier than working with others. Even though you might be afraid of someone stealing your idea or thinking you're not intelligent, you should be much more concerned about wasting huge swaths of your time toiling away on the wrong thing.

Don't become another statistic.

It's All About the Team

So, let's back up now and put all of these ideas together.

The point we've been hammering away at is that, in the realm of programming, lone craftspeople are extremely rare—and even when they do exist, they don't perform superhuman achievements in a vacuum; their world-changing accomplishment is almost always the result of a spark of inspiration followed by a heroic team effort.

A great team makes brilliant use of its superstars, but the whole is always greater than the sum of its parts. But creating a superstar team is fiendishly difficult.

Let's put this idea into simpler words: *software engineering is a team endeavor*.

This concept directly contradicts the inner Genius Programmer fantasy so many of us hold, but it's not enough to be brilliant when you're alone in your hacker's lair. You're not going to change the world or delight millions of computer users by hiding and preparing your secret invention. You need to work with other people. Share your vision. Divide the labor. Learn from others. Create a brilliant team.

Consider this: how many pieces of widely used, successful software can you name that were truly written by a single person? (Some people might say "LaTeX," but it's hardly "widely used," unless you consider the number of people writing scientific papers to be a statistically significant portion of all computer users!)

High-functioning teams are gold and the true key to success. You should be aiming for this experience however you can.

The Three Pillars of Social Interaction

So, if teamwork is the best route to producing great software, how does one build (or find) a great team?

To reach collaborative nirvana, you first need to learn and embrace what I call the "three pillars" of social skills. These three principles aren't just about greasing the wheels of relationships; they're the foundation on which all healthy interaction and collaboration are based:

Pillar 1: Humility
: You are not the center of the universe (nor is your code!). You're neither omniscient nor infallible. You're open to self-improvement.

Pillar 2: Respect
: You genuinely care about others you work with. You treat them kindly and appreciate their abilities and accomplishments.

Pillar 3: Trust
: You believe others are competent and will do the right thing, and you're OK with letting them drive when appropriate.[5]

If you perform a root-cause analysis on almost any social conflict, you can ultimately trace it back to a lack of humility, respect, and/or trust. That might sound implausible at first, but give it a try. Think about some nasty or uncomfortable social situation currently in your life. At the basest level, is everyone being appropriately humble? Are people really respecting one another? Is there mutual trust?

Why Do These Pillars Matter?

When you began this chapter, you probably weren't planning to sign up for some sort of weekly support group. We empathize. Dealing with social problems can be difficult: people are messy, unpredictable, and often annoying to interface with. Rather than putting energy into analyzing social situations and making strategic moves, it's tempting to write off the whole effort. It's much easier to hang out with a predictable compiler, isn't it? Why bother with the social stuff at all?

Here's a quote from a famous lecture by Richard Hamming:

> By taking the trouble to tell jokes to the secretaries and being a little friendly, I got superb secretarial help. For instance, one time for some idiot reason all the reproducing services at Murray Hill were tied up. Don't ask me how, but they were. I wanted something done. My secretary called up somebody at Holmdel, hopped [into] the company car, made the hour-long trip down and got it reproduced, and then came back. It was a payoff for the times I had made an effort to cheer her up, tell her jokes and be friendly; it was that little extra work that later paid off for me. By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires.

The moral is this: do not underestimate the power of playing the social game. It's not about tricking or manipulating people; it's about creating relationships to get things done. Relationships always outlast projects. When you've got richer relationships with your coworkers, they'll be more willing to go the extra mile when you need them.

5 This is incredibly difficult if you've been burned in the past by delegating to incompetent people.

Humility, Respect, and Trust in Practice

All of this preaching about humility, respect, and trust sounds like a sermon. Let's come out of the clouds and think about how to apply these ideas in real-life situations. We're going to examine a list of specific behaviors and examples that you can start with. Many of them might sound obvious at first, but after you begin thinking about them, you'll notice how often you (and your peers) are guilty of not following them—we've certainly noticed this about ourselves!

Lose the ego

OK, this is sort of a simpler way of telling someone without enough humility to lose their 'tude. Nobody wants to work with someone who consistently behaves like they're the most important person in the room. Even if you know you're the wisest person in the discussion, don't wave it in people's faces. For example, do you always feel like you need to have the first or last word on every subject? Do you feel the need to comment on every detail in a proposal or discussion? Or do you know somebody who does these things?

Although it's important to be humble, that doesn't mean you need to be a doormat; there's nothing wrong with self-confidence. Just don't come off like a know-it-all. Even better, think about going for a "collective" ego, instead; rather than worrying about whether you're personally awesome, try to build a sense of team accomplishment and group pride. For example, the Apache Software Foundation has a long history of creating communities around software projects. These communities have incredibly strong identities and reject people who are more concerned with self-promotion.

