University Libraries

A destination for research, learning, and friends

Electronic Course Reserves

Notice: Warning Concerning Copyright Restriction

This material is on Electronic Reserve for:

Instructor:	Gestwicki
Course:	CS 222
CREF:	The Myth of the Genius Programmer

The work from which this copy is made:

Author Name:	Brian Fitzpatrick and Ben Collins-Sussman
Book or Periodical Title:	**Team Geek: A Software Developer's Guide to Working Well with Others**
Vol.(is), Date:	
Year Published:	2012
Publisher & Location:	O'Reilly Media; Sebastopol, CA
Pages in excerpt:	1-23
Total pages in book or Publication:	167

The Myth of the Genius Programmer

Since this is a book about the social perils of software development, it makes sense to focus on the one variable you definitely have control of: you.

People are inherently imperfect. 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. You'll end up spending less energy dealing with people-related problems and more time writing great code.

The critical idea in this chapter is to understand that software development is a team sport. And in order to succeed on an engineering team, you need to reorganize your behaviors around the core principles of humility, respect, and trust.

Before we get ahead of ourselves, let's start by observing how programmers behave in general.

Help Me Hide My Code

The two of us have been speaking at programming conferences quite a bit for the past six years. Since we're part of the original team that launched Google's open source Project Hosting service back in 2006, we used to get lots of questions and requests about

the product. Back in mid-2008, we noticed a distinctive trend in the sort of requests we were getting:

> *Can you guys please give Subversion on Google Code the ability to hide specific branches?*
>
> *Can you guys make it possible to create open source projects that start out hidden to the world, then get 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?

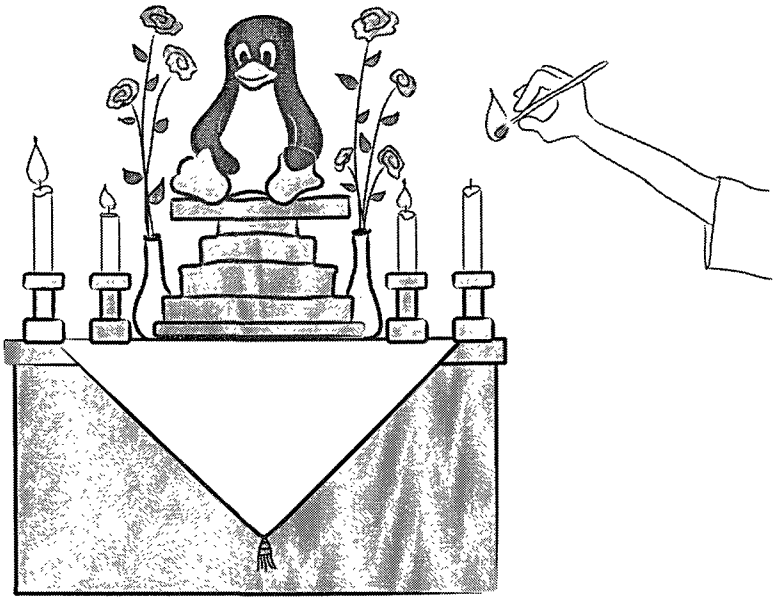
The key motif here is *insecurity*. People are afraid of others seeing and judging their work in progress. In one sense, it's just a part of human nature—nobody likes to be criticized, especially for things that aren't finished. This attitude tipped us off to a trend within software development. Insecurity is actually the symptom of a larger problem.

The Genius Myth

First, let's be clear: we're not actually sports fans. When our wives cheer for baseball or football on TV, we scratch our heads and wonder what's so exciting. Nevertheless, we did live through the early 1990s and witnessed the amazing run of championships by the Chicago Bulls. (That's a basketball team, by the way.) We were both in Chicago during this period, and the national media was saturated for years with stories about this amazing team.

What did we mostly hear about on TV and in newspapers? Not the team, but Michael Jordan, the superstar. Every player around the world wanted to *be* MJ. We watched him dance circles around other players. We watched him in television commercials. We went to see silly movies where he played basketball with cartoon characters. He was a star, and every kid on every court practicing hoops secretly wished to grow up and follow his path.

