Chapter 3. Knowledge Sharing

Written by Nina Chen and Mark Barolak

Edited by Riona MacNamara

Your organization understands your problem domain better than some random person on the internet; your organization should be able to answer most of its own questions. To achieve that, you need both experts who know the answers to those questions *and* mechanisms to distribute their knowledge, which is what we'll explore in this chapter. These mechanisms range from the utterly simple (Ask questions; Write down what you know) to the much more structured, such as tutorials and classes. Most importantly, however, your organization needs a *culture of learning,* and that requires creating the psychological safety that permits people to admit to a lack of knowledge.

Challenges to Learning

Sharing expertise across an organization is not an easy task. Without a strong culture of learning, challenges can emerge. Google has experienced a number of these challenges, especially as the company has scaled:

Lack of psychological safety

: An environment in which people are afraid to take risks or make mistakes in front of others because they fear being punished for it. This often manifests as a culture of fear or a tendency to avoid transparency.

Information islands

: Knowledge fragmentation that occurs in different parts of an organization that don't communicate with one another or use shared resources. In such an environment, each group develops its own way of doing things.[1] This often leads to the following:

Information fragmentation

Each island has an incomplete picture of the bigger whole.

Information duplication

Each island has reinvented its own way of doing something.

Information skew

Each island has its own ways of doing the same thing, and these might or might not conflict.

Single point of failure (SPOF)

A bottleneck that occurs when critical information is available from only a single person. This is related to *bus factor*, which is discussed in more detail in Chapter 2.

SPOFs can arise out of good intentions: it can be easy to fall into a habit of "Let me take care of that for you." But this approach optimizes for short-term efficiency ("It's faster for me to do it") at the cost of poor long-term scalability (the team never learns how to do whatever it is that needs to be done). This mindset also tends to lead to *all-or-nothing expertise*.

All-or-nothing expertise

A group of people that is split between people who know "everything" and novices, with little middle ground. This problem often reinforces itself if experts always do everything themselves and don't take the time to develop new experts through mentoring or documentation. In this scenario, knowledge and responsibilities continue to accumulate on those who already have expertise, and new team members or novices are left to fend for themselves and ramp up more slowly.

Parroting

Mimicry without understanding. This is typically characterized by mindlessly copying patterns or code without understanding their purpose, often under the assumption that said code is needed for unknown reasons.

Haunted graveyards

Places, often in code, that people avoid touching or changing because they are afraid that something might go wrong. Unlike the aforementioned *parroting*, haunted graveyards are characterized by people avoiding action because of fear and superstition.

In the rest of this chapter, we dive into strategies that Google's engineering organizations have found to be successful in addressing these challenges.

Philosophy

Software engineering can be defined as the multiperson development of multiversion programs.[2] People are at the core of software engineering: code is an important output but only a small part of building a product. Crucially, code does not emerge spontaneously out of nothing, and neither does expertise. Every expert was once a novice: an organization's success depends on growing and investing in its people.

Personalized, one-to-one advice from an expert is always invaluable. Different team members have different areas of expertise, and so the best teammate to ask for any given question will vary. But if the expert goes on vacation or switches teams, the team can be left in the lurch. And although one person might be able to provide personalized help for one-to-many, this doesn't scale and is limited to small numbers of "many."

Documented knowledge, on the other hand, can better scale not just to the team but to the entire organization. Mechanisms such as a team wiki enable many authors to share their expertise with a larger group. But even though written documentation is more scalable than one-to-one conversations, that scalability comes with some trade-offs: it might be more generalized and less applicable to individual learners' situations, and it comes with the added maintenance cost required to keep information relevant and up to date over time.

Tribal knowledge exists in the gap between what individual team members know and what is documented. Human experts know these things that aren't written down. If we document that knowledge and maintain it, it is now available not only to somebody with direct one-to-one access to the expert today, but to anybody who can find and view the documentation.

So in a magical world in which everything is always perfectly and immediately documented, we wouldn't need to consult a person any more, right? Not quite. Written knowledge has scaling advantages, but so does targeted human help. A human expert can synthesize their expanse of knowledge. They can assess what information is applicable to the individual's

use case, determine whether the documentation is still relevant, and know where to find it. Or, if they don't know where to find the answers, they might know who does.

Tribal and written knowledge complement each other. Even a perfectly expert team with perfect documentation needs to communicate with one another, coordinate with other teams, and adapt their strategies over time. No single knowledge-sharing approach is the correct solution for all types of learning, and the particulars of a good mix will likely vary based on your organization. Institutional knowledge evolves over time, and the knowledge-sharing methods that work best for your organization will likely change as it grows. Train, focus on learning and growth, and build your own stable of experts: there is no such thing as too much engineering expertise.

Setting the Stage: Psychological Safety

Psychological safety is critical to promoting a learning environment.

To learn, you must first acknowledge that there are things you don't understand. We should welcome such honesty rather than punish it. (Google does this pretty well, but sometimes engineers are reluctant to admit they don't understand something.)

An enormous part of learning is being able to try things and feeling safe to fail. In a healthy environment, people feel comfortable asking questions, being wrong, and learning new things. This is a baseline expectation for all Google teams; indeed, our research has shown that psychological safety is the most important part of an effective team.

