
Technical Coaching
for IT Organizational Transformation
Dave Nicolette

Technical Coaching for IT Organizational Transformation

David Nicolette

This book is for sale at http://leanpub.com/technical-coaching

This version was published on 2019-05-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

This book is dedicated to my wife Malu, without whose encouragement and support I would most likely do nothing at all.

Contents

Preface

After some 42 years in the information technology industry, with the last 35 or so in consulting or contracting, and the last 18 in the Agile and Lean organizational transformation space, I have a few observations to share concerning typical approaches to organizational change and technical coaching.

Some of the observations pertain to consultants and coaches - how they interact with clients and how they effect change. Others pertain to the clients of such consultants - how they engage external helpers, how they track progress, how they manage friction and conflict during the transformation program.

I've had the opportunity to serve in various roles and to engage with clients at levels ranging from the CEO (in small companies) all the way up to direct coaching of technical teams with hands on the keyboard (in companies of all sizes). I've seen a few anti-patterns (recurring failure modes), and I have a few suggestions.

Notes On the Second Edition

The First Edition had a limited amount of information about step-by-step improvements, including incrementally collapsing functional silos, incrementally improving estimation and forecasting practices, and incrementally improving code reviews. There are many more aspects of technical practice and operations to be addressed in an organizational transformation program.

For the Second Edition, I expanded Part 4 with chapters on improving version control and branching strategies (Chapter 32) and operations and production support (Chapter 33). I hope this will make the book slightly more useful.

Part 1: Foundations

> Wings are like dreams. Before each flight, a bird takes a small jump, a leap of faith, believing that its wings will work. That jump can only be made with rock solid feet.
> — J.R. Rim

This approach to organizational change, consulting, and in particular to technical coaching are different enough from conventional thinking that it's advisable to begin with some explanation of concepts, for context.

After so many years of poor outcomes from attempted transformation programs, perhaps the first question to answer is, simply, is organizational transformation possible at all? If not, then we may have been trying to "do the wrong thing righter" all these years, and we need to figure out what the "right thing" is. Otherwise, we need to determine how we've been causing all these poor results, and change whatever we need to change in our own work to improve those results. This question is the topic of *Is Transformation Possible?*

In *Anti-Patterns*, I enumerate several recurring failure modes I've observed in organizational transformation programs over the years, and suggest practical corrective actions to avoid repeating the same mistakes.

9 | The Cycle of Change

> Repetition, repetition, repetion; equals Results.
> — Jeanette Coron

A failure mode in organizational transformation programs that involve IT is that the consultants insist on extremely radical changes in organizational structure, roles and responsibilities, process flow, work item definition, and metrics as a *first step*. It's often more than organizations can handle, and the program stumbles.

Another failure mode is the opposite - the consultants say "start where you are" without making any foundational changes, and attempt to effect incremental improvements. The weight of existing organizational constraints is too great to enable any of the changes to take hold.

To avoid the risks of these extremes, I propose an approach to organizational transformation that involves several initial changes, based on what has been observed to be effective in industry, followed by an ongoing series of "experiments" guided by a plan-do-study-act (PDSA) cycle.

Any PDSA model may be used. In this book, I use one that I created by extending an earlier and simpler model by consultant Corey Ladas. I call it the *cycle of change*, and it looks like this:

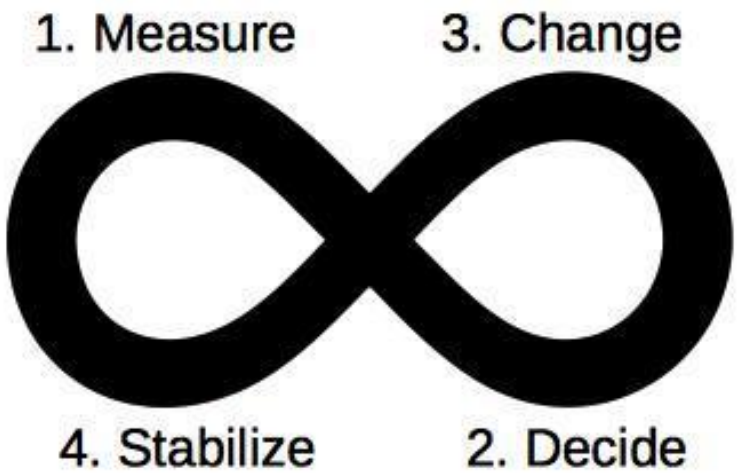

Figure 3.1: Cycle of Change

We begin with measurements of whatever factors are relevant to the thing we aim to improve. That way, we understand the "before" state of the system. We then decide on a change that we think will result in improvement. We make the change.

