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

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

Part V. Conclusion

CHAPTER 8

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.