Ego manifests itself in many ways, and a lot of the time, it can get in the way of your productivity and slow you down. Here's another great story from Hamming's lecture that illustrates this point perfectly (emphasis ours):

> John Tukey almost always dressed very casually. He would go into an important office and it would take a long time before the other fellow realized that this is a first-class man and he had better listen. For a long time, John has had to overcome this kind of hostility. It's wasted effort! I didn't say you should conform; I said, "The appearance of conforming gets you a long way." If you chose to assert your ego in any number of ways, "I am going to do it my way," you pay a small steady price throughout the whole of your professional career. And this, over a whole lifetime, adds up to an enormous amount of needless trouble. […] By realizing you have to use the system and studying how to get the system to do your work, you learn how to adapt the system to your desires. *Or you can fight it steadily, as a small, undeclared war, for the whole of your life.*

Learn to give *and* take criticism

A few years ago, Joe started a new job as a programmer. After his first week, he really began digging into the codebase. Because he cared about what was going on, he started gently questioning other teammates about their contributions. He sent simple code reviews by email, politely asking about design assumptions or pointing out places where logic could be improved. After a couple of weeks, he was summoned to his director's office. "What's the problem?" Joe asked. "Did I do something wrong?" The director looked concerned: "We've had a lot of complaints about your behavior, Joe. Apparently, you've been really harsh toward your teammates, criticizing them left and right. They're upset. You need to tone it down." Joe was utterly baffled. Surely, he thought, his code reviews should have been welcomed and appreciated by his peers. In this case, however, Joe should have been more sensitive to the team's widespread insecurity and should have used a subtler means to introduce code reviews into the culture—perhaps even something as simple as discussing the idea with the team in advance and asking team members to try it out for a few weeks.

In a professional software engineering environment, criticism is almost never personal—it's usually just part of the process of making a better project. The trick is to make sure you (and those around you) understand the difference between a constructive criticism of someone's creative output and a flat-out assault against someone's character. The latter is useless—it's petty and nearly impossible to act on. The former can (and should!) be helpful and give guidance on how to improve. And, most important, it's imbued with respect: the person giving the constructive criticism genuinely cares about the other person and wants them to improve themselves or their work. Learn to respect your peers and give constructive criticism politely. If you truly respect someone, you'll be motivated to choose tactful, helpful phrasing—a skill acquired with much practice. We cover this much more in Chapter 9.

On the other side of the conversation, you need to learn to accept criticism as well. This means not just being humble about your skills, but trusting that the other person has your best interests (and those of your project!) at heart and doesn't actually think you're an idiot. Programming is a skill like anything else: it improves with practice. If a peer pointed out ways in which you could improve your juggling, would you take it as an attack on your character and value as a human being? We hope not. In the same way, your self-worth shouldn't be connected to the code you write—or any creative project you build. To repeat ourselves: *you are not your code*. Say that over and over. You are not what you make. You need to not only believe it yourself, but get your coworkers to believe it, too.

For example, if you have an insecure collaborator, here's what not to say: "Man, you totally got the control flow wrong on that method there. You should be using the standard xyzzy code pattern like everyone else." This feedback is full of antipatterns: you're telling someone they're "wrong" (as if the world were black and white), demanding they change something, and accusing them of creating something that

goes against what everyone else is doing (making them feel stupid). Your coworker will immediately be put on the offense, and their response is bound to be overly emotional.

A better way to say the same thing might be, "Hey, I'm confused by the control flow in this section here. I wonder if the xyzzy code pattern might make this clearer and easier to maintain?" Notice how you're using humility to make the question about you, not them. They're not wrong; you're just having trouble understanding the code. The suggestion is merely offered up as a way to clarify things for poor little you while possibly helping the project's long-term sustainability goals. You're also not demanding anything—you're giving your collaborator the ability to peacefully reject the suggestion. The discussion stays focused on the code itself, not on anyone's value or coding skills.

Fail fast and iterate

There's a well-known urban legend in the business world about a manager who makes a mistake and loses an impressive $10 million. He dejectedly goes into the office the next day and starts packing up his desk, and when he gets the inevitable "the CEO wants to see you in his office" call, he trudges into the CEO's office and quietly slides a piece of paper across the desk.

> "What's this?" asks the CEO.
>
> "My resignation," says the executive. "I assume you called me in here to fire me."
>
> "Fire you?" responds the CEO, incredulously. "Why would I fire you? I just spent $10 million training you!"[6]

It's an extreme story, to be sure, but the CEO in this story understands that firing the executive wouldn't undo the $10 million loss, and it would compound it by losing a valuable executive who he can be very sure won't make that kind of mistake again.

At Google, one of our favorite mottos is that "Failure is an option." It's widely recognized that if you're not failing now and then, you're not being innovative enough or taking enough risks. Failure is viewed as a golden opportunity to learn and improve for the next go-around.[7] In fact, Thomas Edison is often quoted as saying, "If I find 10,000 ways something won't work, I haven't failed. I am not discouraged, because every wrong attempt discarded is another step forward."

