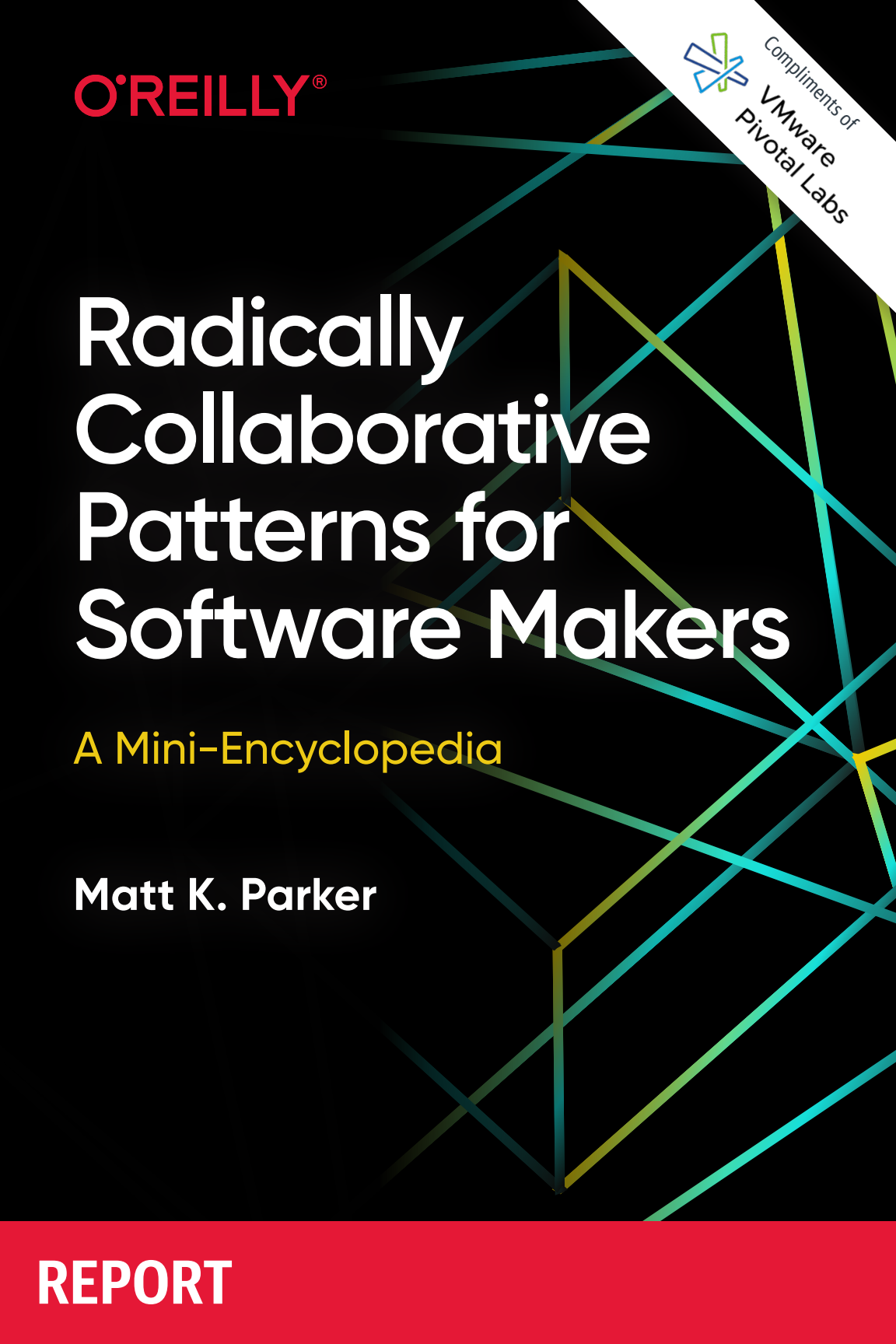
O'REILLY®
Compliments of
VMware
Pivotal Labs
Radically Collaborative Patterns for Software Makers
A Mini-Encyclopedia
Matt K. Parker
REPORT

VMware
Pivotal Labs

Build software a smarter way

Jumpstart app development in an iterative, results-driven way. We help you deliver great apps with proven practices and simple tools. You'll have working software in days, thanks to an approach that starts small and scales fast. Build new apps your customers love and update the ones they already rely on.

Modernize your existing apps
Build innovative new products
Collaborate in a culture of continuous learning

https://tanzu.vmware.com/labs

Radically Collaborative Patterns for Software Makers

A Mini-Encyclopedia

Matt K. Parker

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Radically Collaborative Patterns for Software Makers
by Matt K. Parker

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Developmental Editor: Melissa Potter
Production Editor: Nan Barber
Copyeditor: Octal Publishing, LLC
Proofreader: Justin Billing
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2020: First Edition

Revision History for the First Edition
2020-03-05: First Release

978-1-492-06327-8

[LSI]

Table of Contents

Radically Collaborative Patterns for Software Makers **1**
Introduction 1
Autonomy of Space 4
Balanced Teams 5
Collaborative Story Acceptance 7
Collocation 8
Communal Breakfast 9
Continuous Integration/Continuous Deployment 10
Discovery and Framing 12
Facilitation 14
Free Snacks 34
Information Radiator 35
Iteration Planning Meeting 36
Outsid*er*-In 38
Pair Programming 39
Play Space 44
Promiscuous Pairing 45
Relative Complexity Estimates 45
Rotation 48
Retrospective 49
Team Standup 51
Test-Driven Development 53
Top of Backlog (ToB) 59
User-Driven Architectures 60
User Stories 67
Value-Stream Map 69
Velocity 71

Volatility 72
Workspace Standup 73
Conclusion 75

Radically Collaborative Patterns for Software Makers

A Mini-Encyclopedia

Introduction

This is a little collection of patterns for making software with others. (The word "pattern" in this book simply means a repeatable way of working that makes sense in some, but not necessarily all, situations). These patterns have been observed in organizations that individually subscribe to different and supposedly competing methodologies for making software. Some claim to follow *Scrum*, whereas others subscribe to *Extreme Programming* (XP). Some are *SAFe* certified; others eschewed certifications in general. Because they all tend to disagree with one another about what label to use to describe their "way" of making software—and because they all believe their way is the right way—I've simply referred to them collectively as Radically Collaborative organizations, or RC organizations for short. Perhaps if they begin to adopt this more general, more open way of labeling themselves, they will begin to open themselves up to one another's ideas.

Keep in mind that this book is not an exhaustive list of patterns. RC organizations exhibit many more ways of making software than will be found in this book. Also keep in mind that this book is not a manual. The short essays presented here will not teach you how to practice any given pattern; that is far beyond the scope of this book, and perhaps any one book. This book is simply an attempt to introduce you to some of the most successful patterns seen within RC organizations—and to help you understand how radical these patterns really are.

It's impossible to describe any one pattern in this book without referring to other patterns in this book. That's because the patterns

are differentiated but integrated; they all flow into one another, and sometimes it's not clear where one pattern ends and another begins. When the description of one pattern references another pattern in the book, that reference is indicated in quotes (and is hyperlinked in electronic versions of this book), and that the referenced pattern has its own descriptive essay.

You are not expected to read this book straight through. Instead, I recommend that you browse the table of contents and begin reading the first topic that stands out to you. As you do, you'll come across references to other patterns, and you will naturally begin to jump around the book.

The patterns are arranged alphabetically to make it easier for you to find them, as you jump from one essay to the next. However, the patterns range from high level to low level patterns, and so another way to understand them within the whole is to see them arranged hierarchically. If we begin with high-level patterns—patterns that have organization-wide implications and/or require organization-wide participation—we would begin with:

- "Autonomy of Space"
- "Collocation"
- "Communal Breakfast"
- "Free Snacks"
- "Outsid*er*-In"
- "Play Space"
- "Workspace Standup"

The next level of patterns are team-wide patterns—patterns that either refer to whole-team structures and movements, or which require the participation, or at least the active support, of the whole team:

- "Balanced Teams"
- "Collaborative Story Acceptance"
- "Discovery and Framing"
- "Facilitation"
- "Information Radiator"

- "Iteration Planning Meeting"
- "Relative Complexity Estimates"
- "Retrospective"
- "Rotation"
- "Team Standup"
- "User Stories"
- "Value-Stream Map"
- "Velocity"
- "Volatility"

Lastly, there are patterns that are specific to certain specialized roles. Given that I myself am a software developer, I have chosen to write short essays about certain radically collaborative ways of engineering software that have made a deep impact on me as well as the people I've worked with. I've written them in a way that should be accessible to you, even if you have never programmed. I haven't written about specialized patterns that product designers do, or product managers, or exploratory testers, or data scientists, or any other role that I have no real experience with. That's not to say that I think those patterns shouldn't be written about—but simply that I am not the person to write them. Here, then, are a few patterns specific to software developers that have stood out to me:

- "Continuous Integration/Continuous Deployment"
- "Pair Programming"
- "Promiscuous Pairing"
- "Test-Driven Development"
- "Top of Backlog (ToB)"
- "User-Driven Architectures"

A final thought before you begin. Patterns make sense in certain contexts, but when used outside of that context, they might do more harm than good. In light of this, these essays not only attempt to explain the "what" of the pattern, but the "why," with the hope that knowing the "why" will help you quickly develop an intuitive feel for when the pattern fits the context you find yourself in. This might not only reduce the ill effects of using a pattern outside of its

context, but it might also increase your confidence in trying out the pattern in the first place. Good luck!

Autonomy of Space

RC teams are empowered to shape and reshape their workspace environment.

Who knows best what kind of environment radically collaborative software makers need? Is it the high-priced architectural firms that design "modern" corporate interiors? Clearly, no. Their results are so often bland and devoid of joy, or worse, so strikingly "modern" that they satisfy the aesthetics of only those architects who designed it. So, if it's not the architects, is it the executive leadership of the organization? Also, clearly, no. They are too far removed from the actual value-creating work of their company to understand what kind of environment the makers need.

RC organizations recognize that the software makers themselves know best what kind of environment they need—that the users of the environment must also be the architects and builders of the environment. RC organizations, whether they know it or not, subscribe to a "timeless way of building" originally promoted by the architect and humanist Christopher Alexander. Alexander articulated a post-industrial way of building structures and livable environments based on an ancient, participatory approach to architecture that is responsible for some of the most beautiful human structures on the planet. It is a way of organically creating and evolving living structures that he believed had been largely lost with the rise of corporate and profit-driven construction methods in the twentieth century. Alexander sought to restore our harmony with the environment by empowering communities to decide what to build, and how to build it, and by arming those communities with a catalog of ancient architectural patterns that enriched the structures of old.[1]

Inside RC organizations, you can see a similar process unfold. For example, employees in these workplaces decide where their team "Collocation", what kind of desks to get, and what kind of materials

1 If you'd like to dive further into Alexander's work, I recommend starting with his three-part book series, *The Timeless Way of Building*, *A Pattern Language*, and *The Oregon Experiment*, available from Oxford University Press.

to fill their team environment with. They decide what kind of computers, keyboards, mice, monitors, headsets, peripherals, and so on that they need, and whether to have rolling whiteboards, whiteboard walls, or both. They also decide what kind of "Information Radiator" to buy and where to put them so that their team has a direct line of sight to them. But their decisions go beyond the team space and into the whole office environment in general. They control how many plants to decorate their office with, how much artwork to put on the walls, and what kind of rugs to put on the floor. They decide what kind of games to put in their "Play Space", and what kind of toys and gadgets to have laying around to play with on breaks; what "Free Snacks" to fill their kitchen with, and what types of food to have available at their "Communal Breakfast".

An environment like this is finely tuned to the needs of its inhabitants—because the inhabitants themselves control the environment. Unfortunately, the leaders of most traditional organizations have a difficult time understanding the value of giving their employees autonomy over their workspace. A traditional leader might hear about this pattern and balk at the "cost." If so, you might struggle to convince that leader of the value of this pattern through direct discussion and debate. Instead, I recommend taking them directly into an RC environment and letting them walk though it and experience it. When they experience a workplace environment that is visibly *alive*, that reverberates and resonates innovation throughout, they are much more likely to understand the value of the timeless way.

Balanced Teams

RC teams are leaderless in any formal sense of the word.

RC organizations are filled with something they call *balanced teams*, but to understand what a balanced team is, it's helpful to first look at more traditional software development team structures.

On a traditional team, there is a line manager, typically with a programming background. Everyone on the team (which is composed entirely or almost entirely of programmers) officially reports to this manager; this manager has hire and fire power over everyone on the team. This manager is also responsible for the team's work and, most important, for ensuring the team completes their assigned work on time and on budget. The team is building software according to a specification that has been handed down to it; the line manager

might or might not have been involved in creating the specification. The team typically doesn't include designers or product managers—their work has already been done and handed down to the development team, either in the form of a specification or in the form of pixel-perfect design artifacts. Although all of this makes the work of this team less creative, there are still plenty of times when the team is confronted with situations that no one anticipated when building the specification. In these moments, the manager plays an outsized role. They will take the lead on addressing the situation and might or might not tap others on the team to help out. This traditional team is essentially a software equivalent of an industrial manufacturing process.

On an RC-balanced team, there is no line manager. No one on the team reports to anyone else. There is no specification handed down to them. They're responsible for designing and building a solution (and often, for deciding on the problem to solve in the first place—see "Discovery and Framing"). The team consists of anyone and everyone needed to solve whatever problem the team is tackling. And it does whatever needs to be done, end to end. Research, prototyping, making, deployment, testing. The team puts software, or prototypes, or simply questions in front of users. It learns from those users' feedback, and comes up with new ideas for what to do next. The team does all of this, all of the time.

Team members each have roles. For example, there might be people on the team that identify as product managers or product designers or developers. There might be data scientists, exploratory testers, user researchers, or systems operators. But the roles are fluid. A developer can manage product, and a designer can develop code. A product manager can conduct user research, and a data scientist can administer the system. Of course, they don't have to do any of these things, either. RC-balanced teams are accepting of differences among team members. They accept designers who want to program just as much as they accept product managers who don't.

An RC-balanced team is psychologically safe. The members of the team respect one another for the human beings they are. They listen to and care for one another. If someone has a high need for autonomy, members leave them alone to solve complex problems. If someone has a high need for security, they make plans and then transparently message changes and deviations from said plans. If someone wants to take a risk, the team supports them, even if it

doesn't work out. If someone wants to speak up, they actively listen to their words as well as the emotional meaning behind them.

RC-balanced teams are resilient. They can adapt to new information, even if it causes a dramatic shift in direction. They can grow their team size or they can shrink it, and they can tackle unexpected problems. They can handle failures, and they can learn from them.

Some traditional leaders might initially fear RC-balanced teams because they might see them as an existential threat. RC teams flip traditional organizations on their heads. Instead of the leaders being the experts, the makers become the experts. Instead of the leaders organizing the work of the makers, the makers self-organize their own work. Instead of the leaders deciding what everyone should do, the makers decide what they do. But these leaders, the ones that fear the rise of the RC-balanced team, quite possibly have forgotten why leaders exist in the first place. When an organization begins with one or two people, there are no leaders; they just do whatever needs to be done. They're the original balanced team. And for a time, it's easy enough for the founders to simply hire more people to share the increasing workload without worrying about how to coordinate all of the work or support everyone. But soon enough the day comes when everything has become too big, too complicated, and too chaotic. That's typically when the founders stop doing and start supporting, because that's what real leadership is: support. And that's what some traditional leaders have forgotten.

Collaborative Story Acceptance

> *RC teams accept stories together, instead of leaving it to the product manager.*

After the engineers implement a "User Stories", what next? What needs to happen before the story can be delivered into the hands of users? Many teams, even Agile teams, believe that someone other than the engineers must manually test out the story before it can be delivered into the production environment. But this is really just carrying over the traditional quality assurance (QA) process from waterfall software development methods and grafting it onto an Agile team.