Programmers have that same instinct—to find idols and worship them. Linus Torvalds, Richard Stallman, Bill Gates—all heroes who changed the world with heroic feats. Linus wrote Linux by himself, right?



Beware of the natural instinct to idolize things.

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

On that same note, did Stallman personally write all of the Free Software Foundation's software? He wrote the first generation of Emacs. But hundreds of others were responsible for bash, the GCC tool chain, and all the rest of the software that runs on Linux. Steve Jobs led an entire team that built the Macintosh, and while 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 their collective achievements.

And how about Michael Jordan?

It's the same story. We idolize 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—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. The 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 locate 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 we all secretly wish to be geniuses. The ultimate geek fantasy is to be struck by an awesome new concept. You go into your Bat 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: let's do a reality check. You're probably not a genius.

No offense, of course—we're sure you're a very intelligent guy or gal. But do you realize how rare *actual* geniuses really are? Sure, you write code, and that's a tricky skill that probably puts you in a bracket above a lot of the human population. 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. What's going to make or break your career is how well you collaborate with others.

It turns out that this Genius Myth is just another aspect of our insecurity. Most 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 another programmer from Ben's blog:

> *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 and work, work, work. Nobody will see your goof-ups; you still have a chance to unveil your masterpiece when you're done. Hide away until all of it is perfect.

Another common motivation for holding your cards close to your chest 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 Is Considered Harmful

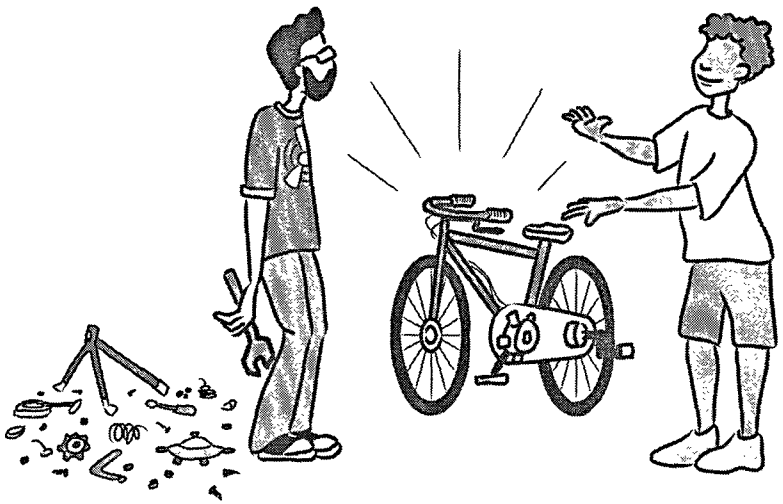
If you spend all your time working alone, you're *increasing* the risk of failure and cheating your potential for growth.

First of all, how do you even know if 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.



Working in isolation often leads to disappointment.

There are a number of lessons to learn here. 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.[1] 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.[2] Remember the tried-and-true mantra of "Fail early, fail fast, fail often"—we'll discuss the importance of failure at length later in the book.

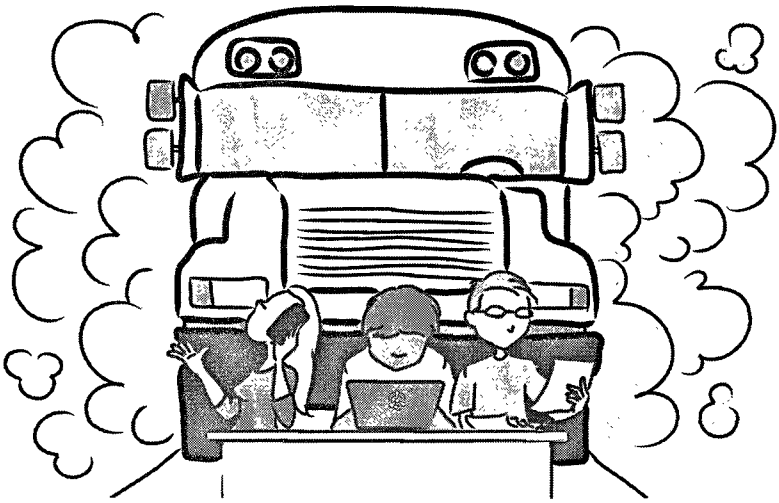
---

[1] Literally, if you are, in fact, a bike designer.

