Chapter 2. How to Work Well on Teams

Written by Brian Fitzpatrick

Edited by Riona MacNamara

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

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

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

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

Help Me Hide My Code

For the past 20 years, my colleague Ben[1] and I have spoken at many programming conferences. In 2006, we launched Google's (now deprecated) open source Project Hosting service, and at first, we used to get lots of questions and requests about the product. But around mid-2008, we began to notice a trend in the sort of requests we were getting:

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

Can you spot a common theme to these requests?

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

The Genius Myth

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

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

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

And what about Michael Jordan?

It's the same story. We idolized him, but the fact is that he didn't win every basketball game by himself. His true genius was in the way he worked with his team. The team's coach, Phil Jackson, was extremely clever, and his coaching techniques are legendary. He recognized that one player alone never wins a championship, and so he assembled an entire "dream team" around MJ. This team was a well-oiled machine—at least as impressive as Michael himself.

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

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

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

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

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

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

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

To quote a friend:

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

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

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

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

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

Hiding Considered Harmful

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

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

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

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

Early Detection

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

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

The Bus Factor

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

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

Beyond the bus factor, there's the issue of overall pace of progress. It's easy to forget that working alone is often a tough slog, much slower than people want to admit. How much do you learn when working alone? How fast do you move? Google and Stack Overflow are great sources of opinions and information, but they're no substitute for actual human experience. Working with other people directly increases the collective wisdom behind the effort. When you become stuck on something absurd, how much time do you waste pulling yourself out of the hole? Think about how different the experience would be if you had a couple of peers to look over your shoulder and tell you—instantly—how you goofed and how to get past the problem. This is exactly why teams sit together (or do pair programming) in software engineering companies. Programming is hard. Software engineering is even harder. You need that second pair of eyes.

Pace of Progress

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

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

CASE STUDY: ENGINEERS AND OFFICES

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

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

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

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

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

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

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

In Short, Don’t Hide

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

Don’t become another statistic.

It’s All About the Team

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

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

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

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

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

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

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

The Three Pillars of Social Interaction

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

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

Pillar 1: Humility

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

Pillar 2: Respect

> You genuinely care about others you work with. You treat them kindly and appreciate their abilities and accomplishments.

Pillar 3: Trust

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

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

Why Do These Pillars Matter?

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

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

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

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

Humility, Respect, and Trust in Practice

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

Lose the ego

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

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

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

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

Learn to give *and* take criticism

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

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

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

For example, if you have an insecure collaborator, here's what not to say: "Man, you totally got the control flow wrong on that method there. You should be using the standard xyzzy code pattern like everyone else." This feedback is full of antipatterns: you're telling someone they're "wrong" (as if the world were black and white), demanding they change something, and accusing them of creating something that goes against what everyone else is doing (making them feel stupid). Your coworker will immediately be put on the offense, and their response is bound to be overly emotional.

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

Fail fast and iterate

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

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

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

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

Over in Google X—the division that works on "moonshots" like self-driving cars and internet access delivered by balloons—failure is deliberately built into its incentive system. People come up with outlandish ideas and coworkers are actively encouraged to shoot them down as fast as possible. Individuals are rewarded (and even compete) to see how many ideas they can disprove or invalidate in a fixed period of time. Only when a concept truly cannot be debunked at a whiteboard by all peers does it proceed to early prototype.

Blameless Post-Mortem Culture

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

A good postmortem should include the following:

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

Learn patience

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

A problem arose when we began pair programming: I’m a bottom-up engineer who is content to dive into the muck and dig my way out by trying a lot of things quickly and skimming over the details. Karl, however, is a top-down engineer who wants to get the full lay of the land and dive into the implementation of almost every method on the call stack before proceeding to tackle the bug. This resulted in some epic interpersonal conflicts, disagreements, and the occasional heated argument. It got to the point at which the two of us simply couldn’t pair-program together: it was too frustrating for us both.

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

Be open to influence

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

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

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

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

Being Googley

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

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

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

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

Thrives in ambiguity

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

Values feedback

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

Challenges status quo

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

Puts the user first

> Has empathy and respect for users of Google's products and pursues actions that are in their best interests.

Cares about the team

> Has empathy and respect for coworkers and actively works to help them without being asked, improving team cohesion.

Does the right thing

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

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

Conclusion

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

TL;DRs

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

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

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

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

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

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

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

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