Over in Google X—the division that works on "moonshots" like self-driving cars and internet access delivered by balloons—failure is deliberately built into its incentive system. People come up with outlandish ideas and coworkers are actively encouraged

6 You can find a dozen variants of this legend on the web, attributed to different famous managers.

7 By the same token, if you do the same thing over and over and keep failing, it's not failure, it's incompetence.

to shoot them down as fast as possible. Individuals are rewarded (and even compete) to see how many ideas they can disprove or invalidate in a fixed period of time. Only when a concept truly cannot be debunked at a whiteboard by all peers does it proceed to early prototype.

Blameless Post-Mortem Culture

The key to learning from your mistakes is to document your failures by performing a root-cause analysis and writing up a "postmortem," as it's called at Google (and many other companies). Take extra care to make sure the postmortem document isn't just a useless list of apologies or excuses or finger-pointing—that's not its purpose. A proper postmortem should always contain an explanation of what was learned and what is going to change as a result of the learning experience. Then, make sure that the postmortem is readily accessible and that the team really follows through on the proposed changes. Properly documenting failures also makes it easier for other people (present and future) to know what happened and avoid repeating history. Don't erase your tracks—light them up like a runway for those who follow you!

A good postmortem should include the following:

- A brief summary of the event
- A timeline of the event, from discovery through investigation to resolution
- The primary cause of the event
- Impact and damage assessment
- A set of action items (with owners) to fix the problem immediately
- A set of action items to prevent the event from happening again
- Lessons learned

Learn patience

Years ago, I was writing a tool to convert CVS repositories to Subversion (and later, Git). Due to the vagaries of CVS, I kept unearthing bizarre bugs. Because my longtime friend and coworker Karl knew CVS quite intimately, we decided we should work together to fix these bugs.

A problem arose when we began pair programming: I'm a bottom-up engineer who is content to dive into the muck and dig my way out by trying a lot of things quickly and skimming over the details. Karl, however, is a top-down engineer who wants to get the full lay of the land and dive into the implementation of almost every method on the call stack before proceeding to tackle the bug. This resulted in some epic interpersonal conflicts, disagreements, and the occasional heated argument. It got to the

point at which the two of us simply couldn't pair-program together: it was too frustrating for us both.

That said, we had a longstanding history of trust and respect for each other. Combined with patience, this helped us work out a new method of collaborating. We would sit together at the computer, identify the bug, and then split up and attack the problem from two directions at once (top-down and bottom-up) before coming back together with our findings. Our patience and willingness to improvise new working styles not only saved the project, but also our friendship.

Be open to influence

The more open you are to influence, the more you are able to influence; the more vulnerable you are, the stronger you appear. These statements sound like bizarre contradictions. But everyone can think of someone they've worked with who is just maddeningly stubborn—no matter how much people try to persuade them, they dig their heels in even more. What eventually happens to such team members? In our experience, people stop listening to their opinions or objections; instead, they end up "routing around" them like an obstacle everyone takes for granted. You certainly don't want to be that person, so keep this idea in your head: it's OK for someone else to change your mind. In the opening chapter of this book, we said that engineering is inherently about trade-offs. It's impossible for you to be right about everything all the time unless you have an unchanging environment and perfect knowledge, so of course you should change your mind when presented with new evidence. Choose your battles carefully: to be heard properly, you first need to listen to others. It's better to do this listening *before* putting a stake in the ground or firmly announcing a decision—if you're constantly changing your mind, people will think you're wishy-washy.

The idea of vulnerability can seem strange, too. If someone admits ignorance of the topic at hand or the solution to a problem, what sort of credibility will they have in a group? Vulnerability is a show of weakness, and that destroys trust, right?

Not true. Admitting that you've made a mistake or you're simply out of your league can increase your status over the long run. In fact, the willingness to express vulnerability is an outward show of humility, it demonstrates accountability and the willingness to take responsibility, and it's a signal that you trust others' opinions. In return, people end up respecting your honesty and strength. Sometimes, the best thing you can do is just say, "I don't know."

Professional politicians, for example, are notorious for never admitting error or ignorance, even when it's patently obvious that they're wrong or unknowledgeable about a subject. This behavior exists primarily because politicians are constantly under attack by their opponents, and it's why most people don't believe a word that politicians say. When you're writing software, however, you don't need to be continually on the

defensive—your teammates are collaborators, not competitors. You all have the same goal.

Being Googley

At Google, we have our own internal version of the principles of "humility, respect, and trust" when it comes to behavior and human interactions.

From the earliest days of our culture, we often referred to actions as being "Googley" or "not Googley." The word was never explicitly defined; rather, everyone just sort of took it to mean "don't be evil" or "do the right thing" or "be good to each other." Over time, people also started using the term "Googley" as an informal test for culture-fit whenever we would interview a candidate for an engineering job, or when writing internal performance reviews of one another. People would often express opinions about others using the term; for example, "the person coded well, but didn't seem to have a very Googley attitude."