Mentorship

At Google, we try to set the tone as soon as a "Noogler" (new Googler) engineer joins the company. One important way of building psychological safety is to assign Nooglers a mentor—someone who is *not* their team member, manager, or tech lead—whose responsibilities explicitly include answering questions and helping the Noogler ramp up. Having an officially assigned mentor to ask for help makes it easier for the newcomer and means that they don't need to worry about taking up too much of their coworkers' time.

A mentor is a volunteer who has been at Google for more than a year and who is available to advise on anything from using Google infrastructure to navigating Google culture. Crucially, the mentor is there to be a safety net to talk to if the mentee doesn't know whom else to ask for advice. This mentor is not on the same team as the mentee, which can make the mentee feel more comfortable about asking for help in tricky situations.

Mentorship formalizes and facilitates learning, but learning itself is an ongoing process. There will always be opportunities for coworkers to learn from one another, whether it's a new employee joining the organization or an experienced engineer learning a new technology. With a healthy team, teammates will be open not just to answering but also to *asking* questions: showing that they don't know something and learning from one another.

Psychological Safety in Large Groups

Asking a nearby teammate for help is easier than approaching a large group of mostly strangers. But as we've seen, one-to-one solutions don't scale well. Group solutions are more scalable, but they are also scarier. It can be intimidating for novices to form a question and ask it of a large group, knowing that their question might be archived for many years. The need for psychological safety is amplified in large groups. Every member of the group has a role to play in creating and maintaining a safe environment that ensures that newcomers are confident asking questions and up-and-coming experts feel empowered to help those newcomers without the fear of having their answers attacked by established experts.

The most important way to achieve this safe and welcoming environment is for group interactions to be cooperative, not adversarial. Table 3-1 lists some examples of recommended patterns (and their corresponding antipatterns) of group interactions.

Table 3-1. Group interaction patterns

Recommended patterns (cooperative)	Antipatterns (adversarial)
Basic questions or mistakes are guided in the proper direction	Basic questions or mistakes are picked on, and the person asking the question is chastised
Explanations are given with the intent of helping the person asking the question learn	Explanations are given with the intent of showing off one's own knowledge
Responses are kind, patient, and helpful	Responses are condescending, snarky, and unconstructive
Interactions are shared discussions for finding solutions	Interactions are arguments with "winners" and "losers"

These antipatterns can emerge unintentionally: someone might be trying to be helpful but is accidentally condescending and unwelcoming. We find the Recurse Center's social rules to be helpful here:

No feigned surprise ("What?! I can't believe you don't know what the stack is!")

: Feigned surprise is a barrier to psychological safety and makes members of the group afraid of admitting to a lack of knowledge.

No "well-actuallys"

: Pedantic corrections that tend to be about grandstanding rather than precision.

No back-seat driving

: Interrupting an existing discussion to offer opinions without committing to the conversation.

No subtle "-isms" ("It's so easy my grandmother could do it!")

: Small expressions of bias (racism, ageism, homophobia) that can make individuals feel unwelcome, disrespected, or unsafe.

Growing Your Knowledge

Knowledge sharing starts with yourself. It is important to recognize that you always have something to learn. The following guidelines allow you to augment your own personal knowledge.

Ask Questions

If you take away only a single thing from this chapter, it is this: always be learning; always be asking questions.

We tell Nooglers that ramping up can take around six months. This extended period is necessary to ramp up on Google's large, complex infrastructure, but it also reinforces the idea that learning is an ongoing, iterative process. One of the biggest mistakes that beginners make is not to ask for help when they're stuck. You might be tempted to struggle through it alone or feel fearful that your questions are "too simple." "I just need to try harder before I ask anyone for help," you think. Don't fall into this trap! Your coworkers are often the best source of information: leverage this valuable resource.

There is no magical day when you suddenly always know exactly what to do in every situation—there's always more to learn. Engineers who have been at Google for years still have areas in which they don't feel like they know what they are doing, and that's OK! Don't be afraid to say "I don't know what that is; could you explain it?" Embrace not knowing things as an area of opportunity rather than one to fear.[3]

It doesn't matter whether you're new to a team or a senior leader: you should always be in an environment in which there's something to learn. If not, you stagnate (and should find a new environment).

It's especially critical for those in leadership roles to model this behavior: it's important not to mistakenly equate "seniority" with "knowing everything." In fact, the more you know, the more you know you don't know. Openly asking questions[4] or expressing gaps in knowledge reinforces that it's OK for others to do the same.

On the receiving end, patience and kindness when answering questions fosters an environment in which people feel safe looking for help. Making it easier to

overcome the initial hesitation to ask a question sets the tone early: reach out to solicit questions, and make it easy for even "trivial" questions to get an answer. Although engineers *could* probably figure out tribal knowledge on their own, they're not here to work in a vacuum. Targeted help allows engineers to be productive faster, which in turn makes their entire team more productive.

Understand Context

Learning is not just about understanding new things; it also includes developing an understanding of the decisions behind the design and implementation of existing things. Suppose that your team inherits a legacy codebase for a critical piece of infrastructure that has existed for many years. The original authors are long gone, and the code is difficult to understand. It can be tempting to rewrite from scratch rather than spend time learning the existing code. But instead of thinking "I don't get it" and ending your thoughts there, dive deeper: what questions should you be asking?

Consider the principle of "Chesterson's fence": before removing or changing something, first understand why it's there.