Normally, there's a period of stabilization after change is introduced to any system. We wait for that to occur so that we won't get a false reading when we measure the "after" state.

When we measure the same factors again, we can tell whether the change we made had any effect on the system's performance. If the change moves us closer to the goal, we keep it and continue the cycle. Otherwise, we decide whether we want to try and adjust things to make the change effective, or reverse the change and try something different.

The approach is based on the idea that an organization of humans is a *complex adaptive system*, not (merely) a *complicated* system. In that sort of system, it isn't possible to predict the results of making a change, as cause-and-effect doesn't operate linearly. Instead, we use an approach called *sense and respond* (Diana). We probe the

system and then respond to the feedback it provides.

The cycle of change is a fractal pattern; that is, it applies at all scales in the organization. Large-scale changes usually require more time to settle in and demonstrate themselves than small-scale changes.

For example, an executive may choose to implement some form of incremental funding and to discontinue the annual budget cycle. The cycle of change in that case might take a year to observe and measure. The scope of the change is large. The impact of the change is large.

In contrast, a technical team may decide to change from a multi-branch code commit strategy to a trunk-based strategy. The effects of this change will become apparent in no more than a couple of weeks. The scope of the change is moderate. The impact of the change is moderate. The impact will increase with repetition by other teams.

Finally, an individual developer may decide to try incrementally refactoring a piece of legacy code. The effects of this change will become apparent (to the developer) in no more than a few minutes. The scope of the change is small. The impact of the change is small. The impact will increase with repetition by other developers and other teams.

The cycle of change is structurally similar to the *scientific method*. There are slight variations in descriptions of the scientific method, but the following steps are pretty common. In paretheses I've added notes relating the steps in the scientific method to the cycle of Change.

- Observation (measure)
- Question (assess the measurement)
- Hypothesis (decide on a change)
- Experiment (implement the change, allow system to stabilize)
- Results (measure again)
- Conclusion (assess the measurement)

From this perspective, the cycle of change appears to be a framework for experimentation. Many of the changes we might attempt in the course of a transformation are in the nature of experiments or trials.

For example, we might wish to adopt a proactive approach to monitoring production operations, rather than the *status quo* reactive approach of waiting for support tickets to be entered. We try out a particular tool to support this. We can measure the stability of production operations and the duration of any outages before and after the change to decide whether to keep the new practices and the tool to support them.

However, there are certain organizational structures, processes, and practices that are strongly supported by experience and observation of other organizations, and we may recommend these directly without any experimentation. For example, the advisability of aligning value-producing resources with value streams is not really in doubt. Restructuring teams in this way is not an experiment, but a correction.

Part 3 | Humanity

> Man is the only creature who refuses to be what he is.
> — Albert Camus

It seems to me that in nearly all organizational transformation programs, not to mention quite a bit of day-to-day management under normal conditions, the fundamental humanity of the people involved is either ignored, denied, suppressed, or punished. As all organizations comprise humans, it might be more effective to acknowledge, seek to understand, accept, and work with people as they are.

This section covers several aspects of the subject, including:

- a chair is a "resource;" a human is not
- Toyota's "respect for humanity" - what does it mean?
- psychological and political safety
- responsibility over accountability
- the effects of stress
- introversion and extraversion
- cognitive biases
- ego development
- attitudes toward formal authority
- the perils of personality profiling
- organizational culture
- consultant-client relations.

14 | Holding Precious What It Is To Be Human

> All sentient beings should have at least one right—the right not to be treated as property
> — Gary L. Francione

The Toyota Production System (TPS) is quite well-known in the Western world. It has two pillars: Respect for People, and Continuous Improvement. Of the two, Continuous Improvement gets by far the most attention in Western organizations and improvement programs.

