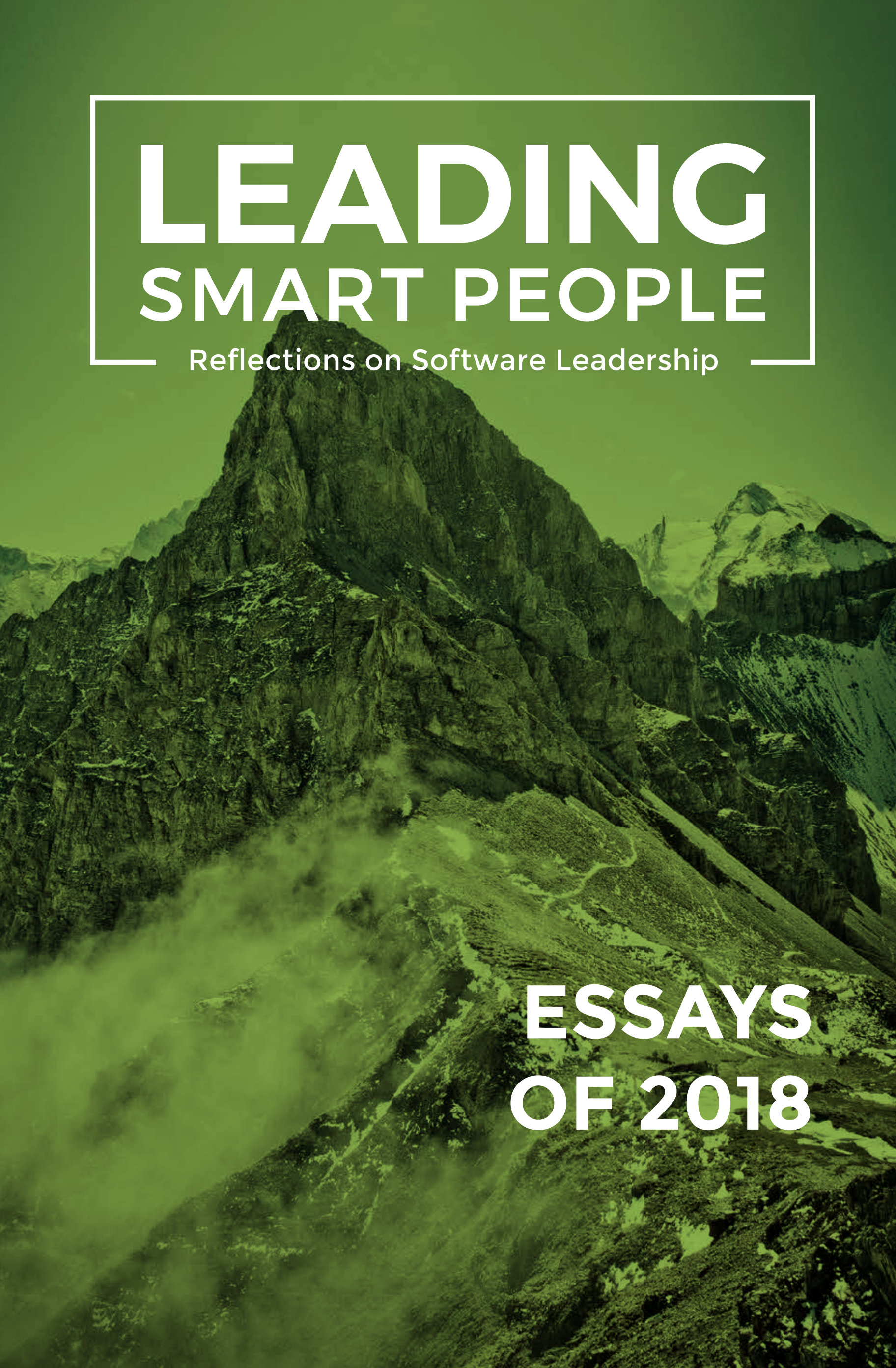


LEADING SMART PEOPLE

Reflections on Software Leadership

ESSAYS OF 2018

Welcome

[illegible]

MARCUS BLANKENSHIP

2018 was a pretty good year. I taught workshops in San Francisco, Oregon and Berlin, coached people from around the world, and developed Software Leader Seminar, a self-paced leadership course. But the thing I've worked hardest on, and the thing I'm proudest of, is my writing. My goal has always been to connect with people, both managers and developers, and help them learn how to build – and be part of – happy teams that produce valuable work.

With that in mind, I decided to collect some of my best writing from 2018 and offer it to you here in one place. Any collection of pieces written over time has a few oddities – bits of chronology that don't quite fit, references to current (now in the past) events. But overall, I hope you can find as much usefulness in these articles as my original readers did. So read on for 2018's best!

The Real Work of Software Management - Part 1

recently received an email from a reader, Anton, asking this question: How do I balance management with real work?

This is a common question. What Anton wants are tools to help him manage his time and his workload. He's having trouble "doing it all," which is a symptom of a larger problem. The underlying problem is the mindset that management is not "real" work.

The most important aspect of time management is prioritization. If you don't understand that management work is THE priority, any time-management techniques you use are bound to fail.

So, before I talk about tactics to balance demands, time

“

You might be called a Lead Programmer, Team Lead, IT Manager, Department Manager, Project Manager or CTO. If you’re responsible for overseeing programmers, then this is for you.

management, prioritization and the like, I want to step back and address the mindset question. Because if you don’t get the mindset right, then the tactics are only a short-term Band-Aid solution that temporarily masks the larger problem.

Management Is Real Work

Let me be as clear as possible: Management IS real work.

Any view that fails to put your management responsibilities first will fail you in the long-term. You might get away with it for a while, and at times it might seem like it’s working for you, but it’s a losing strategy. Even worse, most managers who hold this flawed mindset find out too late that it’s failed them. Projects are delivered late, teams fail to adapt to new demands, employee turnover increases and customers get very frustrated. All of this is because the manager isn't focused on managing their team. Instead, they're still programming.

Quick note: *I realize that different organizations use different titles, but I'm going to use the term "Software Manager" to refer to someone who is responsible for the work of programmers. You might be called a Lead Programmer, Team Lead, IT Manager, Department Manager, Project Manager or CTO. If you're responsible for overseeing programmers, then this is for you.*

What Is Real Work?

In my experience, real work is defined as effort that produces value for an organization. Ditch diggers produce valuable ditches for an organization. It is clearly real work.

Software developers create valuable software for an organization. Even if the ditch diggers don't agree, software development is also real work.

Software managers create valuable software development teams for an organization. Again, even if the software developers don't always agree, software management IS real work.

So what value does a software manager produce?

As a software manager, you provide value to multiple groups in the organization. Your work is more important than that of any one individual programmer because you are a force multiplier. You create a team that is greater than the sum of its parts. Some examples of where software managers create value:

They provide value to their team by...

1. Ensuring the right members are on the team and that they receive the right training,
2. Creating a productive work environment appropriate for software development,
3. Ensuring the team has the best tools possible, and
4. Protecting the team from organizational distractions and politics.

They provide value to their up-chain management by...

1. Ensuring management understands the value of the software the team creates,
2. Ensuring that the team's efforts support organizational goals, and
3. Ensuring that team turnover is low and the members are high-performing contributors.

They provide value to their customers by...

1. Listening to customer's needs and goals,
2. Ensuring that the right software is delivered at the right time, to the right people,
3. Collaborating with the customer about team priorities to deliver the highest priority items soonest, and
4. Taking customer feedback to the team so that the team can improve.

Why Management Must Be Your ONLY Job

No one else on the team can do it.

You are not "one of the gang." Sorry if you don't like this, but your team knows it, so don't get confused and act like it isn't true. You are unique in your group; you are their manager and leader. You have more authority than they do, and everyone knows it.

You are also responsible for the work the team does, the successes and failures of the team. The organization put you in charge to LEAD and MANAGE this group, and when things go wrong, they expect that you will take full responsibility for the problem. This is something only you can do.

Finally, you control the teams' attitude, productivity and working environment. You control who is on the team and who is not on the team. You sit in a unique position to lead, inspire, direct and manage. You alone are appointed to do this work.

Sound terrible? Well, the saying "It's lonely at the top" isn't hypothetical. It's very real, and at times, being the manager feels very lonely. Not always, but sometimes. Your job and the jobs of your team members depend on you creating a team that can deliver the right software, at the right time, to the right people.

If you fail to create a team that delivers software, your management (or your customers) will replace you with someone who will. This might sound harsh, but if you can't do it, they will find someone who will. Besides, the livelihood of every member of your team depends on your ability to create a team that can deliver software! Teams that don't deliver get cut during down-sizing, right-sizing and re-organization efforts. Remember, executive management is always keeping any eye on the cost vs. value ratio, so it's up to you to make sure your team is consistently delivering value.

In addition, you've got to promote the great work your team is doing to your manager through status reports, demonstrations and customer testimonials. You are the PR department for your team, and you've got to make sure their good work gets noticed!

Finally, your team needs constant mentoring, training and coaching to stay sharp and keep growing. Managers that neglect this put their team members at risk, because the poor performers and “dinosaurs” are always the first to be cut when times are tight. Employees ALWAYS want more training, and managers who don’t provide this do their team a great disservice.

Why Management Is Work Worth Doing

When I was first promoted to team lead, I was excited and proud. I was eager to tell my dad about my promotion.

He put his hand on my shoulder and said, “Son, your team deserves a good manager. Don’t let them down.”

As I returned to work, I looked at my team differently.

The brilliant programmers on my team wanted and deserved a good manager, someone who was willing to serve, protect, encourage, discipline and mentor them to do great things. Someone who was willing to put aside their love of programming to do the hard, uncomfortable, dirty work of management. Your team needs the same things.

The number one description of Bad Bosses that I hear from Programmers is this: “My manager didn’t want to manage. He just wanted to code.”

When you got promoted, the team knew exactly what it meant: you are now their boss.

Unfortunately, too many managers try to act like their team's best friend. They want to live in a fantasy world where everyone likes them, there are no difficult decisions, and everyone works together without any need for guidance or direction.

Symptoms of this problem are…

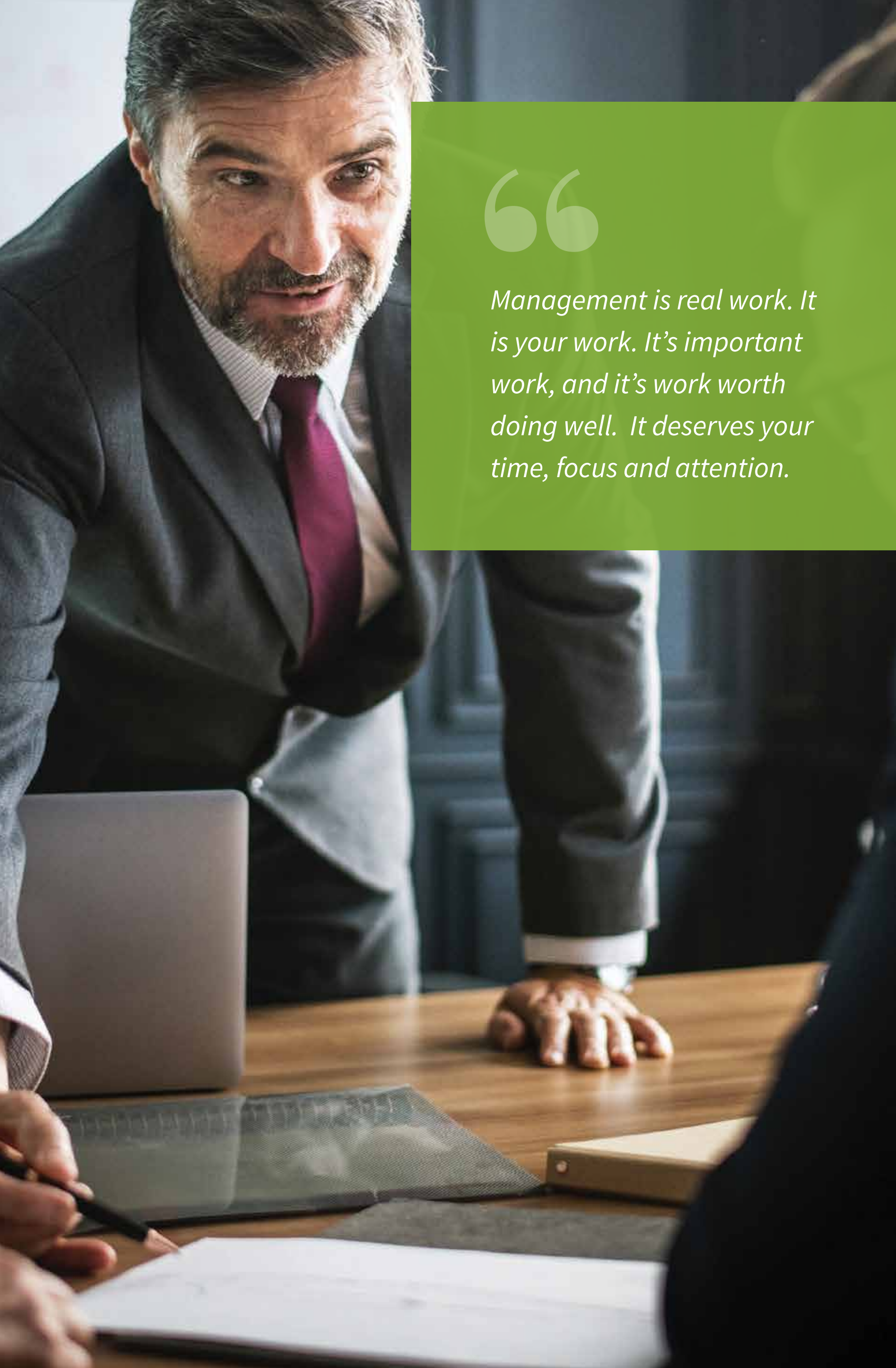
1. Unwillingness to take responsibility when problems arise,
2. Unwillingness to delegate clearly and directly,
3. Unwillingness to set clear expectations about how the work is done and when it is needed, and
4. Unwillingness to discipline disrespect and poor performance.

