

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

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

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: Committing 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

Style Guides and Rules

Written by Shaindel Schwartz
Edited by Tom Manshreck

Most engineering organizations have rules governing their codebases—rules about where source files are stored, rules about the formatting of the code, rules about naming and patterns and exceptions and threads. Most software engineers are working within the bounds of a set of policies that control how they operate. At Google, to manage our codebase, we maintain a set of style guides that define our rules.

Rules are laws. They are not just suggestions or recommendations, but strict, mandatory laws. As such, they are universally enforceable—rules may not be disregarded except as approved on a need-to-use basis. In contrast to rules, guidance provides recommendations and best practices. These bits are good to follow, even highly advisable to follow, but unlike rules, they usually have some room for variance.

We collect the rules that we define, the do's and don'ts of writing code that must be followed, in our programming style guides, which are treated as canon. “Style” might be a bit of a misnomer here, implying a collection limited to formatting practices. Our style guides are more than that; they are the full set of conventions that govern our code. That's not to say that our style guides are strictly prescriptive; style guide rules may call for judgement, such as the rule to use names that are “**as descriptive as possible, within reason.**” Rather, our style guides serve as the definitive source for the rules to which our engineers are held accountable.

We maintain separate style guides for each of the programming languages used at Google.¹ At a high level, all of the guides have similar goals, aiming to steer code

¹ Many of our style guides have external versions, which you can find at <https://google.github.io/styleguide>. We cite numerous examples from these guides within this chapter.

development with an eye to sustainability. At the same time, there is a lot of variation among them in scope, length, and content. Programming languages have different strengths, different features, different priorities, and different historical paths to adoption within Google’s ever-evolving repositories of code. It is far more practical, therefore, to independently tailor each language’s guidelines. Some of our style guides are concise, focusing on a few overarching principles like naming and formatting, as demonstrated in our Dart, R, and Shell guides. Other style guides include far more detail, delving into specific language features and stretching into far lengthier documents—notably, our C++, Python, and Java guides. Some style guides put a premium on typical non-Google use of the language—our Go style guide is very short, adding just a few rules to a summary directive to adhere to the practices outlined in the **externally recognized conventions**. Others include rules that fundamentally differ from external norms; our C++ rules disallow use of exceptions, a language feature widely used outside of Google code.

The wide variance among even our own style guides makes it difficult to pin down the precise description of what a style guide should cover. The decisions guiding the development of Google’s style guides stem from the need to keep our codebase sustainable. Other organizations’ codebases will inherently have different requirements for sustainability that necessitate a different set of tailored rules. This chapter discusses the principles and processes that steer the development of our rules and guidance, pulling examples primarily from Google’s C++, Python, and Java style guides.

Why Have Rules?

So why do we have rules? The goal of having rules in place is to encourage “good” behavior and discourage “bad” behavior. The interpretation of “good” and “bad” varies by organization, depending on what the organization cares about. Such designations are not universal preferences; good versus bad is subjective, and tailored to needs. For some organizations, “good” might promote usage patterns that support a small memory footprint or prioritize potential runtime optimizations. In other organizations, “good” might promote choices that exercise new language features. Sometimes, an organization cares most deeply about consistency, so that anything inconsistent with existing patterns is “bad.” We must first recognize what a given organization values; we use rules and guidance to encourage and discourage behavior accordingly.

As an organization grows, the established rules and guidelines shape the common vocabulary of coding. A common vocabulary allows engineers to concentrate on what their code needs to say rather than how they’re saying it. By shaping this vocabulary, engineers will tend to do the “good” things by default, even subconsciously. Rules thus give us broad leverage to nudge common development patterns in desired directions.

Creating the Rules

When defining a set of rules, the key question is not, “What rules should we have?” The question to ask is, “What goal are we trying to advance?” When we focus on the goal that the rules will be serving, identifying which rules support this goal makes it easier to distill the set of useful rules. At Google, where the style guide serves as law for coding practices, we do not ask, “What goes into the style guide?” but rather, “Why does something go into the style guide?” What does our organization gain by having a set of rules to regulate writing code?

Guiding Principles

Let’s put things in context: Google’s engineering organization is composed of more than 30,000 engineers. That engineering population exhibits a wild variance in skill and background. About 60,000 submissions are made each day to a codebase of more than two billion lines of code that will likely exist for decades. We’re optimizing for a different set of values than most other organizations need, but to some degree, these concerns are ubiquitous—we need to sustain an engineering environment that is resilient to both scale and time.