Many Westerners assume they understand what "respect for people" means - it's about being outwardly polite, maintaining a false smile, not raising one's voice, avoiding naughty words, saying "please" and "thank you," etc.

As this is so obvious, people don't bother to study it. They focus instead on the more mechanical aspects of TPS.

In the TPS philosophy, Respect For People has two components:

- Respect: We respect others, make every effort to understand each other, take responsibility and do our best to build mutual trust.
- Teamwork: We stimulate personal and professional growth, share the opportunities of development and maximize individual and team performance.

Respect for People is usually translated as Respect for Humanity. In a 2008 article on the Gemba Academy site, Lean expert and Japanese-English translator Jon Miller clarifies:

> The phrase 人間尊重 is not rare within the CSR (corporate social responsibility) statements of major Japanese corporations. The word 人間 means "human", "humans" or "people" and 尊重 can be translated as "respect." But the phrase used at Toyota is a bit different. It is 人間性尊重. The observant reader or student of Asian languages may recognize the extra character making "human" or "people" into "humanity" or "humanness."

Humanness is an interesting English coinage, as it seems to imply *the defining characteristics of humans*, as opposed to *humanity*, which suggests, simply, *the members of the set, Human.*

Miller further clarifies:

> So our current understanding of "respect for people" must be broader than simply respecting the rights of every person […] To be wordy, the literal meaning of Toyota's phrase 人間性尊重 is "holding precious what it is to be human" […] "respect for people" in my view is pithy but does not convey the full weight of these words in the original language.

Putting the phrase 人間性尊重 into Google Translate (as of April 15, 2019), the result is "Respect for Humanity." So, the full weight of the words isn't apparent unless you look a little deeper. It's worth a look.

The value for an organizational transformation lies in understanding what the defining characteristics of humans are and taking steps to create the conditions in which humans can thrive. It is the diametric opposite of treating people as "resources."

15 | Safety

> In one study investigating employee experiences with speaking up, 85% of respondents reported at least one occasion when they felt unable to raise a concern with their bosses, even though they believed the issue was important.
> — Amy C. Edmondson

In any learning organization, people need *safety*. In context, that means psychological and political safety to speak out about issues, to try experiments, and to make mistakes.

15.1 | The Business Value of Safety

In any organization that wants to be responsive and adaptable, safety is a prerequisite. The "mechanical" reason is that adaptability and responsiveness require decision-making authority to be pushed "down and out" throughout the organization. (Hope, Bogsnes)

A hierarchical decision-making structure in which every decision must be made (or at least vetted) by a "higher authority" can't quickly adapt to the unexpected, change course to take advantage of emergent opportunities, or innovate to shape new demand.

In most organizations, anyone who makes a mistake or who attempts to apply a new idea that isn't immediately successful will suffer consequences that may range from being marginalized and denied future career opportunities to just being fired outright. If the transformation goals include dynamic interaction with the market, then a work environment that values the humanity of the people working there is essential.

15.2 | Putting People First

Few consultancies that offer organizational transformation services place a high value on this sort of safety. As I write this, I'm aware of only two. Neither of them has the resources or capability to guide a comprehensive organizational transformation program. Each has its particular area of specialization within which they are excellent, but neither covers all the bases for organizational transformation. (It may be worth mentioning that I don't think *any* consultancy covers all the bases. Each covers different bases.)

I want to avoid mentioning specific companies in this book except when it helps illustrate a point, because companies change over time and what was true at the time of writing may not be true at the time of reading. I'll make an exception in this case and mention Industrial Logic, a California-based consultancy that operates in the "agile" space.

This is one of the very few consultancies that places humans at the center of everything, and they have quite a unique approach. A slogan of theirs is "Make people awesome," and it refers to customers, client teams, and their own consultants. Their consultants have the job title, *anzeneer*. It's a coinage based on the Japanese word 安全 (*anzen*) and the English word *engineer*. They are engineers of safety (Kerievsky).