Of course, we eventually realized that the term "Googley" was being overloaded with meaning; worse yet, it could become a source of unconscious bias in hiring or evaluations. If "Googley" means something different to every employee, we run the risk of the term starting to mean "*is just like me.*" Obviously, that's not a good test for hiring—we don't want to hire people "just like me," but people from a diverse set of backgrounds and with different opinions and experiences. An interviewer's personal desire to have a beer with a candidate (or coworker) should *never* be considered a valid signal about somebody else's performance or ability to thrive at Google.

Google eventually fixed the problem by explicitly defining a rubric for what we mean by "Googleyness"—a set of attributes and behaviors that we look for that represent strong leadership and exemplify "humility, respect, and trust":

Thrives in ambiguity
: Can deal with conflicting messages or directions, build consensus, and make progress against a problem, even when the environment is constantly shifting.

Values feedback
: Has humility to both receive and give feedback gracefully and understands how valuable feedback is for personal (and team) development.

Challenges status quo
: Is able to set ambitious goals and pursue them even when there might be resistance or inertia from others.

Puts the user first
: Has empathy and respect for users of Google's products and pursues actions that are in their best interests.

Cares about the team
: Has empathy and respect for coworkers and actively works to help them without being asked, improving team cohesion.

Does the right thing
: Has a strong sense of ethics about everything they do; willing to make difficult or inconvenient decisions to protect the integrity of the team and product.

Now that we have these best-practice behaviors better defined, we've begun to shy away from using the term "Googley." It's always better to be specific about expectations!

Conclusion

The foundation for almost any software endeavor—of almost any size—is a well-functioning team. Although the Genius Myth of the solo software developer still persists, the truth is that no one really goes it alone. For a software organization to stand the test of time, it must have a healthy culture, rooted in humility, trust, and respect that revolves around the team, rather than the individual. Further, the creative nature of software development *requires* that people take risks and occasionally fail; for people to accept that failure, a healthy team environment must exist.

TL;DRs

- Be aware of the trade-offs of working in isolation.
- Acknowledge the amount of time that you and your team spend communicating and in interpersonal conflict. A small investment in understanding personalities and working styles of yourself and others can go a long way toward improving productivity.
- If you want to work effectively with a team or a large organization, be aware of your preferred working style and that of others.

CHAPTER 4
Engineering for Equity

Written by Demma Rodriguez
Edited by Riona MacNamara

In earlier chapters, we've explored the contrast between programming as the production of code that addresses the problem of the moment, and software engineering as the broader application of code, tools, policies, and processes to a dynamic and ambiguous problem that can span decades or even lifetimes. In this chapter, we'll discuss the unique responsibilities of an engineer when designing products for a broad base of users. Further, we evaluate how an organization, by embracing diversity, can design systems that work for everyone, and avoid perpetuating harm against our users.

As new as the field of software engineering is, we're newer still at understanding the impact it has on underrepresented people and diverse societies. We did not write this chapter because we know all the answers. We do not. In fact, understanding how to engineer products that empower and respect all our users is still something Google is learning to do. We have had many public failures in protecting our most vulnerable users, and so we are writing this chapter because the path forward to more equitable products begins with evaluating our own failures and encouraging growth.

We are also writing this chapter because of the increasing imbalance of power between those who make development decisions that impact the world and those who simply must accept and live with those decisions that sometimes disadvantage already marginalized communities globally. It is important to share and reflect on what we've learned so far with the next generation of software engineers. It is even more important that we help influence the next generation of engineers to be better than we are today.

Just picking up this book means that you likely aspire to be an exceptional engineer. You want to solve problems. You aspire to build products that drive positive outcomes for the broadest base of people, including people who are the most difficult to reach. To do this, you will need to consider how the tools you build will be leveraged to change the trajectory of humanity, hopefully for the better.

Bias Is the Default

When engineers do not focus on users of different nationalities, ethnicities, races, genders, ages, socioeconomic statuses, abilities, and belief systems, even the most talented staff will inadvertently fail their users. Such failures are often unintentional; all people have certain biases, and social scientists have recognized over the past several decades that most people exhibit unconscious bias, enforcing and promulgating existing stereotypes. Unconscious bias is insidious and often more difficult to mitigate than intentional acts of exclusion. Even when we want to do the right thing, we might not recognize our own biases. By the same token, our organizations must also recognize that such bias exists and work to address it in their workforces, product development, and user outreach.

Because of bias, Google has at times failed to represent users equitably within their products, with launches over the past several years that did not focus enough on underrepresented groups. Many users attribute our lack of awareness in these cases to the fact that our engineering population is mostly male, mostly White or Asian, and certainly not representative of all the communities that use our products. The lack of representation of such users in our workforce[1] means that we often do not have the requisite diversity to understand how the use of our products can affect underrepresented or vulnerable users.