In this context, the goal of our rules is to manage the complexity of our development environment, keeping the codebase manageable while still allowing engineers to work productively. We are making a trade-off here: the large body of rules that helps us toward this goal does mean we are restricting choice. We lose some flexibility and we might even offend some people, but the gains of consistency and reduced conflict furnished by an authoritative standard win out.

Given this view, we recognize a number of overarching principles that guide the development of our rules, which must:

- Pull their weight
- Optimize for the reader
- Be consistent
- Avoid error-prone and surprising constructs
- Concede to practicalities when necessary

Rules must pull their weight

Not everything should go into a style guide. There is a nonzero cost in asking all of the engineers in an organization to learn and adapt to any new rule that is set. With too many rules,² not only will it become harder for engineers to remember all relevant rules as they write their code, but it also becomes harder for new engineers to learn their way. More rules also make it more challenging and more expensive to maintain the rule set.

To this end, we deliberately chose not to include rules expected to be self-evident. Google's style guide is not intended to be interpreted in a lawyerly fashion; just because something isn't explicitly outlawed does not imply that it is legal. For example, the C++ style guide has no rule against the use of `goto`. C++ programmers already tend to avoid it, so including an explicit rule forbidding it would introduce unnecessary overhead. If just one or two engineers are getting something wrong, adding to everyone's mental load by creating new rules doesn't scale.

Optimize for the reader

Another principle of our rules is to optimize for the reader of the code rather than the author. Given the passage of time, our code will be read far more frequently than it is written. We'd rather the code be tedious to type than difficult to read. In our Python style guide, when discussing conditional expressions, we recognize that they are shorter than `if` statements and therefore more convenient for code authors. However, because they tend to be more difficult for readers to understand than the more verbose `if` statements, **we restrict their usage**. We value "simple to read" over "simple to write." We're making a trade-off here: it can cost more upfront when engineers must repeatedly type potentially longer, descriptive names for variables and types. We choose to pay this cost for the readability it provides for all future readers.

As part of this prioritization, we also require that engineers leave explicit evidence of intended behavior in their code. We want readers to clearly understand what the code is doing as they read it. For example, our Java, JavaScript, and C++ style guides mandate use of the `override` annotation or keyword whenever a method overrides a superclass method. Without the explicit in-place evidence of design, readers can likely figure out this intent, though it would take a bit more digging on the part of each reader working through the code.

2 Tooling matters here. The measure for "too many" is not the raw number of rules in play, but how many an engineer needs to remember. For example, in the bad-old-days pre-clang-format, we needed to remember a ton of formatting rules. Those rules haven't gone away, but with our current tooling, the cost of adherence has fallen dramatically. We've reached a point at which somebody could add an arbitrary number of formatting rules and nobody would care, because the tool just does it for you.

Evidence of intended behavior becomes even more important when it might be surprising. In C++, it is sometimes difficult to track the ownership of a pointer just by reading a snippet of code. If a pointer is passed to a function, without being familiar with the behavior of the function, we can't be sure what to expect. Does the caller still own the pointer? Did the function take ownership? Can I continue using the pointer after the function returns or might it have been deleted? To avoid this problem, our **C++ style guide prefers the use of `std::unique_ptr`** when ownership transfer is intended. `unique_ptr` is a construct that manages pointer ownership, ensuring that only one copy of the pointer ever exists. When a function takes a `unique_ptr` as an argument and intends to take ownership of the pointer, callers must explicitly invoke move semantics:

```
// Function that takes a Foo* and may or may not assume ownership of  
// the passed pointer.  
void TakeFoo(Foo* arg);  
  
// Calls to the function don't tell the reader anything about what to  
// expect with regard to ownership after the function returns.  
Foo* my_foo(NewFoo());  
TakeFoo(my_foo);
```

Compare this to the following:

```
// Function that takes a std::unique_ptr<Foo>.  
void TakeFoo(std::unique_ptr<Foo> arg);  
  
// Any call to the function explicitly shows that ownership is  
// yielded and the unique_ptr cannot be used after the function  
// returns.  
std::unique_ptr<Foo> my_foo(FooFactory());  
TakeFoo(std::move(my_foo));
```

Given the style guide rule, we guarantee that all call sites will include clear evidence of ownership transfer whenever it applies. With this signal in place, readers of the code don't need to understand the behavior of every function call. We provide enough information in the API to reason about its interactions. This clear documentation of behavior at the call sites ensures that code snippets remain readable and understandable. We aim for local reasoning, where the goal is clear understanding of what's happening at the call site without needing to find and reference other code, including the function's implementation.