15.3 | Importance of Safety in Times of Change

I've mentioned several times that stress is a natural effect of change, even when everyone agrees that the changes are necessary and will be beneficial. Under normal operating conditions, most people in most organizations already feel they aren't "safe" in the

sense we're discussing here. When we add the stress induced by a transformation program, they feel even more exposed than usual.

A good organizational transformation program must recognize, acknowledge, and manage this aspect of change. Purely "mechanical" changes in organizational structure, processes, tooling, and practices will have limited impact if the people are fearful of speaking up, trying things, or making mistakes.

With the approach suggested in this book, based on repetitions of the Cycle of Change, experimentation is key to the entire program. It's in the nature of experiments to reveal *better questions* rather than *final answers*. Wrong turns, blind alleys, and errors are normal, expected, and a natural part of the improvement process. Safety is fundamental.

Part 4: Change

> Nothing is so painful to the human mind as a great and sudden change.
> — Mary Wollstonecraft Shelley (*Frankenstein*)

This part deals with effective ways to drive or stimulate improvement. There are a couple of broad points to make on this subject.

First: Developing, delivering, and supporting a product or service require certain structures, processes, tools, and practices. Effecting organizational change is a different activity, calling for its own structures, processes, tools, and practices.

A successful transformation program must recognize these two distinct types of work and create the conditions necessary for both to be performed effectively.

Second: Many change agents are skilled at describing the goal state and identifying the gaps between the current state and the goal state, yet they struggle to define a practical series of steps to move the organization toward the target state. At times, it seems as if change agents expect the organization to flip abruptly to the target state.

A successful transformation program must identify a roadmap from the current state to the target state. The details are subject to change based on lessons learned along the way, but people need to have a clear idea of how to make stepwise progress. In most cases, so many changes are required on so many levels that an abrupt switch-over to the target state is not feasible.

This part addresses the issue on two levels:

- Including specific provisions for change in the structure of the consulting engagement; and
- Making stepwise improvements at the technical level aimed at the desired end state.

29 | Incrementally Collapsing Functional Silos

"Is that it?"
"No. That's a wall."
"It could be disguised."
"You're not very good at looking for things, are you?"
"I'm good at looking for walls. Look, I found another one."
— Derek Landy (*Kingdom of the Wicked*)

Given that changes in *structure* have the greatest impact, changes in *process* the next-greatest, and changes in *practices* the smallest, our series of incremental improvements should be determined mainly by structural changes, with changes in process and practices that support each structural improvement.

29.1 | Initial Organizational Structure

Many IT organizations are structured to maximize *resource utilization* rather than *throughput*. This sort of structure impedes continuous flow by creating numerous cross-team dependencies, leading to hand-offs, miscommunication, delays, errors, back-flows, rework, and unfinished inventory in the form of incomplete work waiting for the next functional silo to pick it up. Any given piece of work bounces around the organization like a Pachinko ball.

This sort of structure also tends to isolate the *makers* of solutions from the *users* of solutions. The result is the makers may have little understanding of, empathy for, or connection with the customers who use their products. They go through the motions of performing their tasks without any real commitment or interest.

The other side of the coin is the organization can enjoy the benefit of only a tiny fraction of the makers' creativity, intelligence, and skill, as they are locked into narrowly-defined roles and expected to repeat the same tasks over and over by rote.

All of that stems from a *utilization* mindset; the belief that results can be obtained efficiently if each person focuses on a single area of specialization, and managers orchestrate their activities. Once people begin to think about *throughput*, they quickly see the *utilization* approach isn't as effective as they had believed.

When we made the initial changes to begin the transformation in earnest, we collapsed functional silos to the extent possible given existing organizational constraints to create product-aligned cross-functional teams. In most large organizations, these initial teams will not include all the responsibilities for supporting their products. Some service teams will remain, and some silos based on technology stacks will remain.

Going forward, we want to continue to break down walls between activities and responsibilities that have become separated over the many years of focusing on *resource utilization*.

Most organizations can't just leap to a perfect structure in a single go. Ironically, to sustain the structure of disconnectedness, too many things are interconnected to allow for a quick or easy fix. The problem is they're not connected in a way that aligns with products or value streams. What might be a reasonable step-by-step approach to collapse functional silos?