If you're not doing these things, you are not the manager your team wants or needs. Your misplaced need to be accepted by them has overshadowed the real need to create a valuable software development team. You are putting everyone at risk, and your team knows it.

Management is real work. It is your work. It's important work, and it's work worth doing well. It deserves your time, focus and attention. Like programming, it's a skill that improves with focus and time.

Now, poor Anton still needs to know how to manage his time, balance competing priorities, and find time to look up from his work. But I hope that he sees that having the management mindset must come first.

Next, we'll look at how everything managers do is "managing," whether it's shooting the breeze at the coffee machine or sitting in a customer meeting. You might be surprised at all the forms that "managing" can take.



Management is real work. It is your work. It's important work, and it's work worth doing well. It deserves your time, focus and attention.

People who don’t “add value” don’t last long at any organization.

The Real Work of Software Management - Part 2

Six months into the job, I started to wonder if being a Team Lead was for me. While I felt important being included in so many meetings, my new job didn’t feel nearly as valuable as my old one. It felt like I didn’t do anything besides sit in meetings or send email. I missed the feeling of accomplishment I used to have at the end of each day when I could see (and show!) the results of my efforts.

I also started to fear that others wouldn’t feel I was doing anything important either. Imaginary conversations played through my mind where team members snickered behind my back or my boss wanted to replace me. After all, people who don’t “add value” don’t last long at any organization, and I certainly didn’t feel I was adding much value!

I dove back into the IDE, trying to code my way into feeling good. This resulted in a dozen half-built projects, which gave me the feeling I was doing real work, but none of it actually got shipped to customers. Worse, now I had a dozen more things to finish hanging over my head! But at least I had gotten some "real work" done that I could show my boss and team (in case they ever asked).

When I finally brought up the subject with my boss, my questions came spilling out: "How am I doing at this management thing? I feel like I don't do anything, but I'm busy and stressed out all the time. Maybe I should just go back to coding."

He leaned back in his chair, a knowing half-smile on his face. "You're doing fine. Your new job is to build your software team and keep it running smoothly, so they produce what's needed. Let's look at how your activities in the past few months are doing that…"

He went on to show me the why behind the what of all my "meaningless" activities. I had learned what to do from watching him and other managers, but I'd only been mimicking their actions without understanding why they were important. This is a normal way to learn something new, but now it was holding me back. I needed to get to the next level.

As he connected the dots for me, my guilt over not coding eased. The fears that my team would think I was a slacker melted away as I saw that my management job had purpose and value. I was the only one on the team who was in the position to do it, and I could see what would happen if the job wasn't done well.

As I found the reasons behind the actions of leading my team, I learned that "Everything you do is Management." Recall from my last article that successful software managers know their only job is to create teams that deliver valuable software for their organization. All of their activities are in support of that goal, no matter how little it looks like management.

Let's look at four things good managers do that might not look anything like management to you…

1) "Just Visiting" With Your Team Members

You build trust with your team when you take the time to visit with them, individually or collectively. When you learn about their families, hobbies or weekend activities, you show that you care about them. People follow leaders they know care about them, their families and their goals. Building individual relationships through actively listening to their struggles, coaching them through challenges and keeping their confidence is vital to effective team management. Visiting with them in casual situations is when this magic happens.

To the outsider, this might look like "shooting the breeze" or "slacking off," but the experienced manager knows that the more she invests in relationships with her team members, the more enthusiastically they will follow her leadership. She knows most people want a boss who cares about them, and in return they will "go down with the ship" for her.

If this is a challenge for you, try keeping a short file or notepad on each employee. Knowing their spouse's name and occupation, the names and ages of their kids, what they enjoy doing on vacation and other personal information is too important to miss. During your 1:1 meeting with them, ask about their challenges, what projects and activities they enjoy or find frustrating, and what their goals are.

Don’t mistake this advice for just acting like you care about them. You actually have to care about your team members, each and every one of them. If you’re struggling with this, you need to step back and ask yourself if they are the right fit for your team, or else you might be subconsciously trying to sabotage their efforts or playing favorites.

2) Helping to Get the Project Shipped out the Door

When you are doing technical work side by side with the team, you are observing and improving your team’s skills. It doesn’t matter if this is coding, debugging, architecting, testing or writing documentation. Managers who pitch in know this is a prime time to invest in their team’s training, give informal feedback and judge skills up close.

Early in my career, my manager and I were working shoulder to shoulder to fix some critical systems, working late into the night. About 11:30 pm the VP of Information Systems (two steps above my manager on the corporate ladder) appeared to see if we needed anything or if he could help. When he heard we had not eaten dinner, he bought sodas and pizza, thanked us for our dedication and told us he was confident we would get it fixed. He stayed about an hour and then slipped away quietly. I never forgot that he cared enough to drop by and see if he could help, and he did help, as I was getting really hungry!

Good managers practice servant leadership, making sure the greatest needs of others are being served. They never hesitate to

```
}

/* =Menu
-------------------------------

#access {
    display: inline-block;
    height: 69px;
    float: right;
    margin: 11px 28px 0px 0px;
    max-width: 800px;
}

#access ul {
    font-size: 13px;
    list-style: none;
    margin: 0 0 0 -0.8125em;
    padding-left: 0;
    z-index: 99999;
    text-align: right;
}

[illegible] {
    display: inline-block;
    text-align: left;
```

invest in their team, knowing that this effort pays back tenfold. Your programmers want to get better at their jobs, they want to improve, and when they do, it benefits you and the company.

Additionally, managers who spend time mentoring their staff create a “train the trainers” culture. This culture encourages programmers to help each other and makes onboarding new developers much easier. Finally, training requires sharing yourself and what you know, which increases the bond between you and your team. People appreciate being trained and invested in, and they pay it forward to others when given the chance.

3) Meeting with Customers

Has your team ever asked, “Why do they want this feature?” or “Why is this due-date important?” These two questions represented

90% of the push-back I got from my team, and they weren’t being disrespectful. They were ensuring that the feature and deadline deserved their commitment.

You see, your team fears investing themselves in work and deadlines that don’t matter. These fears aren’t hypothetical. Most of us have sacrificed our evenings and weekends to hit a due-date that turned out to be unimportant to the customer. We’ve put our sweat and blood into solving hard problems only to learn that the customer didn’t care about the feature. When that happens, we wither inside. We feel deceived; we stop giving 100%. We become cynical, and we start protecting ourselves. We ask those two questions because we don’t want to be hurt again.

The solution? Meet with your customers regularly and make sure the work and deadlines are important. Don’t commit to them unless you are convinced, and keep asking questions until you have answers. Meeting with your customers to learn the specifics of why projects and deadlines are important is the only remedy. I know this might feel like “yet another meeting,” but if you don’t believe in the projects and features, you can never defend them to your team.

Poor managers often have the attitude, “We just do what we’re told, we don’t ask why.” That’s not management, it’s cowardice. You must ensure your team has valid answers for these questions, or you must tell the customers “No.” You sit in the gap, even in the most agile organizations, and you need to have answers to these questions at your fingertips.

You must meet with your customers regularly, and you must communicate what you learned back to your team. It is fundamental to protecting the health and morale of your team.

4) Empowering Your Team, and Letting Them Make Decisions

When you are listening to your team in meetings and work sessions, you are building trust by sharing control. This might not look like management to the uninitiated, but good leaders listen to their team's ideas, concerns and opinions. They know that simply holding the title of manager does not make them smarter than everyone and that the best way to earn trust is to give it.

Listening is a prerequisite to trust building. It shows that you can put your agenda aside and let someone else lead the discussion. The confidence to let others speak and be heard helps others trust you. Many times I've heard employees say, "I didn't like the final decision, but I appreciated that he took my viewpoint into account."

If good leaders listen, then great leaders let the team make the decision, gently guiding along the way. They know that when the team makes the decision, they will be committed and excited about what needs to be done. The manager doesn't need to force their will. They simply need to ensure the team has the best knowledge of goals and constraints (often called "context") to make good decisions. It might appear that managers who let their teams make decisions are abdicating their role, but in reality, they are trading a small amount of control for a large amount of enthusiasm and team building. As Kent Beck says in Extreme Programming Explained, "There is no substitute for enthusiasm." I couldn't agree more.

As the months continued, I became more confident in my new management role. I saw that everything I did was aligned with growing, protecting and guiding my team, even at the coffee machine. I realized that in some ways, management has something in common with parenting: teams, like kids, are always watching to see how I behave, so I'd better be setting a good example. As the years went on, I became subtler and more intentional in my management style. I learned the importance of caring for and serving my team members, of letting others make the decisions, and of protecting the team from demotivating arbitrary tasks and deadlines.

Next time we'll talk about how not to manage. In the meantime, hang out around the coffee machine and visit with your team!

It might appear that managers who let their teams make decisions are abdicating their role, but in reality, they're trading a small amount of control for a large amount of enthusiasm and team building.

7 Tips to Successfully Micromanage Programmers

A bit of satire to make you smile

Since most programmers only want to write code and don’t actually care about what problem they are solving, you need to learn to effectively micromanage them.

Without this, you will have endless discussions about frameworks, tools, ideas, process and the like. This will occur ad nauseam until you tell them what to do. Deep down, programmers want to be told what to do, how to do it, and when to do it. They often pretend this isn’t the case, but don’t let that fool you.

Their best friend, the compiler, is also extremely picky and demanding, which gives them a feeling of security and good boundaries. Either a program compiles, or it doesn’t. Your management should have the same ring of absoluteness to it.

You might think that micromanagement is a lot of work, and you'd be right.

You might think that micromanagement is a lot of work, and you'd be right. You always have to be "on," even during nights, weekends and vacations. But with these seven tips, you know you'll be spending your time effectively and creating a fast-moving team.

Remember you own them. They are YOUR programmers, and you are THEIR boss. This century's old dynamic is a classic for a reason: it works. If it's good enough for the Romans, monarchies and Henry Ford, it's good enough for us. Make sure you never waiver on who makes the decisions on your team.

Tell them exactly how to do it. Create very detailed "specs" that outline everything that needs to be done. Include the architecture, frameworks, database schema, object diagrams, function signatures, naming conventions, everything you can think of. Don't leave anything to chance. Don't worry if you have flaws in the spec

– this is an opportunity for them to “problem solve,” which many of them claim to enjoy. This detailed specificity about WHAT work is done will set the bar high and keep your developers productive!

Tell them exactly how to work together. Create flowcharts and role assignments. Remember, your team wants to be assembled together like cogs, and the team works best when each member has a very specific role. Discourage people from working together, to build experts in each role. Create formal communication channels and expect your employees to use them. They will appreciate that you have figured out the best way for them to work and be efficient.

Figure out the BEST way to do things for them. Programmers only want to write code, so you should figure out the BEST design, framework, coding standards, timeframes, role delegation, etc., in advance. Your programmers will trust your judgement and feel secure that you know best. This also leaves them plenty of coding time, which makes them happy.

Hold their toes to the fire. All good project managers know that teams work best under pressure, with challenging deadlines. This brings out the best in people, and they love rising to a challenge! If things start to slip, begin making passive-aggressive comments and veiled threats. This will show people that the deadlines are important, and they will shift into high-gear!

Check in multiple times each day to change direction. Programmers tend to drift and get distracted, so help them focus by checking in with them frequently. There’s no need to have something valuable to say – your mere presence will remind them to work harder.

Email, slack and call them late at night and on weekends. This is key to keeping up momentum through the long weekends and holidays. They will know you care about them when you call at 10 p.m. with a new feature idea or email them at 2 a.m. about a new bug. They will also know you're working hard and will be reminded that they should be too. You'll win admiration and respect for your work ethic. Can't stay up until 2 a.m.? Just schedule your emails to be sent at a specific time. No one will be the wiser.

With these seven tips you can increase your teams' output while still having time to golf over a long lunch.

If you hear complaints or murmurings, remind your team how lucky they are to have a job and that there are many other people lined up to take their place if they make trouble.

You've heard about the benefits of micromanaging, now you have actionable advice to start getting things done. Good luck!

Instead of focusing only on the leader, LMX Theory studies the relationships the leader has with their individual employees.