Most style guide rules covering comments are also designed to support this goal of in-place evidence for readers. Documentation comments (the block comments prepended to a given file, class, or function) describe the design or intent of the code that follows. Implementation comments (the comments interspersed throughout the code itself) justify or highlight non-obvious choices, explain tricky bits, and underscore important parts of the code. We have style guide rules covering both types of

comments, requiring engineers to provide the explanations another engineer might be looking for when reading through the code.

Be consistent

Our view on consistency within our codebase is similar to the philosophy we apply to our Google offices. With a large, distributed engineering population, teams are frequently split among offices, and Googlers often find themselves traveling to other sites. Although each office maintains its unique personality, embracing local flavor and style, for anything necessary to get work done, things are deliberately kept the same. A visiting Googler's badge will work with all local badge readers; any Google devices will always get WiFi; the video conferencing setup in any conference room will have the same interface. A Googler doesn't need to spend time learning how to get this all set up; they know that it will be the same no matter where they are. It's easy to move between offices and still get work done.

That's what we strive for with our source code. Consistency is what enables any engineer to jump into an unfamiliar part of the codebase and get to work fairly quickly. A local project can have its unique personality, but its tools are the same, its techniques are the same, its libraries are the same, and it all Just Works.

Advantages of consistency

Even though it might feel restrictive for an office to be disallowed from customizing a badge reader or video conferencing interface, the consistency benefits far outweigh the creative freedom we lose. It's the same with code: being consistent may feel constraining at times, but it means more engineers get more work done with less effort.³

- When a codebase is internally consistent in its style and norms, engineers writing code and others reading it can focus on what's getting done rather than how it is presented. To a large degree, this consistency allows for expert chunking.⁴ When we solve our problems with the same interfaces and format the code in a consistent way, it's easier for experts to glance at some code, zero in on what's important, and understand what it's doing. It also makes it easier to modularize code and spot duplication. For these reasons, we focus a lot of attention on consistent naming conventions, consistent use of common patterns, and consistent formatting and structure. There are also many rules that put forth a decision on a seemingly small issue solely to guarantee that things are done in only one way. For

³ Credit to H. Wright for the real-world comparison, made at the point of having visited around 15 different Google offices.

⁴ "Chunking" is a cognitive process that groups pieces of information together into meaningful "chunks" rather than keeping note of them individually. Expert chess players, for example, think about configurations of pieces rather than the positions of the individuals.

example, take the choice of the number of spaces to use for indentation or the limit set on line length.⁵ It's the consistency of having one answer rather than the answer itself that is the valuable part here.

- Consistency enables scaling. Tooling is key for an organization to scale, and consistent code makes it easier to build tools that can understand, edit, and generate code. The full benefits of the tools that depend on uniformity can't be applied if everyone has little pockets of code that differ—if a tool can keep source files updated by adding missing imports or removing unused includes, if different projects are choosing different sorting strategies for their import lists, the tool might not be able to work everywhere. When everyone is using the same components and when everyone's code follows the same rules for structure and organization, we can invest in tooling that works everywhere, building in automation for many of our maintenance tasks. If each team needed to separately invest in a bespoke version of the same tool, tailored for their unique environment, we would lose that advantage.
- Consistency helps when scaling the human part of an organization, too. As an organization grows, the number of engineers working on the codebase increases. Keeping the code that everyone is working on as consistent as possible enables better mobility across projects, minimizing the ramp-up time for an engineer switching teams and building in the ability for the organization to flex and adapt as headcount needs fluctuate. A growing organization also means that people in other roles interact with the code—SREs, library engineers, and code janitors, for example. At Google, these roles often span multiple projects, which means engineers unfamiliar with a given team's project might jump in to work on that project's code. A consistent experience across the codebase makes this efficient.
- Consistency also ensures resilience to time. As time passes, engineers leave projects, new people join, ownership shifts, and projects merge or split. Striving for a consistent codebase ensures that these transitions are low cost and allows us nearly unconstrained fluidity for both the code and the engineers working on it, simplifying the processes necessary for long-term maintenance.

⁵ See [4.2 Block indentation: +2 spaces, Spaces vs. Tabs](#), [4.4 Column limit:100](#) and [Line Length](#).

At Scale

A few years ago, our C++ style guide promised to almost never change style guide rules that would make old code inconsistent: “In some cases, there might be good arguments for changing certain style rules, but we nonetheless keep things as they are in order to preserve consistency.”