RC teams approach story acceptance differently. Instead of engineers throwing stories over the wall to a product manager for QA, the engineers, product managers, designers, and anyone else

relevant to the process accept stories together. They all sit together and walk through the implementation; they try out the interface, explore its nooks and crannies, and see the new feature within the context of the rest of the software. They approach acceptance in this way because they see stories as *tokens for a conversation—and that conversation extends through the entire development process.* They accept stories together to see whether the vision everyone had in their head when they first discussed the story matches the reality of what's been delivered. And when there's a delta between the vision and the reality, they have a conversation about why and what, if anything, they should do about it. To these teams, it's all about the feedback loop—the conversation.

They also accept stories immediately, as soon as the engineers finish programming. This again is quite different from traditional waterfall environments in which it might take days or even weeks for stories to be accepted. But do you know how much code engineers can build on top of those waiting-to-be-accepted stories during that time? How much code they will potentially need to unwind if the story is eventually rejected? How much time will it take them to pair-up again and context-switch back into the code they wrote several days prior? Delaying acceptance creates waste. It wastes time and money. But it's also unnecessary; RC teams see no point in rushing ahead when delivered stories are waiting to be accepted. They know that to rush ahead and implement more stories would be a false expediency. RC teams don't just want to go fast *now*; they want to go fast *forever*, and sometimes that means slowing down before speeding up.

Collocation

RC teams collocate—physically or virtually.

In traditional IT organizations, the notion of "team" can sometimes become so distorted that it loses all meaning. You can commonly find "teams" in which the members all work on different floors, or in different buildings. They rarely talk to one another except in formal meetings that occur infrequently. Their "team" is actually one of many such "teams" that they are assigned to and in which they divide their time between.

RC teams are real teams (see "Balanced Teams"). The members are dedicated to the team 100% of the time, and they all collocate

together—physically or virtually. When physically collocated, it's very obvious where the team is: they all sit together, share computers together, and generally talk together, all the time. They believe that innovation is a team effort—that it's a process of collective insight that's only possible through the intimate interpenetration of minds and mental models. It doesn't happen when everyone sits behind closed doors, or in isolated "cube farms," working through their assigned tasks. It happens when the team thinks together.

RC teams collocate virtually, too. Sometimes, a team member needs to work from home or needs to travel to a different time zone. Sometimes they even need to live in a different time zone. But it doesn't matter how many people on the team need to be physically separated or for how long. The instant that one of the team members is remote, they all become remote, even if the rest are still technically collocated physically. There's no point in treating the remote person like a second-class citizen while everyone else enjoys the ease of collaborating face to face in the "real" world. Diminishing one diminishes all. So they all go "remote." They all collaborate virtually, through screen sharing, video conferencing, virtual whiteboarding, instant messaging, and the like—no matter where they are. And despite the increased technological hurdles this creates for collaboration, the result is better than if they hadn't. They feel better, think better, act better.

Communal Breakfast

RC organizations start every day with a free communal breakfast.

RC software makers optimize for face time—which means they need to agree on some kind of core working hours. It turns out, though, that agreeing on and committing to a team schedule is challenging. This is true for any organization, but it's especially true for organizations making the transition from siloed workers to radically collaborative teams.

One way to incentivize a shared schedule is to offer a warm, freshly made, wholesome breakfast every morning, and a communal space for sharing it together.

A breakfast like this—particularly if it's offered only for a set, limited time in the morning—works wonders at incentivizing schedule conformance. Employees who in previous jobs were able to come and go whenever they wanted now have an extra incentive to show up at

a designated time. Who wants to miss this amazing breakfast? In this sense, a communal breakfast is a sort of culture hack.

A communal breakfast also has the added benefit of ensuring everyone has the energy they need for the day. RC patterns are an intense experience; knowledge workers tend to increase their productivity and focus by several orders of magnitude in RC environments—making it all the more imperative that they start their days with full bellies! Viewed purely through the lens of "cost," many organizations balk at offering a free communal breakfast to their employees—not recognizing that without it, they're also paying a cost: missed innovation because your workforce is hungry, or missed time, because your workers are running out for coffee and snacks! (See the "Free Snacks" pattern for more information about the effects that hunger has on the brain.)

But there's a more fundamental reason that RC workplaces share communal breakfasts. A communal breakfast really sets the tone for the organization. It gives everyone a chance to foster community and connections beyond their teams. Those connections grow and enhance the neural pathways of the organization's hive mind. They also decrease tribalism.

Continuous Integration/Continuous Deployment

A team that can't ship, can't learn.

There are two questions that you need to ask when considering shipping software:

- Can we ship?
- Should we ship?

"Should we ship?" is ultimately a business decision. Is it valuable to the business to immediately put the latest features in the hands of the users? The product manager represents the business interests on the team and must own this decision.

However, the question "Can we ship?" is fundamentally an engineering question. Is the software in a working state? Are we confident it won't fail in production? The goal of radically collaborative engineers is to always—*always*—have a "yes" answer to this

question. *A team that can't ship can't learn.* And the longer you're not learning, the greater the risk that you're wasting time and money building the wrong thing.

The combination of the following three RC practices make it possible for teams to always have a "yes" answer to the question "Can We Ship?":

- "User Stories"
- "Test-Driven Development"
- "Continuous Integration"

What's a *story*? It's a little description of how a user interacts with the system. It's the smallest piece of user value that you can put in front of users to learn from. You believe that the feature described in the story, *on its own*, provides value to users. It can't be *demoware* or *vaporware*. It has to be real. It doesn't need to be a lot—just enough to test your belief.

If your backlog consists of stories that conform to this definition, and your engineers commit implementations of those stories only after the team (product manager, designers, and engineers) agrees the implementation completes the story, you'll never have any half-implemented features in the build.

But does the software work? Well, we've already talked about how XP engineers answer that question: they practice "Test-Driven Development". Any pair, on any pairing station, at any time, can run the tests to determine whether their copy of the code works—whether all of the features of the product work correctly. But on a big team, you have lots of pairs working in parallel, and therefore the codebase exists in multiple states simultaneously; the tests might be passing on one pairing station but failing on another. That's where we get to *Continuous Integration* (CI): the team needs a single source of truth that it can point to in order to answer the question "does it work?" Every time a pair pushes up its code, a new CI build is triggered. And if the CI build is green, it works. You *can* ship the software. Now the product manager must decide whether the team *should* ship the software.

It's worth noting that some PMs automate their decision with respect to shipping. Some always have the default answer: "Ship on green." In effect, they've asked the engineers to add another step to

their build pipeline to automatically promote code to production on a green build. That's called *Continuous Deployment* (CD). However, although the mechanics of it are facilitated by engineers, shipping is still a business responsibility. The default answer of "ship on green" doesn't abdicate the responsibility of the decision to the engineers. The product manager still has the responsibility of understanding how the features are working in production and how users are responding to it—which means the product manager needs to prioritize all of the engineering work necessary to build automated production monitoring capabilities that make CD *responsible*.

Discovery and Framing

> *RC teams don't build solutions unless they actually understand the problem they're trying to solve.*

In most traditional organizations, software teams build software according to the specifications of some senior stakeholder. The specifications are often handed down to the team with little to no discussion. The problem the software is supposed to solve, the user value it's supposed to create, and the metrics that would allow the team to gauge success are rarely, if ever, articulated. Instead, the team is simply expected to build software according to the specification and deliver it "on time and on budget."

The result is, as you might expect, often dismal. Rarely do teams deliver the specification on time and on budget—but even if by some miracle they do deliver it on time and on budget, the result rarely creates any real user value. Instead, it lands on users with a dull thud, while everyone involved with the project celebrates their "success."

RC teams reject this process wholesale. Instead, if a stakeholder comes to them with a concrete idea or specification for software, they'll attentively listen to their idea, and then ask them, "What problem are you trying to solve?" Nine times out of ten, the person will stop dead in their tracks. In disbelief, they'll say, "You must not have heard me correctly; didn't you hear how cool this software will be?" "Oh, it sounds cool, to be sure," they'll reply, "but if we don't know what problem it's trying to solve, how will we know whether what we're building is actually succeeding?"

This simple line of reasoning—that before you build a software solution, you must first identify, understand, and validate the problem

it's trying to solve—lies at the basis for a whole host of practices and techniques collectively referred to as *discovery and framing*.

During the initial discovery phase, teams will attempt to discover, articulate, and prioritize a single problem to solve, and a user base to solve it for. They'll deploy a host of disciplines and techniques in the process—including assumptions and experiments workshops, ethnographic user research, and "Facilitation" techniques like "dump and sort" and "2×2s." They'll create journey maps to narrate what a typical user is thinking, feeling, and interacting with over time. They'll create and maintain personas—provisional representations of their users based on the team's continuously evolving knowledge and assumptions. They'll create service blueprints and/or event-storming process artifacts in order to map out the technologies and processes users use today to try and solve the problem that confronts them. They'll eventually whittle down a broad range of possible problems to solve to just one.

As the teams begin to frame out a trajectory for a software solution, they'll engage in "design studios" to understand the problem from different perspectives and generate sketches of many possible solutions. They'll develop a lean "business model canvas" to see, at a glance, how users, business, and technology meet to address the problem opportunity. They'll map out potential solutions on one or more 2×2s, progressively narrowing down to a single trajectory. And they'll map out an initial batch of user stories that will help them quickly validate their solution with real, working software (rather than prototypes).

Discovery and framing is a whole-team activity; engineers participate just as much as designers and product managers. And although engineers might have some specific work during discovery and framing that only they can do (e.g., validating the feasibility of various technological approaches and understanding specific technological constraints within the problem space), they still generally attempt to participate in the rest of the activities as deeply as possible. It's only when the entire team deeply understands the problem, empathizes with the users, and collectively imagines solutions, that it truly innovates.

Facilitation

Well-run meetings are fun and make money. RC organizations implement these techniques to make joyful collaboration the norm.

Do you enjoy meetings in your organization? Are they productive, enlightening, mind-expanding events? Most people, in most organizations, would answer "no" to these questions. Meetings are the bane of their professional existence. They are frustrating battles of will; arenas where nothing is accomplished—where people talk past one another, advocating for their own views. Where no one listens. Where the loudest voices prevail.

RC organizations are different. If you walked into a meeting inside an RC organization and observed it for a few minutes, you might not believe that what you're seeing is a "meeting," because what the people in the room are doing seems so very different from anything like the "battle of wills" paradigm you are likely used to.

Why? In a nutshell, RC organizations recognize that meetings are moments for collaboration, for developing unique, collective ideas that were only possible by bringing different egos together into a single room. They don't assume that the best idea walked into the room when it began; they believe that the best is waiting to be found, and it can only be found by harnessing the creative potential that only a collection of egos affords.

So, instead of a freeform "battle of wills," they have disciplined, facilitated affairs that attempt to unleash that potential. And during a single "meeting," the tenor of the room can change dramatically. Depending on when you walked into the meeting room, you might think you accidentally stumbled into a quiet Buddhist monastery or into a noisy party.

Read on to find out how RC organizations do it.

Point A to Point B

Anyone can be a facilitator within an RC organization. And the first goal of any RC facilitator is to understand where the participants are at when they walked into the room (Point A) and also where they want to get to (Point B). The facilitator's job is then to shepherd them from the start state to the end state—effecting as many intermediate states as necessary to get them there.

Point A—where the participants begin—is about more than understanding the current state of their work as a team. It's about understanding where each participant is, mentally and emotionally. For example, imagine that you are about to facilitate a meeting with four participants. One of the participants, Jack, has just come out of an annual performance review with his manager, which didn't go as he planned, leaving him feeling upset and distracted. Jane, on the other hand, has just joined the company, and has the potential to offer an outsider's perspective that no one else in the room can bring. Bob, one of the managers in the room, is under pressure to deliver on an unrelated project, and seems distracted. And Alice has a cup of coffee and a stack of charts and papers—she's ready and engaged.

As facilitator, it's not possible to know all of these intimate details exactly. Some you can deduce through observation; others you might discover by casually talking with people before the session begins. Most important, as facilitator, you must maintain *flexible* mental models of the participants in the room; and during the session, you need to update your mental models of the participants based on their actions and your observations. A facilitator has no worse enemy than a fixed mental model of participants.

Point A is also about understanding the integrity of each of the relationship dyads in the room. Do the participants have good working relationships? Personal relationships? Are there power dynamics in the room that can affect the psychological safety of participants? You not only need to reflect on the mental models you hold for each of the participants, you also need to consider the mental models they hold for one another.

Although many of the facilitation techniques in this pattern help participants let go of their implicit, and often fixed, mental models, they're not a silver bullet. The efficacy of the techniques will often be limited by the reality of Point A—and it's the job of the facilitator to find ways to help the participants *themselves* mitigate these constraints in order to unleash their true potential as a team.

Connecting with One Another as Human Beings

Most dysfunctional meetings can be characterized as a battle of wills—that is, as a battle of implicit mental models. In that light, tribalism is one of the most pernicious antipatterns that plague meetings. When engaged in a battle of mental models, we instinctively

categorize everyone in the room as either supporters or detractors—in-groups and out-groups.

To limit this self-defeating phenomena, facilitators—especially when they expect the meeting's subject to be contentious—must start the meeting by helping everyone connect with one another as human beings: to set aside professional differences, even if only for a moment, to remember that there's more to each of them than what's seen and observed at work.

One simple technique RC facilitators use is to start the meeting by asking everyone to tell the room something about themselves that people in the room don't already know. It could be anything—a personal hobby, a hidden skill, a story from their childhood, an organization that they volunteer with, a love of films, a comic-book collection.

Sharing this kind of information requires the participants to let their guards down; to make themselves vulnerable. And because that's a scary prospect, the facilitator themselves typically go first. They genuinely invite the room to learn something personal about themselves that they had never told their coworkers before.

It's amazing how effective this simple technique is. Once, I was asked to facilitate a two-day session with a dozen senior enterprise software architects from a bank; each of these architects represented different functional silos within the bank. They all operated with a great deal of autonomy, but were being told by the organization that they had to start to work together in order to develop a shared services platform for the bank. Their psychological safety was clearly threatened by this process. Furthermore, many of the architects in the room had professional relationships with one another that had deteriorated over the course of several years. Those troubled relationships were actually a big part of the reason that they had each amassed so much autonomy and power in their roles—their problematic interactions in the past had driven them all further and further apart.

It was clear that two days was a very short amount of time to reach the desired end state (an effective path forward for a shared services platform)—even without the added complexity of their troubled relationship history. In the interest of time, it was tempting to skip an activity that would help them all get to know one another better, and yet it's in situations like this in which it's most critical to make

these types of investments. So we started by going around the room. I began by sharing that I was an amateur juggler, which led another architect to reveal that he had, in the past, gone to clown school, and was a master juggler—a fact that he had never had the courage to reveal to his coworkers. We quickly began to juggle the first objects we could find in the room (whiteboard erasers), stealing from each other in mid-air. His fellow coworkers laughed and clapped, cheering him on, and frankly were amazed at his prowess. They asked him questions about his juggling activities; they learned that his son has learned to juggle, as well, and that they entered competitions together.

Another architect told us a story about recently meeting his favorite musician, causing another architect in the room to audibly gasp, and admit that he was also a huge fan. Another architect told us about the village he grew up in—on the other side of the world. And on, and on. This one activity—the introductions—took nearly an hour. And yet it paid dividends throughout the entire two days. They all saw one another in a new light; they forgot, however briefly, of their troubled professional past. They created bonds that allowed them to truly begin to *listen* to each person's ideas. A necessary level of respect among the participants was born.

An expert facilitator in one RC organization told a similar story:

> In our office, we started a new practice called "product lunch": a forum for both consultants and clients to get together and improve their craft of Lean product management and user-centered design.
>
> During our first session, a consultant began the meeting with a presentation on how a certain collaboration technique works, but the room was deadly silent whenever he asked for questions or input. It was clear that the participants in the room were hesitant to speak up. As the forum was new to everyone, no one was quite sure what to expect or what the norms of engagement were. And although the various clients in the room had all been brushing shoulders and eating a "Communal Breakfast" at the same table with the consultants every day for months, they didn't really know each other. There had been very little cross-client interactions up to this point.
>
> It just so happened that the lunch arrived 20 minutes late, which created a natural break in the session. I used this opportunity to ask everyone to take turns telling the room something interesting about themselves.
>
> The impact was remarkable. Participants discovered a number of common interests in the room, including singing and sports.