Case Study: Google Misses the Mark on Racial Inclusion

In 2015, software engineer Jacky Alciné pointed out[2] that the image recognition algorithms in Google Photos were classifying his black friends as "gorillas." Google was slow to respond to these mistakes and incomplete in addressing them.

What caused such a monumental failure? Several things:

- Image recognition algorithms depend on being supplied a "proper" (often meaning "complete") dataset. The photo data fed into Google's image recognition algorithm was clearly incomplete. In short, the data did not represent the population.

1 Google's 2019 Diversity Report.

2 @jackyalcine. 2015. "Google Photos, Y'all Fucked up. My Friend's Not a Gorilla." Twitter, June 29, 2015. *https://twitter.com/jackyalcine/status/615329515909156865*.

- Google itself (and the tech industry in general) did not (and does not) have much black representation,[3] and that affects decisions subjective in the design of such algorithms and the collection of such datasets. The unconscious bias of the organization itself likely led to a more representative product being left on the table.
- Google's target market for image recognition did not adequately include such underrepresented groups. Google's tests did not catch these mistakes; as a result, our users did, which both embarrassed Google and harmed our users.

As late as 2018, Google still had not adequately addressed the underlying problem.[4]

In this example, our product was inadequately designed and executed, failing to properly consider all racial groups, and as a result, failed our users and caused Google bad press. Other technology suffers from similar failures: autocomplete can return offensive or racist results. Google's Ad system could be manipulated to show racist or offensive ads. YouTube might not catch hate speech, though it is technically outlawed on that platform.

In all of these cases, the technology itself is not really to blame. Autocomplete, for example, was not designed to target users or to discriminate. But it was also not resilient enough in its design to exclude discriminatory language that is considered hate speech. As a result, the algorithm returned results that caused harm to our users. The harm to Google itself should also be obvious: reduced user trust and engagement with the company. For example, Black, Latinx, and Jewish applicants could lose faith in Google as a platform or even as an inclusive environment itself, therefore undermining Google's goal of improving representation in hiring.

How could this happen? After all, Google hires technologists with impeccable education and/or professional experience—exceptional programmers who write the best code and test their work. "Build for everyone" is a Google brand statement, but the truth is that we still have a long way to go before we can claim that we do. One way to address these problems is to help the software engineering organization itself look like the populations for whom we build products.

3 Many reports in 2018–2019 pointed to a lack of diversity across tech. Some notables include the National Center for Women & Information Technology, and Diversity in Tech.

4 Tom Simonite, "When It Comes to Gorillas, Google Photos Remains Blind," *Wired*, January 11, 2018.

Understanding the Need for Diversity

At Google, we believe that being an exceptional engineer requires that you also focus on bringing diverse perspectives into product design and implementation. It also means that Googlers responsible for hiring or interviewing other engineers must contribute to building a more representative workforce. For example, if you interview other engineers for positions at your company, it is important to learn how biased outcomes happen in hiring. There are significant prerequisites for understanding how to anticipate harm and prevent it. To get to the point where we can build for everyone, we first must understand our representative populations. We need to encourage engineers to have a wider scope of educational training.

The first order of business is to disrupt the notion that as a person with a computer science degree and/or work experience, you have all the skills you need to become an exceptional engineer. A computer science degree is often a necessary foundation. However, the degree alone (even when coupled with work experience) will not make you an engineer. It is also important to disrupt the idea that only people with computer science degrees can design and build products. Today, most programmers do have a computer science degree; they are successful at building code, establishing theories of change, and applying methodologies for problem solving. However, as the aforementioned examples demonstrate, *this approach is insufficient for inclusive and equitable engineering*.

Engineers should begin by focusing all work within the framing of the complete ecosystem they seek to influence. At minimum, they need to understand the population demographics of their users. Engineers should focus on people who are different than themselves, especially people who might attempt to use their products to cause harm. The most difficult users to consider are those who are disenfranchised by the processes and the environment in which they access technology. To address this challenge, engineering teams need to be representative of their existing and future users. In the absence of diverse representation on engineering teams, individual engineers need to learn how to build for all users.

Building Multicultural Capacity

One mark of an exceptional engineer is the ability to understand how products can advantage and disadvantage different groups of human beings. Engineers are expected to have technical aptitude, but they should also have the *discernment* to know when to build something and when not to. Discernment includes building the capacity to identify and reject features or products that drive adverse outcomes. This is a lofty and difficult goal, because there is an enormous amount of individualism that goes into being a high-performing engineer. Yet to succeed, we must extend our

focus beyond our own communities to the next billion users or to current users who might be disenfranchised or left behind by our products.