29.2 | Starting Point

Let's visualize a starting point for this dimension of improvement in which pretty much all responsibilities are siloed, and see how we could begin to improve the structure.

29.2.1 | A Separate Team For Every Type of Task

Typically, all the work pertaining to a given set of technologies or platforms is isolated from the work for other technologies or platform. For each technology group, all the usual activities surrounding software creation, delivery, operation, and support are duplicated. Within each technology group, each of those activities is isolated or "siloed." Some responsibilities are global, or enterprise-wide.

Here's a grid that illustrates what I mean. Each X is a separate team or set of teams in the organization.

	Global	Mobile & Web	Distributed & Mid-Tier	Back-End
COTS & ERP Support (1 team per product)	X			
Security	X			
Database (1 team per product)		X	X	X
Network (1 team per zone or subnet)	X			
Operations			X	X
Production Support (1 team per set of apps)		X	X	X
Enterprise Architecture	X			
Solution Architecture (1 team per app)		X	X	X
Analysis & Requirements 1 team per set of apps)		X	X	X
UX, Usability, Accessibility		X		
Unit & Component Test (1 team per set of apps)		X	X	X
Integration Test (1 team per set of apps)		X	X	X
End-to-End Functional Test (1 team per set of apps)		X	X	X
System Test (1 team per set of apps)		X	X	X
Solution Design & Programming (multiple teams, "fungible")		X	X	X
Solution Build		X	X	X
Continuous Integration		X	X	
Intrastructure Engineering		X	X	X
Solution Deployment		X	X	X

Figure 29.1: Teams organized as functional silos

Notice there is no direct connection between any team and any product or service provided to customers. Teams are organized around activities rather than business value.

29.2.2 | Similar Tasks Segregated by Technology Stack

Every category of work involved with software creation, delivery, operations, and support is handled by a different team, and nearly all categories of work are segregated by platform or technology stack.

For example, there may be a team that performs end-to-end functional testing for just one application or a small set of applications

on just one technology stack, such as Mobile & Web or Back-End (i.e., mainframe). These test teams may be somewhat aligned with products, but they are divided across the various technology stacks in the organization.

Similarly, there may be a pool of programmers who are assigned to different projects depending on which projects call for programming to be done. These work groups (not "teams" in any meaningful sense) are temporary, not aligned with any single product, and segregated by technology stack. Within any one technology stack, programmers may be further segregated to work only on front-end or only on server-side code.

A production support team will deal with just one technology stack, and will have to support all the applications (or parts of applications) that run on that stack.

29.2.3 | IT Becomes a World Apart

No one has visibility into a value stream or product line. From a human standpoint, to a person working in one of these teams, the world looks like an endless series of one-off, disconnected requests for small tasks. There's no sense of purpose or value.

From a political standpoint, this structure begets a thick middle management layer in which career advancement is all but impossible. That leads mid-level managers to compete with each other for visibility, credit, and promotion opportunities. They often pursue this in ways that further impede continuous flow, as they try to undermine one another by delaying selected requests for the teams they manage.

From a mechanical standpoint, every initiative requires services from multiple teams, each of which has its own work queue and priorities. The probability is high that any given work item will be in a wait state for nearly 100% of the time it's in the system. Cycle

efficiency in organizations that are structured this way tends to be in the 0.5% to 2% range.

29.3 | Organizational Constraints on Silo-Busting

The strategy described in this book calls for some initial changes based on what we already know about what "works well." With respect to team structure, that means establishing the closest thing to product-aligned cross-functional teams as we can within the prevailing organizational constraints.

What are those constraints? Bear in mind this is a general model, and real companies will differ to some extent. Yet, I think this is pretty accurate on the whole.