The Wrong Section of the Bookstore

I've always loved books. As a teenager, I spent weekends browsing the Sci-Fi section of the book store, eventually migrating to the Computer Programming section. Many happy hours were spent among those shelves.

After I had become a Team Lead, I noticed my browsing habits slowly changed. I started to take an interest in the Business and Leadership section. As a die-hard nerd, I felt pretty uncomfortable in this new section.

The covers of the books all showed these fit, tan, well-dressed people.

I didn't match any of those attributes, but I was drawn to the promises I found on those book covers.

I was struggling with my new management role. The technology didn't trouble me. It was my team. This alien section of the book store promised to help me lead, inspire and motivate my team, solving all my problems.

So, I bought and read many of these books. But instead of turning into a great leader as they promised, I felt more disconnected and discouraged than ever.

Part of the problem was that these books described people who seemed so different from me. No matter how hard I tried, emulating them felt unnatural and forced.

Fast forward a few years: after much trial and error, I had finally found a management style that worked for me. I heard it described as MBWA, or Management By Wandering Around. My team called it the "How's it going?" approach. They seemed to like it, and I found it a useful and productive way to manage my team.

LMX Theory

One night, while browsing material about Leadership Theories on the internet, I came across a term I'd never seen before: Leader-Member Exchange Theory (LMX Theory). Excitedly, I realized that this approach matched my own experience, and the research into the theory suddenly gave me the why behind the what of my practices.

According to the Oxford Handbook of Leader-Member Exchange, by Talya Bauer, LMX theory "views the dyadic relationship quality between leaders and members as the key to understanding leader effects on members, teams, and organizations." Huh?

Instead of focusing only on the leader, LMX Theory studies the relationships the leader has with their individual employees. The result? LMX studies have found that a huge factor in an employee's job performance and job satisfaction is their relationship with their manager.

It's not about being a born leader. It's about creating great relationships with your team.

However, there's a catch.

LMX theory asserts that leaders form high-quality, trust, affect and respect-based relationships with a subset of their team, whereas with other members, they tend to have a lower-quality exchange that is limited to the employees' and leader's job descriptions.

That is, you have great relationships with some people on your team and maybe not-so-great relationships with other people on your team.

Your great relationships display trust, affection and respect. But your poor relationships are shallower because they only focus on the duties both parties have according to their job descriptions.

This results in two distinct groups, the "in-group" and the "out-group."

My In-Group and Out-Group

Of course, I liked some people on my team more than others. I'll bet you do too. It's only natural. Without realizing it, I was letting my team relationships be dictated by matches between personality or interests rather than intentionally investing in all my relationships. Looking back, this uneven investment created my in-groups and out-groups.

This had significant consequences for my team. I trusted my in-group more. They got the better assignments. I was more likely to recommend them for promotion. I was more likely to overlook problems and give them the benefit of the doubt. I spent more time in casual, friendly conversation and had deeper, more personal relationships with them. I was more likely to mentor them and invest in them.

At the same time, I worried more about my out-group. I watched them more closely, questioned their judgment and decisions, and felt they were less reliable. I held them at arm's length, being more "professional," and they did the same to me. I was more likely to give them mundane assignments and less likely to have personal discussions with them. I didn't invest in them as much, and I didn't spend time mentoring them. I didn't think of them as "bad," but I didn't describe them as my "superstars" either.

Traditional Leadership Theories

Let's go back to those calm, confident, attractive leaders on those book covers: They almost all represented traditional leadership theories that can be lumped into a category called the "Great Man Theory." This theory emphasizes the special qualities of great military, political and business leaders throughout history. Though I found it fascinating to read about these people, I didn't find it useful in my management work. When I tried to act like these "great men," it came across as fake to my team, which hurt my relationship with them.

How LMX Theory Is Different

Instead of trying to change myself to be more like history's great leaders, LMX Theory prioritizes something we are all born knowing how to do: create strong relationships with people.

Let's go back to that phrase "dyadic relationships." A dyad is something that consists of two elements or parts. In this case, the two parts are you and each team member. Let me be clear – this isn't your relationship with "the team," this is your relationship with the individuals who make up your team, not the group that is your team.

The early developers of LMX wanted to understand and measure the impact of the boss–employee relationship on the employee. What they found was that bosses have an astoundingly high impact on their employees' job performance and job satisfaction. In fact, Gerstner and Day, in an academic study published in the **Journal of Applied Psychology**, [1] went so far as to say that "The relationship with one's boss is a lens through which the entire work experience is viewed."

In addition, Gallup's 2015 report "**The State of the American Manager**", [2] says that "Managers account for at least 70% of the variance of employee engagement scores across business units."

In short, managers matter. A lot. Not managers acting like "great

When I tried to act like these "great men," it came across as fake to my team.

men," but managers who create strong relationships with their team.

The Good News

The good news is that we all have experience creating human relationships. We know how to create and invest in friendships, establish and nurture romantic relationships, and have strong, trusted professional relationships with our peers.

So, we already have the tools we need to start creating great relationships. As Robert Fulghum's brilliantly titled book reminds us, All I Really Need to Know I Learned in Kindergarten. (This book is a gem, and you should pick it up if you haven't already read it.)

So, if manager–developer relationships matter, and we know we're good at creating relationships with other people, let's put in place some actions that create great relationships with our team.

The goal is to do things that move people from the out-group to the in-group so that everyone enjoys the benefits of greater productivity and job satisfaction.

Step 1: Take Inventory

First, now that you know how important this is, I suggest that you take an inventory of your relationships with your team members. Make a list of each person on your team and rate the relationship you have with them on a HIGH–MEDIUM–LOW scale.

As you look at your ranking, can you start to see members of your in-group? Everyone not in your in-group is automatically in your out-group. Place an “I” or an “O” next to their name.

Step 2: Commit to Small Improvements

Now you need to figure out ways to move people into your in-group. You’re trying to improve individual relationships, so you’ll probably choose different actions and questions for each person. Think about what you know and don’t know about the person. Like Ed Schein suggests in his book Humble Inquiry, “access your ignorance” to become honestly curious about the other person. Couple this curiosity with honest caring about the person to engage them in more personal and important discussions.

Next, to the name of each person in your out-group, write a note about how you’d like to improve your relationship or get to know them better. Jot down a question you’d like to ask or a topic you’d like to discuss with them and when you’ll do it. I suggest using your one-on-one meeting to do this, or possibly a coffee or lunch meeting where you can talk privately.

Step 3: Try, Try Again

Relationships take time to build, so don’t expect folks in the out-group to open up to you immediately. Your mantra should be to “try, try again” until you earn their trust. Remember, they might be in the out-group because your personalities didn’t immediately “click.” You probably haven’t invested enough energy or time into your relationship with them. It’s understandable that they might be suspicious of your motives. I suggest being upfront about your motives: you want to have a better professional relationship with them. You hope to establish better communication, trust and collaboration with them.

Take responsibility for not initiating this sooner, but remind them that it's never too late for things to improve.

Making relationship building a regular part of your meetings and discussions will impress upon your team members that you care about them, that you want a deeper professional relationship with them.

I've found I have to try about a dozen times before someone starts to believe I that want a different relationship. Consistency is key here. If you give up trying, they may feel your effort wasn't sincere, which will make the relationship worse. And of course, you have to actually care about them. We can all sense when someone is faking interest in us, and it quickly leads to distrust.

Your relationship with your team matters a tremendous amount, more than most managers realize. LMX Theory is the science that proves this and gives us a framework for seeing our team that illuminates great differences in job performance and job satisfaction between the in-group and out-group.

While it may not be possible to move everyone from the out-group to the in-group, regular relationship building activities with your out-group moves interested members into the in-group and opens up new possibilities for trust, collaboration and deeper professional relationships.

Be Yourself

Learning to manage people is hard. Don’t make the same mistake I did and try to become someone else.

I’ve never been a khaki-and-tie sort of fella, and it felt (and looked!) awkward when I acted that way. Finding LMX Theory affirmed what I learned the hard way: You already have the interpersonal skills you need to create trusted relationships.

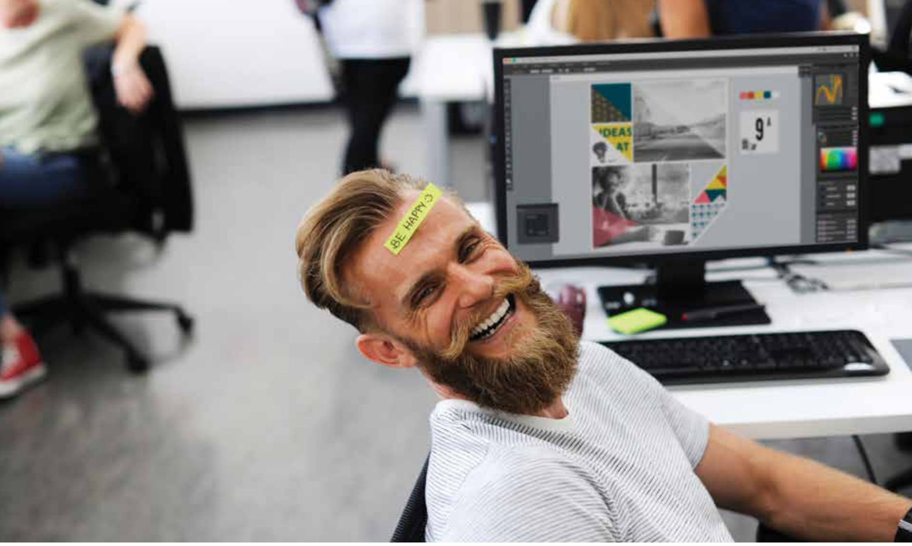
Be yourself. Work to create trusted, deep professional relationships. Connect with your team members as individuals, and humbly ask that they connect with you. These are the actions that create great relationships and build loyalty and trust with your team.

Keep working. You can do this.

---

[1] Gerstner, C. R., & Day, D. V. (1997). Meta-analytic review of leader-member exchange theory: Correlates and construct issues. Journal of Applied Psychology, 82, 827–844.

[2] https://www.gallup.com/services/182138/state-american-manager.aspx



The Science of Happy Developers: How to Improve Your Dev Team's Mojo

Software managers aspire to build high-performing teams, but they often stumble by devaluing the impact their individual relationships with developers have on team performance.

Well-meaning software managers often focus on their own contributions to improving the software process, technical architecture or political climate. While these efforts add value, they don't yield the individual and team performance increases needed by the organization.

Wikipedia defines a high-performance team as "as a group of people with specific roles and complementary talents and skills, aligned with and committed to a common purpose, who consistently show high levels of collaboration and innovation, that produce superior results." [3]

Many managers feel that having highly skilled programmers is the most important step toward creating high-performing teams. While it is true that high-performing software teams are made up of highly skilled programmers, simply grouping highly skilled programmers together rarely makes them a high-performing team.

Technical managers are often promoted from technical roles, so a belief that technical skills equate to team success often leads them to continue to focus on their own technical skills. This is not surprising as these are likely the skills that got them promoted into leadership in the first place.

Instead, managers should see their role in creating successful teams as an extension of the fruitful individual relationships they have with their programmers.

When it comes to building high-performing teams, a technical manager's relationship with her developers is more important than her own engineering prowess. I saw this first hand as a struggling technical manager, and I continue to see this in my clients as a technical leadership and process coach.

Leader-Member Exchange Theory (LMX Theory)

In the early 1980s, sociologists and organizational psychologists began to look at leadership from a new perspective. Traditional leadership theories have long been based on the "**Great Man Theory**," [4] which attributes a team's performance to qualities of the leader. Setting this theory aside in the twentieth century, sociologists began to study how the quality of relationships between managers and employees effected outcomes. This has evolved into an area of research known as Leader-Member Exchange Theory. This research has revealed that “the relationship with one’s boss is a lens through which the entire work experience is viewed.” [5]

They found a significant correlation between the quality of relationship developers have with their manager, and the developer’s job performance and job satisfaction. This focus on the quality of relationship between leader and employee, rather than only the traits of the leader, provides hope for all managers who who want high-performing teams should cultivate high-quality individual relationships with their team members.

Two Steps to Create Better Relationships

Hold weekly one-on-one meetings. In his book The 27 Problems Managers Face, Bruce Tulgan states, “When things are going wrong, the common denominator is unstructured, low-substance hit-or-miss communication.” One-on-one meetings are the cornerstone practice that creates and sustains strong relationships between managers and developers. One-on-one meetings aren’t simply status meetings, or delegation meetings, or even coffee meetings. Instead, these meetings provide a private, consistent place where leaders invest in their team, trust is built, feedback is given (and received) and the relationships deepens.

In short, one-on-one meetings are an activity meant to build deep professional trust, reveal training and feedback opportunities, and allow the developer a safe place to give the manager the candid feedback they need to improve.

Frequency and consistency are key attributes of these meetings.

Managers who see one-on-ones as an investment understand that cancelling them sends the wrong message and hurts the relationship.

They choose a frequency they can commit to, and they remain consistent in keeping and conducting their scheduled one-on-one meetings. While not every meeting is the same, they ensure that time is given to discuss project impediments, give affirming and corrective feedback, and ask for input and feedback on their own management practice. Without this feedback, managers are "driving blind," without a way to understand what their team needs from them.