[2] We should note that sometimes it's dangerous to get *too* much feedback too early in the process, but we'll cover that in a later chapter.

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.

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



What's your team's bus factor?

How dispersed is the knowledge and know-how in your project? If you're the only person who understands how the prototype code works, it may be nice job security, but it also means the project is toast if you get hit by a bus. If you're working with a friend, however, you've doubled the bus factor. And if you've got a small team designing and prototyping together, things are even better—the project won't be over when a team member disappears. Remember: team members may not literally get hit by buses, but other unpredictable life events still happen. Someone may get married, have to move away, leave the company, or have to take care of a sick relative. You need to future-proof a project's success by managing the bus factor.

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? The Web is a great dumping ground of opinions and information, but it's no substitute for actual human experience. Working with other people directly increases

the collective wisdom behind the effort. When you get 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: you often find yourself needing a second pair of eyes.

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 or weeks writing code, then when you think it's all done and completely perfect, press the "compile" button for the very first time? Of course you don't. Can you imagine what sort of disaster would result, trying to compile 50,000 virgin lines of code? As programmers we work best in *tight* feedback loops: write a new function, compile. Add a test, compile. Refactor some code, compile. We get the typos and bugs fixed as soon as possible after generating code. We want the compiler at our side for every little step, playing wingman; 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 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 have to adapt to changing environments as they go. Projects run into unpredictable design obstacles, or political obstacles, or 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. Eric Raymond is often quoted as saying, "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 awake to discover that while their original vision may be complete, the world has changed and made the product irrelevant.

Engineers and Offices

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

We think that it's not only unnecessary for most engineers[3] to be in a private office, it's 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. Walk into the offices of any fast-growing high-tech company that started in the 21st century and you'll find engineers clustered together in shared cubicles (a.k.a., "bullpens") or shared desk areas, but rarely will you find them locked up in offices away from one another.

Of course, you'll still need a way to filter out noise and interruptions, which is why most teams we've seen have developed a way to communicate that they're currently busy and that you should limit interruptions. We 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 give out noise-canceling headphones to engineers to make it easier to deal with the noise in the area—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." Still 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.

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 even more.

[3] We do however acknowledge that serious introverts likely need more peace, quiet, and alone time than most people and may benefit from a more quiet environment if not their own office.

So what it boils down to is this: *working alone is inherently riskier than working with others.* While you may be afraid of someone stealing your idea or thinking you're dumb, you should be much more scared of wasting huge swaths of your time toiling away on the wrong thing.

Sadly, this problem of "clutching ideas to the chest" isn't unique to software engineering—it's a pervasive problem across all fields. For example, professional science is *supposed* to be about the free and open exchange of information. But the desperate need to "publish or perish" and to compete for grants has had exactly the opposite effect. Great thinkers don't share ideas. They cling to them obsessively, do their research in private, hide all mistakes along the path, and then ultimately publish a paper making it sound like the whole process was effortless and obvious. And the results are often disastrous: they accidentally duplicated someone else's work, or they made an undetected mistake early on, or they produced something that used to be interesting but is now regarded as useless. The amount of wasted time and effort is tragic.

Don't become another statistic.

It's All About the Team

So let's back up now and put all these ideas together.

The point we've been hammering is that in the realm of programming, lone craftsmen 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.

Creating a superstar *team* is the real goal, and is fiendishly difficult. The best teams make brilliant use of their superstars, but the whole is always greater than the sum of its parts.

Let's put this idea into simpler words:

Software development is a team sport.

This may be a difficult concept at first, since it directly contradicts our inner Genius Programmer fantasy. Try chanting it as a mantra.

Remember that software development is a team sport.

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!)

We're going to repeat this team-sport concept over and over throughout the book. High-functioning teams are gold and the true key to success. You should be aiming for this experience however you can; that's what this book is all about.

The Three Pillars

So the point about working in teams has been made. If teamwork is the best route to producing great software, how does one build (or find) a great team?

It's not quite that simple. In order to reach collaborative nirvana, you first need to learn and embrace what we 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.