When the codebase was smaller and there were fewer old, dusty corners, that made sense.

When the codebase grew bigger and older, that stopped being a thing to prioritize. This was (for the arbiters behind our C++ style guide, at least) a conscious change: when striking this bit, we were explicitly stating that the C++ codebase would never again be completely consistent, nor were we even aiming for that.

It would simply be too much of a burden to not only update the rules to current best practices, but to also require that we apply those rules to everything that’s ever been written. Our Large Scale Change tooling and processes allow us to update almost all of our code to follow nearly every new pattern or syntax so that most old code exhibits the most recent approved style (see [Chapter 22](#)). Such mechanisms aren’t perfect, however; when the codebase gets as large as it is, we can’t be sure every bit of old code can conform to the new best practices. Requiring perfect consistency has reached the point where there’s too much cost for the value.

Setting the standard. When we advocate for consistency, we tend to focus on internal consistency. Sometimes, local conventions spring up before global ones are adopted, and it isn’t reasonable to adjust everything to match. In that case, we advocate a hierarchy of consistency: “Be consistent” starts locally, where the norms within a given file precede those of a given team, which precede those of the larger project, which precede those of the overall codebase. In fact, the style guides contain a number of rules that explicitly defer to local conventions,⁶ valuing this local consistency over a scientific technical choice.

However, it is not always enough for an organization to create and stick to a set of internal conventions. Sometimes, the standards adopted by the external community should be taken into account.

⁶ Use of `const`, for example.

Counting Spaces

The Python style guide at Google initially mandated two-space indents for all of our Python code. The standard Python style guide, used by the external Python community, uses four-space indents. Most of our early Python development was in direct support of our C++ projects, not for actual Python applications. We therefore chose to use two-space indentation to be consistent with our C++ code, which was already formatted in that manner. As time went by, we saw that this rationale didn't really hold up. Engineers who write Python code read and write other Python code much more often than they read and write C++ code. We were costing our engineers extra effort every time they needed to look something up or reference external code snippets. We were also going through a lot of pain each time we tried to export pieces of our code into open source, spending time reconciling the differences between our internal code and the external world we wanted to join.

When the time came for **Starlark** (a Python-based language designed at Google to serve as the build description language) to have its own style guide, we chose to change to using four-space indents to be consistent with the outside world.⁷

If conventions already exist, it is usually a good idea for an organization to be consistent with the outside world. For small, self-contained, and short-lived efforts, it likely won't make a difference; internal consistency matters more than anything happening outside the project's limited scope. Once the passage of time and potential scaling become factors, the likelihood of your code interacting with outside projects or even ending up in the outside world increase. Looking long-term, adhering to the widely accepted standard will likely pay off.

Avoid error-prone and surprising constructs

Our style guides restrict the use of some of the more surprising, unusual, or tricky constructs in the languages that we use. Complex features often have subtle pitfalls not obvious at first glance. Using these features without thoroughly understanding their complexities makes it easy to misuse them and introduce bugs. Even if a construct is well understood by a project's engineers, future project members and maintainers are not guaranteed to have the same understanding.

This reasoning is behind our Python style guide ruling to avoid using **power features** such as reflection. The reflective Python functions `hasattr()` and `getattr()` allow a user to access attributes of objects using strings:

⁷ Style formatting for BUILD files implemented with Starlark is applied by buildifier. See <https://github.com/bazelbuild/buildtools>.

```
if hasattr(my_object, 'foo'):
    some_var = getattr(my_object, 'foo')
```

Now, with that example, everything might seem fine. But consider this:

some_file.py:

```
A_CONSTANT = [
    'foo',
    'bar',
    'baz',
]
```

other_file.py:

```
values = []
for field in some_file.A_CONSTANT:
    values.append(getattr(my_object, field))
```

When searching through code, how do you know that the fields `foo`, `bar`, and `baz` are being accessed here? There's no clear evidence left for the reader. You don't easily see and therefore can't easily validate which strings are used to access attributes of your object. What if, instead of reading those values from `A_CONSTANT`, we read them from a Remote Procedure Call (RPC) request message or from a data store? Such obfuscated code could cause a major security flaw, one that would be very difficult to notice, simply by validating the message incorrectly. It's also difficult to test and verify such code.

Python's dynamic nature allows such behavior, and in very limited circumstances, using `hasattr()` and `getattr()` is valid. In most cases, however, they just cause obfuscation and introduce bugs.