Some managers object to spending time this way, pointing to their "open door" policy as a better style of management. In her Harvard Business Review article "Cancelling One-on-One Meetings Destroys Your Productivity," [6] Elizabeth Grace Sounders states "When you cancel one-on-ones and compensate with an open door policy, your time investment mimics that of a call center employee who takes requests in the order they are received, instead of an effective manager and executive who aligns his time investment with his priorities."

Adding to that, experienced managers understand that team members are reluctant to use their open-door policy for fear of "wasting the manager's time." Without a scheduled, consistent one-on-one meeting, managers must depend on developers to use their best judgement in deciding when and what to consult with their manager about. This reluctance often leads to unnecessary miscommunication, assumptions and frustration on both sides.

Hire selectively and prioritize for soft skills. LMX Theory found that employees fall into one of two groups under their manager, the in-group and the out-group. Members of the in-group enjoy more advancement opportunities, greater trust, and more autonomy and are happier and more productive than their out-group peers. With this in mind, managers should seek to hire programmers who understand and value strong manager relationships and who have the soft skills necessary to participate in them. While hard technical skills are important, the soft skills are an important indicator of success in relationships with their manager.
Many managers perform “culture fit” interviews, hoping to find programmers who will succeed at the company. This is a good start, but adding questions that explore past managerial relationships and ability (and willingness) to give candid feedback and understanding of the importance of good relationships shows that you care about relationships and helps you find employees who will also care.

Next time, we’ll talk about better ways to onboard those new hires.

---

[3] https://en.wikipedia.org/wiki/High-performance_teams

[4] https://en.wikipedia.org/wiki/Great_man_theory

[5] Gerstner & Day

[6] https://hbr.org/2015/03/cancelling-one-on-one-meetings-destroys-your-productivity

Onboard People, Not Technology

At my first programming job, it took three weeks to get my dev environment fully set up. I was only the second developer to work at the company, ever, so nothing was documented. The first person quit, which is why I had the job. At my second, it only took four days, because I was the eighth person in the programming department, so I had seven other people to help me. Today, my clients tell me things like, “we use Docker, so it takes less than an hour to onboard a new developer.”

I’m glad new devs don’t have to face the same frustrations I did. But setting up a productive dev environment isn’t onboarding. Onboarding is setting up a person to work productively on your team.

A Leadership Smell

Some managers might fall into the trap of believing that once they're done setting up the dev environment, they've done their part to make the new developer successful and that the rest is up to the new employee. That all they need is a computer, a chair, a dev environment, and a project to work on.

This is a dangerous leadership smell. Danger for the manager, yes, but mostly for the new developer.

The danger is the unspoken idea that after we apply onboarding to a new developer, they have everything they need to be productive. Of course, when you read it, it seems sort of ridiculous. But I get a whiff of this from managers at many organizations I work with. It typically comes out in a conversation like this:

Me: How is your new programmer working out?
Manager: He's doing… okay. Not quite what I hoped, but he'll be fine.
Me: What makes you say that?
Manager: Well, we got everything set up quickly, and we got him all his access and onboarding. But he's just a lot slower than I expected.
Me: Why did you expect he'd be faster?
Manager: First, he was supposed to be an expert in Java development. Second, we've automated our dev environment setup at great cost and effort. So, there shouldn't be anything standing in his way.

As you might guess, this situation is more dangerous for the programmer than the manager, because it's the manager's expectations that aren't being met. It's the manager who's disappointed in the speed at which things are progressing. And it's the manager who feels that their onboarding process, whatever it is, should yield a developer that's "up to speed and ready to run."

If the manager believes they've done their part, the problem must be with the developer's motivation, drive or skill. This ignores the most important keys to productivity: good relationships and a healthy environment

These are the things your developer needs to be onboarded to, and it takes much longer than just a few hours.

Onboarding Is Leading

Onboarding is a key activity of technical mangers and leaders. It's not the responsibility of HR, and it's certainly not simply setting up the dev environment.

In my experience, onboarding should include or result in the following:

- Positive relationships with the developer's team
- Introductions to the project stakeholders
- Clarity about the team's current goals
- Knowledge of key upcoming projects
- Clarity about the team's values
- Knowing who to go to for help and how often it's okay to ask for help
- First-hand experience with the team's development process

and ceremonies (agile, scrum, stand-ups, retros, etc.)
- First-hand experience with the key communication channels (team meetings, 1:1s, etc.)
- Receiving a piece of adjusting and affirming feedback from their manager
- Giving a piece of adjusting and affirming feedback to their manager and teammates
- A clear sense of the tech and infrastructure used by the team
- A working dev environment

How long should this take?

It should take you, the manager, at least a few weeks to initially onboard a new developer. After that, expect it to take a few quarters (three to nine months) for a new developer to “come up to speed.” Of course, this will vary wildly according to the kind of projects you are working on. I’ve led teams where I spent a year onboarding a dev to work on a large ERP system. You might have experienced even longer.

Onboarding is important leadership work that you need to be actively involved with. Like other work, you don’t have to do it all yourself, but you need to make sure it all gets done – and done well. Onboarding sets the stage for many years of productive work. Don’t shortchange your team by pretending it’s just about tools, paperwork or diversity videos. It’s about intentionally getting the right relationships in place from day one, which are the key to productive development teams.

Ask "Why?" with humility and genuine interest, seeking to understand.

Why Your Programmers Fantasize About a Big Rewrite

A fictional account of real things I have said and heard.

"Wouldn't it be awesome to re-write this from scratch?"
"Yes, that would be awesome," the team agreed.
"Then we could use a modern framework and that new language."
"Then the code would be so clean."
"We would have complete test coverage."
"We would use a modern UI."
"It would be so easy to debug."
"Then we would remove all the crufty parts."
"Then we would understand it completely."
"Then we could put new ideas into practice."
"We could really be agile about it."
"We'd have control over how we did it."

"We'd have a voice in how we work."
"We'd be taken seriously!"
"Our ideas would be respected!"
"Instead of being treated like code monkeys!"
"Then we could do great work."
"Then we would really show the boss what good software looks like."
"If only we had the chance to do a rewrite."
"Then everything would be great."
"If only."

Of course, this may not be the only reason your programmers want a re-write, but there are important elements of truth in here that you'd be advised not to ignore. The next time your programmers talk about a re-write, lean in and listen closely. It might not actually be about legacy code, old languages or complexity.

Instead, they may feel…

- Unable to get buy-in about their smaller ideas
- Prevented from doing great work by the culture or environment
- Trapped in a codebase they don't fully understand
- Lacking ownership of part of the system

Or, something else entirely.

A good way to learn more about this? Ask "Why?" with humility and genuine interest, seeking to understand what's behind their rewrite idea.

Other relevant questions might include "What's behind that?" "How would that impact the team?" "How would you feel about working on that project?"

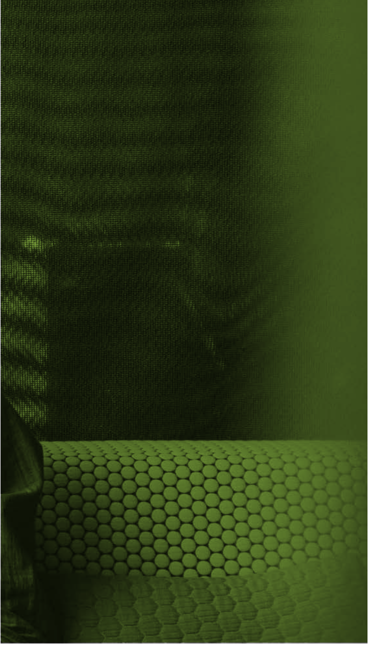
Create an environment where everyone fully contributes to problem-solving and everyone is fully engaged, and your team may have fewer rewrite fantasies.

Why Your Programmer Just Wants to Code

When I interviewed Jamie for a position at ZenTech, he seemed like an enthusiastic engineer. With solid tech skills, ideas for process and product improvement and a great team attitude, he was the obvious choice.

But, two years later, Jamie was “that guy.” You know, the one who wants to code without being bothered.

I should have noticed the signs. He didn’t speak up in retrospectives, he didn’t contribute process or product ideas like I expected, and his “team-friendly” interactions were usually sarcastic. He often talked about technical debt, our lack of innovation and the “stupid” decisions holding us back. An irritating “I told you so” sentiment plagued his comments and feedback.



No matter what the reason, dismissing or devaluing your programmer's ideas – especially in the first few months – is a bad move.

Jamie may have thought about leaving the company. If he did, I couldn't tell. Although, I certainly wished he would have. But we were shorthanded, and I needed all the help I could get.

The result?

Another cliché programmer who just wanted to code and be left alone.

People Are Shaped by Environment

Too many managers believe the problem in this scenario lies with Jamie. If he was a better employee, dedicated worker, or at least cared more, then this wouldn't have happened. Right?

Unfortunately, no.

The transition from enthusiastic programmer to polarized programmer doesn't happen overnight. But it starts sooner than you think.

The First Suggestions Matter - A Lot

How you handle ideas from new programmers sends an important signal. Good or bad, it sets the stage for what they expect. This determines whether they share more ideas in the future… or keep their mouths shut.

Sure, some ideas might not be feasible in your environment. Some might get put on the back burner to be discussed "*when we're not so busy*." Some ideas seem great, but they run against unspoken cultural norms.

No matter what the reason, dismissing or devaluing your programmer's ideas – especially in the first few months – is a bad move.

Damaged by all the naysaying, he'll try a few more times to present his ideas differently, aiming for a successful outcome. If he continues to feel punished, though, he'll realize that the only way to win is to not play.

Which is exactly what you don't want your programmers learning. He will stop presenting ideas, asking to meet customers, and genuinely trying to understand the business.

Ultimately, it's a lose-lose.

The Bigger the Idea, the Bigger the Risk

Remember that your programmer is taking a risk when they offer a new idea. The bigger the idea, the bigger the risk.

Why is it a risk? Because our ideas reflect ourselves, our views and our passions. We don't advance ideas we don't care about or that we don't think will work. We put forth our best ideas with the hope they will be received.

This requires vulnerability, which only happens if we're fairly certain we won't be humiliated. If we believe our ideas won't be accepted, we stop offering them.

Feedback About Ideas Shapes Behavior

It's only natural, then, that your programmer is reduced to doing only what brings him success: coding.

His enthusiasm for creation, innovation and development, sadly, are lost. Perhaps it transforms into unrealistic ideas about code quality or code metrics. His concern for market share and business health is replaced with a concern for titles and pay scales. He becomes more worried about how much he earns, what his title is and how he looks on LinkedIn. His enthusiasm for changing the world is replaced with nit-picking the development process.

Worst of all, though, his concern that "We aren't building the right thing" will be replaced with "We aren't building the thing right." He's learned to not give input on what is built, so he becomes obsessed with how it's built.

Your culture, for him, has become survival of the fittest.

What's Your Onboarding Teaching?

While you would never say this directly, your onboarding and culture may be teaching some pretty negative lessons:

"Our company doesn't like big ideas from little people."

“You just focus on building stuff. We’ll figure out what the customer needs.”
“You are just a code monkey.”
“Why are you asking so many questions? Don’t you have coding to do?”

What Is Your Real Culture?

Culture isn’t the slogan on your wall, or how you describe your mission during an interview. Culture is the way people actually act and what they actually care about.

Texas A&M Professor Ifte Choudhury states, “A culture is a way of life of a group of people – the behaviors, beliefs, values, and symbols that they accept, generally without thinking about them, and that are passed along by communication and imitation from one generation to the next.” [7]

If you wonder what kind of culture you have, start watching how people behave. If you don’t like what you see, change it. Culture isn’t dictated. It’s learned, modeled and imitated. As a leader, it’s your job to be worthy of imitation. Because the culture isn’t Jamie’s fault. It’s ours – Team Leads, Software Managers and CTOs.

So, stop blaming Jamie and start making the changes that your culture demands. The sooner, the better.

[7] http://people.tamu.edu/~i-choudhury/culture.html

A Wake-Up Call for Tech Managers

The most popular article I've written is called "Why Your Programmer Just Wants to Code." The article tells the story of Jamie, a programmer who joins a new company full of enthusiasm and ideas. Fast-forward a couple years, and Jamie is one of those programmers "who just wants to code." A programmer who doesn't contribute new ideas, doesn't offer new ways of doing things, and just wants to be left alone to write code.

Sadly, I received almost no response from managers or leaders about this story.

It appears some of you missed the point, so let me be blunt.
Tech managers, these situations are your fault.

You must accept responsibility for unmotivated programmers who “only want to code” or who appear to be concerned only with flashy, new tech. As the leader, you are responsible for creating an environment where everyone can contribute to solving the problems at hand.

Instead, it appears many programmers are treated like idiot savants: brilliant children capable only of coding.

Stop it. Seriously.

The massive response to this article should scare the hell out of you.

The market is hot, giving your developers the power to resign and replace YOU with a better leader.

I get the sense that they are mad as hell, and they aren’t going to take it anymore.

Don’t believe me? Read on…

I wrote the article in hopes that leaders would recognize that programmers want to bring their whole brain to work but that too often the environment prevents them from doing so.