> *In the matter of reforming things, as distinct from deforming them, there is one plain and simple principle; a principle which will probably be called a paradox. There exists in such a case a certain institution or law; let us say, for the sake of simplicity, a fence or gate erected across a road. The more modern type of reformer goes gaily up to it and says, "I don't see the use of this; let us clear it away." To which the more intelligent type of reformer will do well to answer: "If you don't see the use of it, I certainly won't let you clear it away. Go away and think. Then, when you can come back and tell me that you do see the use of it, I may allow you to destroy it."*

This doesn't mean that code can't lack clarity or that existing design patterns can't be wrong, but engineers have a tendency to reach for "this is bad!" far more quickly than is often warranted, especially for unfamiliar code, languages, or paradigms. Google is not immune to this. Seek out and understand context, especially for decisions that seem unusual. After you've understood the context and purpose of the code, consider whether your change still makes sense. If it does, go ahead and make it; if it doesn't, document your reasoning for future readers.

Many Google style guides explicitly include context to help readers understand the rationale behind the style guidelines instead of just memorizing a list of arbitrary rules. More subtly, understanding the rationale behind a given guideline allows authors to make informed decisions about when the guideline shouldn't apply or whether the guideline needs updating. See Chapter 8.

Scaling Your Questions: Ask the Community

Getting one-to-one help is high bandwidth but necessarily limited in scale. And as a learner, it can be difficult to remember every detail. Do your future self a favor: when you learn something from a one-to-one discussion, *write it down*.

Chances are that future newcomers will have the same questions you had. Do them a favor, too, and *share what you write down*.

Although sharing the answers you receive can be useful, it's also beneficial to seek help not from individuals but from the greater community. In this section, we examine different forms of community-based learning. Each of these approaches—group chats, mailing lists, and question-and-answer systems—have different trade-offs and complement one another. But each of them enables the knowledge seeker to get help from a broader community of peers and experts and also ensures that answers are broadly available to current and future members of that community.

Group Chats

When you have a question, it can sometimes be difficult to get help from the right person. Maybe you're not sure who knows the answer, or the person you want to ask is busy. In these situations, group chats are great, because you can ask your question to many people at once and have a quick back-and-forth conversation with whoever is available. As a bonus, other members of the group chat can learn from the question and answer, and many forms of group chat can be automatically archived and searched later.

Group chats tend to be devoted either to topics or to teams. Topic-driven group chats are typically open so that anyone can drop in to ask a question. They tend to attract experts and can grow quite large, so questions are usually answered quickly. Team-oriented chats, on the other hand, tend to be smaller and restrict membership. As a result, they might not have the same reach as a topic-driven chat, but their smaller size can feel safer to a newcomer.

Although group chats are great for quick questions, they don't provide much structure, which can make it difficult to extract meaningful information from a conversation in which you're not actively involved. As soon as you need to share information outside of the group, or make it available to refer back to later, you should write a document or email a mailing list.

Mailing Lists

Most topics at Google have a topic-users@ or topic-discuss@ Google Groups mailing list that anyone at the company can join or email. Asking a question on a public mailing list is a lot like asking a group chat: the question reaches a lot of people who could potentially answer it and anyone following the list can learn from the answer. Unlike group chats, though, public mailing lists are easy to share with a wider audience: they are packaged into searchable archives, and email threads provide more structure than group chats. At Google, mailing lists are also indexed and can be discovered by Moma, Google's intranet search engine.

When you find an answer to a question you asked on a mailing list, it can be tempting to get on with your work. Don't do it! You never know when someone will need the same information in the future, so it's a best practice to post the answer back to the list.

Mailing lists are not without their trade-offs. They're well suited for complicated questions that require a lot of context, but they're clumsy for the quick back-and-forth exchanges at which group chats excel. A thread about a particular problem is generally most useful while it is active. Email archives are immutable, and it can be hard to determine whether an answer discovered in an old discussion thread is still relevant to a present-day situation. Additionally, the signal-to-noise ratio can be lower than other mediums like formal documentation because the problem that someone is having with their specific workflow might not be applicable to you.

EMAIL AT GOOGLE

Google culture is infamously email-centric and email-heavy. Google engineers receive hundreds of emails (if not more) each day, with varying degrees of actionability. Nooglers can spend days just setting up email filters to deal with the volume of notifications coming from groups that they've been autosubscribed to; some people just give up and don't try to keep up with the flow. Some groups CC large mailing lists onto every discussion by default, without trying to target information to those who are likely to be specifically interested in it; as a result, the signal-to-noise ratio can be a real problem.

Google tends toward email-based workflows by default. This isn't necessarily because email is a better medium than other communications options—it often isn't—rather, it's because that's what our culture is accustomed to. Keep this in mind as your organization considers what forms of communication to encourage or invest in.

YAQS: Question-and-Answer Platform

YAQS ("Yet Another Question System") is a Google-internal version of a Stack Overflow–like website, making it easy for Googlers to link to existing or work-in-progress code as well as discuss confidential information.

Like Stack Overflow, YAQS shares many of the same advantages of mailing lists and adds refinements: answers marked as helpful are promoted in the user interface, and users can edit questions and answers so that they remain accurate and useful as code and facts change. As a result, some mailing lists have been superseded by YAQS, whereas others have evolved into more general discussion lists that are less focused on problem solving.

Scaling Your Knowledge: You Always Have Something to Teach

Teaching is not limited to experts, nor is expertise a binary state in which you are either a novice or an expert. Expertise is a multidimensional vector of what you know: everyone has varying levels of expertise across different areas. This is one of the reasons why diversity is critical to organizational success: different people bring different perspectives and expertise to the table

(see Chapter 4). Google engineers teach others in a variety of ways, such as office hours, giving tech talks, teaching classes, writing documentation, and reviewing code.