> Participants shared interests that even their own teammates hadn't been aware of—teammates that they had worked closely with for months!
>
> One client, a young well-dressed woman who came in every day with makeup and earrings, made the room collectively gasp when she shared that she was part of an air-rifle club. It made everyone sit up and look at each other differently, while reinforcing the point that you can't judge anyone by their looks.
>
> The discussions, after this, became much livelier. The clients opened up and shared their thoughts and opinions on the stressful issues they are facing during their training, and even came up with their preferred topic for the next session: how to manage their stakeholders when they return to their companies, and how to effectively onboard other people in their company into a new way of working, thinking, and collaborating.

So remember: RC organizations have learned that we all listen to one another the most when we see in the room as more than the sum of our professional interactions; when we remember that we are all more than our roles, our jobs, our work; when we connect with our colleagues not just as "professionals," but as human beings.

From Many, to One

Facilitators help a room whittle down many things into one thing. In this section, we describe broadly how this happens, but know that this isn't a technique so much as a guiding purpose for facilitators.

To effectively move the room from Point A to Point B, RC facilitators need to help the room uncover thoughts and ideas. Many of them. And over the course of the meeting, they'll help the group whittle them down until they arrive at a single outcome.

Consider the following example. It's OK if you don't understand each of the individual techniques yet. You can reread this after diving into them. But start by considering the overall movement of the meeting—and the minds.

Imagine that you're helping a team of five people align on their next highest-priority problem to solve for their customers. Left to their own devices, they will likely engage in the "battle of wills" meeting antipattern. So, as the facilitator, you begin by asking them to perform "Silent Generation" of high-priority problems. This leads to 10 problems per person, or a total of 50 problems. As a first reduction

step, you ask them to "Self-Edit", immediately cutting the number of problems in half.

Then, you work with them to Dump and Sort, incorporating a Silent Read. This creates affinity groups for problems, and you work with the team to synthesize each group into a single problem statement. This can also involve the generation of new insights and problem statements now possible through the communion of ideas and collective thinking. Either way, imagine that these techniques eventually reduce the total number of problems to about a third—or eight altogether.

Next, you construct a 2×2, labeling the x-axis as "Harder" on the left, to "Easier" on the right, and the y-axis as "More Important" at the top, to "Less Important" at the bottom. You negotiate the placement of each problem into the 2×2, eventually arriving at four problems "above the line" and four "below the line."

You throw away the four problems "below the line," and facilitate a collective stack rank of the problems that remained. At the end of this process, you arrive at the highest-priority problem to solve.

From individual thinking, to collective thinking. From many problems, to one.

No Laptops, No Phones

Ground rules. Norms of behavior. Every group needs them. Meetings are no different.

The radical collaboration techniques in this pattern require participants to engage with one another in the analog world. They'll use sticky notes and Sharpies. They'll use whiteboard markers and dot stickers. They'll move from individual, silent generation to collective thinking.

But, to achieve this level of collaboration, we must eliminate the distractions that continuously pull our focus away: emails, instant messages, news alerts, push notifications, texts.

The simplest way to eliminate these distractions is to tell the participants at the beginning of the meeting to close their laptops and put them away. To put their phones in their pockets, bags, purses, and not to pull them out unless it's an emergency. Tell them that if there's something critical that comes up that they need to immediately

address on their phone or laptop, that they must leave the room to do so.

Of course, explain why. Be firm—*but empathetic*. Most participants will be surprised; some will even see this as a threat to their psychological safety need for autonomy. Tell them that you understand that "life" happens; things come up. They can leave the room if necessary, no judgement. Even better—tell everyone that this could happen to you, the facilitator. I have, on more than one occasion, stepped out of a meeting I was facilitating in order to take an incoming call from my child's school. Life has a funny way of finding you at the least convenient moments.

It's not just the participants that have norms to respect; you do to, as the facilitator. For example, a facilitator must be seen as an unbiased actor in the room. You can't be a participant and a facilitator. You need to choose. This demands that you leave your ego at the door.

Silent Generation

Have you ever struggled to voice your ideas in a meeting? Perhaps you're naturally shy; perhaps the loud voices in the room drown yours out. Perhaps you are underrepresented in your field and find yourself cut out of the conversation by the unconscious bias of others.

Whatever the reason—you're not alone. An untold number of powerful, vital ideas have failed to surface in organizations because of the hostile arenas in which they are asked to appear.

One of the most essential jobs of the RC facilitator is to help the group take advantage of the diversity of minds present; to be able to consider a situation from as many different angles as possible, in the hopes that through this diversity of sight, a new understanding is born that would not have been possible otherwise.

RC facilitators typically begin this process through *silent generation*. When giving the group a prompt (e.g., "What are the problems our customers face that we could help them solve?"), instead of asking everyone to simply start talking at the room, they give everyone a chance to write down their ideas in silence. They provide everyone with their own pad of sticky notes and tell them to write one idea per sticky. They tell them to write in all caps, with a Sharpie (for readability and conciseness).

Even with this time, people often still struggle to produce ideas—because they suffer from an internal censor. A self-limiting mind-goblin that tells them their ideas aren't good enough; that makes them fear sharing their ideas with others. These mind-goblins are often reinforced by painful memories, sometimes dating all the way back to childhood. A participant might have endured the laughter and ridicule of their classmates in grade school when they had given the teacher a wrong answer to a direct question; this experience had birthed a censoring mind-goblin that continues to plague them to this day. And that censor has grown stronger with each subsequent negative experience in which their psychological need for esteem was threatened.

One of the most effective ways facilitators can silence the mind-goblins is to give everyone an *idea minimum*. For example, an RC facilitator might tell everyone that they need to produce at least 10 stickies; in other words, 10 ideas. Participants are often shocked and wide eyed when given this requirement. Ten stickies? They had just been struggling to come up with even one sticky! But their struggle wasn't due to a lack of ideas: they have plenty of ideas. They're simply being held back by their doubts and fears.

To validate their shock and assuage their concerns, RC facilitators tell participants that their first few ideas are often not their best ideas, and that sometimes the best ideas are waiting to be found, deep beneath the surface of their conscious thoughts. Giving themselves an *idea minimum* will help them get into the flow of producing ideas. RC facilitators remind the participants not to worry about the quality of their ideas; everyone has good ideas and bad ideas, but for this process to work, the participants must overcome their doubts and write them all down.

This process of silent generation accomplishes two important goals: individual reflection and equal participation.

Most people need time to quietly reflect on a prompt in order to reveal their ideas. Instead of immediately hearing the ideas of others, which would bias and limit their own insights, they can instead draw on their own experiences, analysis, and intuition. Exposing the *individual* diversity of ideas within the room is a necessary step toward discovering the *collective* insights of the group.

And secondly, silent generation gives everyone an equal chance to participate. Those who in the past struggled to make their ideas

heard amidst the din of the "battle of wills" now find themselves on equal footing. By starting with silence, their voice will be heard louder than ever before.

One last point: consider giving everyone the exact same color of stickies. When the stickies move from the table to the wall in later activities, the single color of stickies will help the group feel collective ownership over the ideas. Conversely, if you gave everyone different colored sticky pads, you could unintentionally reinforce the "battle of wills" antipattern when participants see their stickies juxtaposed against others.

Self-Edit

The process of silent generation helps participants begin to *separate their ideas from their identities.* But it's only the first step. Though the participants have made their ideas physical and visible, they still feel ownership over them. These are their intellectual children, manifested before their eyes.

So, to continue to increase the intellectual and emotional separation between the participants identities and their ideas, RC facilitators ask everyone to *self-edit*. "If you could show only five of these ideas to the group, which ones would they be?" After everyone has chosen, the facilitators ask the participants to set aside their top five for the moment. "Now, gather up the rest of your stickies; these are the stickies that you didn't choose to be in your top five. Hold them up in your left hand. Pass them to the person to your left. Don't read what's passed to you. Now, rip up whatever was just passed to you."

The reaction is predictable. When groups go through a self-edit for the first time, there's an immediate outburst, ranging from laughter to incredulity. Participants have an emotional connection to their ideas. You might even see some participants go through an accelerated form of the five stages of grief.

As a facilitator, it's important to let this process play out. The group will tend to lean on one another for humor and support. They're all going through the same experience at the same time, and they're all sharing responsibility for it because they're ripping up one another's ideas. It creates bonds of empathy and solidarity among the participants.

In addition to increasing the separation between ideas and identities, self-editing also serves a practical purpose: the participants have simply generated too many ideas to realistically consider in the time that's left. For example, with a group of five people, the process of silent generation generates 50 (or more) ideas. Most teams of five would have neither the time nor the mental stamina to consider all 50 ideas. Through self-edit, you can cut the number of ideas in half.

Working at the Wall

The following techniques (silent read, dump and sort, 2×2, dot voting, and stack ranking) all require participants to work together at the "wall"—so before diving into the particulars of those techniques, it's important to understand why RC software makers "work at the wall" in the first place.

Up to this point, participants have been working alone at their seats. They've silently generated ideas and silently self-edited. Each of these activities have helped them begin the process of separating their ideas from their identities, so that they can share their ideas with one another in psychologically safe ways. But how to share them? How to work with them together as a group? If participants stayed in their chairs and simply resorted to talking about their ideas, the group would quickly devolve back into the "battle of wills."

To work together and to take advantage of the idea/identity separation that's already started, they need to bring the ideas into physical contact with one another. And the wall—a single plane on which all ideas can stand together with equality—represents the most participatory medium for collaboration available to the group.

The act of standing up to work together at the wall creates a clear demarcation line for the meeting—between individual work and group work: from thinking alone, to thinking together. It gives the body and mind a variety of positional attitudes that promotes fresh thinking and new associations. It brings the participants together, giving them the freedom to mingle and collaborate. And, lastly, as the ideas represented on the stickies mingle and merge with the ideas of others—through affinity grouping, synthesis, and so on—it replaces individual ownership of ideas with collective ownership, giving participants the emotional freedom to consider and advocate not just for their own ideas, but the ideas of others. Discussions over

stickies will take place, without the participants even knowing who wrote the sticky in the first place.

Keep in mind that "working at the wall" can be physically tiring for participants because it typically involves standing. Make sure that participants can take frequent breaks and sit when needed. And of course, some participants might not even be able to physically stand. RC facilitators must make sure that participants can "work at the wall" in a way that makes it possible for everyone to participate equally, even if that means not using a wall at all.

Silent read

The *silent read* is typically one of the first activities performed after a team brings their ideas up to the wall. For example, after self-editing, RC facilitators might have everyone place their stickies on a whiteboard. The placement of the sticky doesn't matter for now—they're simply trying to make everyone's stickies available and accessible to the group. They make sure that the stickies spread out enough so that everyone can stand together at the whiteboard and be able to see and read some of the stickies regardless of where they're standing.

After everyone has brought their stickies up to the wall, RC facilitators give everyone a timebox in which to silently read the stickies. There's no discussion allowed during this time; if someone has a question or a concern about a sticky, they can simply take a Sharpie and put a small dot on the sticky. After the five minutes are up, the facilitator can find all of the stickies that have dots next to them and ask the creator of each sticky to explain the sticky's meaning while allowing others to ask clarifying questions.

The process of the silent read creates space for everyone to consider the entire collection of ideas together as well as to quietly reflect on them. It keeps participants from falling back into presentation or "selling" mode for their stickies. The ideas stand on their own. And if people have questions (or concerns) about them, instead of letting the questioner and the creator directly discuss it or debate it, the facilitator acts as an intermediary. The questioner can, and often does, remain anonymous—there's no signature next to the dot they left on a sticky.

Dump and sort

It's not uncommon for multiple participants to generate similar, or even identical, ideas. So, continuing to work at the wall, the facilitator asks everyone to group the stickies together. Anyone and everyone are allowed to walk up to the wall and move the stickies around, grouping them together.

At some point, the facilitator can even empower the participants to circle and name the groupings by giving everyone whiteboard markers. This leads the participants to collaboratively synthesize categories.

This process tends to be iterative. Stickies grouped together by one participant will be pulled apart by another participant—revealing differences in understanding, and leading to a discussion about what the stickies really represent.

And it's often through this process that new stickies—new ideas—are born. Because by exposing these differences in understanding, new categories emerge. For example, a participant might discover that their interpretation of a particular sticky is quite different from the creator's intent—and that their interpretation isn't represented by any particular sticky at all.

Depending on the level of insights and new ideas that begin to emerge through this synthesis, the facilitator can even start a new round of silent generation and self-editing. Although an inexperienced facilitator might see this as counterproductive to their goal of going "from many, to one," an experienced facilitator knows that this is where the real magic of group dynamics emerges. The insights and new ideas that spring from collaboration are the most important to foster and support—it's the entire point of the whole process! We don't meet together to simply decide on who walked into the room with the best idea. Although that's sometimes the end result, our real goal is to discover the *collective idea* that was visible only through the multitude of individual perspectives.

2×2

What if you could wave a magic wand at a set of stickies to discover which stickies are a definite "yes" for the group to proceed with? And of those that remain, which ones are a definite "no," which ones are worth debating, and which ones are nothing more than seductive distractions?

Although magic seems to be in short supply in our universe, we can rely on the next best thing to reveal these categories to us: the 2×2 (pronounced "two by two"). A 2×2 is nothing more than a finite cartesian coordinate plane on which the two axes represent independent spectrums that are meaningful within the context of the stickies.

Imagine that we are again concentrating on "finding the next problem the team wants to focus on solving for its customer." To do so, we can construct a 2×2 for which the y-axis represents how important the problem is to the customer (less importance on the bottom, more importance on the top), and the x-axis represents how difficult the team thinks the problem is to solve (more difficult on the left, easier on the right), as illustrated in Figure 1.

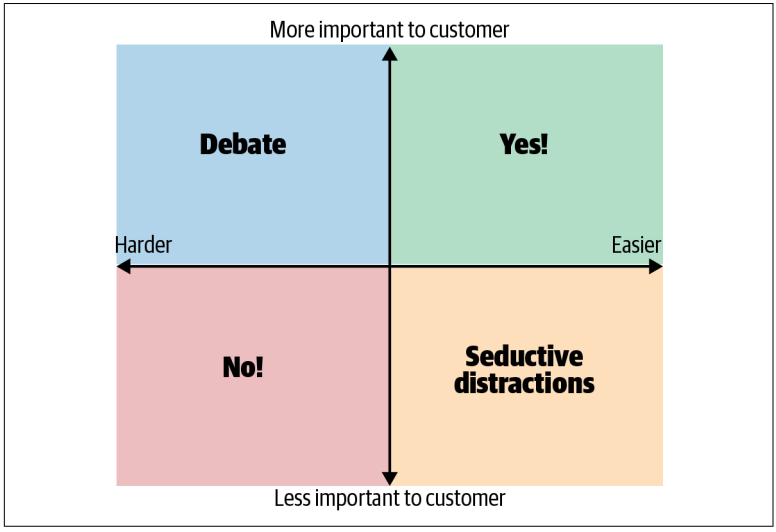

Figure 1. The meaning of the four quadrants in a 2×2

As you can see, the four quadrants reveal the four categories that we sought to expose with our magic wand.

The problems in the upper-right quadrant—those that are really important to the customer and really easy to solve—are the no-brainer problems. The "oh-duh" problems. The problems that the team would look silly for not bothering to solve. They're easy to solve, and they really matter to the customer—so do it!

The problems that make their way into the upper-left quadrant, on the other hand, are equally important to the customer—but they're

more difficult to solve. That doesn't mean they're not worth solving, but given that the team can do only so much work at once, it needs to ask itself some hard questions—are these problems really important to the customer? And, if so, should the team tackle them first, before the easy/important problems?

The problems that make their way into the lower-left quadrant clearly need to be ripped up and thrown away. They're difficult to solve—and they're not very important to the customer anyhow! The team shouldn't waste any time on them.