Instead, I received thousands of responses (claps, comments, messages) from programmers who wish their managers would pay attention. Who wish their culture welcomed, discussed and debated ideas.

Here are some of the comments that stood out to me…
“OMG, preach! This “idea/feedback negging” phenomenon is the deadliest innovation killer of all the land, and it hurts every department (not just a coder problem).”

“I came in to the workforce guns blazing, ready to make a difference. Now, I struggle to suppress my true thoughts every day and just deal with how things are… I REALLY hope leaders start figuring this out soon.”

“I went through something similar, I even stopped working on my pet projects because coding at work was sucky and demanding, glad I just left them after 5 months!”

“It’s sad that actually the coding culture in my current environment is that programmers are merely interested in finishing a task instead of thinking of sharing ideas.”

Another reader takes a slightly different view.

“What we need is not ‘acceptance’ of ideas. We need to see our ideas discussed and debated, and then see that the decision is based on the merit of the idea, not our position. If I present an idea and it gets discussed and then dismissed, that’s ok. If I present an idea and just get told that I should not do that and just focus on my current assignment, well then that is a clear signal that I’m not allowed to do anything but comply with orders like a grunt. When that happens, I’ll basically be looking for the next job.”

That last one sums it up. Create an environment where your programmers can fully contribute, or the best ones will leave.
Let’s get real. If you see this problem on your team, it’s not going to

fix itself overnight. BUT, you can take huge steps toward reversing this problem.

Let's change things.

Start Listening and Stop Telling

If you've got developers who just want to be left alone, today is a great day to turn things around. Start in your next 1:1 with them. (Don't have 1:1s? Start. Today.)

Step 1: Be humble.

Have a reset conversation with each team member, where you honestly ask if you've been deaf to their ideas, treating them like "resources," and frustrating them. Regardless of what they say, tell them you don't want to be that sort of boss, and that you're sorry. (Yes, it is good to apologize to people you may have hurt. Yes, even if you're the boss.) Next, tell them you need their help. You rely on their feedback to improve. Give them permission to stop you the next time you do it. Give them permission to give you feedback in private about your behavior. Finally, thank them for being there and for all their hard work. Thank them for listening and for helping you grow into the kind of manager they need.

Step 2: Listen more, tell less.

In all your interactions with the team, talk half as much. Then, half as much again. This will probably catch them off guard, especially if you've been "leading from the front" and telling them what to do. Instead, listen to how they talk to each other. How do they talk about customers, bosses or other teams? Who controls the flow?

Who's still trying to put ideas out there? Who seems completely shut down? See if you can get everyone to participate through open-ended questions. Consider using a "talking stick" if some people suck all the air out of the room. Gently set expectations that you want to hear everyone's contribution to solving problems.

Step 3: Ask more often than tell.

Most engineering managers take the default approach of telling engineers how something should be done. This is probably because they used to be engineers, and they "see the answer clearly." Yet, telling doesn't build the team up – it shuts them down. So, start asking more questions. Lots of WHY? questions. Of course, you have to tap into your curiosity and lean in to hear the answer. Truth be told, you depend on them to get the work done, and you depend on them to make a million small decisions. You should be very interested in what they think and in bringing their ideas to light. The book Humble Inquiry by Ed Schein is a wonderful resource for learning to ask better questions.

Tech leaders, you've got work to do. You'd better start pulling all-nighters to fix things, before it's too late.

Let's Stop Learned Helplessness in Software Engineering

Over the past 24 hours, two of my articles, "Why Your Programmer Just Wants to Code" and "A Wake-Up Call for Tech Managers," received over 96,000 reads on Medium and over 900 Reddit comments.

It appears we have a bigger problem than I thought.

Yes, we have some bad managers at tech companies. And yes, I was hard on them, placing the blame for programmer apathy directly in their laps.

But I want to help programmers who feel frustrated, disenfranchised or shut-down. It was you – programmers – who posted the vast

We teach managers how to treat us by what we tolerate.

majority of comments, describing terrible conditions and awful management. You raised your hand to say, “I’m tired of this.”

Starting today, let’s change this. We, as programmers, have accepted this behavior. We teach managers how to treat us by what we tolerate. Let’s get real. We can’t change everything alone, but we have more power than we think. Here are some actionable ways you can radically change your work environment.

Recognize That Your Manager Wants to Do a Good Job

Chances are good your boss used to be an engineer, just like you. Tech companies embrace the Peter Principle, and they promote people to the level where they begin to fail.

It doesn't help that your manager probably received less training than your barista. (In fact, a survey I conducted with a fellow management consultant showed that 76% of tech managers receive less than 8 hours of training for their role!)

No one is born knowing how to manage or lead. The phrase "natural-born leader" is crap.

Many tech managers wonder if they are "cut out" for their job. Honestly, most of them wish they were still programming!

The transition from programmer to manager is hard, so let's start by giving managers a break. Yes, they need to change. But who doesn't? No one is perfect. So, let's find ways to change the environment so we can improve, together.

Your Manager Works in a Bad Environment (and So Do You)!

Both you and your boss are part of an environment with problems. Chances are, your boss didn't create the environment. In fact, they likely feel as victimized by bad environments as you do.

Tech leaders almost never control…

- What features / bugs programmers work on
- How much programmers get paid
- How much vacation programmers receive
- What benefits they can offer programmers
- Where programmers sit and what type of computer they use
- If and when programmers can work remotely
- What languages and frameworks are used

Usually, the company culture and environment dictate those things, and it frustrates the hell out of your boss.

Recognize that no PERSON is stopping you, but the ENVIRONMENT might be. Bad environments have smells, just like bad code has smells. Here are some environmental smells you might notice:

- Quiet (or not so quiet) talk about what they would improve if they were in control
- Lack of curiosity about project value and outcomes
- A feeling of powerlessness to make improvements
- A feeling that “everyone else can screw-up, but we are held accountable”
- Continuing to do things “like we always have” for fear of making a mistake (and being punished)
- Lots of talk about change; very little actual change

And many more…

I’ve worked in oppressive environments, and it’s stifling. It feels like you can’t breathe. It leaves you frustrated and angry. So, what can you do about the bad environment?

Leadership Isn’t Granted, It’s Grasped

Think about the folks you work side by side with for a moment. Are there any that you’d describe as an informal leader? (Companies appoint Managers, but leaders seem to pop up everywhere.) Watch these people closely. What do they do differently? Why do you suppose they are doing it? How do others respond to them?
At my first job, I noticed that a co-worker, Milind, acted a bit differently than others. He was an informal leader on the team, even though I didn’t realize it at first. For example, Milind asked key questions about WHY particular approaches were taken. He

admitted his ignorance in a group setting and asked the boss to explain ideas more clearly. He never hesitated to call the customer to clarify or even to negotiate a requirement. He stayed hyper-focused on delivering real software rather than just discussing his tools. Finally, he insisted that we understand the root causes of problems.

Milind changed our environment by his actions, and he certainly changed my view of "Software Engineering." He even changed how our boss acted! He showed me that I had much more power than I thought… if I would only stop expecting to be spoon-fed everything. His actions showed me that true leadership isn't granted, it's grasped. With time, I changed my behavior, and others did as well. So can you.

Talk About the Environment, Not Just the Process

Most teams discuss their agile process. What worked, what didn't, how can things improve. That's what an agile retrospective is for. The best teams regularly talk about it and even make changes to improve things. Great teams also talk about their environment.

They spend time discussing environmental issues, like:

- How people are working together
- How much they trust each other
- How emotionally safe the environment is
- How communication can be improved
- How people can help each other more and receive help more easily
- How they can become better problem-solvers
- How ego and self-image impact the way they work

Bad environments have smells, just like bad code has smells.

Begin to make your environment a topic of discussion, and you may well see things change. You don't have to get approval from anyone for this. Don't expect your boss to hold a meeting entitled "How to Improve the Software Development Environment." Don't propose it for the next team meeting.

Just start talking about it. Start questioning assumptions and bringing up environmental topics in your retrospectives. The more you talk about it, the more others will too.

Schedule a One-on-One with Your Boss

If you don't have a regular one-on-one with your boss, ask for one. You don't have to wait for them to initiate it. Most people feel awkward doing this, but every time I've done it, my manager was thrilled I asked.

See, most managers see the value in one-on-ones, but they suspect you don't. Asking for one changes the dynamic completely.
Send a brief agenda in advance and come prepared. (Unsure what to talk about? Grab a copy of the One-on-One Framework for some ideas.)

A sample agenda might be:

1. Discuss upcoming projects
2. Receive feedback from manager
3. Report on current projects
4. Offer feedback to manager

In your one-on-one, consider telling your boss that you're working to change yourself and to change the environment. Tell them that you want to be a more effective engineer and to create a more effective team environment. Tell that that you know tech skills are only a part of what makes a good engineer and that you want to improve your leadership skills.

They won't feel threatened. You aren't trying to take their job. You're trying to do your job better.

Every manager wants self-led programmers and self-managed teams. That's the promise of agile, right? Your efforts will make their life easier.

Give this a try, and let me know how it goes.

“

Empowering your team takes more work, not less work.

The Surprising Misery of Empowered Teams

My cousin is a remote worker at a company that embraces “employee empowerment.”

The CEO doesn’t try to make the decisions, he leaves that to the team.

He trusts the team to do the right thing.

Sounds good on paper, right? But the reality is miserable.
For example, my cousin spent two hours last night trying to “get on the same page” with four co-workers over slack about what date to publish a blog post he’d written. Not about the content of the blog post, or even the headline or call-to-action, but about which date it should be published on their site.

Why?

In his words: "The CEO tells us we're empowered, but that feels like an excuse to never show up or be available to us. He tells us that he trusts us to 'do the right thing,' but no one has any idea what's right. Worse, because no one knows what's right, everyone just argues for their opinion. We're told to 'get everyone on the same page,' so the arguing continues as people tire and drop out. The last man standing wins."

That isn't empowerment, it's chaos.

Needless to say, he's looking for a new job.

Empowering your team takes more work, not less work. Do the work to empower your team. Not sure where to start? Ask them; they know what they need.

Advanced PeopleOps - 1:1 Retrospectives

Carmen's heart sunk as she looked at her calendar. Back-to-back 1:1 meetings filled her day, overflowing into the next.

"Ugh… maybe I could call in sick. Or make up an excuse to work from home. My boss wouldn't care. My team would be thrilled to skip them."

"It's not too late, you can still call in sick," she thought as she stood in the Starbucks line, "but then what kind of boss would you be? It sucks, and everyone hates it, but you have to do it."

"Sheesh, what are we gonna talk about? I guess I'll just ask people what they're working on this week, and hopefully I can get each one done in 5 minutes. Oh! Or maybe we could do them in small groups!

That would take SO much less time."
"I'd better order an extra-large coffee with quad shots… I'm going to need it."

Apply What You Already Know

I'm going to share a head-smackingly simple lesson that has served me well. Ready?

Make every fourth one-on-one meeting a retrospective to discuss improvements to your one-on-ones.

This is similar to a sprint retrospective, and you can use the same format. The point of a sprint retrospective is for the team to improve. The point of this retro is to improve your one-on-ones, making them more valuable for both of you.

That's it. Go do it.

But if you need a nudge…

Here are five steps to help you start:

- Let each team member know that the next one-on-one meeting will be used to discuss your one-on-one meetings.
- Ask them to write down what's working for them, what's not working and ideas for change. You will do the same.
- During the meeting, discuss what you both wrote, just like in a normal retro.
- Brainstorm a list together of possible actions that will improve the meetings.
- Choose a few actions, again together, to try for the next three meetings, and then discuss them in your next one-on-one retro.

Simply start talking about your 1:1s with the other person and discuss how they could be better.

What If You're Not the Boss?

What can you do to improve a one-on-one that is inflicted on you? Here are some simple, but maybe not easy, ways to broach the subject with your boss:

- Forward this article to your boss, with a note that you'd like to try 1:1 retros.
- In your next 1:1 meeting, ask if you can take a few minutes to discuss how the meeting can be improved.
- Brainstorm a retro-style "glad-sad-mad" list about the meeting and bring it to the next meeting.
- Ask your boss what the real goal of the meeting is and whether they feel this format is working.
- Let your boss know that the current meeting format frustrates you and that you'd like to discuss changing it.
- Tell your boss the way you feel about your 1:1 meetings, and then ask how they feel about them.

Talk about what's really happening. Stop pretending your 1:1s are great, or that they can't be changed, or that you're benefiting from them as much as you could be.

Best case: The meetings will improve, your boss will appreciate your initiative, and you'll do better work.

Worst case: Your boss says "No, things are fine as-is. How dare you suggest they could be improved? Just give me your status update." (If the worst case happens, you have bigger problems than crummy 1:1 meetings.)

New possibilities bring hope that the future doesn't have to be like the past.

What Kinds of Things Can Be Changed About a 1:1 Meeting?

It's easy to fall into a rut with your 1:1 meetings, like an old married couple can fall into a pattern about how they spend Friday nights. Here are ten things about your 1:1s that you could change but might not have considered. (There are surely many more, but this should get your creative juices flowing.)