I see three levels of constraints with respect to restructuring IT organizations. From large to small, they are:

- Separation of work and people by technology stack, e.g.
  - Mobile & Web
  - Distributed & Mid-Tier
  - Back-End, such as mainframe and legacy midrange systems
- Separation of people by the types of activities they perform
  - Operations
  - Production Support
  - Security
  - Software design, creation, delivery
  - Infrastructure support
- Separation of people by role or tasks
  - Analysis & Requirements
  - Testing (multiple types)

- Enterprise Architecture
- Solution Architecture
- Solution Design & Programming
- Solution Build
- Continuous Integration
- Deployment
- *more*

Notice what's absent here: No team anywhere in the organization is explicitly aligned end-to-end with a system that serves a category of customers.

It's a Matryoshka doll. The largest doll is the technology stack. Inside that one there's a "types of activities" doll. Inside that, there's a "role or task" doll. So, we have (for instance) "end-to-end test" teams for mobile & web, for mid-tier systems, and for back-end systems. It's the same general structure across all roles and tasks. Here's a highly artistic visual representation:

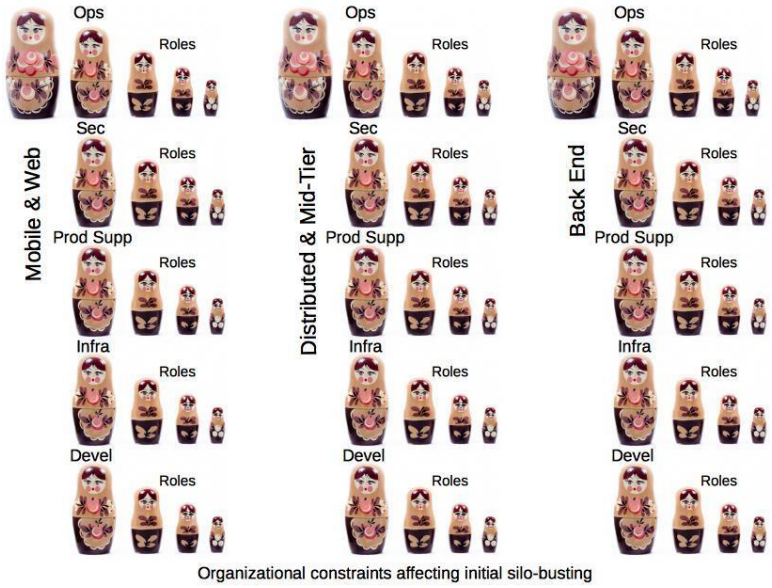

Figure 29.2: Silo-busting constraints

29.4 | Temporary Scaffolding

There is a fundamental concept that most change agents seem to overlook: When you move toward an end state one step at a time, you are not necessarily implementing permanent changes all along the way. You might introduce a change that will become obsolete after further improvements have been made.

I think of this as analogous to ancient construction methods. When people in Egypt, Greece, Rome, or the pre-columbian Americas built pyramids and temples, and when ancient Europeans built structures of large stone monoliths as at Stonehenge, they had to lift very large, heavy blocks to great heights.

To accomplish this, they carved the blocks with *bosses*, or protrusions that stuck out on the sides. They could secure ropes to

these bosses and use various rope-and-pulley systems (depending on the technologies of the particular civilization) to raise the stones into position on top of other stones. Then they carved away the bosses and finished the exterior of the wall or column in whatever way they wished, either by hand-carving the final designs into the surface, or adding a plaster or limestone coating.

This photo shows unfinished column drums on the north wall of the Acropolis in Greece. You can see peg-like protrusions sticking out from the sides of several of them. Those are the bosses. They would have been used to lift the drums to their final position, and then they would have been carved off.

Figure 29.3: Unfinished column drums, Acropolis

To construct columns out of "drums," the ancient Greeks and Romans cut guide holes in the center of each drum and used wooden pegs to guide the next stone into the proper position on top of the previous stone. Then they finished the exterior of the column so that it looked like a single piece.

Coaches who are guiding teams through a series of "baby steps"

toward a more-effective way of working may have to do something similar.

When we introduce a practice such as defining a "definition of done" for handing off requirements from an analysis team to a programming team, we intend to discontinue that practice once the analysts, testers, and programmers have learned to collaborate directly.

When we introduce an idea such as "relative sizing" to support team-level estimation, we intend to discontinue the practice once the team has learned to slice work items vertically into similarly-sized chunks, and to use forecasting to plan their work.