And the problems that make their way into the lower-right quadrant are the most dangerous of all—they're really easy to solve and therefore tempting for the team to sink its collective teeth into. But they don't actually matter all that much to the customer compared to the stickies above the line. These are the "seductive distractions."

When it works, the 2×2 is an amazing tool. Unfortunately, it doesn't always work out perfectly. Sometimes, a team will debate the relative placement of stickies, only to discover that they don't really have enough data to make a determination about where it should go on the 2×2. It's up to the facilitator to help the team navigate this situation; sometimes the facilitator can push the team to make a decision with imperfect data. (For situations in which you're dealing with a team that's commonly afflicted with "analysis paralysis," this can be a necessary nudge.) At other times, you might sense that this uncertainty is important to dig into. You might decide to use the remaining time to dive deeper into the issue, developing assumptions and experiments that the team can run to reveal data and insights that they're lacking, or validate or invalidate assumptions that they recognize about the sticky in question. Or, you might even decide to set it aside, dedicating an entirely separate session to the examination of that single sticky. There's no "one size fits all" answer in this type of situation. The facilitator needs to draw on their experiences, analysis, and intuition to find a path forward.

Another common problem you'll encounter with 2×2s is that the team will want to place *every sticky* above the line. "Everything is important," members will say. It's a common dilemma, particularly among passionate teams that really care about their customers.

Fortunately, if this happens, there's a simple solution: simply move the x-axis up, thereby positioning half of the stickies "below the line," as depicted in Figure 2.

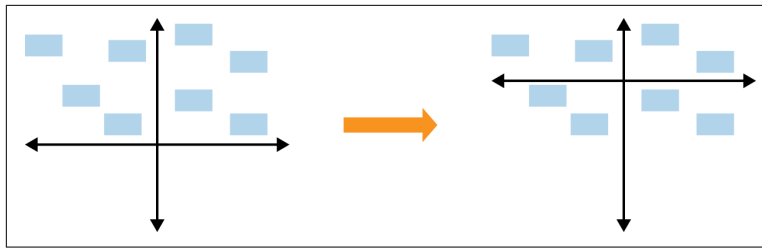

Figure 2. "Fixing" the x-axis

Some participants might object that this is "cheating," but remind them that there were no units on the axes in the first place; that the placement of every sticky was simply relative to every other sticky.

In fact, because this situation is so common, it's often useful to construct the 2×2 axes on the whiteboard with blue painter's tape, instead of drawing the axes with a whiteboard marker. It's really easy to pick up the x-axis and adjust its placement when it's painters' tape! And it also makes for a visual spectacle for those experiencing this process for the first time.

Dot voting

Imagine that you have 10 items on the wall that ideally the group would examine, discuss, and develop further—but you have only enough time to get through one or two. Which ones do you start with?

Dot voting is a simple technique for understanding the items with which the group is most motivated to continue.

RC facilitators often begin by giving everyone three dots. These could be literal dot-shaped stickers, or simply a whiteboard marker for the participants to draw dots.

Next, working at the wall, they ask everyone to silently place their dots on the items with which they're most interested in moving forward. Participants are free to use their dots however they like; for example, if a participant is really keen to see the group move forward with one particular item, that participant could place all three of their dots on that particular item. Or perhaps they see several items they'd like to consider; they can spread their dots among them.

Note that, because everyone can see this process unfold, those that place their dots later in the process are likely to be biased and influenced by the sentiments they already see visually unfolding on the wall.

You might also see one or two participants hang back, waiting for everyone else to place their dots first to see if there are competing items vying for consideration. They then can use their three dots to tip the scales in favor of one of the items.

Stack ranking

After a team has analyzed its stickies with a 2×2, it will wind up with an even smaller set of stickies to work with. The team should gather up the "Yes!" quadrant stickies as well as any "Debate" stickies that it felt strongly about, and facilitate a collective stack ranking; that is, a linear prioritization of the remaining stickies.

Moving from a 2×2 to a stack rank eliminates an entire dimension of thought—we're literally moving from the two-dimensional cartesian coordinate plane of the 2×2, to the unidimensional straight line of the stack rank.

For example, if we're trying to discover and decide on the next most important problem to solve for our customers, we can stack rank the stickies "above the line" to collectively decide on which problem we should tackle first.

Because this can often involve a considerable amount of discussion and wrangling, it's important that we have only a smaller number of stickies to rank in the first place. As a general rule of thumb, don't stack rank until you have seven or fewer stickies to prioritize. If you have more than seven, continue to use other analysis and prioritization techniques to whittle them down.

RC facilitators sometimes begin by sorting the stickies into a single line themselves, using any criteria that seems appropriate. For example, the stickies "above the line" were already sorted into most difficult to easiest, so they might start there.

But after they've placed them, they turn to the group and, starting with the bottom two stickies, ask "does this placement make sense? Is it really more important to work on this one before this one?" Disagreements are often an opportunity to liberate knowledge—to discover that different people know different things about the stick-

ies at hand, and that only by understanding the sum total of the group's knowledge can the team effectively decide on the correct order. RC facilitators repeat this process, working their way from the bottom of the stack rank to the top, until the group reaches an agreement about which sticky deserves top billing. It might matter less whether the order of the remaining stickies is completely accurate—it really depends on the team and its situation.

The top ranking sticky sometimes surprises participants; it's not uncommon for the top sticky to represent something that no participant walked into the meeting thinking was the most important. Typically, this is a really positive sign: when the team believes in the results of its stack rank, even if the result is unexpected, it creates yet more distance between the team's ideas and its identities. Team members begin to truly believe that together they are smarter than any single individual on the team.

But it's also an important moment to pause and sense-check the result; a participant might realize that the team arrived at their result because that participant had failed to mention some critical piece of information.

For example, once I facilitated a session with the executive team of a fortune 50 company in which members wanted to focus on how to obtain a larger "wallet share" for a particular customer segment. During the session, the participants each silently generated a persona that they believed represented this customer segment. It turned out that the participants had many different ideas for who this customer segment really was. They brought their personas up to the wall and then practiced a silent read followed by a dot vote. One of the participants, the CFO, openly questioned this result; she told everyone that she believed another persona made much more sense to focus on—for reasons that no one else on the executive team had known about. However, when the team heard her reasoning, everyone agreed.

In this situation, the facilitator (me) didn't have to nudge the participants into sense checking the outcome. The CFO felt safe and confident in this situation to question the result—even though she knew that the CEO had used his dot votes on the persona with which she disagreed. However, it's much more likely that you, as facilitator, will need to explicitly create space and permission within the session for everyone to sense check their results—particularly for situations in

which there are power dynamics in the room that naturally threaten the psychological safety of the participants.

Affinity Groups

When dot voting, an RC facilitator might decide to proceed with a single item for the entire group to dive into. But they could also decide to break the group up into smaller groups to consider multiple items simultaneously. Obviously, this works best when the original group is large enough to be broken up into smaller subgroups in the first place.

Affinity groups are a natural complement to dot voting. Imagine that, through dot voting, a team has discovered two or more items that have received a significant number of dots. Instead of choosing just one, the RC facilitator could choose two or three of the items, and ask everyone to break up into smaller groups based on whichever item for which they have the most affinity.

RC facilitators must make sure to give very clear instructions to everyone before they break out into groups. This includes giving them a timebox for their affinity group work as well as instructions for how each group will "report" back to the larger group.

Affinity groups tend to work best when the goal of the session is to uncover information, not necessarily make decisions.

Materials and Space

A discussion of facilitation wouldn't be complete without talking about materials. Practically all of the techniques discussed in this pattern require the organization to invest in space and materials to make collaboration possible.

The RC organization needs to invest in collaboration rooms that allow for individual work at a table as well as collective work at a wall (note that most meeting rooms you'll find in most organizations fail to meet *both* of these requirements).

The organization also needs whiteboards in all of the rooms—not little whiteboards, but giant ones that cover the entire wall. And not just one whiteboard, but multiple whiteboards.

The rooms must be stocked every day with copious collaboration materials, including the following:

- Whiteboard markers of all different colors
- Whiteboard erasers
- Pads of sticky notes of all different colors
- Thick, black Sharpies—lots of them!
- Sheets of paper (for activities like persona generation)
- Dots (stickers)
- Name tags

This is just a small list of possible collaboration materials. Ultimately, the radical collaborators themselves should decide on what they want inside their collaboration rooms—because they practice "Autonomy of Space".

Introducing Facilitation into Your Organization

With contributions from Daphne Lin, Weiman Kow, and Carl Coryell-Martin.

As valuable as they are, introducing these facilitation techniques into your organization can unintentionally *reduce* psychological safety if you're not careful.

For starters, any attempt to change the meeting culture and norms within your organization can represent a basic threat to an individual's need for security—that is, an individual's desire for consistency, or, conversely, their resistance to change.

Several of these techniques require participants to share ideas that can be seen as "wrong," which can threaten their need for esteem. Silent generation, for example, encourages participants to generate a large number of ideas, and it's simply not possible to do that if participants are overly concerned with whether others will see their ideas as "good." To risk being wrong requires an environment that makes interpersonal risk-taking possible—in other words, an environment of trust.

Facilitation will also directly challenge the power of individuals in the organization who had heretofore dominated meetings via the "battle of wills" antipattern. They will see facilitation as a loss of control as well as a loss of autonomy. During the "battle of wills," they had been free to act as they pleased, with no real structure imposed on their participation. They could interject at will, veer

conversations off course, or even ignore the battle altogether. But now, they're presented with a facilitator who will guide the group; they'll be forced to ideate silently and synthesize collaboratively. They'll rarely be given the floor to talk at the group, instead being guided through small group activities like affinity groups. In other words, the current organizational culture has predisposed them to resist facilitation.

For all of these reasons, it's critical that facilitators consciously cultivate the psychological safety of any groups new to these techniques, starting with setting expectations. Weiman Kow has discovered that many of the clients she works with come from a corporate culture in which they are expected by their leadership to "always have the right answer," seriously limiting their participation in meetings. In these situations, she carefully and deliberately sets and resets their expectations at each stage of the meeting; for example, she tells participants that for generative activities like silent generation, synthesis, and affinity groups, they should have fun and let their ideas run wild. "It's freeing for them," she explains, often witnessing waves of ideas and creativity flow over the group that they had previously held back out of fear. "I explain to them that their goal is to get into the flow of generating ideas. Your first few ideas often aren't your best ideas. Good and bad ideas will naturally come to mind—the only rule is that you must write all of them down."

Facilitators can also significantly assuage the fear participants face when first encountering these new techniques by *narrating* their experience. For example, when presented with radically different ideas, many participants might naturally feel concerned, sensing a "battle of wills" brewing. A facilitator can defuse this tension by celebrating the diversity of ideas. "Wow, it's really amazing to see how different all of these ideas are—I love the creativity in the room, keep it coming!" As Weiman notes, comments like this, "keep the mood up, building confidence that the room is heading in the right direction."

Similarly, facilitators can help the room connect disparate ideas and detect trends; the participants—particularly when breaking out into smaller groups—need the facilitators help to reconnect to the larger picture. Simple statements such as "We are seeing a pattern here" and "These ideas build on each other" reinforce the movement of the entire group from individual thinking to collective thinking.

With careful facilitation, the fear and reluctance participants feel when experiencing facilitation will fade. The act of joyful, positive collaboration these techniques make possible will erode their fears and win them over. Even the powerful will embrace facilitation, recognizing that their loss of power has been offset by their reduction in stress and the increase in innovation within their teams and organization.

Free Snacks

RC software makers have access to free snacks and drinks whenever they want them.

Hungry brains don't work. You can't think when you're hungry. OK, that's not entirely true—if you fast for 16 hours, your brain will flip a switch and move into a hyper-focused state that helped us survive in the *before time* when saber-toothed tigers hunted us day and night. But within the context of a normal, modern-day sustainable work life, you're unlikely to regularly go to work after 16 hours of fasting, so let's ignore this phenomenon for now.

If you're hungry, your glucose levels drop. And that's not good—glucose gives your body energy, and your brain requires about 10 times more energy than any other organ in your body.[2] If your brain is starved for energy, you can't think; you appear tired and irritable as your "lower," unconscious, primitive brain (the amygdala) increasingly takes the reins from your "higher," conscious brain (the prefrontal cortex).

Now think about the efficacy of a hungry, irritable workforce. A modern organization relies on the creativity, innovation, problem solving, collaboration, and rational decision making of its knowledge workers. If they're hungry, all of that goes away. So, RC organizations created a simple, straightforward solution: give everyone free snacks. At any time of the day, employees can and should be able to reach for a snack if they're feeling hungry. This extends to drinks as well. Free coffee, tea, beer, wine, flavored water—whatever they ask for.

2 Johannes Gutenberg Universitaet Mainz. "Individual stress susceptibility and glucose metabolism are linked to brain function: Perturbations in brain glucose metabolism identified as cause for stress-induced spatial memory impairments." *ScienceDaily*, 24 October 2018. *www.sciencedaily.com/releases/2018/10/181024122411.htm*.

You might balk at the organizational "cost" of providing everyone with free snacks, but consider this: what's more costly? Free snacks, or the loss in productivity, creativity, and innovation that results from a hungry workforce?

This solution works only in as much as the software makers actually want to eat the snacks that are available. Thus, although it's wonderful if they have access to healthy snacks rich in good fats, antioxidants, and vegetables (all shown to benefit the immune system)—just make sure they're not so healthy that no one wants to eat them! RC organizations empower the software makers themselves to decide what snacks to fill the office kitchen with (see "Autonomy of Space").

Note that in a chronic high-stress environment—one in which the psychological safety of individuals is constantly threatened—free snacks, particularly unhealthy ones, can facilitate emotional eating, which can simply lead to more stress. So this pattern, despite its simplicity, is intimately connected to a number of other patterns in this book that promote psychologically safe environments.

Information Radiator

RC teams transparently share the status of their work with anyone who wanders by.

What's the point of hiding the status of your team's work from other people with whom you work? From other teams? From managers, directors, vice presidents? Wouldn't it be more interesting to share the status with anyone who walks by so that they could ask questions if they wanted to? If they're another team, wouldn't it be valuable to share insights, wins, and challenges with each other? If they're a leader, wouldn't it be interesting to share with them the real, unvarnished status of the work that the leader ostensibly supports?

Of course, in a traditional IT organization, the thought of transparently radiating the status of the team's work with anyone who walks by is terrifying. Most traditional teams live on secrecy; on hiding the bad from anyone who could punish them, and the good from anyone who could steal the credit from them.

RC teams, by contrast, live on transparency. They fearlessly share the status of their work with anyone who walks by, because they know that they have nothing to fear. The leader who would punish

them for mistakes is not a leader who deserves their fealty. The leader who would steal the credit for their successes is not a leader who deserves their attention. It helps, of course, that RC organizations as a whole don't suffer from those types of "leaders." But even if one did happen to slip through the cracks, RC teams would just smile at them, laugh, and go on about their business.

One common way they transparently share the status of their team's work is with a *digital information radiator*—like a TV that displays the real time status of their CI builds, their team's velocity and volatility, their work-in-progress in the backlog, their production metrics that tell them how well their application is succeeding at providing user value.

These radiators, of course, also benefit the teams themselves. Members of a team will all be sitting with line of sight to an information radiator. They'll all see the second a build turns red, or a production outage occurs, or a user metric falters. Then, they can fluidly self-organize in order to address whatever the problem is.

Iteration Planning Meeting

> *At the beginning of every week, RC teams sit down and ask themselves: "What should we work on this week? Why should we work on it? How will we know whether our week is successful?"*

On waterfall teams, it's not uncommon to see months (or even years) of work planned out for developers. But RC teams know that the best software in the world isn't built through a process of extensive upfront planning; instead, it's built through learning—by putting software in the hands of users, seeing whether the users truly find it valuable, and adjusting as necessary. Future work is informed by learnings from the previous work. You can't iterate if you can't adjust.

But, deciding on exactly how to prioritize upcoming work based on learnings isn't always obvious or trivial. Prioritization is a *whole-team* effort. You need to combine the ideas, insights, learnings, intuitions, and risk assessments of everyone on the team. And you need to do it often. Every week, a team learns several new, vital pieces of information that inform their work—new insights from users, new priorities from the organization, new challenges and opportunities with technologies, and so on.