Office Hours

Sometimes it's really important to have a human to talk to, and in those instances, office hours can be a good solution. Office hours are a regularly scheduled (typically weekly) event during which one or more people make themselves available to answer questions about a particular topic. Office hours are almost never the first choice for knowledge sharing: if you have an urgent question, it can be painful to wait for the next session for an answer; and if you're hosting office hours, they take up time and need to be regularly promoted. That said, they do provide a way for people to talk to an expert in person. This is particularly useful if the problem is still ambiguous enough that the engineer doesn't yet know what questions to ask (such as when they're just starting to design a new service) or whether the problem is about something so specialized that there just isn't documentation on it.

Tech Talks and Classes

Google has a robust culture of both internal and external[5] tech talks and classes. Our engEDU (Engineering Education) team focuses on providing Computer Science education to many audiences, ranging from Google engineers to students around the world. At a more grassroots level, our g2g (Googler2Googler) program lets Googlers sign up to give or attend talks and classes from fellow Googlers.[6] The program is wildly successful, with thousands of participating Googlers teaching topics from the technical (e.g., "Understanding Vectorization in Modern CPUs") to the just-for-fun (e.g., "Beginner Swing Dance").

Tech talks typically consist of a speaker presenting directly to an audience. Classes, on the other hand, can have a lecture component but often center on in-class exercises and therefore require more active participation from attendees. As a result, instructor-led classes are typically more demanding and expensive to create and maintain than tech talks and are reserved for the most important or difficult topics. That said, after a class has been created, it can be scaled relatively easily because many instructors can teach a class from the

same course materials. We've found that classes tend to work best when the following circumstances exist:

- The topic is complicated enough that it's a frequent source of misunderstanding. Classes take a lot of work to create, so they should be developed only when they're addressing specific needs.
- The topic is relatively stable. Updating class materials is a lot of work, so if the subject is rapidly evolving, other forms of sharing knowledge will have a better bang for the buck.
- The topic benefits from having teachers available to answer questions and provide personalized help. If students can easily learn without directed help, self-serve mediums like documentation or recordings are more efficient. A number of introductory classes at Google also have self-study versions.
- There is enough demand to offer the class regularly. Otherwise, potential learners will get the information they need in other ways rather than waiting for the class. At Google, this is particularly a problem for small, geographically remote offices.

Documentation

Documentation is written knowledge whose primary goal is to help its readers learn something. Not all written knowledge is necessarily documentation, although it can be useful as a paper trail. For example, it's possible to find an answer to a problem in a mailing list thread, but the primary goal of the original question on the thread was to seek answers, and only secondarily to document the discussion for others.

In this section, we focus on spotting opportunities for contributing to and creating formal documentation, from small things like fixing a typo to larger efforts such as documenting tribal knowledge.

NOTE

For a more comprehensive discussion of documentation, see Chapter 10.

Updating documentation

The first time you learn something is the best time to see ways that the existing documentation and training materials can be improved. By the time you've absorbed and understood a new process or system, you might have forgotten what was difficult or what simple steps were missing from the "Getting Started" documentation. At this stage, if you find a mistake or omission in the documentation, fix it! Leave the campground cleaner than you found it,[7] and try to update the documents yourself, even when that documentation is owned by a different part of the organization.

At Google, engineers feel empowered to update documentation regardless of who owns it—and we often do—even if the fix is as small as correcting a typo. This level of community upkeep increased notably with the introduction of g3doc,[8] which made it much easier for Googlers to find a documentation owner to review their suggestion. It also leaves an auditable trail of change history no different than that for code.

Creating documentation

As your proficiency grows, write your own documentation and update existing docs. For example, if you set up a new development flow, document the steps. You can then make it easier for others to follow in your path by pointing them to your document. Even better, make it easier for people to find the document themselves. Any sufficiently undiscoverable or unsearchable documentation might as well not exist. This is another area in which g3doc shines because the documentation is predictably located right next to the source code, as opposed to off in an (unfindable) document or webpage somewhere.

Finally, make sure there's a mechanism for feedback. If there's no easy and direct way for readers to indicate that documentation is outdated or inaccurate, they are likely not to bother telling anyone, and the next newcomer will come across the same problem. People are more willing to contribute changes if they feel that someone will actually notice and consider their suggestions. At Google, you can file a documentation bug directly from the document itself.

In addition, Googlers can easily leave comments on g3doc pages. Other Googlers can see and respond to these comments and, because leaving a

comment automatically files a bug for the documentation owner, the reader doesn't need to figure out who to contact.

Promoting documentation

Traditionally, encouraging engineers to document their work can be difficult. Writing documentation takes time and effort that could be spent on coding, and the benefits that result from that work are not immediate and are mostly reaped by others. Asymmetrical trade-offs like these are good for the organization as a whole given that many people can benefit from the time investment of a few, but without good incentives, it can be challenging to encourage such behavior. We discuss some of these structural incentives in the section "Incentives and recognition".

However, a document author can often directly benefit from writing documentation. Suppose that team members always ask you for help debugging certain kinds of production failures. Documenting your procedures requires an upfront investment of time, but after that work is done, you can save time in the future by pointing team members to the documentation and providing hands-on help only when needed.