Over time, you might build tools that billions of people use daily—tools that influence how people think about the value of human lives, tools that monitor human activity, and tools that capture and persist sensitive data, such as images of their children and loved ones, as well as other types of sensitive data. As an engineer, you might wield more power than you realize: the power to literally change society. It's critical that on your journey to becoming an exceptional engineer, you understand the innate responsibility needed to exercise power without causing harm. The first step is to recognize the default state of your bias caused by many societal and educational factors. After you recognize this, you'll be able to consider the often-forgotten use cases or users who can benefit or be harmed by the products you build.

The industry continues to move forward, building new use cases for artificial intelligence (AI) and machine learning at an ever-increasing speed. To stay competitive, we drive toward scale and efficacy in building a high-talent engineering and technology workforce. Yet we need to pause and consider the fact that today, some people have the ability to design the future of technology and others do not. We need to understand whether the software systems we build will eliminate the potential for entire populations to experience shared prosperity and provide equal access to technology.

Historically, companies faced with a decision between completing a strategic objective that drives market dominance and revenue and one that potentially slows momentum toward that goal have opted for speed and shareholder value. This tendency is exacerbated by the fact that many companies value individual performance and excellence, yet often fail to effectively drive accountability on product equity across all areas. Focusing on underrepresented users is a clear opportunity to promote equity. To continue to be competitive in the technology sector, we need to learn to engineer for global equity.

Today, we worry when companies design technology to scan, capture, and identify people walking down the street. We worry about privacy and how governments might use this information now and in the future. Yet most technologists do not have the requisite perspective of underrepresented groups to understand the impact of racial variance in facial recognition or to understand how applying AI can drive harmful and inaccurate results.

Currently, AI-driven facial-recognition software continues to disadvantage people of color or ethnic minorities. Our research is not comprehensive enough and does not include a wide enough range of different skin tones. We cannot expect the output to be valid if both the training data and those creating the software represent only a small subsection of people. In those cases, we should be willing to delay development in favor of trying to get more complete and accurate data, and a more comprehensive and inclusive product.

Data science itself is challenging for humans to evaluate, however. Even when we do have representation, a training set can still be biased and produce invalid results. A study completed in 2016 found that more than 117 million American adults are in a law enforcement facial recognition database.[5] Due to the disproportionate policing of Black communities and disparate outcomes in arrests, there could be racially biased error rates in utilizing such a database in facial recognition. Although the software is being developed and deployed at ever-increasing rates, the independent testing is not. To correct for this egregious misstep, we need to have the integrity to slow down and ensure that our inputs contain as little bias as possible. Google now offers statistical training within the context of AI to help ensure that datasets are not intrinsically biased.

Therefore, shifting the focus of your industry experience to include more comprehensive, multicultural, race and gender studies education is not only *your* responsibility, but also the *responsibility of your employer.* Technology companies must ensure that their employees are continually receiving professional development and that this development is comprehensive and multidisciplinary. The requirement is not that one individual take it upon themselves to learn about other cultures or other demographics alone. Change requires that each of us, individually or as leaders of teams, invest in continuous professional development that builds not just our software development and leadership skills, but also our capacity to understand the diverse experiences throughout humanity.

Making Diversity Actionable

Systemic equity and fairness are attainable if we are willing to accept that we are all accountable for the systemic discrimination we see in the technology sector. We are accountable for the failures in the system. Deferring or abstracting away personal accountability is ineffective, and depending on your role, it could be irresponsible. It is also irresponsible to fully attribute dynamics at your specific company or within your team to the larger societal issues that contribute to inequity. A favorite line among diversity proponents and detractors alike goes something like this: "We are working hard to fix (insert systemic discrimination topic), but accountability is hard. How do we combat (insert hundreds of years) of historical discrimination?" This line of inquiry is a detour to a more philosophical or academic conversation and away from focused efforts to improve work conditions or outcomes. Part of building multicultural capacity requires a more comprehensive understanding of how systems of inequality in society impact the workplace, especially in the technology sector.

5 Stephen Gaines and Sara Williams. "The Perpetual Lineup: Unregulated Police Face Recognition in America." *Center on Privacy & Technology at Georgetown Law*, October 18, 2016.

If you are an engineering manager working on hiring more people from underrepresented groups, deferring to the historical impact of discrimination in the world is a useful academic exercise. However, it is critical to move beyond the academic conversation to a focus on quantifiable and actionable steps that you can take to drive equity and fairness. For example, as a hiring software engineer manager, you're accountable for ensuring that your candidate slates are balanced. Are there women or other underrepresented groups in the pool of candidates' reviews? After you hire someone, what opportunities for growth have you provided, and is the distribution of opportunities equitable? Every technology lead or software engineering manager has the means to augment equity on their teams. It is important that we acknowledge that, although there are significant systemic challenges, we are all part of the system. It is our problem to fix.

Reject Singular Approaches

We cannot perpetuate solutions that present a single philosophy or methodology for fixing inequity in the technology sector. Our problems are complex and multifactorial. Therefore, we must disrupt singular approaches to advancing representation in the workplace, even if they are promoted by people we admire or who have institutional power.