- How often you meet (it doesn't have to be the same frequency with each person)
- What time you meet (it doesn't have to always be at the same time)
- Who runs the meeting (how could you take turns running the meeting?)
- Where you meet (consider a walking meeting or a breakfast meeting)
- What preparation both of you do for the meeting (try more, or less, prep)
- The agenda for the meeting
- The goal of the meeting
- The length of the meeting
- The communication medium (face-to-face, telephone, slack, Skype, etc.)
- What you could combine it with (a meal, a walk, a commute, etc.)

Do You Wait Too Long to Consider a Change?

When I start to feel in a rut, I ask myself some questions. In particular, I ask myself if my current practices still fit with my current situation or reality. Often, I find that this one question allows me to be more agile, more creative and less judging. It lets me see new possibilities that I'd been missing.

For example, if I was dreading my 1:1s, I might ask questions such as...

- What is my secret goal for these meetings?
- What is my spoken goal for them?
- What is the other person's goal for them?
- What is the company's goal for them?
- What would happen if we stopped doing them?
- What parts are valuable to me and which feel like a waste?
- What parts are valuable to them and which feel like a waste?

- What is the least we could do and still have a valuable 1:1 meeting?
- What do we need to add? Subtract? Change?

Grandma's Ham

Albert Einstein summed it up pretty well when he said, "The important thing is not to stop questioning."

Jane asked her mother, "Why do you cut the ends off the ham before baking it?"

Her mother answered, "Because that's how your grandma taught me to do it. Ask Grandma."

When Jane asked her Grandma, she replied, "My roasting pan was small, so I had to cut the ends off the ham to fit it in the pan."
It's hard to question the status quo.

Not just because you want to avoid looking dumb, or rocking the boat, or breaking tradition, but because you may not realize there's a question to ask.

New possibilities are wonderful. New choices and options do exist about how we work together. New possibilities bring hope that the future doesn't have to be like the past and that we can grow and improve.

What a great thought!

Now, you might be thinking of someone who might be feeling stuck that needs to hear this. Go ahead and forward this article it to them.

The Myth of Many Hands

Fred Brooks is a rebel. In his 1975 book T*he Mythical Man-Month*, Brooks observed the impact of adding people to late software projects. The simplified version of his observation became known as Brooks' Law: "Adding human resources to a late software project makes it later."

I've heard this law quoted often but only in jest by programmers who failed to meet a project deadline. As they look back over the project, searching for root causes of the failure, this law is often trotted out as a sort of joking excuse. Yet, after 25 years working in and with software teams, I have yet to hear a manager of a late project say, "Hmm… The very worst thing I could do is add more people. Brooks' Law cautions me that adding people will actually

Brooks' Law: "Adding human resources to a late software project makes it later."

make the project later!"

I've also never heard a manager of a late project look back on it and comment, "My big mistake was adding more people to the project. That just made things worse."

Unfortunately, it appears that Brooks' Law has become an internet platitude, a moral saying that contains truth but is largely disregarded.

Why Brooks' Law Is (Mostly) Ignored

"Many hands make light work" – John Haywood (1497–1580)
It's counterintuitive. When we're young, we learn to work in various situations. Usually that work is done with our hands, arms and feet,

and it's usually completed faster when more people work together. For example, raking a large yard of leaves is faster with four people than with two and weeding a flowerbed is faster with two people than with one. This assumes, of course, that there are enough rakes and trowels for everyone to have their own.

Early on, we learn a formula for work that looks like this: "If X people take Y days to finish Z tasks, then (2X) people will take (Y/2) days to finish Z tasks." It's even a common math story problem in school! Now, writing software doesn't seem much like raking or weeding, but that old formula is hard to shake. After all, it's so darn logical! In actuality, many of us secretly embrace a different law than Brooks' Law, which we could call Skoorb's Law: Adding human resources to a late software project helps it finish on time.

The name might sound silly, but since it's how we act, it might as well have one.

We're under pressure. A senior director of project management at a Fortune 1000 company interviewed for this story reports, "Executive management's first question about late projects is almost always: 'Can we add more people to the project?' Our answer is usually, 'No, the nature of the project is such that adding people won't get it done faster.' While this might not be a popular answer, we understand why they ask the question. Adding people may be the most intuitive move to take, yet it's often a bad idea."

When projects are late, there's immense pressure to take steps to get the project back on track. When this pressure comes from non-technical executives, it can be difficult to resist – and futile to explain that software "doesn't work that way." In many organizations, doing something is better than doing nothing, as it shows we're working hard to fix the problem. Additionally, it feels safe to add people to a late project, because it's hard to imagine it will make things worse. Yet that's exactly what Brooks' Law predicts. Recall that Brooks' Law does not state…

- Adding human resources to a late software project has no effect.
- Adding human resources to a late software project has a marginal effect.
- Adding human resources to a late software project has an unknown effect.

Instead, it states that adding human resources to a late software project has a negative effect. This one management activity, meant to help, will cause the project to deliver even later than expected. The law is at once profound, provocative and a bit upsetting. Someone got lucky (or thinks they did!) Companies have their own mythology. Stories about project successes often find their way into the cannon of myths and become local legends. Like many such laws, Brooks' Law isn't absolute. Managers regularly add people to late projects, and some of those projects finish on time. As they say, luck happens.

Unfortunately, confirmation bias might be working against us. Confirmation bias is the tendency to interpret information in ways that confirm our own beliefs. This can easily impact how we view the outcome of a project. For example, if Mary decides to add a programmer to her late project, and the project finishes on time, she may see this as a confirmation that Brooks' Law is a fallacy. The truth is that, before GitPrime, it was very difficult (impossible?) to measure the impact of a team member added late in the project, either good, bad or indifferent. Yet confirmation bias, past luck, and local legends can quickly create a belief that Brooks' Law is an antiquated axiom from the past.

Loopholes

Fortunately, there are some loopholes in Brooks' law, and once you know them, you can use them. First, if you see a project is off track, add resources quickly. Adding people at the 25% or 50% mark allows them to come up to speed and spend time contributing to the project. Fred Brooks cited the time it takes for a new programmer to come up to speed, or ramp-up time, as an oft-overlooked factor. While adding more people early in a project may appear wasteful (you may not be certain yet that you need them), there is always the option to remove them later if they aren't needed.

This brings us to the second loophole: padding. Pad the number of people you think you need by one or two, just in case. And of course, pad the project schedule. Estimates, after all, are just guesses. As Jerry Weinberg[8] told me once, "Consider your best-case estimate for when the project will be done if everything goes perfectly. Have you ever seen anything go perfectly? No, this logic tells us that your delivery date must be later than that."

Add only "the best" people. I don't mean the rock star programmer, or the smartest person on your team. "Best," in this case, refers to four very project-specific attributes:

1. Has worked with the team on significant projects in the past
2. Has good relationships with team members
3. Knows the code base
4. Knows the problem domain

These four factors will have a huge impact on the outcome of adding someone to a project, especially a project that's already underway (read: late).

Late projects are often full of chaos, miscommunications, stress, responsibility hand-offs and a frantic pace. This is the worst possible environment for creating new trust relationships and getting a new member to gel with a team. In fact, one of the reasons Brooks' Law is true is that the team will spend a great deal of time trying to bring the new person up to speed and figuring out whether they can trust them. This puts everyone farther behind.

Adding someone the team knows and trusts, maybe someone they have worked with before, is your best bet. In fact, early in the project is an excellent time to ask the team, "If we could add someone to the team, who would you like that to be? What could they do to contribute to the project?" The team is in the best position to know, and they'll be most accepting of a new person if they have input on the decision.

Roles and Work Partitions

Pulling weeds or raking leaves are examples of work that can be partitioned between people. Of course, any parent with two kids and one rake knows a shortage of tools can be a recipe for trouble. Only one kid can take the rake at a time, which causes strife between team members (and everyone else!). If you do choose to add a new member, discuss with the team which pieces can be partitioned and determine whether you need to obtain more rakes so everyone can be productive. You might feel you know exactly how to divide the work, but your team will probably have better ideas.

Watch Out for Your Team's Feelings

Don't forget that the addition of a new project member also sends signals to the team about how others perceive them and their work on the project. You may be sending the message that they're somehow not good enough. This makes it doubly important for the new person to be "pulled in" by the team rather than "pushed in" by management. Pushing in a new team member without the team's buy-in may send the signal that the team is not trusted by management, is not capable of finishing or is coming up short in some other area. This can create resentment or hostility, resulting in a dramatic loss of focus and productivity.

Brook's Law reveals fundamental truths about building software:

Technical teams are highly dependent on each other, the work is often tightly coupled and difficult to partition, and positive trust relationships among team members are a key factor to success.

However, Brooks' Law is not a constant – like the speed of light – but a guide – like highway speed limits. Following it will keep you out of trouble most of the time, but in special circumstances, it's appropriate to break it.

[8] http://www.geraldmweinberg.com/Site/Home.html

Professionals aren't content with generalities or vague requirements. They stop and ask for specifics, even at the risk of looking dumb.

The 4-Letter Word That Makes My Blood Boil

"JUST"

It's one of the worst four-letter words I know. Whenever I catch myself using it, I stop and apologize. And when I hear it, I hold up my hand and stop the person speaking.

Let me give you some examples from last week…

"Just put a form up to collect their e-mail…"
"Just make it so they can login with Facebook…"
"I'll just throw it in a new database field."
"We can just launch a new database server…"
"Let's just let them post notes, like Twitter does…"
A synonym I often hear is "simply."

"Let's simply use Redis for this…"
"We'll simply spin up another AWS server…"
"It should be simple to reuse the Atlas library for that."

If you use the words "just" or "simply," you might have forgotten how hard the technical details can be. I cover how to fix this in Chapter 2 of my book, 7 Habits That Ruin Your Technical Team.

Or, you might be pushing the team too hard and glossing over the details. That's covered in Chapter 5 of the book.

What if you're not saying it, but you're hearing it?

Then it's time to stop the conversation and politely ask for the missing details. This used to be hard for me because it made me feel like I was asking "stupid" questions. For many years, I felt that if I asked people to explain what they meant, I'd look dumb. Or unprofessional. Or I'd be wasting their time.

I finally realized that professionals aren't content with generalities or vague requirements. They stop and ask for specifics, even at the risk of looking dumb. They have the confidence to know they aren't dumb and to not pretend to understand something they don't.
You can use phrases like…

"Let's pause so I can clarify what you mean. Are you suggesting that we…"
"Wait, before we continue, can you explain that feature more?"
"Going back to what you said, can you explain how you would implement that?"
"I might be a bit slow here, but can you explain?"

Lullaby Language

Jerry Weinberg calls “just” an example of Lullaby Language, which “lulls your mind into a false sense of security, yet remains ambiguous enough to allow for the opposite interpretation.”[9] He groups it with words like “should,” “soon,” “very” and “trivial.” All perfectly nice words that we see every day, but words that can carry a lot of hidden ambiguity and assumptions.

How I Learned to Stop the Conversation

My boss, Milind, was great at this. When I was promoted to Team Lead, I was brought into a whole new world of meetings and discussions, and I would keep my mouth shut when someone used the word “Just” or spoke in vague terms. I didn’t want someone to think I wasn’t fit for the job or that I was having trouble keeping up. Instead, I nodded and smiled and tried to look like I was tracking with them.

But Milind knew it was dangerous to accept generalities or misunderstandings. He would stop a large group conversation with the phrase, “Maybe I’m missing something here, but can you explain that in more detail?” Everyone would look at him, the speaker would pause and then back up to cover the “just” part in more detail.

And lo and behold, 90% of the time it was revealed that the person who glossed over the details had oversimplified something important. Or was wrong about an assumption. That means 90% of the time we were able to correct the discussion in the moment and move forward with better information.

And the 10% of the time there wasn't a problem? The explanation clarified everyone's understanding and we quickly moved forward. Or it opened the door to other unspoken questions from the group. Watching Milind do this made me feel confident enough to try it. Now I do it often, as really understanding what someone is telling me is the most important thing. It allows me to correct misunderstandings and assumptions in the moment instead of wasting time working in the wrong direction.

Now It's Your Turn

How often do you hear the word "just" or "simply" and nod in agreement?

How could you pause the conversation and change it to move in a different direction?

How often do you use these words yourself, especially when setting expectations or defining requirements?

[9] https://www.humansystemsinaction.com/lullaby-language/

Your Agile Assembly Line Workers

In 1913, Henry Ford installed the first moving assembly line in the world and reduced the time required to build a car from 12 hours to 2.5 hours.

Ford's assembly line used a motorized conveyer belt to move products through a set of workstations. Each worker had a very specific job, and that was their only job. Today, it seems like the most logical thing in the world, but it was a radical and difficult change for workers.

How Things Were Built Before the Assembly Line

Before the Industrial Revolution, most manufactured products were made individually by hand. A single craftsman or team of craftsmen would create each part of a product. They would use their skills and their tools to create the individual parts. Then, they would assemble them into the final product, making cut-and-try changes in the parts until they fit and worked together in a process often referred to as craft production. [10]

You might think that workers loved the assembly line. Less walking around, more focus on the task at hand, tools and parts located nearby, and so on.