Writing documentation also helps your team and organization scale. First, the information in the documentation becomes canonicalized as a reference: team members can refer to the shared document and even update it themselves. Second, the canonicalization may spread outside the team. Perhaps some parts of the documentation are not unique to the team's configuration and become useful for other teams looking to resolve similar problems.

Code

At a meta level, code *is* knowledge, so the very act of writing code can be considered a form of knowledge transcription. Although knowledge sharing might not be a direct intent of production code, it is often an emergent side effect, which can be facilitated by code readability and clarity.

Code documentation is one way to share knowledge; clear documentation not only benefits consumers of the library, but also future maintainers. Similarly, implementation comments transmit knowledge across time: you're writing these comments expressly for the sake of future readers (including Future You!). In terms of trade-offs, code comments are subject to the same

downsides as general documentation: they need to be actively maintained or they can quickly become out of date, as anyone who has ever read a comment that directly contradicts the code can attest.

Code reviews (see Chapter 9) are often a learning opportunity for both author(s) and reviewer(s). For example, a reviewer's suggestion might introduce the author to a new testing pattern, or a reviewer might learn of a new library by seeing the author use it in their code. Google standardizes mentoring through code review with the *readability process*, as detailed in the case study at the end of this chapter.

Scaling Your Organization's Knowledge

Ensuring that expertise is appropriately shared across the organization becomes more difficult as the organization grows. Some things, like culture, are important at every stage of growth, whereas others, like establishing canonical sources of information, might be more beneficial for more mature organizations.

Cultivating a Knowledge-Sharing Culture

Organizational culture is the squishy human thing that many companies treat as an afterthought. But at Google, we believe that focusing on the culture and environment first[9] results in better outcomes than focusing on only the output—such as the code—of that environment.

Making major organizational shifts is difficult, and countless books have been written on the topic. We don't pretend to have all the answers, but we can share specific steps Google has taken to create a culture that promotes learning.

See the book *Work Rules!*[10] for a more in-depth examination of Google's culture.

Respect

The bad behavior of just a few individuals can make an entire team or community unwelcoming. In such an environment, novices learn to take their questions elsewhere, and potential new experts stop trying and don't have

room to grow. In the worst cases, the group reduces to its most toxic members. It can be difficult to recover from this state.

Knowledge sharing can and should be done with kindness and respect. In tech, tolerance—or worse, reverence—of the "brilliant jerk" is both pervasive and harmful, but being an expert and being kind are not mutually exclusive. The Leadership section of Google's software engineering job ladder outlines this clearly:

> *Although a measure of technical leadership is expected at higher levels, not all leadership is directed at technical problems. Leaders improve the quality of the people around them, improve the team's psychological safety, create a culture of teamwork and collaboration, defuse tensions within the team, set an example of Google's culture and values, and make Google a more vibrant and exciting place to work. Jerks are not good leaders.*

This expectation is modeled by senior leadership: Urs Hölzle (Senior Vice President of Technical Infrastructure) and Ben Treynor Sloss (Vice President, Founder of Google SRE) wrote a regularly cited internal document ("No Jerks") about why Googlers should care about respectful behavior at work and what to do about it.

Incentives and recognition

Good culture must be actively nurtured, and encouraging a culture of knowledge sharing requires a commitment to recognizing and rewarding it at a systemic level. It's a common mistake for organizations to pay lip service to a set of values while actively rewarding behavior that does not enforce those values. People react to incentives over platitudes, and so it's important to put your money where your mouth is by putting in place a system of compensation and awards.

Google uses a variety of recognition mechanisms, from company-wide standards such as performance review and promotion criteria to peer-to-peer awards between Googlers.

Our software engineering ladder, which we use to calibrate rewards like compensation and promotion across the company, encourages engineers to share knowledge by noting these expectations explicitly. At more senior

levels, the ladder explicitly calls out the importance of wider influence, and this expectation increases as seniority increases. At the highest levels, examples of leadership include the following:

- Growing future leaders by serving as mentors to junior staff, helping them develop both technically and in their Google role
- Sustaining and developing the software community at Google via code and design reviews, engineering education and development, and expert guidance to others in the field

---

NOTE

See Chapters 5 and 6 for more on leadership.

Job ladder expectations are a top-down way to direct a culture, but culture is also formed from the bottom up. At Google, the peer bonus program is one way we embrace the bottom-up culture. Peer bonuses are a monetary award and formal recognition that any Googler can bestow on any other Googler for above-and-beyond work.[11] For example, when Ravi sends a peer bonus to Julia for being a top contributor to a mailing list—regularly answering questions that benefit many readers—he is publicly recognizing her knowledge-sharing work and its impact beyond her team. Because peer bonuses are employee driven, not management driven, they can have an important and powerful grassroots effect.

Similar to peer bonuses are kudos: public acknowledgement of contributions (typically smaller in impact or effort than those meriting a peer bonus) that boost the visibility of peer-to-peer contributions.

When a Googler gives another Googler a peer bonus or kudos, they can choose to copy additional groups or individuals on the award email, boosting recognition of the peer's work. It's also common for the recipient's manager to forward the award email to the team to celebrate one another's achievements.

A system in which people can formally and easily recognize their peers is a powerful tool for encouraging peers to keep doing the awesome things they do. It's not the bonus that matters: it's the peer acknowledgement.