One singular narrative held dear in the technology industry is that lack of representation in the workforce can be addressed solely by fixing the hiring pipelines. Yes, that is a fundamental step, but that is not the immediate issue we need to fix. We need to recognize systemic inequity in progression and retention while simultaneously focusing on more representative hiring and educational disparities across lines of race, gender, and socioeconomic and immigration status, for example.

In the technology industry, many people from underrepresented groups are passed over daily for opportunities and advancement. Attrition among Black+ Google employees outpaces attrition from all other groups and confounds progress on representation goals. If we want to drive change and increase representation, we need to evaluate whether we're creating an ecosystem in which all aspiring engineers and other technology professionals can thrive.

Fully understanding an entire problem space is critical to determining how to fix it. This holds true for everything from a critical data migration to the hiring of a representative workforce. For example, if you are an engineering manager who wants to hire more women, don't just focus on building a pipeline. Focus on other aspects of the hiring, retention, and progression ecosystem and how inclusive it might or might not be to women. Consider whether your recruiters are demonstrating the ability to identify strong candidates who are women as well as men. If you manage a diverse engineering team, focus on psychological safety and invest in increasing multicultural capacity on the team so that new team members feel welcome.

A common methodology today is to build for the majority use case first, leaving improvements and features that address edge cases for later. But this approach is flawed; it gives users who are already advantaged in access to technology a head start, which increases inequity. Relegating the consideration of all user groups to the point when design has been nearly completed is to lower the bar of what it means to be an excellent engineer. Instead, by building in inclusive design from the start and raising development standards for development to make tools delightful and accessible for people who struggle to access technology, we enhance the experience for *all* users.

Designing for the user who is least like you is not just wise, it's a best practice. There are pragmatic and immediate next steps that all technologists, regardless of domain, should consider when developing products that avoid disadvantaging or underrepresenting users. It begins with more comprehensive user-experience research. This research should be done with user groups that are multilingual and multicultural and that span multiple countries, socioeconomic class, abilities, and age ranges. Focus on the most difficult or least represented use case first.

Challenge Established Processes

Challenging yourself to build more equitable systems goes beyond designing more inclusive product specifications. Building equitable systems sometimes means challenging established processes that drive invalid results.

Consider a recent case evaluated for equity implications. At Google, several engineering teams worked to build a global hiring requisition system. The system supports both external hiring and internal mobility. The engineers and product managers involved did a great job of listening to the requests of what they considered to be their core user group: recruiters. The recruiters were focused on minimizing wasted time for hiring managers and applicants, and they presented the development team with use cases focused on scale and efficiency for those people. To drive efficiency, the recruiters asked the engineering team to include a feature that would highlight performance ratings—specifically lower ratings—to the hiring manager and recruiter as soon as an internal transfer expressed interest in a job.

On its face, expediting the evaluation process and helping jobseekers save time is a great goal. So where is the potential equity concern? The following equity questions were raised:

- Are developmental assessments a predictive measure of performance?
- Are the performance assessments being presented to prospective managers free of individual bias?
- Are performance assessment scores standardized across organizations?

If the answer to any of these questions is "no," presenting performance ratings could still drive inequitable, and therefore invalid, results.

When an exceptional engineer questioned whether past performance was in fact predictive of future performance, the reviewing team decided to conduct a thorough review. In the end, it was determined that candidates who had received a poor performance rating were likely to overcome the poor rating if they found a new team. In fact, they were just as likely to receive a satisfactory or exemplary performance rating as candidates who had never received a poor rating. In short, performance ratings are indicative only of how a person is performing in their given role *at the time they are being evaluated*. Ratings, although an important way to measure performance during a specific period, are not predictive of future performance and should not be used to gauge readiness for a future role or qualify an internal candidate for a different team. (They can, however, be used to evaluate whether an employee is properly or improperly slotted on their current team; therefore, they can provide an opportunity to evaluate how to better support an internal candidate moving forward.)

This analysis definitely took up significant project time, but the positive trade-off was a more equitable internal mobility process.

Values Versus Outcomes

Google has a strong track record of investing in hiring. As the previous example illustrates, we also continually evaluate our processes in order to improve equity and inclusion. More broadly, our core values are based on respect and an unwavering commitment to a diverse and inclusive workforce. Yet, year after year, we have also missed our mark on hiring a representative workforce that reflects our users around the globe. The struggle to improve our equitable outcomes persists despite the policies and programs in place to help support inclusion initiatives and promote excellence in hiring and progression. The failure point is not in the values, intentions, or investments of the company, but rather in the application of those policies at the *implementation* level.

Old habits are hard to break. The users you might be used to designing for today—the ones you are used to getting feedback from—might not be representative of all the users you need to reach. We see this play out frequently across all kinds of products, from wearables that do not work for women's bodies to video-conferencing software that does not work well for people with darker skin tones.