Although these advanced language features might perfectly solve a problem for an expert who knows how to leverage them, power features are often more difficult to understand and are not very widely used. We need all of our engineers able to operate in the codebase, not just the experts. It's not just support for the novice software engineer, but it's also a better environment for SREs—if an SRE is debugging a production outage, they will jump into any bit of suspect code, even code written in a language in which they are not fluent. We place higher value on simplified, straightforward code that is easier to understand and maintain.

Concede to practicalities

In the words of Ralph Waldo Emerson: “A foolish consistency is the hobgoblin of little minds.” In our quest for a consistent, simplified codebase, we do not want to blindly ignore all else. We know that some of the rules in our style guides will encounter cases that warrant exceptions, and that's OK. When necessary, we permit concessions to optimizations and practicalities that might otherwise conflict with our rules.

Performance matters. Sometimes, even if it means sacrificing consistency or readability, it just makes sense to accommodate performance optimizations. For example, although our C++ style guide prohibits use of exceptions, it includes a rule that allows the use of `noexcept`, an exception-related language specifier that can trigger compiler optimizations.

Interoperability also matters. Code that is designed to work with specific non-Google pieces might do better if tailored for its target. For example, our C++ style guide includes an exception to the general CamelCase naming guideline that permits use of the standard library's `snake_case` style for entities that mimic standard library features.⁸ The C++ style guide also allows `exemptions for Windows programming`, where compatibility with platform features requires multiple inheritance, something explicitly forbidden for all other C++ code. Both our Java and JavaScript style guides explicitly state that generated code, which frequently interfaces with or depends on components outside of a project's ownership, is out of scope for the guide's rules.⁹ Consistency is vital; adaptation is key.

The Style Guide

So, what does go into a language style guide? There are roughly three categories into which all style guide rules fall:

- Rules to avoid dangers
- Rules to enforce best practices
- Rules to ensure consistency

Avoiding danger

First and foremost, our style guides include rules about language features that either must or must not be done for technical reasons. We have rules about how to use static members and variables; rules about using lambda expressions; rules about handling exceptions; rules about building for threading, access control, and class inheritance. We cover which language features to use and which constructs to avoid. We call out standard vocabulary types that may be used and for what purposes. We specifically include rulings on the hard-to-use and the hard-to-use-correctly—some language features have nuanced usage patterns that might not be intuitive or easy to apply

⁸ See `Exceptions to Naming Rules`. As an example, our open sourced Abseil libraries use `snake_case` naming for types intended to be replacements for standard types. See the types defined in <https://github.com/abseil/abseil-cpp/blob/master/absl/utility/utility.h>. These are C++11 implementation of C++14 standard types and therefore use the standard's favored `snake_case` style instead of Google's preferred CamelCase form.

⁹ See `Generated code: mostly exempt`.

properly, causing subtle bugs to creep in. For each ruling in the guide, we aim to include the pros and cons that were weighed with an explanation of the decision that was reached. Most of these decisions are based on the need for resilience to time, supporting and encouraging maintainable language usage.

Enforcing best practices

Our style guides also include rules enforcing some best practices of writing source code. These rules help keep the codebase healthy and maintainable. For example, we specify where and how code authors must include comments.¹⁰ Our rules for comments cover general conventions for commenting and extend to include specific cases that must include in-code documentation—cases in which intent is not always obvious, such as fall-through in switch statements, empty exception catch blocks, and template metaprogramming. We also have rules detailing the structuring of source files, outlining the organization of expected content. We have rules about naming: naming of packages, of classes, of functions, of variables. All of these rules are intended to guide engineers to practices that support healthier, more sustainable code.

Some of the best practices enforced by our style guides are designed to make source code more readable. Many formatting rules fall under this category. Our style guides specify when and how to use vertical and horizontal whitespace in order to improve readability. They also cover line length limits and brace alignment. For some languages, we cover formatting requirements by deferring to autoformatting tools—gofmt for Go, dartfmt for Dart. Itemizing a detailed list of formatting requirements or naming a tool that must be applied, the goal is the same: we have a consistent set of formatting rules designed to improve readability that we apply to all of our code.