So, radically collaborative teams start every week by going into a room and asking themselves the following questions:

- "What should we work on this week?"
- "Why should we work on it?"
- "How will we know whether this work is successful?"

During this meeting, they might pencil in new "User Stories" based on last week's learnings. Or, if they already have a pile of stories to consider, they use "Facilitation" techniques to whittle them down and prioritize them. Programmers on the team will quickly estimate stories based on their relative complexity (see "Relative Complexity Estimates")—after all, the complexity of a story can affect how a team prioritizes it!

RC teams make sure that everyone understands the "why" behind each story that gets prioritized—what is the story's value hypothesis? What outcome is it trying to achieve? And how will the team measure the story's success? So many "Agile" teams forget to close this most critical feedback loop (feedback from users), assuming someone else will do it, or that they'll eventually get around to it. RC teams make sure to bake right into each and every story the ways to measure success on the outcome the story is trying to achieve. And they make sure they have a plan for how they'll follow through on these measurements as quickly as possible (see, for example, "Continuous Integration/Continuous Deployment"). There should be a clear path around the idea-to-feedback circle for every single story, and the team should establish how to make that lap around the circle as quick as possible for every single story.

At the end of the iteration planning meeting, make sure the team has a prioritized backlog, answering the three questions and allowing developers to practice ToB.

Note that although the "Iteration Planning Meeting" at the start of the week produces a prioritized backlog for that week, that doesn't mean that the backlog can't change throughout the week. In fact, on healthy teams, it absolutely will change throughout the week. See the "Team Standup" pattern for more information about the kind of information that a team can learn throughout the week that would cause it to adjust its backlog by mid-week.

Outsid*er*-In

Every week, RC software makers invite outsiders into their organizations to tell them that they're doing it wrong.

Organizations are natural echo chambers. How often do you hear someone challenge the status quo within your organization? Are new ideas valued? Are they safe? When an organization begins, ideas flow freely, as while in its infancy, the organization has yet to develop hard-and-fast perspectives. To survive, the fledgling organization must quickly create value in the world—and that value isn't guaranteed. The organization races, trying out different ideas in quick succession in its search for market fit.

At some point, either the organization fails, gives up, or folds. Or, a *particular idea*, a particular form of value, gains traction. Before this turning point, new ideas were celebrated. They were the lifeblood of the organization, the hope for the future, but now, the *new idea* is seen with suspicion. It threatens to derail the organization's focus from the particular idea that has gained traction. The curious, observant, inquisitive minds, so great at uncovering and exploring new ideas, suddenly find their skills in question as the organization transitions from exploration to optimization.

The longer your organization exists, the worse this problem becomes. Most organizations become hyper focused on optimizing their cash cow, forgetting that the rest of the world is continuing to change and evolve. They forget that just as they once disrupted the market, so too will they be disrupted.

The patterns of your organization cement themselves. And for this reason, every organization is an exercise in inertia, and that inertia creates the seeds for disruption. Thus, in many ways, the most important patterns within your organization are the patterns that promote change, evolution, disruption, transformation: the patterns that will make it possible for your organization to reinvent itself time and again, as it adapts to our ever-changing world.

When everyone in your organization becomes hyperfocused and specialized on the enormous complexity of maintaining and optimizing your cash cow, they have little capacity to step back and see the bigger picture and imagine new forms of value and new solutions. The forest is lost, each worker tending only to their own tree.

But when the leadership of an organization creates a forum for sharing ideas, and when they invite outsiders in to share perspectives that differ from their own, they create an environment in which alternative ideas are not only possible, but *valued*. They inspire a learning culture that in turn empowers organizational innovation and personal motivation.

Every week, RC software makers listen to a presentation from an *outsider*. It might be a fellow software developer from the local community talking about their open source project. It might be the founder of extreme programming, Kent Beck,[3] challenging them to reframe their ideas about software development into his new "3X" model (eXplore, eXpand, eXtract). It might be a former Ruby evangelist, like Steve Klabnik,[4] talking about why he stopped using Ruby and devoted himself to building a community for a very different programming language—Rust.

But regardless of who it is, the goal is the same: everyone, from the CEO down to the practitioner, needs to be challenged by the outsider who has nothing to lose by telling them that they're doing it wrong.

Pair Programming

> *RC teams program in pairs. Two engineers, two keyboards, two mice—one computer.*

Will your project fail if one of your team members is hit by a bus? What's the *bus count* of your team? Admittedly, the "bus count" question is a grim thought experiment. But ask any team this question, and you'll discover that most, if not all of them, have never considered it.

When engineers work alone on production code—a practice called *soloing*—they become knowledge silos. They begin to "own" particular parts of the system. Other engineers fear touching their code; the

3 Kent Beck created one of the first Agile software development practices, known as Extreme Programming, or XP for short. For further information about his ideas, see his book *Extreme Programming Explained: Embrace Change*, 2nd ed., (Addison-Wesley, 2004).

4 After playing an influential role in the Ruby on Rails community, Steve Klabnik went on to work on the Rust programming language. See the book *The Rust Programming Language*, 2nd ed., that he coauthored with Carol Nichols (No Starch Press, 2019).

other engineers don't understand it, and they're afraid they'll break it. The owners of the knowledge silos begin to jealously guard their corners of the system, recognizing it as a form of power. They know that their job security grows proportionally with the size of the silo (this is sometimes referred to as *mortgage-driven development*).

But the cost of the silo is high. It poisons team power dynamics and puts the continued development of the product at risk—a disservice to both the users of the product and the organization. Even the owner of the silo suffers. Creating benefits for yourself at the cost of everyone else can't lead to fulfillment.

One of the simplest, most straightforward, and most powerful ways of breaking down knowledge silos is pair programming.

Imagine a typical scenario in our industry: two engineers, with two computers, two keyboards, two mice, sitting at adjacent desks, writing code all day long—but with their headphones on, never interacting with each other—in other words, they are soloing. Now simply erase one of the computers from your mental image; plug both keyboards and both mice into the remaining computer. Sit the two engineers next to each other at the same desk. Put the monitor between them so that they can both see it. Now they are pairing—giving each other real time feedback and sharing knowledge.

But they do more than program together. The pairing process runs through the entire life cycle of the software development process. For example, imagine that our pair of engineers pull a "User Stories" from the "Top of Backlog (ToB)" and proceed to implement it—together. They talk about the "User-Driven Architectures" that the user experience seems to demand. They decide what behavior to test first, and exactly where within their modular codebase to write the first test. They practice "Test-Driven Development" through ping-pong pairing—one engineer writes a test, the other makes it pass, and together they refactor. They discover that one of them knows more about a particular section of the code they have to work with, but through pair programming, they spread that knowledge equally between them. They each catch edge cases and bugs that the other misses. One engineer calls out a code smell by name, whereas the other suggests a design pattern to refactor to. Every hour or two, they stand up and take a break. One makes tea; one plays guitar. Or, they catch a game of actual ping-pong with other coworkers on break.

Sooner or later, they complete the engineering for the story—but they're not done. They now turn around and show it to their product manager and designers for "Collaborative Story Acceptance". They learn that they had missed the intent behind a particular aspect of the user interface and adjust it with the help of the designers. They deploy it to an acceptance environment, then a staging environment, and then a production environment, ensuring that the appropriate *human*, *proactive*, and *reactive* quality controls are observed at each step in the pipeline.

While analyzing the usage metrics in their production environment, they discover that a large percentage of users are bailing out at a critical stage of the feature. A heat map from the production user interface indicates that some users are never seeing and clicking the button that's necessary for progression at this stage—because it's *below the fold* in two out of the three most popular web browsers due to a styling glitch. They quickly work with a designer to fix the button's location so that it shows correctly in all browsers and redeploy. Afterward, they talk as a team about how they could ensure a problem like that doesn't make it into production again.

As you can see, pair programming isn't just two engineers writing code—it's two engineers collaboratively solving problems and learning together. It creates rapid feedback loops through a sort of real-time code review.

There are many patterns of pair programming; we've already mentioned ping-pong pairing (in which one engineer writes the test, the other makes it pass, and together they refactor). There's also driver-navigator, mute, remote, and others. But one of the most powerful forms of pair programming, expert-expert, is characterized by a state of organic flow—testing, writing, debugging, discussion, shared intuition—that can come only from a deep well of familiarity, empathy, trust, and mastery between the two engineers. The pair of engineers think and act as one in a way that's difficult to grasp when seen from the outside, as if they have achieved some level of telepathy, some Vulcan mind-meld. They finish each other's thoughts, verbally and at the keyboard. One senses when the other wants to take control and so leans back in their chair accordingly, hands receding from the keyboard. One seems two steps ahead of the other, only to smile as the other mentally leapfrogs over them. They end each day exhausted, fulfilled, and energized all at the same time.

This might sound like a beautiful process—indeed, it is! It's a unique experience in the world of software engineering as well as one that's highly fulfilling and inherently addictive. But it's important that you know that there's a dark side to pairing, particularly when transitioning from a traditional, siloed development team to a pair programming team. Pair programming, to those going through this transition, can represent a profound loss of autonomy—one of the three key ingredients for *intrinsic* motivation. On your traditional team, when you were given a task by a project manager, you were also given the autonomy for how to implement it. But now you share that responsibility with another engineer. You no longer do only what you want, you need to write code together, and you must agree on the tests and implementations.

This problem is exacerbated in traditional organizations by a fear of mistakes. Many traditional organizations measure success not by whether a team achieved goals like creating true user value, but by whether they made mistakes: were there bugs in the software, did the team miss deadlines, did it overrun its budget. Organizations that focus on these metrics as the measure of success create a culture of fear, blame, and scapegoating. In traditional organizations, employees lack the psychological safety necessary to think out loud, and if they can't think out loud, they can't pair.

Pair programming brings to the surface all kinds of feelings of inadequacy. In a traditional organization, your worth to the organization directly correlates to your individual knowledge and productivity. In the new world, your worth is wrapped up in your team's ability to iterate and learn quickly. The individual talents of any one person on the team matter much less than the capabilities of the team combined. But at first, the feelings of worth equating to personal prowess will be difficult to shake. When paired with someone else, every time their partner seems to exhibit knowledge or skills that they lack, they will feel real, palpable fear.

For these reasons, any attempt to *force* pair programming on engineers is destined to fail. They will reject it. Pairing must be opt-in. If you're transitioning from a traditional organization to a radically collaborative one, tell everyone that you're creating a new team with pair programming, and ask for volunteers. Give the curious a chance to try it out by letting them pair with an expert pair programmer on a short code kata. An engineer can maintain their feeling of autonomy only if they *choose* to pair. Over time, as teams

practicing pair programming show positive results, many skeptical practitioners will be swayed.

But what of the skeptical managers? When asked to consider moving their teams to pair programming, they'll typically start by asking two related questions:

- Won't everything take twice as long now?
- Won't I need to hire twice as many engineers to get the same amount of work done?

The first question is based on an assumption that programming is their bottleneck. The second is based on the *Mythical Man-Month*[5] fallacy—the idea that the more humans you throw at a software problem, the faster that problem is solved! Yet time and time again, when teams thoughtfully map out their idea-to-feedback cycle, they discover (much to their surprise) that the programming itself isn't the bottleneck. They uncover all kinds of other waste:

- Technical debt
- Backlog mismanagement
- Too much work in progress not actually being actively worked on
- Ambiguous definitions of done
- Overly large stories
- Large teams
- Unnecessary complexity
- Psychological distress
- Context-switching
- Knowledge loss

5 Brooks, Frederick P., Jr. *Mythical Man Month.* (Addison-Wesley Professional, Anniversary edition; August 12, 1995).

- Deployment issues
- Organizational bureaucracy

These are just a handful of challenges that limit a team's ability to effectively build software together. If you'd like to dive deeper into the sources of waste on software development teams, start by reading the research paper "Software Development Waste," by Todd Sedano, Paul Ralph, and Cécile Péraire.[6]

Play Space

RC organizations create time and space within their day to play.

RC is an intensely focused experience—and the human brain can stay in that state of focus for only so long. After a while, the "focus" pathways of the brains tire, and you transition from a collaborative, deliberative, creative state into a uncollaborative, rash, and irritable state.

That's why it's critical to create play spaces in your place of work—spaces that are physically separated, yet easily accessible, from team areas. Spaces that condone, and even incentivize, rest and play.

The RC employees themselves decide what to fill the space with. For example, some of the software makers might be musicians, so they would make sure the space has guitars, keyboards, electric drums, and other instruments. Perhaps some of the software makers love board games and card games; thus, they would make sure that the play space includes games like Settlers of Catan, chess, checkers, UNO, playing cards, and so on. Perhaps there are several gamers in the organization; they would decide to put TVs, gaming systems, and couches in the space. Still others might like a space to practice yoga, or to perform quiet meditation, or to take naps. And so it would be built.

In other words, RC organizations do whatever they need to do to successfully unfocus from their team work (work which, when going well, is really no different from play anyways). When viewed through the lens of "cost," most traditional organizations balk at this

6 Sedano, Todd; Ralph, Paul; and Péraire, Cécile. *Software Development Waste*. IEEE/ACM 39th International Conference on Software Engineering (2017): 130-140. https://doi.org/10.1109/ICSE.2017.20

pattern. But they're playing a dangerous game: those organizations are actively being disrupted by the organizations that understand the real value of fostering psychologically safe environments that foster creativity, innovation, and optimal brain states. If they don't transform, those traditional organizations will be disrupted.

Promiscuous Pairing

RC teams swap pairs—every day.

The purpose of "Pair Programming"—to learn and emancipate knowledge—is quickly undermined when the pairs on a team don't change (an antipattern known as "pair-married"). Knowledge silos, originally broken down by "Pair Programming", begin to reform in specific pairs; pairs begin to specialize; knowledge of significant refactorings don't escape the pair boundary.

Furthermore, hearing the same perspective day in and day out from your pair leads to a loss of learning, makes pairs blind to broken windows, creates boredom, or worse, tension and stress, and even burnout.

But there's a very simple antidote: promiscuous pairing—a cheeky name for a simple practice. Swap your pair every day, at the start of the day. It doesn't matter whether a "User Stories" is in flight. It doesn't matter whether a pair is working on a "critical issue." That's all the more reason to swap. Fresh perspectives help ensure quality work.

At the end of "Team Standup", RC team members self-organize into pairs. And if there were stories in progress at the end of the previous day, they quickly divide their team into two sets—one set of engineers that will stay on the story that's in progress, and another set that will rotate. By simply requiring that everyone pair with someone different than the day before, they ensure that every story in flight receives both continuity and fresh perspective.

Relative Complexity Estimates

RC engineers estimate "User Stories" by ordering upcoming stories from least complex to most complex. They then assign points to the stories to reflect their relative complexity.

Time estimates in our industry are the definition of insanity. Since the start of our industry, organizations have been asking, "How long

will this take?" And for decades, developers have been responding with time estimates. Kent Beck, the inventor of Extreme Programming, has a rule for time estimates: "Whatever they say, double the number, and increment the unit." In other words, if a programmer says it will take "one day," it will take two weeks. "One week"? Two months. Of course, he's not seriously recommending this method of estimation to anyone, but he is hinting at a real truth in our industry. Time estimates tend to be horribly, terribly, catastrophically wrong.

In a traditional waterfall/leader-follower organization, success isn't measured by the realization of user value, but by the ability of teams to deliver predetermined requirements on time and on budget, hence the focus on estimating in *time*. RC organizations, on the other hand, incrementally build and deliver "User Stories" into the hands of users and iterate based on concrete feedback. RC teams don't typically have months or years of work planned upfront. Instead, they have goals that they're trying to achieve with their product and outcomes that their measuring against; the specific functionality that will achieve the goal isn't all determined upfront. Instead, it's discovered through validated learning.