Establishing Canonical Sources of Information

Canonical sources of information are centralized, company-wide corpuses of information that provide a way to standardize and propagate expert knowledge. They work best for information that is relevant to all engineers within the organization, which is otherwise prone to information islands. For example, a guide to setting up a basic developer workflow should be made canonical, whereas a guide for running a local Frobber instance is more relevant just to the engineers working on Frobber.

Establishing canonical sources of information requires higher investment than maintaining more localized information such as team documentation, but it also has broader benefits. Providing centralized references for the entire organization makes broadly required information easier and more predictable to find and counters problems with information fragmentation that can arise when multiple teams grappling with similar problems produce their own—often conflicting—guides.

Because canonical information is highly visible and intended to provide a shared understanding at the organizational level, it's important that the content is actively maintained and vetted by subject matter experts. The more complex a topic, the more critical it is that canonical content has explicit owners. Well-meaning readers might see that something is out of date but lack the expertise to make the significant structural changes needed to fix it, even if tooling makes it easy to suggest updates.

Creating and maintaining centralized, canonical sources of information is expensive and time consuming, and not all content needs to be shared at an organizational level. When considering how much effort to invest in this resource, consider your audience. Who benefits from this information? You? Your team? Your product area? All engineers?

Developer guides

Google has a broad and deep set of official guidance for engineers, including style guides, official software engineering best practices,[12] guides for code review[13] and testing,[14] and Tips of the Week (TotW).[15]

The corpus of information is so large that it's impractical to expect engineers to read it all end to end, much less be able to absorb so much information at

once. Instead, a human expert already familiar with a guideline can send a link to a fellow engineer, who then can read the reference and learn more. The expert saves time by not needing to personally explain a company-wide practice, and the learner now knows that there is a canonical source of trustworthy information that they can access whenever necessary. Such a process scales knowledge because it enables human experts to recognize and solve a specific information need by leveraging common, scalable resources.

go/ links

go/ links (sometimes referred to as goto/ links) are Google's internal URL shortener.[16] Most Google-internal references have at least one internal go/ link. For example, "go/spanner" provides information about Spanner, "go/python" is Google's Python developer guide. The content can live in any repository (g3doc, Google Drive, Google Sites, etc.), but having a go/ link that points to it provides a predictable, memorable way to access it. This yields some nice benefits:

- go/ links are so short that it's easy to share them in conversation ("You should check out go/frobber!"). This is much easier than having to go find a link and then send a message to all interested parties. Having a low-friction way to share references makes it more likely that that knowledge will be shared in the first place.
- go/ links provide a permalink to the content, even if the underlying URL changes. When an owner moves content to a different repository (for example, moving content from a Google doc to g3doc), they can simply update the go/ link's target URL. The go/ link itself remains unchanged.

go/ links are so ingrained into Google culture that a virtuous cycle has emerged: a Googler looking for information about Frobber will likely first check go/frobber. If the go/ link doesn't point to the Frobber Developer Guide (as expected), the Googler will generally configure the link themselves. As a result, Googlers can usually guess the correct go/ link on the first try.

Codelabs

Google codelabs are guided, hands-on tutorials that teach engineers new concepts or processes by combining explanations, working best-practice example code, and code exercises.[17] A canonical collection of codelabs for technologies broadly used across Google is available at go/codelab. These

codelabs go through several rounds of formal review and testing before publication. Codelabs are an interesting halfway point between static documentation and instructor-led classes, and they share the best and worst features of each. Their hands-on nature makes them more engaging than traditional documentation, but engineers can still access them on demand and complete them on their own; but they are expensive to maintain and are not tailored to the learner's specific needs.

Static analysis

Static analysis tools are a powerful way to share best practices that can be checked programmatically. Every programming language has its own particular static analysis tools, but they have the same general purpose: to alert code authors and reviewers to ways in which code can be improved to follow style and best practices. Some tools go one step further and offer to automatically apply those improvements to the code.

NOTE

See Chapter 20 for details on static analysis tools and how they're used at Google.

Setting up static analysis tools requires an upfront investment, but as soon as they are in place, they scale efficiently. When a check for a best practice is added to a tool, every engineer using that tool becomes aware of that best practice. This also frees up engineers to teach other things: the time and effort that would have gone into manually teaching the (now automated) best practice can instead be used to teach something else. Static analysis tools augment engineers' knowledge. They enable an organization to apply more best practices and apply them more consistently than would otherwise be possible.

Staying in the Loop

Some information is critical to do one's job, such as knowing how to do a typical development workflow. Other information, such as updates on popular productivity tools, is less critical but still useful. For this type of knowledge, the formality of the information sharing medium depends on the importance of the information being delivered. For example, users expect official

documentation to be kept up to date, but typically have no such expectation for newsletter content, which therefore requires less maintenance and upkeep from the owner.

Newsletters

Google has a number of company-wide newsletters that are sent to all engineers, including *EngNews* (engineering news), *Ownd* (Privacy/Security news), and *Google's Greatest Hits* (report of the most interesting outages of the quarter). These are a good way to communicate information that is of interest to engineers but isn't mission critical. For this type of update, we've found that newsletters get better engagement when they are sent less frequently and contain more useful, interesting content. Otherwise, newsletters can be perceived as spam.

Even though most Google newsletters are sent via email, some are more creative in their distribution. *Testing on the Toilet* (testing tips) and *Learning on the Loo* (productivity tips) are single-page newsletters posted inside toilet stalls. This unique delivery medium helps the *Testing on the Toilet* and *Learning on the Loo* stand out from other newsletters, and all issues are archived online.