Humility

> You are not the center of the universe. You're neither omniscient nor infallible. You're open to self-improvement.

Respect

> You genuinely care about others you work with. You treat them as human beings, and appreciate their abilities and accomplishments.

Trust

> You believe others are competent and will do the right thing, and you're OK with letting them drive when appropriate.[4]

Together, we refer to these principles as HRT. We pronounce this as "heart" and not "hurt" because it's all about *decreasing* pain and not about injuring people. In fact, our main thesis is built directly on these pillars:

> *Almost every social conflict can ultimately be traced back to a lack of humility, respect, or trust.*

It may sound implausible at first, but give it a try. Think about some nasty or uncomfortable social situation in your life right now. At the basest level, is everyone being appropriately humble? Are people really respecting one another? Is there mutual trust?

We believe these principles are so important that we've even structured this book around them.

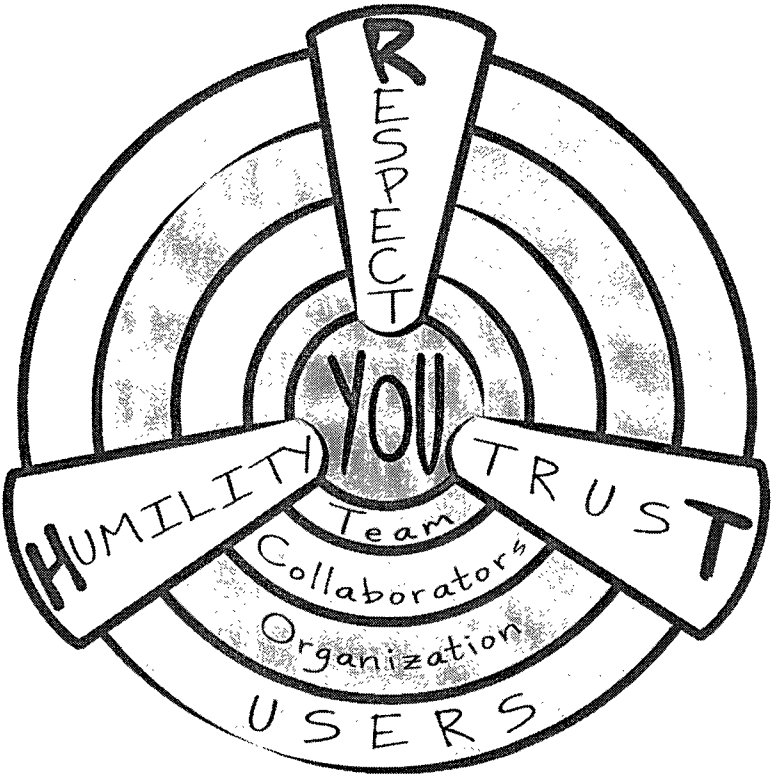
This book begins with you: getting you to embrace HRT and really internalize what it means to put HRT at the center of your interactions. That's what this first chapter is about. From there we create ever-expanding circles of influence.

In Chapter 2 we discuss the challenge of building a team based on the three pillars. Creating a team culture is the critical next step to success—this is the "dream team" discussed earlier.

---

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

We then examine people who are interacting with your team on a daily basis, but may not be part of the core team culture. These may be coworkers from other teams, or just volunteers offering to help on your project. Many of them not only disregard HRT, but they can be downright poisonous! Learning to defend your team from them is the first order of business. Removing their fangs and sucking them into your culture should be the ultimate goal, however. It's a great way to expand a team.



Embrace HRT for collaborative nirvana.

Most teams work within a larger company, and this environment can often be just as much of an impediment as poisonous people. Learning how to navigate these organizational obstacles can be the difference between launching a product and getting that very same product canceled.

Finally, we consider the users of your software. Sometimes we forget they exist, but they are the lifeblood of your project. Without users,

your software has no purpose. The same HRT principles that thrive in your team can and should be applied to the way you interact with your users, and the benefits reaped are tremendous.

Let's pause for a moment.

When you picked up this book, you probably weren't thinking you were signing 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:[5]

> *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, and relationships *always* outlast projects.