When we introduce a time-boxed iterative process model to help a team manage its work flow, we expect that eventually the team will learn to operate in continuous-flow mode, and we can dispense with the explicit iterations.

We will not necessarily move directly from the current state to the ultimate target state in a single step, and some of the artifacts we put into place early in the process may be removed as we advance.

So, as you read about step-by-step ways to move toward the goal state, don't worry if you see practices being recommended that you know aren't ideal. They are being recommended at particular stages in the progress of the organization or team, for specific reasons; typically, because the team hasn't yet learned to collaborate at a more-effective level, or hasn't yet learned some particular technique or practice. Much of this temporary scaffolding can be discontinued at an appropriate point in the future.

29.5 | First Steps in Silo-Busting

The larger the doll, the harder it is to break. Therefore, a sensible starting point for step-by-step structural improvement is the

smallest doll: Collapse specialist teams into cross-functional teams *within* a given category of activity *within* each technology area.

This will affect primarily the *software design, creation, delivery* and *infrastructure support* categories , as they tend to be broken up into more specialized teams than the *operations* and *production support* categories.

Bear in mind this chapter is strictly about silo-busting. There will be corresponding and mutually-reinforcing changes in *process* and *technical practices* happening at the same time. Those will also be done in a measured, stepwise way.

29.5.1 | Analysis, Test, Design, Programming, Build, CI

Let's look at the category of work around analysis and requirements, solution design, low-level testing, programming, solution build, and continuous integration.

29.5.1.1 | Step 1: Create Product-Aligned Development Teams

As part of our initial restructuring, before we begin the recurring series of experiments to drive incremental improvement, we'll preemptively break up some of the functional silos within this area of practice. Where the organization had a matrix structure comprising "fungible" teams for analysis, testing, design, programming, and build scripts and continuous integration, we'll combine those functions in ways that are relatively easy to accomplish given current organizational constraints.

We'll reshuffle people into product-aligned development teams with the following characteristics:

- aligned with products - each team supports a product; they

are not "fungible resources" that can be assigned tasks pertaining to different products; note this goes hand-in-hand with shifting from a project-focused delivery model to a product-focused one;
- stable - the teams are meant to exist indefinitely; rather than forming a team when a project comes up, we prefer to give long-lived teams work pertaining to a product; note: this doesn't mean individuals have no options to change teams in order to learn new things and advance their careers;
- cross-functional - each team has people on board who have the skills necessary to support the product; in this general category of activity, those skills will mainly concern analysis, testing, application design, and application programming.

Note that these teams will not be able to take on end-to-end responsibility for an application that cuts through the enterprise technical infrastructure. That's a longer-term goal, but this early in the transformation program the groundwork has not been laid for it. These aligned teams will have responsibility for a given product within a given technology stack only.

The initial teams also won't be able to support all aspects of the application. This example concerns activities such as solution design, programming, testing, and requirements analysis. It doesn't include activities around builds, continuous integration, database support, network support, deployment, infrastructure management, operations, or production support. Baby steps.

We'll establish "service" teams for some of the other functions in this category, with the longer-term goal of folding these activities into the development teams, as well:

- UX, Usability, Accessibility, Localization
- Build Scripts, Continuous Integration support

Other existing service teams will remain in place at this time:

- Database
- Network
- Security

29.5.1.2 | Step 2: Encourage Collaboration Across Certain Roles

Some of the roles in this general area of activity have natural affinities. It's sensible to encourage collaboration between people in these particular roles as a way to begin softening the boundaries between roles.