Our style guides also include limitations on new and not-yet-well-understood language features. The goal is to preemptively install safety fences around a feature's potential pitfalls while we all go through the learning process. At the same time, before everyone takes off running, limiting use gives us a chance to watch the usage patterns that develop and extract best practices from the examples we observe. For these new features, at the outset, we are sometimes not sure of the proper guidance to give. As adoption spreads, engineers wanting to use the new features in different ways discuss their examples with the style guide owners, asking for allowances to permit additional use cases beyond those covered by the initial restrictions. Watching the waiver requests that come in, we get a sense of how the feature is getting used and eventually collect enough examples to generalize good practice from bad. After we

¹⁰ See <https://google.github.io/styleguide/cppguide.html#Comments>, <http://google.github.io/styleguide/pyguide#38-comments-and-docstrings>, and <https://google.github.io/styleguide/javaguide.html#s7-javadoc>, where multiple languages define general comment rules.

have that information, we can circle back to the restrictive ruling and amend it to allow wider use.

Case Study: Introducing `std::unique_ptr`

When C++11 introduced `std::unique_ptr`, a smart pointer type that expresses exclusive ownership of a dynamically allocated object and deletes the object when the `unique_ptr` goes out of scope, our style guide initially disallowed usage. The behavior of the `unique_ptr` was unfamiliar to most engineers, and the related move semantics that the language introduced were very new and, to most engineers, very confusing. Preventing the introduction of `std::unique_ptr` in the codebase seemed the safer choice. We updated our tooling to catch references to the disallowed type and kept our existing guidance recommending other types of existing smart pointers.

Time passed. Engineers had a chance to adjust to the implications of move semantics and we became increasingly convinced that using `std::unique_ptr` was directly in line with the goals of our style guidance. The information regarding object ownership that a `std::unique_ptr` facilitates at a function call site makes it far easier for a reader to understand that code. The added complexity of introducing this new type, and the novel move semantics that come with it, was still a strong concern, but the significant improvement in the long-term overall state of the codebase made the adoption of `std::unique_ptr` a worthwhile trade-off.

Building in consistency

Our style guides also contain rules that cover a lot of the smaller stuff. For these rules, we make and document a decision primarily to make and document a decision. Many rules in this category don't have significant technical impact. Things like naming conventions, indentation spacing, import ordering: there is usually no clear, measurable, technical benefit for one form over another, which might be why the technical community tends to keep debating them.¹¹ By choosing one, we've dropped out of the endless debate cycle and can just move on. Our engineers no longer spend time discussing two spaces versus four. The important bit for this category of rules is not *what* we've chosen for a given rule so much as the fact that we *have* chosen.

¹¹ Such discussions are really just [bikeshedding](#), an illustration of [Parkinson's law of triviality](#).

About the Authors

Titus Winters is a Senior Staff Software Engineer at Google, where he has worked since 2010. Today, he is the chair of the global subcommittee for the design of the C++ standard library. At Google, he is the library lead for Google's C++ codebase: 250 million lines of code that will be edited by 12,000 distinct engineers in a month. For the last seven years, Titus and his teams have been organizing, maintaining, and evolving the foundational components of Google's C++ codebase using modern automation and tooling. Along the way, he has started several Google projects that are believed to be in the top-10 largest refactorings in human history. As a direct result of helping to build out refactoring tooling and automation, Titus has encountered first-hand a huge swath of the shortcuts that engineers and programmers may take to “just get something working.” That unique scale and perspective has informed all of his thinking on the care and feeding of software systems.

Tom Manshreck is a Staff Technical Writer within Software Engineering at Google since 2005, responsible for developing and maintaining many of Google's core programming guides in infrastructure and language. Since 2011, he has been a member of Google's C++ Library Team, developing Google's C++ documentation set, launching (with Titus Winters) Google's C++ training classes, and documenting Abseil, Google's open source C++ code. Tom holds a BS in Political Science and a BS in History from the Massachusetts Institute of Technology. Before Google, Tom worked as a Managing Editor at Pearson/Prentice Hall and various startups.

Hyrum Wright is a Staff Software Engineer at Google, where he has worked since 2012, mainly in the areas of large-scale maintenance of Google's C++ codebase. Hyrum has made more individual edits to Google's codebase than any other engineer in the history of the company, and leads Google's automated change tooling group. Hyrum received a PhD in Software Engineering from the University of Texas at Austin and also holds an MS from the University of Texas and a BS from Brigham Young University, and is an occasional visiting faculty member at Carnegie Mellon University. He is an active speaker at conferences and contributor to the academic literature on software maintenance and evolution.

The O'Reilly logo is displayed in white, bold, sans-serif capital letters. The background of the entire advertisement is a vibrant red-to-orange gradient, overlaid with several large, semi-transparent, overlapping circles in varying shades of red and orange, creating a dynamic, abstract pattern.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning