O'REILLY®

Software Engineering at Google

Lessons Learned from Programming Over Time

Curated by Titus Winters, Tom Manshreck & Hyrum Wright

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

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

Copyright © 2020 Google, LLC. All rights reserved.

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.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Software Engineering at Google*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-08279-8

[LSI]

Table of Contents

Foreword xvii

Preface xix

Part I. Thesis

1. What Is Software Engineering? 3

- Time and Change 6
  - Hyrum's Law 8
  - Example: Hash Ordering 9
  - Why Not Just Aim for "Nothing Changes"? 10
- Scale and Efficiency 11
  - Policies That Don't Scale 12
  - Policies That Scale Well 14
  - Example: Compiler Upgrade 14
  - Shifting Left 17
- Trade-offs and Costs 18
  - Example: Markers 19
  - Inputs to Decision Making 20
  - Example: Distributed Builds 20
  - Example: Deciding Between Time and Scale 22
  - Revisiting Decisions, Making Mistakes 22
- Software Engineering Versus Programming 23
- Conclusion 24
- TL;DRs 24

Part II. Culture

2. How to Work Well on Teams. 27
Help Me Hide My Code 27
The Genius Myth 28
Hiding Considered Harmful 30
Early Detection 31
The Bus Factor 31
Pace of Progress 32
In Short, Don't Hide 34
It's All About the Team 34
The Three Pillars of Social Interaction 34
Why Do These Pillars Matter? 35
Humility, Respect, and Trust in Practice 36
Blameless Post-Mortem Culture 39
Being Googley 41
Conclusion 42
TL;DRs 42

3. Knowledge Sharing. 43
Challenges to Learning 43
Philosophy 45
Setting the Stage: Psychological Safety 46
Mentorship 46
Psychological Safety in Large Groups 47
Growing Your Knowledge 48
Ask Questions 48
Understand Context 49
Scaling Your Questions: Ask the Community 50
Group Chats 50
Mailing Lists 50
YAQS: Question-and-Answer Platform 51
Scaling Your Knowledge: You Always Have Something to Teach 52
Office Hours 52
Tech Talks and Classes 52
Documentation 53
Code 56
Scaling Your Organization's Knowledge 56
Cultivating a Knowledge-Sharing Culture 56
Establishing Canonical Sources of Information 58

Staying in the Loop 61
Readability: Standardized Mentorship Through Code Review 62
What Is the Readability Process? 63
Why Have This Process? 64
Conclusion 66
TL;DRs 67

4. Engineering for Equity. 69
Bias Is the Default 70
Understanding the Need for Diversity 72
Building Multicultural Capacity 72
Making Diversity Actionable 74
Reject Singular Approaches 75
Challenge Established Processes 76
Values Versus Outcomes 77
Stay Curious, Push Forward 78
Conclusion 79
TL;DRs 79

5. How to Lead a Team. 81
Managers and Tech Leads (and Both) 81
The Engineering Manager 82
The Tech Lead 82
The Tech Lead Manager 82
Moving from an Individual Contributor Role to a Leadership Role 83
The Only Thing to Fear Is…Well, Everything 84
Servant Leadership 85
The Engineering Manager 86
Manager Is a Four-Letter Word 86
Today’s Engineering Manager 87
Antipatterns 88
Antipattern: Hire Pushovers 89
Antipattern: Ignore Low Performers 89
Antipattern: Ignore Human Issues 90
Antipattern: Be Everyone’s Friend 91
Antipattern: Compromise the Hiring Bar 92
Antipattern: Treat Your Team Like Children 92
Positive Patterns 93
Lose the Ego 93
Be a Zen Master 94
Be a Catalyst 96

Remove Roadblocks 96
Be a Teacher and a Mentor 97
Set Clear Goals 97
Be Honest 98
Track Happiness 99
The Unexpected Question 100
Other Tips and Tricks 101
People Are Like Plants 103
Intrinsic Versus Extrinsic Motivation 104
Conclusion 105
TL;DRs 105

6. Leading at Scale. 107
Always Be Deciding 108
The Parable of the Airplane 108
Identify the Blinders 109
Identify the Key Trade-Offs 109
Decide, Then Iterate 110
Always Be Leaving 112
Your Mission: Build a "Self-Driving" Team 112
Dividing the Problem Space 113
Always Be Scaling 116
The Cycle of Success 116
Important Versus Urgent 118
Learn to Drop Balls 119
Protecting Your Energy 120
Conclusion 122
TL;DRs 122

7. Measuring Engineering Productivity. 123
Why Should We Measure Engineering Productivity? 123
Triage: Is It Even Worth Measuring? 125
Selecting Meaningful Metrics with Goals and Signals 129
Goals 130
Signals 132
Metrics 132
Using Data to Validate Metrics 133
Taking Action and Tracking Results 137
Conclusion 137
TL;DRs 137

Part III. Processes

8. Style Guides and Rules. 141
Why Have Rules? 142
Creating the Rules 143
  Guiding Principles 143
  The Style Guide 151
Changing the Rules 154
  The Process 155
  The Style Arbiters 156
  Exceptions 156
Guidance 157
Applying the Rules 158
  Error Checkers 160
  Code Formatters 161
Conclusion 163
TL;DRs 163

9. Code Review. 165
Code Review Flow 166
How Code Review Works at Google 167
Code Review Benefits 170
  Code Correctness 171
  Comprehension of Code 172
  Code Consistency 173
  Psychological and Cultural Benefits 174
  Knowledge Sharing 175
Code Review Best Practices 176
  Be Polite and Professional 176
  Write Small Changes 177
  Write Good Change Descriptions 178
  Keep Reviewers to a Minimum 179
  Automate Where Possible 179
Types of Code Reviews 180
  Greenfield Code Reviews 180
  Behavioral Changes, Improvements, and Optimizations 181
  Bug Fixes and Rollbacks 181
  Refactorings and Large-Scale Changes 182
Conclusion 182
TL;DRs 183

10. Documentation. 185
What Qualifies as Documentation? 185
Why Is Documentation Needed? 186
Documentation Is Like Code 188
Know Your Audience 190
Types of Audiences 191
Documentation Types 192
Reference Documentation 193
Design Docs 195
Tutorials 196
Conceptual Documentation 198
Landing Pages 198
Documentation Reviews 199
Documentation Philosophy 201
WHO, WHAT, WHEN, WHERE, and WHY 201
The Beginning, Middle, and End 202
The Parameters of Good Documentation 202
Deprecating Documents 203
When Do You Need Technical Writers? 204
Conclusion 204
TL;DRs 205

11. Testing Overview. 207
Why Do We Write Tests? 208
The Story of Google Web Server 209
Testing at the Speed of Modern Development 210
Write, Run, React 212
Benefits of Testing Code 213
Designing a Test Suite 214
Test Size 215
Test Scope 219
The Beyoncé Rule 221
A Note on Code Coverage 222
Testing at Google Scale 223
The Pitfalls of a Large Test Suite 224
History of Testing at Google 225
Orientation Classes 226
Test Certified 227
Testing on the Toilet 227
Testing Culture Today 228

The Limits of Automated Testing 229
Conclusion 230
TL;DRs 230

12. Unit Testing. 231
The Importance of Maintainability 232
Preventing Brittle Tests 233
  Strive for Unchanging Tests 233
  Test via Public APIs 234
  Test State, Not Interactions 238
Writing Clear Tests 239
  Make Your Tests Complete and Concise 240
  Test Behaviors, Not Methods 241
  Don’t Put Logic in Tests 246
  Write Clear Failure Messages 247
Tests and Code Sharing: DAMP, Not DRY 248
  Shared Values 251
  Shared Setup 253
  Shared Helpers and Validation 254
  Defining Test Infrastructure 255
Conclusion 256
TL;DRs 256

13. Test Doubles. 257
The Impact of Test Doubles on Software Development 258
Test Doubles at Google 258
Basic Concepts 259
  An Example Test Double 259
  Seams 260
  Mocking Frameworks 261
Techniques for Using Test Doubles 262
  Faking 263
  Stubbing 263
  Interaction Testing 264
Real Implementations 264
  Prefer Realism Over Isolation 265
  How to Decide When to Use a Real Implementation 266
Faking 269
  Why Are Fakes Important? 270
  When Should Fakes Be Written? 270
  The Fidelity of Fakes 271

Fakes Should Be Tested 272
What to Do If a Fake Is Not Available 272
Stubbing 272
The Dangers of Overusing Stubbing 273
When Is Stubbing Appropriate? 275
Interaction Testing 275
Prefer State Testing Over Interaction Testing 275
When Is Interaction Testing Appropriate? 277
Best Practices for Interaction Testing 277
Conclusion 280
TL;DRs 280

14. Larger Testing. 281
What Are Larger Tests? 281
Fidelity 282
Common Gaps in Unit Tests 283
Why Not Have Larger Tests? 285
Larger Tests at Google 286
Larger Tests and Time 286
Larger Tests at Google Scale 288
Structure of a Large Test 289
The System Under Test 290
Test Data 294
Verification 295
Types of Larger Tests 296
Functional Testing of One or More Interacting Binaries 297
Browser and Device Testing 297
Performance, Load, and Stress testing 297
Deployment Configuration Testing 298
Exploratory Testing 298
A/B Diff Regression Testing 299
UAT 301
Probers and Canary Analysis 301
Disaster Recovery and Chaos Engineering 302
User Evaluation 303
Large Tests and the Developer Workflow 304
Authoring Large Tests 305
Running Large Tests 305
Owning Large Tests 308
Conclusion 309
TL;DRs 309