HRT in Practice

All of this preaching about humility, respect, and trust sounds like sermon material. Let's come out of the clouds and think about how to apply these ideas in real-life situations. We're looking for

[5] "You and Your Research," *http://www.cs.virginia.edu/~robins/YouAndYourResearch.pdf*

practical suggestions and so we're going to examine a list of specific behaviors and examples you can start with. Many of them may sound obvious at first, but once you start thinking about them you'll notice how often you (and your peers) are guilty of *not* following them.

Lose the Ego

OK, this is sort of a simpler way of telling someone without enough *humility* to lose his 'tude. Nobody wants to work with someone who consistently behaves like he's 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?

Note that "being humble" is *not* the same as saying one should be an utter 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. 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 about 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:*
>
> *"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."

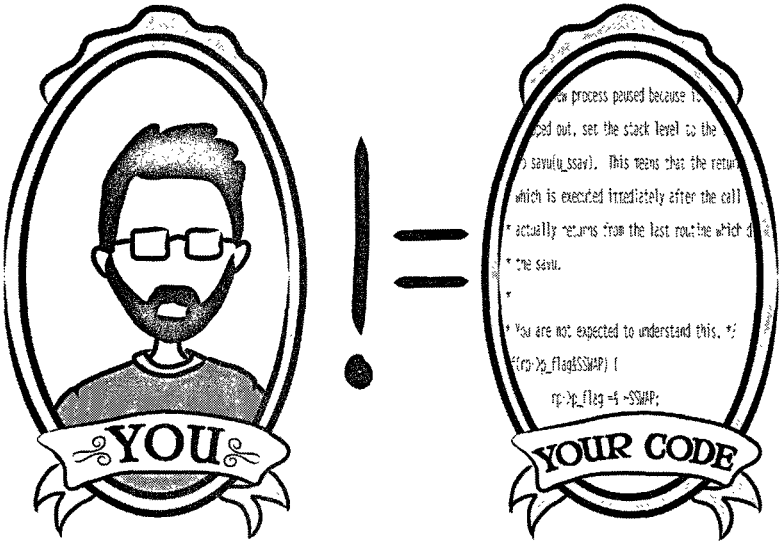
Learn to Both Deal Out and Handle Criticism

Joe started a new job as a programmer. After his first week he really started digging into the code base. 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. In a strong culture based on HRT, Joe's 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 subtler means to introduce code reviews into the culture.

Criticism is almost never personal in a professional software engineering environment—it's usually just part of the process of making a better product. The trick is to make sure you (and those around you) understand the difference between constructive criticism of someone's creative output and flat-out assaults against someone's character. The latter is useless—it's petty and nearly impossible to act on. The former is always helpful and gives guidance on how to improve. And most importantly, it's imbued with *respect*: the person giving the constructive criticism genuinely cares about the other person and wants her to improve herself or her 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.

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.* To repeat ourselves: you are not your code. Say that over and over. *You are not your code.* You need to not only believe it yourself, but get your coworkers to believe it too.



Don't equate your self-worth with your code quality.

For example, if you have a possibly 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 he's "wrong" (as if the world were black and white!), demanding he change something, and accusing him of creating something that goes against what everyone else is doing (making him feel stupid). The response is going to be overly emotional, coming from someone put on the defense.

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 him. He's 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, and 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 in the realm of the code itself and isn't about anyone's value or coding skills.

Fail Fast; Learn; Iterate

There's a well-known (and clichéd) 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 to the CEO.

"What's this?" asks the CEO.

"My resignation," says the exec. "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 exec wouldn't undo the $10 million loss, and it would compound it by losing a valuable executive who you can be damned sure won't make that kind of mistake again.

The two of us work at Google, and one of our favorite mottoes of Google's is "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. 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."

Google often follows the concept of "not hiding in the cave until it's perfect" (which we discussed previously): as soon as something

[6] A dozen variants of this legend can be found on the Web, attributed to different famous managers.

is vaguely usable, it gets released in raw form to the public. This is what Google Labs was all about. It becomes apparent very quickly where the successes and failures are, and so the programming team is expected to learn, iterate, and push a new version out as quickly as possible. The downside is that Google occasionally gets teased for having things like Gmail in "beta" for four-plus years. The upside is the ability to maneuver and adapt quickly, producing an amazing product in a very short amount of time. All it requires is some *humility*—that it's OK to show imperfect software to users, and some *trust* that your users really do appreciate your efforts and are eager to see rapid improvements.