Which leads to an interesting question: if success in an RC organization is about the realization of user value and not about delivering a predetermined set of features on time and on budget, do RC teams need to estimate? Perhaps, surprisingly, the answer is "yes." Sometimes, they literally need to know when a certain set of stories will be done. For example, imagine there's an important conference on the horizon, and everyone would like to know what they'll be able to tell the world what the product contains on that date. It's not easy to develop speeches, marketing materials, and so on for an event like that when you don't have a clue as to what the product will contain by that time.

But even without dates like that, estimating work can still provide value. The trick is, you need to estimate with *relative complexity points,* not time. Because even though programmers are terrible at estimating work with time, they're really great at estimating how easy or difficult one user story is compared to other user stories.

For example, line up three stories in front of the development team, and ask the team to rank the stories from least complex to most

complex, relative to one another. They'll quickly sort it out. The conversation might go something like this:

- "Hmmmm. Well, this one will be really easy, it's really just some light updates to the frontend."
- "Yeah, but this one, though, will require work in multiple modules, so it's definitely more complex."
- "This third one touches all of that as well as the third-party rules engine integration, which is always a pain—there's quite a bit of technical debt to address there. It's definitely the most complex."

And now you have three stories, estimated with relative complexity. Assign each of them a certain number of points; for example, "1" for the easiest, "3" for the most difficult, and "2" for the story in between. We said this process can provide value—but what value has the team actually realized by going through this process?

Even if it did nothing else, the team already had a conversation that illuminated the complexity that each developer saw in a story. This process often uncovers knowledge silos on the team or highlights different levels of experience. For example, imagine one engineer declares a story to be the most complex, whereas another says it's the least complex. Differences in perspectives like this aren't as uncommon as you might think, and can reveal knowledge silos about the underlying system, or different levels of experience with certain types of engineering work. And revealing those differences gives the team the opportunity to take advantage of one another's skills and knowledge better, and learn more effectively. In this example, it would be great to pair the two engineers together for that story so that the knowledge that makes the story "low complexity" (or "high complexity," depending on whose perspective was more accurate in this situation) is spread through the team.

But if teams take this a little further—for example, if they keep track of how many user story points they deliver each week—they can understand their team's "Velocity", as well their team's "Volatility". They can use the velocity to predict when stories will be done based on math, instead of guessing when something will be done based on a death-march-inducing combination of fear and ego. And they can monitor their volatility as a trailing indicator of dysfunction.

Rotation

RC teams routinely rotate outsiders into their teams in order to point out the problems that the team can no longer see.

Every team needs two things. First, it needs a core group of practitioners that deeply understand the problem the team is solving and the users for whom it's solving it so that it can iterate on user feedback, day in, day out. But it's easy for a team like this to become blind to problems and team inefficiencies over time. This is why the team also needs fresh perspectives. It needs outsiders to join the team and tell it all of the things that have gone wrong, that team members themselves can't see. This process of bringing outsiders into a team is called "rotation."

How often do you need the fresh perspectives of outsiders? Ask yourselves: is everyone comfortable? Has the team gelled? Does each day feel familiar? If the answer is "yes," you need a fresh perspective because it means no one on the team is an outsider; everyone is comfortable with the team's process. And if everyone is comfortable, your team is suffering from flaws that none of you can see. You have all become happy, comfortable, boiling frogs.

The time it takes for an outsider to be assimilated into the team and lose their outsider perspective varies. But here's a good rule of thumb: rotate someone new into your team every eight weeks. A warning, though: feelings of tribalism will grow whenever someone new joins the team. Instincts within the human psyche will take hold. The feedback of the newcomer will be rejected; the defensive circuits of the brain will activate, lowering both the individual and the collective IQ of the team. Human evolution, which saved us from the saber-toothed tiger, now works against us.

You can't force an outsider onto a team. For this pattern to really work, the team members need to invite the outsider in themselves. They need to genuinely ask for the feedback and fresh perspective. This willingness to be vulnerable is fostered by an empowering feedback culture that rests on a foundation of individual and organizational psychological safety.

Retrospective

Once a week, RC teams sit down and reflect on what's working well and what could be better. A team that doesn't adapt doesn't thrive.

Within this book, you'll find a number of patterns that you can seed new teams with ("Discovery and Framing", "Balanced Teams", "Pair Programming", "Iteration Planning Meeting", "Test-Driven Development", "Collocation", "Collaborative Story Acceptance", etc.). But there are two forces always at work to undermine teams: the inevitable degradation of norms, and the team's evolving context. We've studied the psychological basis for the former in the "Rotation" pattern.

However, what about the second force: a team's ever-evolving context? The world outside the team never stops changing. Team compositions, organizational directions, user expectations, technology fads—they're all changing. Unless a team adapts to its ever-evolving context, unless it mutates its ways of working to meet its new circumstances, it will wither. Every week, the team will encounter more and more challenges that will go unmet. Retrospectives are a simple pattern for purposeful pattern mutation. At the end of every week, RC teams go into a room and reflect on what's going well and what could go better.

There are many formats for doing this. Here's two popular variants that will give you a sense for the possibilities:

The "Happy, Meh, Sad" three-column retro
: Everyone writes down their reflections on the weeks (one reflection per sticky—in all caps, with a Sharpie, for readability) and puts them on the board under the appropriate columns. Then, someone pulls stickies down one by one, and facilitates a discussion about each sticky, capturing action items along the way. Variations on this format include altering the column names to "Keep, Question, Change."

The boat retro
: The facilitator draws a boat in an ocean on the whiteboard. They draw lines to indicate wind in its sails, and then they draw an island that the boat is clearly heading to. However, they also draw an anchor holding it back as well as sharks in the water, between the boat and the island. The boat represents the team,

the anchors represent the current blockers, and the sharks are potential future problems. The wind represents the things that are already working well for the team, the island stands for the team's goals. Everyone writes down (one per sticky) examples of wind, anchors, and sharks. They also write down what they believe their island to be—what their goal, or destination, is. They all put their stickies on the board in the appropriate places, and then the facilitator guides a discussion about each sticky, capturing action items along the way.

There are entire books filled with retrospective formats, but the point of every format is the same: adaptation. Identify what's working well, and keep doing it. Determine what's not working well, and change it.

Here are some signs that your retrospectives aren't as effective as they could be:

- The same problems show up week after week (action items ineffective)
- The same action items show up week after week (no one is following through on the action items each week)
- There are obvious omissions each week that no one is courageous enough to call out (there's a lack of psychological safety in your retrospectives)

Whenever you see problems like this, it's important to call them out, even if it feels scary to do so. Otherwise, retrospectives on your team will become a hollow imitation of what they're intended to be. In addition to watching out for the aforementioned signs, there are also some simple do's and don'ts:

- Don't let management into the retrospective (this undermines psychological safety)—make sure the retrospective consists solely of the "Balanced Teams"
- Do assign owners for the action items, make them visible in the team's space, and track their progress
- Don't allow specific personalities to dominate conversations
- Do rotate facilitation duties around the entire team

These do's and don'ts go a long toward increasing the efficacy of a team's retrospective practice and connecting the reflections from the retrospective to the daily work and rhythm of the team.

A word of caution: retrospectives, when not balanced with root-cause analysis and a "Value-Stream Map", and when not grounded in a foundation of systems thinking, can easily lead a team toward local optimizations—one of the key antipatterns inhibiting learning and efficacy in organizations today.

Team Standup

> *Before an RC team starts pairing for the day, they take a moment to see whether anyone on the team is blocked or has vital information to share.*

Have you ever spent all day working on something only to discover that all of your work was for naught because of some vital piece of information that a member of your team had neglected to share with you? Or have you ever struggled to get work done only to discover that someone else on the team could have unblocked you had only they known you needed help?

Because of situations like these, RC teams begin every day with a team standup: a moment for anyone and everyone to expose blockers or share vital information that's not yet widely known on the team. In the early days of Agile, it was common to see teams use a format for standup in which everyone stood, everyone spoke in turn, and everyone said what they worked on yesterday, what they're working on today, and if they're blocked on anything.

And it was common to find this format produce little to no value for the team. For example, some people would tell the team information that everyone on the team already knew—perhaps the team practiced "Collocation", and lots of information was already widely known from the day before. Others would ramble, trying to remember what they did the day before, relaying all kinds of information that was neither urgent nor important. Teams with a prioritized backlog that obeyed "Top of Backlog (ToB)" had no need to say what they were going to work on today—that was dictated by ToB, and had already been decided on during that week's "Iteration Planning Meeting". Team members would commonly zone out, simply waiting for their turn to talk, and everyone was visibly relieved when the ritual was over.

This is an example of *cargo-culting*. The team had copied a ritual without understanding its deeper intent, causing the ritual to lose all meaning and purpose.

You can avoid making the same mistake by focusing on the goal of the standup: removing blockers and disseminating knowledge. If no one is blocked and no one has gained vital knowledge that's not already widely known on the team, the standup can literally take seconds.

Here are some examples of the vital knowledge conveyed during a standup:

- A team member on pager duty was awakened during the middle of the night by a production outage.
- A pair learned yesterday that a feature they had built and deployed had low adoption, invalidating the value hypothesis behind the feature.
- Another pair discovered that a third-party integration was going to be much more challenging to implement than originally anticipated.
- A product manager participated in a workshop to ideate and prioritize business goals for the upcoming year.

Similarly, here are some examples of blockers worthy of mention in a standup:

- The "Continuous Integration/Continuous Deployment" environment went down overnight—everyone needs to stop work on "User Stories" until the impediment is resolved.
- An entire track of prioritized work must be put on hold due to delays on the completion of a downstream dependency by another team.
- A product manager has been sucked into too many status-report meetings that provide no value to the team or the organization—causing backups and delays in "Collaborative Story Acceptance".

All of these examples represent siloed knowledge that needs to be shared with the entire team. It's the responsibility of the team as

whole to absorb this knowledge because it's likely to affect the collective goals, directions, and mental models.

Test-Driven Development

> *RC programmers test to see whether the program does what they want it to do—before they've programmed it. No, they're not insane.*

Consider the two graphs of team speed over time illustrated in Figure 3, and ask yourself which team you want to be on:

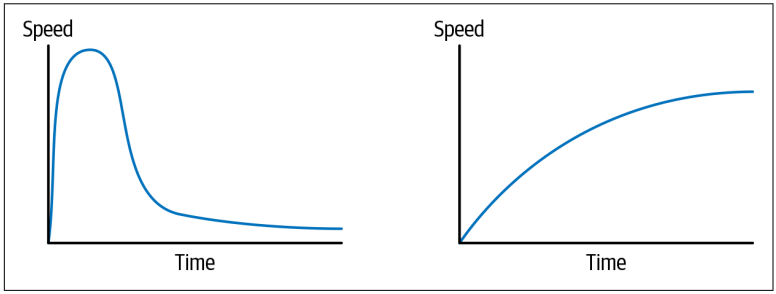

Figure 3. Two possible graphs of team speed over time.

Look at the first curve. They went really fast at first—but it didn't last. Their speed dropped precipitously, until reaching a depressing asymptote. That asymptote kills teams. It kills their spirit. It's the death march.

RC teams believe that the best software is built through a process of learning, which requires them to continuously deliver software into the hands of users, solicit their feedback, and iterate on the product in response to that. You can't do that if you're on that first curve. You need the second curve. You need to *go fast forever*.

There's lots of reasons why a team slows down. But there's one thing that will always slow a team down: bad code. A team that looks like the first curve cut corners (e.g., it didn't clean its code, it created lots of defects). You can go really fast if you cut all the corners—but only for a little while. It's false expedience. What made it easy to go super fast in the short term makes it impossible to go fast in the long term. The codebase very quickly resembles a ball of mud. And eventually, it reaches the point of no return, and those engineers will have no choice but to start over and try again.

If you want to *go fast forever*, you need to have *clean code*. But you don't just clean your code once. Cleaning code is like taking a bath.

You need to bathe regularly if you want to stay clean. The longer you wait, the dirtier you get. It's the same with code.

So, to keep your code clean, you must constantly refactor it. Every new feature you add to your codebase challenges the assumptions behind the code's design. Here's where you have two options: you can either find a way to workaround the invalidated design in your code, or you can take the time right then to fix it—to *refactor* it. The former leads to that rollercoaster first curve, which ends with a death spiral. Refactoring leads to the second curve—consistent speed—by holding the behavior of the system constant while cleaning up the underlying design.

Regrettably, out of fear, most teams don't refactor. They know the code is rotting, but they're not sure whether everything will still work after they clean it up. To continuously refactor your code, you have to have *confidence* that the refactoring has still resulted in working software, that you haven't introduced regressions.

Unfortunately, there's no magic wand that you can wave to prove that your system still works. You need to do the work yourself. If that work is too difficult, or takes too long, you're unlikely to do it. That's why manual QA teams don't work—it takes too long for the engineers to know whether the changes they made resulted in working software. The only way to get that confidence quickly is to write tests. Tests give you the confidence you need to refactor your code to keep it clean so that you can go fast forever.

So the question becomes: when do you do it? When do you write tests? And the answer is obvious. Because everything depends on tests, you do them first—they're the most important thing! This is why RC teams practice test-driven development (TDD)—the process of writing a test *first*, running it, watching it fail, writing the simplest production code to make it pass, and then refactoring (see Figure 4).

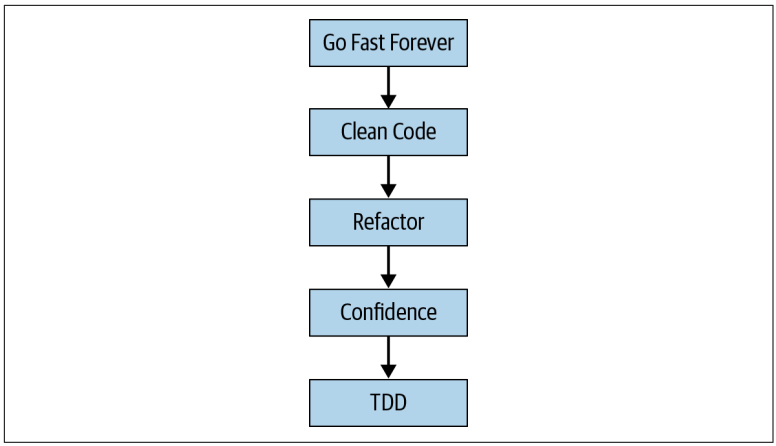

Figure 4. The go fast forever" dependency chain

There are, of course, many more reasons why RC teams write tests before instead of after they write the production code. Writing it first helps them tease out and think through their APIs. It forces them to clarify exactly what behavior they're trying to build because they can't write a test without that clarity. It also lets them know when they're done. It helps them triangulate on simple, maintainable implementations by making each test pass one by one and writing just enough code to make each test pass. Furthermore, writing the test before the implementation gives them the confidence that their test suites aren't giving them false positives; they first watch the test fail for the reason they expect and then do the simplest thing they can to make it pass.

A word of warning though: TDD is difficult. Progressing from novice to expert at TDD can take years. The payoff is large, but the danger is this: telling a workforce of engineers who have never practiced TDD to simply start doing it *is guaranteed to fail*. Without anyone to teach them, guide them, grow them, and rescue them, they will write brittle, flaky, unreadable, unmaintainable test suites that do much more harm than good. Furthermore, an existing workforce working on existing products and codebases, none of which were built with TDD, will have the extra complexity of attempting to introduce tests into legacy codebases not built for it. This isn't TDD 101. Working effectively with legacy code requires mastering both the skill of TDD *and* the skill of incrementally retrofitting tests into codebases that

weren't built with automated testing in mind. Both skills require a great deal of time and expertise to master.

Don't set up teams for failure. Introduce TDD slowly. Begin with a single team, and hire people into the team who are already experts at TDD. Use expert–novice "Pair Programming" to teach the skill; rotate pairs frequently, and occasionally set up novice–novice pairings to help everyone stay aware of how far they've truly progressed.

Prove success with one team before expanding the practice to more teams. This process takes long enough that you also need to focus on retention. The skills you're cultivating into your workforce are long-term investments. Don't watch your investment vanish due to poor retention rates.