15. Deprecation. 311
Why Deprecate? 312
Why Is Deprecation So Hard? 313
  Deprecation During Design 315
Types of Deprecation 316
  Advisory Deprecation 316
  Compulsory Deprecation 317
  Deprecation Warnings 318
Managing the Deprecation Process 319
  Process Owners 320
  Milestones 320
  Deprecation Tooling 321
Conclusion 322
TL;DRs 323

Part IV. Tools

16. Version Control and Branch Management. 327
What Is Version Control? 327
  Why Is Version Control Important? 329
  Centralized VCS Versus Distributed VCS 331
  Source of Truth 334
  Version Control Versus Dependency Management 336
Branch Management 336
  Work in Progress Is Akin to a Branch 336
  Dev Branches 337
  Release Branches 339
Version Control at Google 340
  One Version 340
  Scenario: Multiple Available Versions 341
  The “One-Version” Rule 342
  (Nearly) No Long-Lived Branches 343
  What About Release Branches? 344
Monorepos 345
Future of Version Control 346
Conclusion 348
TL;DRs 349

17. Code Search. 351
The Code Search UI 352
How Do Googlers Use Code Search? 353
Where? 353
What? 354
How? 354
Why? 354
Who and When? 355
Why a Separate Web Tool? 355
Scale 355
Zero Setup Global Code View 356
Specialization 356
Integration with Other Developer Tools 356
API Exposure 359
Impact of Scale on Design 359
Search Query Latency 359
Index Latency 360
Google's Implementation 361
Search Index 361
Ranking 363
Selected Trade-Offs 366
Completeness: Repository at Head 366
Completeness: All Versus Most-Relevant Results 366
Completeness: Head Versus Branches Versus All History Versus Workspaces 367
Expressiveness: Token Versus Substring Versus Regex 368
Conclusion 369
TL;DRs 370

18. Build Systems and Build Philosophy. 371
Purpose of a Build System 371
What Happens Without a Build System? 372
But All I Need Is a Compiler! 373
Shell Scripts to the Rescue? 373
Modern Build Systems 375
It's All About Dependencies 375
Task-Based Build Systems 376
Artifact-Based Build Systems 380
Distributed Builds 386
Time, Scale, Trade-Offs 390

Dealing with Modules and Dependencies 390
Using Fine-Grained Modules and the 1:1:1 Rule 391
Minimizing Module Visibility 392
Managing Dependencies 392
Conclusion 397
TL;DRs 397

19. Critique: Google's Code Review Tool. 399
Code Review Tooling Principles 399
Code Review Flow 400
Notifications 402
Stage 1: Create a Change 402
Diffing 403
Analysis Results 404
Tight Tool Integration 406
Stage 2: Request Review 406
Stages 3 and 4: Understanding and Commenting on a Change 408
Commenting 408
Understanding the State of a Change 410
Stage 5: Change Approvals (Scoring a Change) 412
Stage 6: Commiting a Change 413
After Commit: Tracking History 414
Conclusion 415
TL;DRs 416

20. Static Analysis. 417
Characteristics of Effective Static Analysis 418
Scalability 418
Usability 418
Key Lessons in Making Static Analysis Work 419
Focus on Developer Happiness 419
Make Static Analysis a Part of the Core Developer Workflow 420
Empower Users to Contribute 420
Tricorder: Google's Static Analysis Platform 421
Integrated Tools 422
Integrated Feedback Channels 423
Suggested Fixes 424
Per-Project Customization 424
Presubmits 425
Compiler Integration 426
Analysis While Editing and Browsing Code 427

Conclusion 428
TL;DRs 428

21. Dependency Management. 429
Why Is Dependency Management So Difficult? 431
Conflicting Requirements and Diamond Dependencies 431
Importing Dependencies 433
Compatibility Promises 433
Considerations When Importing 436
How Google Handles Importing Dependencies 437
Dependency Management, In Theory 439
Nothing Changes (aka The Static Dependency Model) 439
Semantic Versioning 440
Bundled Distribution Models 441
Live at Head 442
The Limitations of SemVer 443
SemVer Might Overconstrain 444
SemVer Might Overpromise 445
Motivations 446
Minimum Version Selection 447
So, Does SemVer Work? 448
Dependency Management with Infinite Resources 449
Exporting Dependencies 452
Conclusion 456
TL;DRs 456