NOTE

See Chapter 11 for a history of how *Testing on the Toilet* came to be.

Communities

Googlers like to form cross-organizational communities around various topics to share knowledge. These open channels make it easier to learn from others outside your immediate circle and avoid information islands and duplication. Google Groups are especially popular: Google has thousands of internal groups with varying levels of formality. Some are dedicated to troubleshooting; others, like the Code Health group, are more for discussion and guidance. Internal Google+ is also popular among Googlers as a source of informal information because people will post interesting technical breakdowns or details about projects they are working on.

Readability: Standardized Mentorship Through Code Review

At Google, “readability” refers to more than just code readability; it is a standardized, Google-wide mentorship process for disseminating programming language best practices. Readability covers a wide breadth of expertise, including but not limited to language idioms, code structure, API design, appropriate use of common libraries, documentation, and test coverage.

Readability started as a one-person effort. In Google’s early days, Craig Silverstein (employee ID #3) would sit down in person with every new hire and do a line-by-line “readability review” of their first major code commit. It was a nitpicky review that covered everything from ways the code could be improved to whitespace conventions. This gave Google’s codebase a uniform appearance but, more important, it taught best practices, highlighted what shared infrastructure was available, and showed new hires what it’s like to write code at Google.

Inevitably, Google’s hiring rate grew beyond what one person could keep up with. So many engineers found the process valuable that they volunteered their own time to scale the program. Today, around 20% of Google engineers are participating in the readability process at any given time, as either reviewers or code authors.

What Is the Readability Process?

Code review is mandatory at Google. Every changelist (CL)[18] requires *readability approval*, which indicates that someone who has *readability certification* for that language has approved the CL. Certified authors implicitly provide readability approval of their own CLs; otherwise, one or more qualified reviewers must explicitly give readability approval for the CL. This requirement was added after Google grew to a point where it was no longer possible to enforce that every engineer received code reviews that taught best practices to the desired rigor.

NOTE

See Chapter 9 for an overview of the Google code review process and what Approval means in this context.

Within Google, having readability certification is commonly referred to as "having readability" for a language. Engineers with readability have demonstrated that they consistently write clear, idiomatic, and maintainable code that exemplifies Google's best practices and coding style for a given language. They do this by submitting CLs through the readability process, during which a centralized group of *readability reviewers* review the CLs and give feedback on how much it demonstrates the various areas of mastery. As authors internalize the readability guidelines, they receive fewer and fewer comments on their CLs until they eventually graduate from the process and formally receive readability. Readability brings increased responsibility: engineers with readability are trusted to continue to apply their knowledge to their own code and to act as reviewers for other engineers' code.

Around 1 to 2% of Google engineers are readability reviewers. All reviewers are volunteers, and anyone with readability is welcome to self-nominate to become a readability reviewer. Readability reviewers are held to the highest standards because they are expected not just to have deep language expertise, but also an aptitude for teaching through code review. They are expected to treat readability as first and foremost a mentoring and cooperative process, not a gatekeeping or adversarial one. Readability reviewers and CL authors alike are encouraged to have discussions during the review process. Reviewers provide relevant citations for their comments so that authors can learn about the rationales that went into the style guidelines ("Chesterson's fence"). If the rationale for any given guideline is unclear, authors should ask for clarification ("ask questions").

Readability is deliberately a human-driven process that aims to scale knowledge in a standardized yet personalized way. As a complementary blend of written and tribal knowledge, readability combines the advantages of written documentation, which can be accessed with citable references, with the advantages of expert human reviewers, who know which guidelines to cite. Canonical guidelines and language recommendations are comprehensively documented—which is good!—but the corpus of

information is so large[19] that it can be overwhelming, especially to newcomers.

Why Have This Process?

Code is read far more than it is written, and this effect is magnified at Google's scale and in our (very large) monorepo.[20] Any engineer can look at and learn from the wealth of knowledge that is the code of other teams, and powerful tools like Kythe make it easy to find references throughout the entire codebase (see Chapter 17). An important feature of documented best practices (see Chapter 8) is that they provide consistent standards for all Google code to follow. Readability is both an enforcement and propagation mechanism for these standards.

One of the primary advantages of the readability program is that it exposes engineers to more than just their own team's tribal knowledge. To earn readability in a given language, engineers must send CLs through a centralized set of readability reviewers who review code across the entire company. Centralizing the process makes a significant trade-off: the program is limited to scaling linearly rather than sublinearly with organization growth, but it makes it easier to enforce consistency, avoid islands, and avoid (often unintentional) drifting from established norms.

The value of codebase-wide consistency cannot be overstated: even with tens of thousands of engineers writing code over decades, it ensures that code in a given language will look similar across the corpus. This enables readers to focus on what the code does rather than being distracted by why it looks different than code that they're used to. Large-scale change authors (see Chapter 22) can more easily make changes across the entire monorepo, crossing the boundaries of thousands of teams. People can change teams and be confident that the way that the new team uses a given language is not drastically different than their previous team.

These benefits come with some costs: readability is a heavyweight process compared to other mediums like documentation and classes because it is mandatory and enforced by Google tooling (see Chapter 19). These costs are nontrivial and include the following:

- Increased friction for teams that do not have any team members with readability, because they need to find reviewers from outside their team to

give readability approval on CLs.
- Potential for additional rounds of code review for authors who need readability review.
- Scaling disadvantages of being a human-driven process. Limited to scaling linearly to organization growth because it depends on human reviewers doing specialized code reviews.