Crafting a good test suite is difficult. Teams need to keep in mind the four goals of a test suite:

- Fast
- Clean
- Confidence
- Freedom

We explore each of these goals in turn and then end by considering why these goals are actually really challenging to achieve completely.

Fast

A test suite should be fast for the simple reason that if it's slow, you're less likely to run it. And the less likely you are to run it, the longer it takes for you to know whether you've broken anything while refactoring. And the longer that takes, the less likely you'll be to refactor.

How fast should a test suite be? I asked a team recently, and received several answers:

- Short enough that your mind doesn't wander
- Short enough that it's not an excuse to take a break
- Less than the time it takes to get a cup of coffee

Although these answers are good, I prefer the simple goal of "instantaneous." The ideal test suite time is zero seconds.

Clean

RC programmers write tests to help them keep their production code clean so that they can *go fast forever*. But test suite code is susceptible to all of the same problems that plague production code. No one wants an unreadable, unmaintainable test suite. For example, without proper grooming, you can quickly end up with a test suite that's riddled with knowledge duplication. And that knowledge duplication leads to fragility—a simple change you'd like to make to the production code causes tons of tests to break, and each test must be fixed individually.

You need to keep your test suite clean (i.e., clear, readable, and maintainable), not just the production code. Otherwise, you won't be able to maintain your test suite over time. It will become such a burden that team members will stop writing tests.

Confidence

If your test suite is green, how confident should you be that the software works? Obviously, the ideal answer is: 100% confident! If my product manager asks me, "Can we ship the software?" I want to be able to point to a green CI build and say, "Yep—it's green. There's nothing more I need to do to know that we have a working release. Ship it!"

Freedom

Even if you've been practicing TDD for a while, you might have been wondering what is meant by "freedom." A test suite should give you freedom: the freedom to refactor your code! This might sound obvious, but actually, it's not uncommon to discover teams have written test suites that make it more difficult for them to refactor their production code instead of easier. Sometimes, it's because they've tested implementation instead of behavior. Sometimes, it's because they've coupled their tests to the underlying design of their production code, making it difficult to refactor that design because the tests are all aware of it.

This problem often stems from a common, but incorrect, definition of "unit testing": teams mistakenly believe that unit testing means that for every class, and every public method of every class, there must be a corresponding unit test. In fact, the "unit" in "unit testing"

simply meant that each test could be run in isolation of all of the other tests—that each test was a unit unto itself, that there were no ordering dependencies between the tests.

The Great Balancing Act

Imagine it: a test suite that runs instantaneously, gives you 100% confidence that the software works, is incredibly readable and maintainable, and gives you unfettered freedom to refactor your production code. That would be a fearsome thing to behold!

But actually, it's challenging to balance all of these qualities. Each test you write will take a little bit of time to run; even if each individual test takes only a few milliseconds, they'll add up. And you might very well discover that in order to feel really confident in your test suite, you'll need to write an end-to-end integration test or two, which will further slow down your test suite. But then maybe you'll decide that to speed it up, you'll use test doubles to "mock everything out"—so many teams have practically ruined their test suites by overzealously "mocking" everything out in the pursuit of speed. The test suite runs fast, but they're not confident that it's actually testing anything real anymore. And the tests themselves are tightly coupled to the underlying implementations, making it increasingly difficult to refactor.

As you and your team are practicing TDD, you should constantly ask yourself the following questions:

- Is this test helping us meet our test suite goals of *Fast, Freedom, Clean*, and *Confidence*?
- Or are we testing in such a way that it undermines one or more of our goals?
- And, if so, is there a different way we could test this?

These kinds of objective goals can lead to less heated, more reasoned discussions about different ways in which you could employ TDD. Aligning a team around goals leads to less religious debates and more productive conversations, experimentation, and innovation.

Top of Backlog (ToB)

RC teams don't assign work to specific developers. Instead, they create a single prioritized backlog of "User Stories", and have every pair of programmers pull stories straight from the ToB.

In traditional waterfall organizations, all of the work for a software development product is planned up front; the work is then divided into hundreds or even thousands of tasks. Any one task, on its own, wouldn't be valuable; it's only after the sum of all of the tasks are completed that the software "works." The tasks themselves are often assigned out to individual developers by a project manager.

RC teams operate in a radically different fashion. They don't plan all conceivable features and tasks up front; instead, they focus on the problem they want to solve and then create and deploy "User Stories" that they believe will help them start to solve that problem. The user story, unlike the waterfall task, represents incremental value.

But which stories do pairs work on at any given time? Is the work assigned out to specific pairs by a project manager? Of course not! Each week, in an "Iteration Planning Meeting", the team prioritizes any stories that they've written into a backlog, and each pair pulls from the "Top of Backlog (ToB)" until the backlog is exhausted (this process is sometimes simply referred to as "tob'ing").

Why? Well, the story at the top of the backlog is the highest priority story. The team wants to complete that story and put it in the hands of users sooner than any other story in the backlog. They believe that story will provide the value their users need the most at this moment. Or, conversely, they believe that if that story proves unsuccessful at providing value to users, it has the biggest potential to change the team's future trajectory, and future stories.

Obeying ToB has other benefits, too; in a traditional waterfall organization, developers are often assigned tasks that have them work on only a single part of the tech stack. The knowledge of their work in that part of the tech stack is siloed into their brains; other engineers don't know or understand that part of the stack. But when you "tob," you end up working on every part of the tech stack of your product; frontend, backend, middleware, persistence, and so on. ToB, combined with "Pair Programming", eliminates knowledge silos and reduces the risk that those silos represented.

User-Driven Architectures

Have you ever put 20 engineers in a room and asked them to write down their definition of software architecture? I did once and I got 20 different answers. Some were negative, some positive, and several contradictory.

It's not their fault. The software industry has yet to actually agree upon a definition of software architecture, and that's causing real problems. There are a great many battles happening within organizations, between different engineers with different roles and different levels of power, simply because they don't agree on—or even understand—what architecture is and what makes it "good."

Before we can show how engineers within RC organizations approach architecture, we first must define architecture. There are several great people improving our industry's understanding and practice of architecture (Matt Stine, Simon Brown, Robert Martin, Grady Booch, and Martin Fowler come to mind) and the following definitions are beginning to emerge:

Architecture
: Decisions that are difficult to reverse.

Software Architecture
: The high-level shape and flow of the software that's independent of the problem domain, but dependent on the desired user experience.

Let's look at each of these definitions in detail.

Architecture, Generally

In his two-volume work *Software Architecture for Developers*,[7] Simon Brown explores some of Grady Booch's writings on architecture, including the following gem: "Architecture represents the significant design decisions that shape a system, where significance is measured by cost of change."

Simon then goes on to note, "The architectural decisions are those that you can't reverse without some degree of effort."

7 Brown, Simon. *Software Architecture For Developers*, volumes 1 and 2. (Lean Pub, 2018).

I'm sure that with that definition, even the most die-hard "anti-architecture" Agile developer will admit that they've made architectural decisions. For example, every engineer has at some point chosen a programming language for a project. The programming language is a very significant decision. Imagine that you choose Java on day one of a project and then three months later, you decide Go would have been a better language choice. Well, too bad. You're three months into writing Java. You're unlikely to quickly reverse that decision. The programming language, then, is architecture.

Software Architecture, Specifically

Software architecture refers to something more specific that's difficult to reverse: it's the high-level shape and flow of the software that's *independent* of the problem domain and *dependent* on the desired user experience. To understand what that means, let's look at three common categories of software architecture:

Request/response
: I make a discrete request to the system; I get a discrete response.

Event driven
: Events happen. Actors independently react to those events and can generate new events in the process. Actors can be both human and nonhuman. There is no central coordination.

Batch
: The system processes data without user interaction or involvement, including error handling and recovery.

Most web applications have request/response architectures. If you examine the codebases of those web applications, you'll find that they have a very similar high-level shape and flow. How can that be? Those web applications are all solving different problems for their users. But remember: the software architecture is independent of the problem domain but dependent on the user experience. So even though those web applications are all solving different problems, they all have a very similar user experience. Click a link, navigate to a new page. Submit a form, see validation errors. Submit request. Get response. Start. Stop. Request. Response. Get it?

Interestingly, it's not just most web applications that have request/response architectures. So do most command-line programs. Type *whoami* into your Unix terminal, see your username. Type *pwd*, see

your current working directory. Type command. See output. Start. Stop. Request. Response.

This means that most command-line applications and most web applications have the same user experience! And if you examine the codebases, you'll see that those command-line applications and those web applications have the same high-level shape and flow—a request/response software architecture.

Let's look at another category of software architecture. Have you ever played a video game? A sidescroller like Super Mario Bros? Or a first-person-shooter like Doom? (Am I aging myself?) A video game user experience feels very different from those web and command-line applications that we just discussed. With these games, you press Start and a whole world is set in motion. Your character is just one of the actors in those worlds. You press the right-arrow key and Mario moves to the right. The game notices your move event and reacts by updating the screen so that you are centered in it. While you're moving, a goombah strolls into the frame and collides with you. The sound engine notices the collision event and reacts by playing a sound. Many actors. Many events. Many reactions. Get it?

Most games have event-driven architectures, but so do a lot of distributed systems. For example, Diego, the core distributed system powering Cloud Foundry, is event driven. And if you examine the codebases, you'll find that all of these applications have a very similar high-level shape and flow—despite the fact that a distributed system is made up of many different processes running on different machines on a network, and a game might be made up of a single process running on a single machine.

Let's look at one more software architecture. Do you get a paycheck? Perhaps it's mailed to you every couple of weeks, or it's deposited into your bank account. Do you have to submit a request to some software every two weeks to make sure you get a paycheck? No! Of course not. That paycheck is sent to you (and likewise to all of your fellow employees) every two weeks without you having to interact with any software. That's because, behind the scenes, the system is processing the payroll on a schedule, without direct user involvement.

That's the defining characteristic of a batch architecture: the system is processing requests for the users without the users having to

actually make the request! And that has a profound consequence on the shape and flow of the software. When a user submits a request to a website, if the response is "503 Service Unavailable," the user can simply refresh and try again. In other words, in a request/response architecture, the responsibility to "retry" is given to the users. But if a batch job is processing some data behind the scenes and suddenly encounters some transient exception, there's no user there to notice it and make the software retry processing the job. The batch architecture has to do that for us, automatically, without our intervention.

So obviously, payroll-processing applications have a batch architecture. But so do many other systems: automated clearing houses and subscription management systems, for example. And, you guessed it: examine the codebases, and the high-level shapes and flows are the same.

Because the shape of the architecture fits the user experience, and because RC teams create user experiences that enhance users' lives—experiences that their users actually enjoy and benefit from. I refer to this type of architecture as user-driven architecture.

But Why Is Software Architecture "Architecture"?

Remember, architecture, generally speaking, refers to the decisions we make that are difficult to reverse. And the software architecture—the high-level shape and flow—is not an easy thing to change. All of your code—and tests—are built around that flow. After you decide to go with a specific software architecture, you can't easily reverse it.

Software architecture is expensive. You need to make sure you're making the correct decision when you choose one. But before we talk about how we choose, let's talk about when we choose.

When Do RC Engineers "Architect"?

Agreeing on the definition of architecture is only half the equation; the engineers also need to agree *when* to architect the software. Do they determine it all at the outset? Does it evolve over time? Does it emerge naturally through refactoring?

The mistake that many Agile engineers make is assuming that the software architecture will emerge naturally through the process of "Test-Driven Development" (TDD) and refactoring. It won't. Soft-

ware architecture is a significant decision. As Robert Martin notes in his essay "The Domain Discontinuity," you can't even begin to TDD until you know the software architecture within which you will TDD—because your tests are, by necessity, coupled to the software architecture. This implies that we must determine the software architecture before development.

However, RC engineers don't foolishly attempt to plan all of the features they're going to build into the software upfront, before beginning development. They know that good software is built through a process of validated learning, not through extensive foreplanning. They know that the quicker they can put working software in front of users and get their feedback, the quicker they can discover the product assumptions that they got wrong. And lastly, they know that software is never "done." Software solves problems for people. Because people are always changing, software must be ready to change along with them or risk obsolescence. This means that on an RC team, the user experience of the product will grow and evolve over time in ways that can't be predicted or planned at the outset. Most products grow to encompass multiple user experiences—and therefore multiple architectures.

This implies that we must determine the software architecture during development, which might seem paradoxical. First we say we need to decide before development and then we say we must decide during development. But in fact, those two statements are not mutually exclusive. Let's look at how RC engineers determine the software architecture before development first.

How Do RC Engineers "Architect" Before Development?

Before you can write your first test, you need to know the software architecture. So how do you determine that? You do it by establishing the desired user experience—by talking to your product owners and product designers about the features that you're going to build, and understanding how the users are interacting with the product.

Imagine that we're building an application for the game "Rock, Paper, Scissors." Our product manager shows you the following story:

```
Feature: Play

  Scenario: Rock v. Scissors
```

```
Given player one throws rock
And player two throws scissors
Then player one wins
```

What software architecture does this story demand? Request/response? Event driven? Batch?

The answer is that it's impossible to know based on this story. Why? Because this story doesn't give us enough information about the desired user experience. Do player one and player two throw rocks and scissors in real life, and then enter their throws into the application to see who the winner is? Or are they actually using the application to play the game?

We ask our product manager to clarify; they send us an updated version of the story:

```
Feature: Play

  Scenario: Rock v. Scissors
    Given player one throws rock in real life
    And player two throws scissors in real life
    When one of them enters the throws into the application
    Then the application tells them that player one wins
```

That's much clearer. Now we need to decide which software architecture maps to this user experience. After talking about it a bit, we decide that it's probably request/response. The "When" statement seems to indicate that a user enters throws into the system, and the "Then" statement indicates a response. We ask our product manager for a wireframe to confirm our suspicions. Figure 5 depicts what we received back.

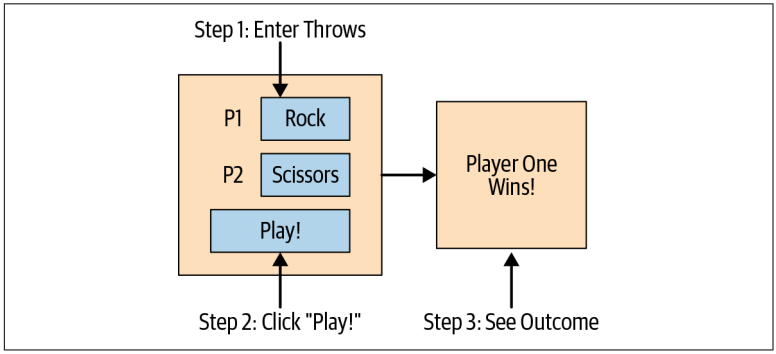

Figure 5. Wireframe for "Play" "User Stories"

It's as we suspected! The user enters the details into a simple web form and presses the "Play!" button. The system replies with a result. This is a request/response user experience. We know what the high-level shape and flow of a request/response architecture looks like and we know the boundaries within which we'll TDD.

But what if...

Imagine for a second that the product manager had replied to our clarification query with the following story:

```
Feature: Play
  Scenario: Rock v. Scissors
    Given player one submits a "rock" throw
    When player two submits a "scissors" throw
    Then player one should see that they won
    And player two should see that they lost
```

This feels a bit different. This doesn't feel like a start/stop, request/response flow. This feels like there are multiple actors in the system. We ask the product owner for wireframes to confirm our suspicions. Figure 6 shows what they gave us.