22. Large-Scale Changes. 459
What Is a Large-Scale Change? 460
Who Deals with LSCs? 461
Barriers to Atomic Changes 463
Technical Limitations 463
Merge Conflicts 463
No Haunted Graveyards 464
Heterogeneity 464
Testing 465
Code Review 467
LSC Infrastructure 468
Policies and Culture 469
Codebase Insight 470
Change Management 470
Testing 471

Language Support 471
The LSC Process 472
Authorization 473
Change Creation 473
Sharding and Submitting 474
Cleanup 477
Conclusion 477
TL;DRs 478

23. Continuous Integration. 479
CI Concepts 481
Fast Feedback Loops 481
Automation 483
Continuous Testing 485
CI Challenges 490
Hermetic Testing 491
CI at Google 493
CI Case Study: Google Takeout 496
But I Can't Afford CI 503
Conclusion 503
TL;DRs 503

24. Continuous Delivery. 505
Idioms of Continuous Delivery at Google 506
Velocity Is a Team Sport: How to Break Up a Deployment into Manageable Pieces 507
Evaluating Changes in Isolation: Flag-Guarding Features 508
Striving for Agility: Setting Up a Release Train 509
No Binary Is Perfect 509
Meet Your Release Deadline 510
Quality and User-Focus: Ship Only What Gets Used 511
Shifting Left: Making Data-Driven Decisions Earlier 512
Changing Team Culture: Building Discipline into Deployment 513
Conclusion 514
TL;DRs 514

25. Compute as a Service. 517
Taming the Compute Environment 518
Automation of Toil 518
Containerization and Multitenancy 520
Summary 523

Writing Software for Managed Compute 523
Architecting for Failure 523
Batch Versus Serving 525
Managing State 527
Connecting to a Service 528
One-Off Code 529
CaaS Over Time and Scale 530
Containers as an Abstraction 530
One Service to Rule Them All 533
Submitted Configuration 535
Choosing a Compute Service 535
Centralization Versus Customization 537
Level of Abstraction: Serverless 539
Public Versus Private 543
Conclusion 544
TL;DRs 545

Part V. Conclusion

Afterword. 549

Index. 551

Foreword

I have always been endlessly fascinated with the details of how Google does things. I have grilled my Googler friends for information about the way things really work inside of the company. How do they manage such a massive, monolithic code repository without falling over? How do tens of thousands of engineers successfully collaborate on thousands of projects? How do they maintain the quality of their systems?

Working with former Googlers has only increased my curiosity. If you've ever worked with a former Google engineer (or "Xoogler," as they're sometimes called), you've no doubt heard the phrase "at Google we…" Coming out of Google into other companies seems to be a shocking experience, at least from the engineering side of things. As far as this outsider can tell, the systems and processes for writing code at Google must be among the best in the world, given both the scale of the company and how often people sing their praises.

In *Software Engineering at Google*, a set of Googlers (and some Xooglers) gives us a lengthy blueprint for many of the practices, tools, and even cultural elements that underlie software engineering at Google. It's easy to overfocus on the amazing tools that Google has built to support writing code, and this book provides a lot of details about those tools. But it also goes beyond simply describing the tooling to give us the philosophy and processes that the teams at Google follow. These can be adapted to fit a variety of circumstances, whether or not you have the scale and tooling. To my delight, there are several chapters that go deep on various aspects of automated testing, a topic that continues to meet with too much resistance in our industry.

The great thing about tech is that there is never only one way to do something. Instead, there is a series of trade-offs we all must make depending on the circumstances of our team and situation. What can we cheaply take from open source? What can our team build? What makes sense to support for our scale? When I was grilling my Googler friends, I wanted to hear about the world at the extreme end of scale: resource rich, in both talent and money, with high demands on the software being built. This anecdotal information gave me ideas on some options that I might not otherwise have considered.

With this book, we've written down those options for everyone to read. Of course, Google is a unique company, and it would be foolish to assume that the right way to run your software engineering organization is to precisely copy their formula. Applied practically, this book will give you ideas on how things could be done, and a lot of information that you can use to bolster your arguments for adopting best practices like testing, knowledge sharing, and building collaborative teams.

You may never need to build Google yourself, and you may not even want to reach for the same techniques they apply in your organization. But if you aren't familiar with the practices Google has developed, you're missing a perspective on software engineering that comes from tens of thousands of engineers working collaboratively on software over the course of more than two decades. That knowledge is far too valuable to ignore.

— *Camille Fournier*
Author, The Manager's Path