Maybe some did, but many more hated it. In fact, the Henry Ford Museum has an entire collection of letters from workers (and their spouses) about the terrible working conditions and the effects of the new moving assembly line. Amanda Ross, of the Henry Ford Museum, has this to say:

"Each task was timed to determine how long it should take. The assembly line was set to move at that pace. Speed was the key. If a worker had 6 seconds to complete a task, then he had to get it done on time every time. Whether he was ready or not, the next car chassis would be in front of him in 6 seconds.

Hours upon hours of performing the same, mindless task was very difficult for the workers to accept. Morale was often low. Also, line work – due to its quick pace and repetitive nature – was dangerous. In 1916, the Ford Highland Park plant recorded almost 200 severed fingers and over 75,000 cuts, burns and puncture wounds.

It was the tyrannical motorized assembly line that sparked the unionization movement that accompanied the industrial revolution. Aroused and angered by impossible production-line speeds and work standards, serious safety and health concerns, fears of unemployment, and overly abusive foremen, the United Automobile

Workers Union was at the center of the social and economic revolution associated with the rise of industrial unionism." [11]

The Tyranny of Velocity

One of the huge changes Ford's workers had to adjust to was the relentless pace at which they had to work. Remember, before the assembly line, they swarmed or mobbed around their work to complete it. This was called craft production.

How each person performed was less important than how the team performed as a whole.

Production metrics were simple and valuable for everyone: the number of cars built per day was what mattered.

But the new assembly lines never slowed or paused. Management decided how fast they should go after observing and timing workers at each station. Six seconds here, 20 seconds there, 42 seconds at the next station…

Managers decide how work is partitioned and what activities are performed at each station.

Managers decide where each station is placed and how they are sequenced.

Managers decide what tools and raw materials each station will use. Managers determine quality standards and metrics for each station.

Managers decide how many seconds each station has to complete the work.

That's a lot of managerial decisions inflicted on workers.

Imagine Going Through This Change

You've spent years assembling cars and learning how to assemble many different parts of the car. Sure, you love the interior detail work, but you're as good at installing engines as anyone. Working together with your team, you're proud to build a complete car each day. Everyone on the team can take credit for the final product. Your team's goal is clear: one high-quality car per day.

Then, one day, the plant changes. New equipment is installed, and you have the title of "Windshield Installer I." You are escorted to your workstation where you are told you have thirty seconds to attach the ten bolts that secure the windshield. You are expected to do five hundred windshields per day. The work comes at an overwhelming pace from a motor-driven assembly line. Your only concern is attaching five thousand bolts per day.

As an experienced car assembler, you see quality problems zoom by, but no one listens when you point them out. Plus, every second you take pointing them out means working even faster to catch up and keep up.

After all, a new windshield arrives every thirty seconds, no matter what.

It's easy to see why workers were left feeling worthless and unappreciated. And why unions formed and workers demanded better working conditions. In fact, it appears assembly lines haven't changed much.

Your Agile Assembly Line

Too many Agile teams feel they're working on an assembly line, trying to keep up with the pace set by management. This pace, usually called a team's "velocity," may create a virtual assembly line. Unfortunately, this is turning your Software Engineers into Agile Assembly Line Workers, and the negative effects are similar to those experienced by Ford's workers.

Like Ford, you might pay them well, but your workers will be stuck with poor working conditions. And like the assembly line workers of the past, they may decide to unionize, revolt or lash out in other ways. In fact, my guess is that they already are, even if you haven't noticed.

Your Challenge Starts Now

If you lead software teams and some of this smells familiar, I dare you to share this story with your team and ask, "Is this happening here? Does it feel like an assembly line?"

This takes tremendous bravery, and it may come at a personal cost, but I encourage you to take the risk. This challenge might be hard for a number of reasons: a) you're afraid of the answer, b) you don't think you can change things even if it's bad, or c) you're a clueless boss and your ego has blinded you to the truth.

Your team knows whether you have an Agile assembly line. Ask them collectively or in your one-on-ones and start collaborating with them to improve their working environment. Your developers – and your customers – will thank you.

---

10 https://en.wikipedia.org/wiki/Craft_production

11 https://51154787.weebly.com/negative-impact.html

Why Software Quality Is So Confusing (And How We Can Fix It)

If your team strives to build quality software, I applaud them. If your team has created specific standards that ensure quality, then they are among the top 15% of developers in the world (based on my experience.)

But if your team recognizes that quality is always subjective, personal and political, then something amazing has happened. Not quite there yet? Read on…

See, quality is subjective, not objective. There is no such thing as objectively high-quality software, yet it's a common fantasy most programmers and managers hold to.

Quality is subjective, not objective. There is no such thing as objectively high-quality software.

Defining quality is making a statement about what a person values at a particular time. Some examples might help:

Novice users may value software that is easy to learn.

- Experienced users may value software that has many shortcut keys.
- Developers may value software built with modern tools.
- Managers may value software that is ready when it's needed.
- Clients may value software that costs as little as possible.

Quality is also always political because we cannot make everyone happy all the time. Instead, we must pick and choose whose definition of quality we will use. This requires negotiation, prioritization and discussion between the various parties.

The First Step to Improving Quality

First, stop talking about quality in absolute terms. Don't allow your developers, users, clients, customers, managers, founders or sales folks to use fairy-tale language about quality. After all, who could disagree with someone who states "*We need to build good quality software.*" That statement will get everyone's head nodding, even though no one has any idea what it means.

However, it's even worse than that. Everyone will have a different idea of what it means. Having different ideas about what quality means is more harmful than having no idea what quality means. When this happens, each team member will strive to produce software that matches their personal, unspoken definition of quality. When you hear someone talking about quality in absolute terms, stop the conversation and ask whose definition they are using. Then teach them what you've just learned about software quality. Then you can have a productive discussion about what quality means to the team, the clients, the users and the managers.

An Illustration from Gardening

When I was twelve, I was a reluctant gardener.

Every year my father would order a dump-truck load of mulch to be delivered to the front of the house, near the road. Then he'd hand me a shovel and a couple of wheelbarrows. The algorithm went like this:

1. I would shovel mulch into the wheelbarrow to fill it.
2. I would push the full wheelbarrow 150 feet to where my father was spreading mulch.

3. He would give me the empty wheelbarrow.
4. Goto Step 1

My father's definition of quality was simple:

1. Don't spill the mulch on the ground when you shovel it.
2. Have a full wheelbarrow to him about the time he was out of mulch.

For many reasons, I did not enjoy this job.
To make it tolerable, I created an additional definition of quality:

1. Fill the wheelbarrow using the fewest number of shovelfuls of mulch.

My quality rule made the process more interesting and fun for me. It also made me feel in control of a situation I didn't enjoy. I thought I could add my rule to my father's quality rules without any impact. It turns out, that was false.

Counting shovelfuls don't take much time, but creating heaping-huge-heavy shovelfuls did. Moving heaping-huge-heavy shovelfuls without spilling caused me to work more slowly. Worse, no matter how slowly I moved the heaping-huge-heavy shovelfuls, some mulch spilled on the ground. I was aware of these tradeoffs, but I thought "No big deal. Look how big my shovelfuls are!"

It wasn't long before my father noticed that his Quality Rule #2 ("get mulch to him quickly") was consistently being violated, so like a good manager, he decided to check on the worker. He was quite surprised to see me very slowly moving a heaping 16" tall shovelful of mulch into the wheelbarrow, spilling about 25% of it on the ground in the process.

Surprised, and upset.

I’ll leave the rest of the story to your imagination, but needless to say, my Quality Rule got tossed.

Why We Get Confused About Quality

Let me present the concept of a packed phrase.
Words such as quality, performance, complexity and scalability are packed phrases. Packed phrases are packed with so much potential information that we miss what’s important, rendering them useless without unpacking them.

Like lullaby language, packed phrases give us the impression we’re clearly communicating even when we aren’t. When you hear someone use a packed phrase, here’s a single, tiny question you can ask to start the unpacking process:

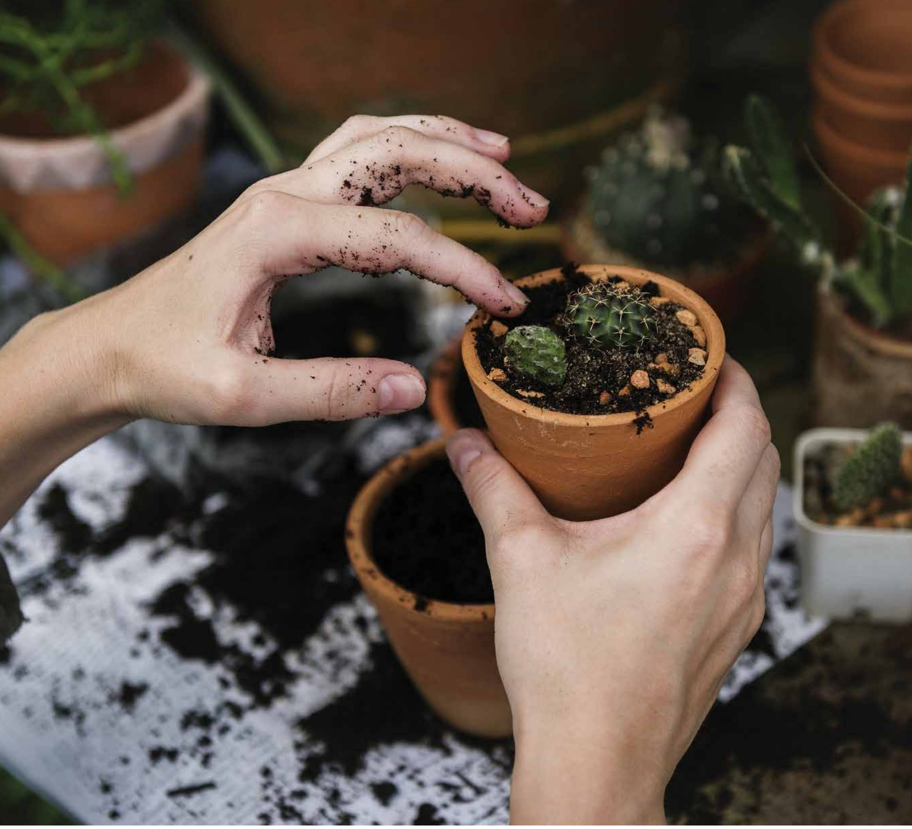
“Compared to what?”

For example…

“The system needs to be fast.” “Compared to what?”
“The legacy code is filled with bugs.” “Compared to what?”
“The app needs to scale.” “Compared to what?”
“Our process sucks.” “Compared to what?”

Asking this question usually receives one of two responses:

- Information about the attribute the speaker feels is important
- A blank stare



Both responses are useful. More information is always useful. But sometimes new information also contains packed phrases. Don't worry, just repeat the question: "Compared to what?"

After doing this two or three times, people will catch on and strive to give more concrete ideas. Then you can continue your discussion. And if you get the blank stare?

Just for you, I'm going to break the rules and reveal one of the Secrets of the Highest Order of Software Consultants.

Don’t report me to the grand poobah, okay?

If you get the blank stare… stare back at them.

Stare until it gets awkward, which it will!

Don’t glare, just look.

If no one has spoken after you’ve silently counted to 1000, ask: “*What’s happening for you now?*”

This should jiggle them a bit and allow you to resume the discussion.

Your goal is to help them become more specific in how they think (and talk) about these abstract concepts. They are probably right that something isn’t good, but you can’t fix it without a clear definition.

Once you stop being content with ambiguities, they can stop, too. This is a huge win–win for everyone.

Why do you value potential customers more than existing customers?

The Trap of Sales-Driven Development

The CTO of a leading SaaS approached me to get help with his software team.

“They have too many bugs,” he said.
“They don’t hit their estimates,” he said.
“They don’t plan their features well,” he said.
“When things go wrong, they blame the product owners. And the product owners blame them right back.”

After thirty minutes of discussion, we uncovered a few things:
The QA team was being pushed around by the engineering team, who wanted to ship software fast.

The engineering team was being pushed around by the product team, who’d been told to create new features.

The product team was being pushed around by the sales team, who felt they could only close a sale if they promised new features to each prospect.

The sales team received commissions on new sales but not on existing customers.

The customer service team was measured by existing customer satisfaction and the number of people who left the platform. Finally, I asked, “Why do you value potential customers more than existing customers?”

Shocked, the CTO realized that’s exactly what they were doing.

The SDD Cycle

We might call this system Sales-Driven Development. The sales team demands a constant stream of new features that only serve to close sales deals. What does this have to do with the problem of “too many bugs”? The current customers are left with engineering leftovers because the teams aren’t aligned and incentivized to serve them.

Follow this logic for a moment…

“Why do they have so many bugs?”
“They don’t have time to fix them.”
“Why don’t they have time to fix them?”
“They have to hit the estimates.”
“Why did they give those estimates?”

“The product team already promised them to the sales group.”
“Why did the product team promise them?”
“To close a sale, the sales team promised a potential customer it would be done soon.”
“Why did the sales team make that promise?”
“The customer had to check a box on their RFP.”
“Why is that feature important?”
“We have no idea.”
“How will it be used?”
“We have no clue.”

Bingo.