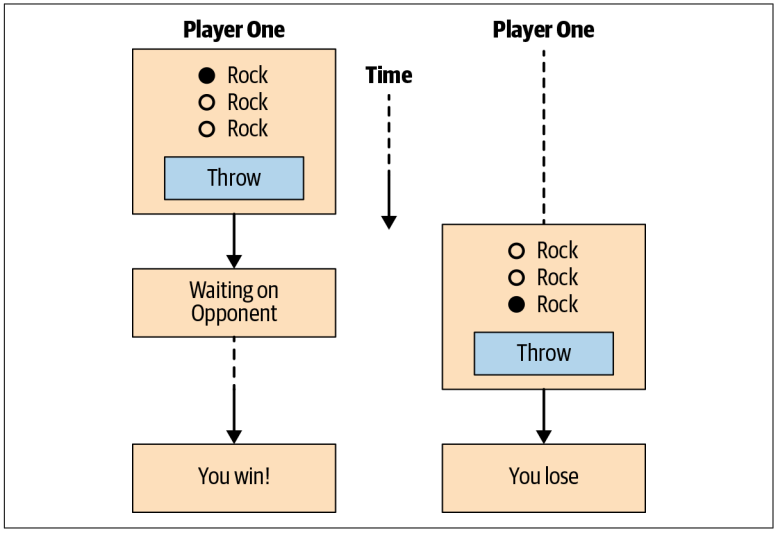

Figure 6. Alternative wireframe/user experience for the Play feature

As we suspected, this is an event-driven user experience! There are two users simultaneously using the application. The users are generating "throw" events. The system is watching those "throw" events,

and computing and generating "result" events after both players have thrown. The user interface is watching "result" events and updating the display accordingly. And again, now that we know that we need an event-driven architecture, we know the boundaries within which we will TDD.

How Do RC Engineers "Architect" During Development?

Software architecture happens both before and during development. You can't begin to TDD until you know the software architecture that you'll TDD within. The software architecture flows from the desired user experience.

But products grow and evolve over time, and the desired user experience grows and evolves over time, too. On an RC team, at any time, you could pick up a story that demands a different user experience than the one your application has facilitated until now. You must ask yourself, with each and every story, "What user experience does this story demand?" If the answer is different from the user experience that you've been building to this point, now you'll have a product that facilitates two different types of user experiences. Which means your code will need to house multiple software architectures simultaneously. That's a complexity booster for your codebase; maintaining the multiple necessary architectures requires care and craft. In the long run, this is of course simpler than choosing "One Architecture to Rule Them All" upfront, and then adhering to that architecture come hell or high water.

User Stories

User stories are the primary unit of currency on RC teams.

Building software is complex. Between the technology, teams, processes, practices, not to mention individual personalities and interpersonal dynamics, it's shockingly easy to lose sight of the single most important actor in all of this: the user—the *human*—for whom you're building the software.

When you lose sight of the user, when you lose focus on creating value for them, software development transitions from a purposeful, mission-fulfilling process into a meaningless, frustrating struggle without purpose. And if you and your organization don't right the ship, your users will abandon it for the first life raft they see.

RC teams stay laser-focused on the users by organizing all of their efforts around a single purpose: delivering features, one by one, into the hands of users and getting feedback on them. And they describe those features to one another through user stories. A user story is a narrative description of a single task or activity that a human can perform with your software. It represents *potential incremental value*—in fact, it represents the smallest amount of *potential incremental value* that you can put in the hands of users in order to learn.

Note that we say *potential*. The creator of the user story believes that the story will create value for the users. But they also know that it's just a hypothesis, that they might be wrong; so instead of building lots of functionality and then releasing it all at once, they build a single, small user story and put that in the hands of the users. Releasing little by little gives teams the ability to adapt, adjust, and reprioritize based on concrete learnings from those who are actually using the product.

Teams often struggle with the size of their user stories. They ask, "How big (or small) should a story be?" But to answer this, they must simply ask themselves one or two questions. "Could I remove functionality from this story and still provide value to the users?" If the answer is yes, the story is too big. Conversely: "If I complete this story, will it actually provide value to the user?" If the answer is no, the story is too small. For example, imagine a story that describes a user filling out a web form—yet the story doesn't require that the form do anything when the user presses the submit button. That story is too small. Asking users to fill out a form that doesn't actually work isn't adding value.

The "Continuous Integration/Continuous Deployment" of user stories reduces the risk inherent in the process of building software. But these patterns require particular traits that are sometimes difficult to come by in the software development industry: *humility* and *courage*. The creator of the user story must have the humility to admit that their assumptions might be wrong, that the story might not actually provide value; and they must have the *courage* to test their assumptions by releasing the completed story into the hands of users.

Value-Stream Map

> *If you can't describe what you're doing as a value stream, you don't know what you're doing.*
>
> —Karen Martin, *Value Stream Mapping*

What does it take for a software team to take an idea, turn it into software, and put it into the hands of users? How many steps are involved? How many ideas are in progress simultaneously? Clearly, software developers live within this process—turning ideas into reality—day in, day out; it's familiar. Yet, to paraphrase the German philosopher Georg Hegel: what is familiar *is not known*.

I once learned this lesson the hard way, 11 months into a greenfield project. We were developing a very complicated software product with both shrink-wrap components and server-side components, including an API as well as a web-based GUI. We'd launched a beta version of the product in three months—a record at the company we were at—and went GA after another two months. We were proud of what we'd accomplished, and we thought our development process was nearly flawless.

With that optimism, we decided to investigate our process for turning ideas into working software in the hopes that we would find a way to make it completely flawless. We reserved a conference room with a long whiteboard and sketched out all the steps in our software development process. Then, we used green and red sticky notes to write down what we thought was working well at each stage and what could be better (respectively). When we finished decorating the board and stepped back to see the whole picture, we were fairly shocked to discover a veritable sea of red stickies, occasionally peppered with a green sticky, here or there. Not only that, but at an even higher level, our "flawless" process, which we thought was very Agile and very Lean, was actually terribly long and complicated. It looked less like a Lean software development process and more like the blueprints for a Rube Goldberg machine.

We didn't know it at the time, but we had stumbled our way into a Lean product development practice now widely known as *value-stream mapping*, an activity that helps organizations ask, "How do we deliver value to our customers today, and how could we do that better tomorrow?" The practice was originally created by Toyota Motor Corporation, and revealed to the rest of the world in the

1990s through the books of James Womack, Daniel Jones, and Daniel Roos—particularly, *The Machine the Changed the World*[8] and *Lean Thinking*.[9] However, it wasn't until Mike Rother and John Shook wrote their 1999 work, *Learning to See*,[10] that the term "value stream mapping" made its way into the Lean lexicon.

Our industry now has an entire book dedicated specifically to the practice of value-stream mapping, thanks to Karen Martin and Mike Osterling: *Value Stream Mapping: How To Visualize Work and Align Leadership for Organizational Transformation*.[11] Through two decades of consulting efforts, Karen and Mike have captured a set of recommendations and patterns for successfully implementing value-stream mapping within an organization—particularly for situations in which a single team does not own the entire value stream, but instead the value stream touches many discrete portions of the organization. They have a multistep process for mapping that includes the following:

1. Gathering the best mapping team, often including an executive sponsor, a value stream champion, and a facilitator
2. Developing and socializing a charter that creates alignment within an organization for the challenges the organization is facing and motivation for improvement/value stream mapping
3. Generating the current-state value-stream mapping through an iterative, back-and-forth process of sketching and physically walking the value stream ("going to the *gemba*"), progressively layering it with metadata (e.g., pain points like barriers to flow, excessive batching, system downtime, shared resources, inaccessible staff, excessive task-switching and interruptions, and incompatible priorities across different divisions) and metrics (such as process time, lead time, and percent complete and accurate)

8 Roos, Daniel, Ph.D.; Womack, James P., Ph.D.; and Jones, Daniel T. *The Machine That Changed the World: The Story of Lean Production* (Harper Perennial, 1991).

9 Womack, James P. and Jones, Daniel T. *Lean Thinking* (Simon & Schuster, 1996).

10 Rather, Mike and Shook, John. *Learning to See: Value Stream Mapping to Add Value and Eliminate Muda*. (Lean Enterprise Institute, 2003).

11 Martin, Karen, and Osterling, Mike. *Value Stream Mapping: How To Visualize Work and Align Leadership for Organizational Transformation* (McGraw-Hill Education, 2013).

4. Designing a future-state value stream through an iterative process prioritizing important problems to solve, and removing wasteful steps (e.g., local optimizations) and/or adding upstream time-saving steps (i.e., global optimizations in systems theory speak)
5. And lastly, designing and executing a transformation plan that will enable the entire organization to collaboratively realize the future-state plan

Both works—*Learning to See* and *Value Stream Mapping*—provide a fantastic starting point for understanding and realizing the transformational possibilities of value-stream mapping. However, if you're already on an autonomous, empowered, self-organizing feature team, you likely won't need to read an entire book to reap the benefits of value-stream mapping. (Although reading books definitely won't hurt!) You can do as we did—start by simply making your value stream visible to yourselves, and take it from there. You will undoubtedly discover ample room for improvement.

Velocity

> *RC teams use velocity to statistically predict when stories will be delivered instead of pretending that they can intuitively determine how long a piece of work might take to complete.*

We've already talked about the value teams can derive from "Relative Complexity Estimates" (as opposed to time estimates). However, when you take those estimates a step further and begin to keep track of how many points a team delivers week over week, you uncover two powerful metrics: velocity and "Volatility".

Velocity is nothing more than a rolling average of how many points a team delivers in a week, based on the last three weeks of data. In other words, add up the number of points delivered in the three previous weeks (W_{-3} to W_{-1}), and divide it by 3:

$$V = \left(\sum_{i=-3}^{-1} w_i \right) / 3$$

With that number, you can estimate the delivery time of any story in the backlog. Pick a story in the backlog that you want to estimate delivery of (let's call it *Story "n"*, or S_n) and then simply add up how

many points are between it and the first story at the "Top of Backlog (ToB)" (we'll call that S_1) and divide it by the team's velocity (V) to get the estimated time of delivery (ETD) in work weeks:

$$ETD(S_n) = \frac{\Sigma_{i=1}^{n} \ points(S_i)}{V}$$

Why is this valuable? Because a stable velocity allows a team to predict when stories will be delivered based on concrete data and probabilities. A stable velocity is far more effective at making predictions than a developer estimating work in time. As we discussed in "Relative Complexity Estimates", time estimates tend to be terribly, horribly, catastrophically wrong.

A couple of points to make: don't compare velocity between teams. For starters, different teams might use different pointing scales. One might use a simple 1-2-3-4 pointing scale, whereas another uses a subset of the Fibonacci sequence (e.g., 1-2-3-5-8).

But even if all teams used the same pointing sequence, you still can't compare velocities between teams. The number tells you how many points a team can deliver in a single week—but the points themselves won't mean the same thing between teams. One team's "1" might be another team's "3." Points are *not* a standard unit of measurement.

You can (and should), however, look at "Volatility" differences between teams.

Volatility

> *Volatility is more important than "Velocity". RC teams strive to keep their velocity stable by keeping volatility low. When volatility is high, they relentlessly root out the cause.*

For velocity to work, for it to be *predictive,* it must be stable. A volatile velocity—one that fluctuates wildly week over week—can't be used to predict when something will be done with any degree of reliability. So, keep track of how volatile your team's velocity is. You can measure volatility (X) as the standard deviation of the last three week's delivered points (W_i), divided by the velocity for that same time period:

$$X = \left(\left| \sqrt{\left(\sum_{i=-3}^{-1} \left(W_i - V \right)^2 \right) / 3} \right| / V \right)$$

RC teams use this number to establish the range for the estimated time of delivery for a particular story:

$$Range(S_n) = ETD(S_n) \pm x * ETD(S_n)$$

When volatility is high, RC teams find out why. Volatility is a trailing indicator of dysfunction—perhaps on your team, or in your organization, or in your deployment stack, or in your codebase. Whatever it is, find it and fix it. For example, examine it week over week in your team's "Retrospective". The real problem with volatility is that it's affecting your team's ability to reliably and consistently put value in the hands of users and quickly iterate on the product based on concrete feedback.

Workspace Standup

> *RC organizations begin every day with a workspace-wide standup—a way for everyone in their workspace to quickly and easily tap into the local hive mind.*

Every day, within an RC organization, practitioners discover, discuss, investigate, and solve problems. Every day, they learn new techniques, skills, methods, and disciplines. Every day, they innovate. When the organization is small, when it's just a handful of people, it's easy to share all of this knowledge throughout the entire organization. But as the organization grows, this natural hive mind tends to wane. Most organizations are quickly confronted with a range of troubling questions as they rapidly scale: How can they take advantage of and amplify the innovation in their organization? How do they scale the learnings? How do they ensure that teams can use existing solutions, instead of reinventing the wheel, time and time again? How can they avoid someone struggling with a problem that others have already solved?

It's not easy. There is no silver bullet. The larger an organization becomes, the greater the challenge becomes. There are a variety of patterns that RC organizations need to employ in order to work at this pattern, and this is but one.

At the start of every day, many RC organizations create a simple way for everyone in the organization to tap into the hive mind. To give everyone the ability to ask for help, broadcast events, introduce new faces, or share interesting learnings, solutions, challenges, innovations, and experiments. They accomplish this through a workspace standup—a standup that brings together every single person working in the same general space together.

Many RC organizations run their workspace standup immediately after "Communal Breakfast": they let it mark the end of breakfast and the beginning of the workday. They often create a ceremonial way to start it. They might ring a cowbell, bang a gong, strum a guitar, or clap.

They tend to rotate facilitation duties week to week; they let anyone and everyone in their organization take a turn at facilitating the workspace standup. The facilitator will ask everyone who has congregated together for new faces, helps, interestings, and events. But, most important—they make it a psychologically safe environment: an inviting place for people who need help to ask for it; for people who have an idea to share it; for people who have had a recent success to celebrate it. Then, they create a ritualized, ceremonial way to end the standup—like with a synchronized clap, where everyone claps "on three."

If the RC organization is large; if it spans multiple locations, or comprises hundreds or thousands of people, they scale this practice by creating multiple standups. One in each location, or even one on each floor, if they have multiple densely populated floors in the same building. This helps them achieve smaller hive minds.

Note that, over time, the larger the standup grows, the more work they need to do to maintain psychological safety. The larger the group, the scarier it is to speak up. Also, over time, the format will become stale. Participation will lessen. They have to mix it up; change the location; change the structure; if they were standing in a circle, they switch the format and stand in a mob. If no one says anything, they introduce a "talking stick" and pass it to someone, challenging them to share something. Or they ask pointed questions: "Did anyone struggle with a technological challenge yesterday that they were unable to solve?" "Who went into production yesterday? Who deployed multiple times to production yesterday? Who's going into production today?" "Who put a new feature in the hands of

users and learned something?" "Who's speaking at an upcoming event?" "Who tried a new productivity hack?" And so on.

Conclusion

I hope that this little book has quickly given you insight into both the "how" and the "why" behind patterns commonly seen within radically collaborative software organizations.

If these practices are new to you, you might be tempted to begin introducing them into your organization. I wholeheartedly hope you do. However, please keep in mind that changing the way people work is hard—incredibly hard. There are many reasons for this, but one of the most basic reasons has to do with the fact that change is a spontaneous, inner movement of the human being and thus cannot be forced from the outside. People resist change that originates from outside of themselves; they find that it threatens their need for autonomy, security, and even respect.

To complicate matters, the structure of your organization might unintentionally resist radically collaborative ways of working. RC patterns require that the "front-line" software maker—the individual contributor—be the most important and empowered member of the organization, whereas everyone else in the organization plays a supporting role, subordinate to their needs. This tends to stand the traditional organization on its head. In some sense, you might say that to become an RC organization, the leaders of the traditional organization must become the followers, and the followers must become the leaders.

But there's good news. There's no dividing line separating RC organizations from traditional organizations. Those are just abstractions that we can use for intellectual convenience. Real organizations will always be some mix of both. In fact, they already are. Even in the most draconian organizations, there's space enough to work in radically collaborative ways, though it might require a fair bit of bravery on the part of those attempting it.

And you also don't need to force anybody to do anything. Tell others about these patterns, and soon enough someone else will be interested in trying it. Work with those people. And have fun. When you worry less about the future, and stop trying to force everything to

change, you'll find that you have plenty of time to stop and enjoy the experience.

About the Author

Matt K. Parker is a third-generation programmer who has played a variety of roles in the software industry. From small startups to large enterprises, he's been a developer, manager, director, and global head of engineering. He's currently researching and investigating the day-to-day experience-in-consciousness of software makers working in radically collaborative environments.