The key to learning from your mistakes is to *document* your failures. Write up "postmortems," as they're often called in our business. Take extra care to make sure the postmortem document isn't just a useless list of apologies or excuses—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 you put it in an easy-to-find place and really follow through on the proposed changes. Remember that 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
- 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 to fix the problem immediately
- A set of action items to prevent the event from happening again
- Lessons learned

Leave Time for Learning

Cindy was a superstar—a software engineer who had truly mastered her specialized area. She was promoted to technical lead, the responsibility increased, and she rose to the challenge. Before long, she was mentoring everyone around her and teaching them the ropes. She was speaking at conferences on her subject and pretty soon ended up in charge of multiple teams. She absolutely loved being the "expert" all the time. And yet, she started to get bored. Somewhere along the way she stopped learning new things. The novelty of being the wisest, most experienced expert in the room started to wear thin. Despite all of the outward signs of mastery and success, something was missing. One day she got to work and realized that her chosen field simply wasn't so relevant anymore; people had moved on to other topics of interest. Where did she go wrong?

Let's face it: it is *fun* to be the most knowledgeable person in the room, and mentoring others can be incredibly rewarding. The problem is that once you reach a local maximum on your team, you stop learning. And when you stop learning, you get bored. Or accidentally become obsolete. It's really easy to get addicted to being a leading player; but only by giving up some ego will you ever change directions and get exposed to new things. Again, it's about increasing *humility* and being willing to learn as much as teach. Put yourself outside your comfort zone now and then; find a fishbowl with bigger fish than you and rise to whatever challenges they hand out to you. You'll be much happier in the long run.

Learn Patience

Years ago, Fitz was writing a tool to convert CVS repositories to Subversion (and later, Git), and, due to the vagaries of RCS and CVS, he kept unearthing bizarre bugs with invalid RCS files that CVS would happily devour. Since his longtime friend and coworker Karl knew CVS and RCS quite intimately, he and Karl decided they should work together to fix these bugs.

A problem arose when they started pair programming together: Fitz was a bottom-up engineer who was content to dive into the muck and dig his way out by trying a lot of things quickly and skimming over the details. Karl, however, was a top-down engineer who wanted 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 took a herculean effort, focus, and no small amount of HRT for Fitz and Karl to accomplish the task at hand. In the end, HRT not only helped save the project, but it also saved their friendship.

Be Open to Influence

The more you are open 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 him, he digs his heels in even more. What eventually happens to such team members? In our experience, they end up just getting "routed around" like an obstacle everyone takes for granted. People stop listening to their opinions or objections. You certainly don't want that happening to you, so keep this idea in your head: it's OK for someone else to change your mind. Choose your battles carefully. Remember that in order to be heard properly, you first need to listen to others. In the case of being influenced, this listening should take place before you've put a stake in the ground or firmly declared that you've decided on something—if you're constantly changing your mind, people will think you're wishy-washy.

On the subject of vulnerability, this seems a bit strange at first too. If someone admitted she was ignorant of the topic at hand or didn't know how to solve a problem, what sort of credibility would she have in a group? Vulnerability is a show of weakness, and that destroys confidence, right?

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

Honesty and humility are not kryptonite.

Consider professional politicians; they're notorious for never admitting error or ignorance, even when it's patently obvious that they're wrong or unknowledgeable about a subject. And for that reason most people don't believe a word that politicians say. This behavior exists primarily because politicians are constantly under attack by their opponents. When you're writing software, however, it's unnecessary to live in a constant state of defense—your teammates are collaborators, not competitors.

Next Steps

If you've made it this far, you're well on your way to mastering the art of "playing well with others." You've got to start with examining and meditating on your own behaviors. Once you've incorporated these strategies into your daily life, you'll find that

collaboration will become much more natural and your engineering productivity will begin to noticeably increase.

The important changes begin with you and then spread outward to others. In the next chapter, we're going to talk about how to create a culture of HRT within your immediate team.