That is the root of many problems. Teams building software that no one will use, without a clue why they’re building it. They have been relegated to digital ditch diggers. Shockingly, the engineering team still cares about the product and customers! In fact, their biggest complaint is that they aren’t allowed to fix bugs and make small improvements that could benefit the thousands of paying customers they already have. Even in the midst of a terrible system, engineers still exhibit a strong drive to build high-quality software that really helps people.

What to Do?

Now that you see who the engineering process is actually serving (the sales group), decide whether that’s who the engineering process should serve. Once you know that, align financial incentives between sales, customer service, product and engineering teams. Once that’s done, get the heck out of the way. Because when everyone is pulling in the same direction, things will happen faster than you can believe.

Past situations, environments and people taught them how to act.

The Unexpected Danger of Typecasting in Engineering Teams

Does this sound familiar?
Your team has a Wizard programmer who solves hard problems but lacks follow-through.

Or maybe a Firefighter who only delivers under pressure.
Or maybe you have a front-end programmer who “can’t figure out SQL,” or a Lone Wolf who insists on working in isolation until their code is perfect.

These types of programmers can be great… but they can also drive you crazy. If you’re like me, you’ve found yourself wishing they were less stereotypical and trying to figure out how you could get them to act differently than their “native type.”

Maybe you've also thought, "But what can I do? That's just who they are. A leopard can't change its spots."

I used to think that way, too.

But after fifteen years of leading software teams, I've come to recognize that it does more harm than good to typecast programmers.

The biggest danger of typecasting is that it limits the feedback you can offer members of your team. After all, you wouldn't ask a leopard to become a lion. Or a hearing-impaired person to "listen better." You wouldn't ask me, who stands 5-foot 5-inches tall, to become 6-foot tall.

Asking those things is both pointless and offensive. They are characteristics that cannot be changed. Our culture is very sensitive about offending people. Suggesting that people make any change always risks offending someone.

But when your Wizard flounders with the "boring" parts of a project, or your Lone Wolf avoids collaborating on a feature, it's not because they can't do those things. It's because they won't. And the reason they won't is simple. Past situations, environments and people taught them how to act. We learn from experiences, finding what leads to success (or failure!).

Okay, once again.

Your Wizard can learn to successfully work on boring projects.
Your Lone Wolf can learn to successfully collaborate.

Your Firefighter can learn to successfully plan their work.

And anyone who can learn #&^$ CSS can certainly learn SQL!

Each person can learn to change, but they need your help to do it.

You help them by giving them feedback, not by keeping silent.

You can do this by coaching with words like:

- "I know you can do this."
- "Here's how you can approach this."
- "Here's how your team would benefit."
- "Here's how you'll personally benefit."
- "Let's talk about one small step you could take."

But as long as you believe "that's just the way they are," you'll never ask them to change.

Think about your favorite teacher in school. When you struggled with a subject, did they say, "Oh, you'll never figure it out. You're just not a math | English | science type"? Probably not. Instead, they probably said, "Keep practicing. Don't give up. You've got this." They knew that the material could be learned by anyone who applied themselves, not just by certain kinds of people. These teachers were probably your favorite because they pushed you to be better, and they believed you could achieve more.

In her book Mindset, Carol Dweck, Ph.D., describes two distinct mindsets that impact how we see ourselves: the fixed mindset and the growth mindset. Someone with a fixed mindset sees limitations. Someone with a growth mindset sees challenges.

Someone with a fixed mindset believes that if they are good at something, it's because they were born that way. Someone with a growth mindset believes that if they are good at something, it's because they learned through effort and practice.

Your job as leader, manager and coach might be to believe what the other person does not believe. To envision what they cannot yet see. You need to have a growth mindset about your programmers, even if they have a fixed mindset about themselves.

Personally, I've found that people who believe in me have a tremendous effect on my belief in myself. Have you found this as well?

They Didn't Type Themselves, You Typed Them

While some people will loudly declaim about what type of person they are, your team members (probably) didn't label themselves with these types. In fact, they might not agree with them at all. In that case, you may have developed a fixed mindset about them. In this case, it's you who needs to change perspectives.

It's easy and convenient to see people through the lens of stereotypes, and it appears useful at first. After many months of unsuccessfully prodding my Wizard to finish a project, I approached my boss about how to handle the issue.

My boss told me, “Jim is a Wizard, and you need to keep your Wizards happy. Give the project to Jane to finish, and let Jim play with something else. You don’t want a team full of Wizards, but you do need one.” Unfortunately, my boss wasn’t right this time.
It’s easy to imagine that great software teams are assembled from a combination of types, like a recipe, and that all you have to do is find the right proportions:

1 Wizard
2 Lone Wolves
1 Firefighter
3 Ninjas

Unfortunately, not only is that silly, it’s harmful.
Yet I’ve personally talked to hundreds of engineering leaders who, when trying to figure out how to deliver a late project, resist the idea

of asking someone to work outside their type: “I can’t ask Diane to help deliver the front-end, she’s more of a Wizard.” Of course, they can ask Diane to help deliver on the front-end, and she may do a great job. But they never know because they never ask.

Types Are Not Strengths

Some leaders confuse types with strengths. Strengths-based leadership, popularized by Gallup[13], is not the same thing as typecasting. Each person brings strengths and weaknesses to a team. Leaders who only play to people’s strengths never give them the opportunity to grow. People grow when their team allows them to use their strengths and challenges them to improve their weaknesses (and supports them in doing so). The acknowledgment that no one is perfect and that everyone brings both strengths and weaknesses to the table creates an environment where it is safe to work and learn. Perfection is not expected, but growth can be achieved.

In the same way, some leaders confuse types with communications styles. Communication styles are formed early in life. Personality tests, such as the DiSC Profile, [14] may help teams understand each other better and improve internal communication. Yet, these are very different from harmful programmer stereotypes that prevent us from offering constructive and truly useful feedback.

How to Begin

If you have withheld feedback because you’ve typecast someone, it’s time to think again. Start by considering your own expectations:

- What unspoken expectations do you have of them that you don’t have of others on the team?

- What unspoken expectations do you have of others that you no longer have of them?
- What opportunities could you give them to grow in a new direction?
- How can you reset your expectations, and possibly their own, and encourage them to think outside of their type?
- Is it time to apologize for typecasting and clearly discuss your expectations?

While the trends of micro-specialization may continue in the job market, you don't have to continue to typecast your team members. Instead, discuss and negotiate expectations with your team and provide opportunities for them to grow.

With your coaching and encouragement, your programmers can do the best work of their lives, which will make you the best boss they've ever had.

[13] https://www.gallup.com/press/176588/strengths-based-leadership.aspx
[14] https://www.discprofile.com/what-is-disc/overview/

9 Lessons from Teams Who Anticipate Learning

Does your team eagerly look for opportunities to learn? Do they frame failure and success as opportunities to grow?

Here's a true story about what happened when we taught coders to learn from "play" experiences…

Round One

The Sassy Vagrants were frantic. Their four-foot tall house of cards was falling over. They refactored, reinforced and reworked the tower, but still it sagged and swayed. They cheered each other on, working faster and faster… until time ran out.

They had met all the engineering requirements except one: their tower must reach four feet high and stand without support. That was the most important requirement.

Taking their hands off the tower, they watched it slowly topple to the floor. The look on their faces said it all: “We failed.”

Meanwhile, the Cutting Edge was confident, even… cocky. They’d met their four-foot mark with apparent ease and were now adorning their tower with enhancements. Their tower started life as a square that rose from the floor, but now was, in their words, using an architecture from “a Japanese apartment building.”

When the time ran out, their tower stood tall, and they all cheered. Everyone cheered with them, basking in their victory.

It’s a fundamental part of the leader’s job to create a place that’s safe to learn, try and grow.

Round Two

In this round, the Vagrants decided for a more traditional approach to architecture, so they spent time looking at other teams designs. This was perfectly acceptable, but it did cost them some time.

Interestingly enough, the Vagrants were the only team who took the time to observe other designs before beginning their tower.

After their design review, they did a bit of sketching and planning, and then they got to work.

The Cutting Edge decided to revise their architecture a bit, but not dramatically, believing their previous design would allow them to reach the new goal height of eight feet.

As they built, their team was quiet. No one spoke. Each person had a job to do, and now it was time to get it done.

That is, until the tower was six feet tall, when a loud “Oh, crud!” was heard.

Their design had reached its natural limits and was swaying dangerously. It was suggested that Felipe, the youngest team member, should hold onto the structure while the rest of the team continued to build upwards.

Their plan was to extend the tower all the way to the ceiling, where they believed they “could create downward pressure on the tower, squeezing it between the ceiling and floor, and allowing it to stand without Felipe’s aid.”

To accomplish this, the Cutting Edge would have to far exceed the eight-foot requirement, as the room’s ceilings were ten feet high. Later they would note that “failure wasn’t seriously considered, because they knew they were good at this.”

The Vagrants chattered as they worked, building, throwing out ideas, revising designs, planning, gathering materials and laughing. While the Cutting Edge remained mostly silent during Round Two, the Vagrants were engaged in continual discussion, encouragement and idea generation.

As the Vagrants' tower rose, one team member would stand on a chair to place pieces, while others would encircle the tower, looking for problems. "Wait, it's swaying." "Yes, that looks good here." "No, hold on, bend that piece a bit more" were heard by the team.

The Vagrants recognized that the person placing the pieces had the worst view of the tower, even though they were the only person "doing actual work." This created a new informal, yet vital, role on the team: spotter. If you weren't placing pieces, you were actively engaged as a spotter, giving feedback to the team.

The Cutting Edge reached the ceiling and let out a WHOOP! Now all that was left was stuffing enough pieces between the ceiling and tower to create downward pressure. Felipe switched arms rapidly as fatigue set in and a team member mounted a ladder and continued to build. Everyone looked on in silence, until the team member announced, "Okay, Felipe, it's safe to let go."

"Are you sure?" Felipe asked. "Yes," the person on the ladder said, "it seems good from up here."

Felipe stepped away, rubbing his sore shoulders.

With impossible slowness, the tower leaned, then quivered, as the bottom pieces strained to hold the downward pressure. The Cutting Edge watched in silence as the tower slowly toppled under its own weight.

The Vagrants reached their goal of eight feet with time to spare and decided to stop. They put a flag on top to celebrate their achievement and spent the last ten minutes enjoying their coffee and snack, admiring their handiwork.

Our Lessons

The above story is true, and the experience generated many lessons for the team members as well as the learning leaders.

Here are some of the learnings we experienced together:

1. Success can fool you into believing you can't fail, encouraging you to take risks.
2. Failure can encourage you to become more conservative and take the time to design.
3. Success can give you the impression "We know what we're doing", which can limit the conversation and ideas brought out during the build phase.
4. Past success may encourage you to take unreasonable risks.

5. Past failures can cause a team to be satisfied with accomplishing the goal and then stopping to rest.
6. The "spotter" role is valuable, especially to success.
7. Individuals can embrace the spotter role once they understand its value, despite the fact that the spotter isn't "building" anything.
8. The person doing the building might have the worst view of the overall project.
9. Teams that feel they know what they are doing don't talk to each other much.

Each of us left with many lessons that day, gained from simple simulations where we trained our eyes and ears to eagerly anticipate learning. It's a fundamental part of the leader's job to create a place that's safe to learn, try and grow. You need not attend a training, workshop or presentation. Your team's work provides ample opportunities to learn each day, if you decide to start looking.

Recommended Reading

Throughout these articles, I mention a number of books that I particularly like. Here's the complete information, along with a few more books I recommend.

Jerry Weinberg, *Becoming a Technical Leader: An Organic Problem Solving Approach* (1986), Dorset House Publishing

Johanna Rothman & Esther Derby, *Behind Closed Doors: Secrets of Great Management* (2005) Pragmatic Press

Ron Lichty & Mickey Mantle, *Managing the Unmanageable* (2012) Addision-Wesley Professional

Kent Beck, *Extreme Programming Explained* (1999) O'Reilly Publishing

Edgar H. Schein, *Humble Inquiry: The Gentle Art of Asking Instead of Telling* (2013) Berrett–Koehler

Robert Fulghum, *All I Really Need to Know I Learned in Kindergarten: Uncommon Thoughts on Common Things* (1988) Villard

Talya N. Bauer, *Oxford Handbook of Leader–Member Exchange* (2015) Oxford University Press

Bruce Tulgan, *The 27 Challenges Managers Face: Step-by-Step Solutions to (Nearly) All of Your Management Problems* (2014) Jossey-Bass

Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering* (1975) Addison-Wesely

Kim Scott, *Radical Candor: Be a Kick-Ass Boss Without Losing Your Humanity* (2017) Macmillan

Carol S. Dweck, *Mindset: The New Psychology of Success* (2007) Ballantine

Camille Fournier, *The Manager's Path* (2017), O'Reilly Publishing

LSP

t: 541.805.2736

e: marcus@marcusblankenship.com

w: www.marcusblankenship.com

Whether it's supporting your developers, holding better one-on-ones, or learning to really listen, technical management consultant Marcus Blankenship's essays, stories, and personal experiences lead the way to better managers, happier teams, and more productive environments. Between these covers, you'll find 20 of his best essays, articles and blog posts from 2018. Leading smart people isn't always easy, but it doesn't have to be hard: Marcus shows you how.