So, what's the way out?

1. **Take a hard look in the mirror.** At Google, we have the brand slogan, "Build For Everyone." How can we build for everyone when we do not have a representative workforce or engagement model that centralizes community feedback first? We

can't. The truth is that we have at times very publicly failed to protect our most vulnerable users from racist, antisemitic, and homophobic content.

2. **Don't build for everyone. Build *with* everyone.** We are not building for everyone yet. That work does not happen in a vacuum, and it certainly doesn't happen when the technology is still not representative of the population as a whole. That said, we can't pack up and go home. So how do we build for everyone? We build with our users. We need to engage our users across the spectrum of humanity and be intentional about putting the most vulnerable communities at the center of our design. They should not be an afterthought.
3. **Design for the user who will have the most difficulty using your product.** Building for those with additional challenges will make the product better for everyone. Another way of thinking about this is: don't trade equity for short-term velocity.
4. **Don't assume equity; measure equity throughout your systems.** Recognize that decision makers are also subject to bias and might be undereducated about the causes of inequity. You might not have the expertise to identify or measure the scope of an equity issue. Catering to a single userbase might mean disenfranchising another; these trade-offs can be difficult to spot and impossible to reverse. Partner with individuals or teams that are subject matter experts in diversity, equity, and inclusion.
5. **Change is possible.** The problems we're facing with technology today, from surveillance to disinformation to online harassment, are genuinely overwhelming. We can't solve these with the failed approaches of the past or with just the skills we already have. We need to change.

Stay Curious, Push Forward

The path to equity is long and complex. However, we can and should transition from simply building tools and services to growing our understanding of how the products we engineer impact humanity. Challenging our education, influencing our teams and managers, and doing more comprehensive user research are all ways to make progress. Although change is uncomfortable and the path to high performance can be painful, it is possible through collaboration and creativity.

Lastly, as future exceptional engineers, we should focus first on the users most impacted by bias and discrimination. Together, we can work to accelerate progress by focusing on Continuous Improvement and owning our failures. Becoming an engineer is an involved and continual process. The goal is to make changes that push humanity forward without further disenfranchising the disadvantaged. As future exceptional engineers, we have faith that we can prevent future failures in the system.

Conclusion

Developing software, and developing a software organization, is a team effort. As a software organization scales, it must respond and adequately design for its user base, which in the interconnected world of computing today involves everyone, locally and around the world. More effort must be made to make both the development teams that design software and the products that they produce reflect the values of such a diverse and encompassing set of users. And, if an engineering organization wants to scale, it cannot ignore underrepresented groups; not only do such engineers from these groups augment the organization itself, they provide unique and necessary perspectives for the design and implementation of software that is truly useful to the world at large.

TL;DRs

- Bias is the default.
- Diversity is necessary to design properly for a comprehensive user base.
- Inclusivity is critical not just to improving the hiring pipeline for underrepresented groups, but to providing a truly supportive work environment for all people.
- Product velocity must be evaluated against providing a product that is truly useful to all users. It's better to slow down than to release a product that might cause harm to some users.

O'REILLY®

Software Engineering at Google

Today, software engineers need to know not only how to program effectively but also how to develop proper engineering practices to make their codebase sustainable and healthy. This book emphasizes this difference between programming and software engineering.

How can software engineers manage a living codebase that evolves and responds to changing requirements and demands over the length of its life? Based on their experience at Google, software engineers Titus Winters and Hyrum Wright, along with technical writer Tom Manshreck, present a candid and insightful look at how some of the world's leading practitioners construct and maintain software. This book covers Google's unique engineering culture, processes, and tools and how these aspects contribute to the effectiveness of an engineering organization.

You'll explore three fundamental principles that software organizations should keep in mind when designing, architecting, writing, and maintaining code:

- How *time* affects the sustainability of software and how to make your code resilient over time
- How *scale* affects the viability of software practices within an engineering organization
- What *trade-offs* a typical engineer needs to make when evaluating design and development decisions

"While being upfront about trade-offs, this book explains the Google way of doing software engineering, which makes me most productive and happy."

—Eric Haugh
Software Engineer at Google

Titus Winters, a senior staff software engineer at Google, is the library lead for Google's C++ codebase: 250 million lines of code edited by thousands of distinct engineers per month.

Tom Manshreck is a staff technical writer within Software Engineering at Google. He's a member of the C++ Library Team, developing documentation, launching training classes, and documenting Abseil, Google's open source C++ code.

Hyrum Wright is a staff software engineer at Google, where he leads Google's automated change tooling group. Hyrum has made more individual edits to Google's codebase than any other engineer in the history of the company.

SOFTWARE ENGINEERING

US $59.99 CAN $79.99
ISBN: 978-1-492-08279-8

Twitter: @oreillymedia
facebook.com/oreilly