The question, then, is whether the benefits outweigh the costs. There's also the factor of time: the full effect of the benefits versus the costs are not on the same timescale. The program makes a deliberate trade-off of increased short-term code-review latency and upfront costs for the long-term payoffs of higher-quality code, repository-wide code consistency, and increased engineer expertise. The longer timescale of the benefits comes with the expectation that code is written with a potential lifetime of years, if not decades.[21]

As with most—or perhaps all—engineering processes, there's always room for improvement. Some of the costs can be mitigated with tooling. A number of readability comments address issues that could be detected statically and commented on automatically by static analysis tooling. As we continue to invest in static analysis, readability reviewers can increasingly focus on higher-order areas, like whether a particular block of code is understandable by outside readers who are not intimately familiar with the codebase instead of automatable detections like whether a line has trailing whitespace.

But aspirations aren't enough. Readability is a controversial program: some engineers complain that it's an unnecessary bureaucratic hurdle and a poor use of engineer time. Are readability's trade-offs worthwhile? For the answer, we turned to our trusty Engineering Productivity Research (EPR) team.

The EPR team performed in-depth studies of readability, including but not limited to whether people were hindered by the process, learned anything, or changed their behavior after graduating. These studies showed that readability has a net positive impact on engineering velocity. CLs by authors with readability take statistically significantly less time to review and submit than CLs by authors who do not have readability.[22] Self-reported engineer satisfaction with their code quality—lacking more objective measures for code quality—is higher among engineers who have readability versus those who do not. A significant majority of engineers who complete the program report satisfaction with the process and find it worthwhile. They report

learning from reviewers and changing their own behavior to avoid readability issues when writing *and* reviewing code.

NOTE

For an in-depth look at this study and Google's internal engineering productivity research, see Chapter 7.

Google has a very strong culture of code review, and readability is a natural extension of that culture. Readability grew from the passion of a single engineer to a formal program of human experts mentoring all Google engineers. It evolved and changed with Google's growth, and it will continue to evolve as Google's needs change.

Conclusion

Knowledge is in some ways the most important (though intangible) capital of a software engineering organization, and sharing of that knowledge is crucial for making an organization resilient and redundant in the face of change. A culture that promotes open and honest knowledge sharing distributes that knowledge efficiently across the organization and allows that organization to scale over time. In most cases, investments into easier knowledge sharing reap manyfold dividends over the life of a company.

TL;DRs

- *Psychological safety* is the foundation for fostering a knowledge-sharing environment.
- Start small: ask questions and write things down.
- Make it easy for people to get the help they need from both human experts and documented references.
- At a systemic level, encourage and reward those who take time to teach and broaden their expertise beyond just themselves, their team, or their organization.
- There is no silver bullet: empowering a knowledge-sharing culture requires a combination of multiple strategies, and the exact mix that works best for your organization will likely change over time.

1 In other words, rather than developing a single global maximum, we have a bunch of local maxima.

2 David Lorge Parnas, *Software Engineering: Multi-person Development of Multi-version Programs* (Heidelberg: Springer-Verlag Berlin, 2011).

3 Impostor syndrome is not uncommon among high achievers, and Googlers are no exception—in fact, a majority of this book's authors have impostor syndrome. We acknowledge that fear of failure can be difficult for those with impostor syndrome and can reinforce an inclination to avoid branching out.

4 See "How to ask good questions."

5 *https://talksat.withgoogle.com* and *https://www.youtube.com/GoogleTechTalks*, to name a few.

6 The g2g program is detailed in: Laszlo Bock, *Work Rules!: Insights from Inside Google That Will Transform How You Live and Lead* (New York: Twelve Books, 2015). It includes descriptions of different aspects of the program as well as how to evaluate impact and recommendations for what to focus on when setting up similar programs.

7 See "The Boy Scout Rule" and Kevlin Henney, *97 Things Every Programmer Should Know* (Boston: O'Reilly, 2010).

8 g3doc stands for "google3 documentation." google3 is the name of the current incarnation of Google's monolithic source repository.

9 Laszlo Bock, *Work Rules!: Insights from Inside Google That Will Transform How You Live and Lead* (New York: Twelve Books, 2015).

10 Ibid.

11 Peer bonuses include a cash award and a certificate as well as being a permanent part of a Googler's award record in an internal tool called gThanks.

12 Such as books about software engineering at Google.

13 See Chapter 9.

14 See Chapter 11.

15 Available for multiple languages. Externally available for C++ at *https://abseil.io/tips*.

16 go/ links are unrelated to the Go language.

17 External codelabs are available at *https://codelabs.developers.google.com*.

18 A *changelist* is a list of files that make up a change in a version control system. A changelist is synonymous with a *changeset*.

19 As of 2019, just the Google C++ style guide is 40 pages long. The secondary material making up the complete corpus of best practices is many times longer.

20 For why Google uses a monorepo, see *https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext*. Note also that not all of Google's code lives within the monorepo; readability as described here applies only to the monorepo because it is a notion of within-repository consistency.

21 For this reason, code that is known to have a short time span is exempt from readability requirements. Examples include the *experimental/* directory (explicitly designated for experimental code and cannot push to production) and the Area 120 program, a workshop for Google's experimental products.

22 This includes controlling for a variety of factors, including tenure at Google and the fact that CLs for authors who do not have readability typically need additional rounds of review compared to authors who already have readability.