Functional silos with natural affinity:

- analysis and testing
- design and programming

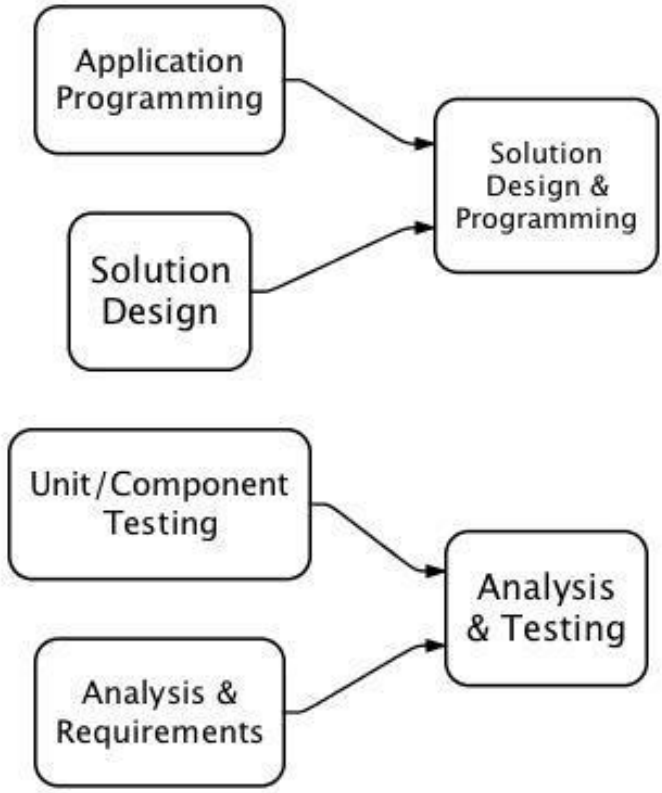

Figure 25.4: Role affinities in development (1)

There's natural affinity between the work of requirements elaboration and the work of creating test plans. It tends to be easy for analysts and testers to collaborate.

There's natural affinity between software design and programming. In organizations that segregate these activities, it tends to be easy to collapse those silos. In many organizations, these activities will not be split to begin with, and there's a little less work to do to break silos.

29.5.1.3 | Step 3: Create Definition of Done and Ready

The terms *definition of done* and *definition of ready* are widely used in the Agile community to refer to explicit rules for handing off entire User Stories. People familiar with kanban systems speak of *explicit policies for pull.*

Here, the idea is generally the same, but the scope is smaller. When we have, say, an analyst and a tester on a development team, and they're still at the level of proficiency where they hand off incomplete items to one another for specialized work, then the usual starting point is that they have no explicit rules for those hand-offs.

Each person hands off work items when they feel they've done what they're expected to do. It's common for the recipient to need further clarification or refinement of the item before they can begin their part of the work. The result is irregularity in flow, and a possibility of miscommunication.

An initial step to smooth this out can be to make the hand-off policies explicit. These written rules can be discarded later, when the parties have learned better ways to collaborate. This is only a baby step to get things moving in the right direction.

29.4.1.4 | Step 4: From Hard Hand-Offs to Safe Hand-Offs

In a 2013 blog post, "How Does Collaboration Begin?" I share a story about that question. It was posted online, and a number of well-intentioned Agile practitioners replied; but they didn't answer the question. They merely reiterated a description of how well-functioning Agile teams work together. The question wasn't about end states; it was about *how to begin.* What's the first step? What's the second step?

I used an analogy from a handgun safety class my family and I took. Here's an excerpt:

One of the things you might do with your firearm is to hand it to someone else. As you might imagine from many years of watching movies and television, there are many different ways to hand a firearm to someone else. You could, for example, drop it on the floor and kick it across the room, loaded and cocked. This is analogous to the way software development specialists hand off work between functional silos; you know, the old "throw it over the wall" thing.

…here's what we were taught about handing off a firearm: First, you clear it, then you hand it off. "Clear" means you check the magazine, action, and chamber to make sure there's no ammunition in the gun. Then you ask someone else to look in the magazine, action, and chamber to confirm there's no ammunition in the gun. You're still holding the gun at that point. Making sure the gun is pointed in a safe direction at all times, you can then hand it to someone else. The procedure for that is to look at the person while he/she looks at the firearm. You present it to him/her, pointing it in a safe direction at all times, stock first. You don't let go of it until he/she verbally confirms that he/she has a firm grip on it. You look at the firearm to see for yourself that it appears as if he/she does, indeed, have a firm grip on it. Then you can let go.

On a software development team of specialists, a typical work flow is for an analyst to hand off an artifact to a programmer, who later hands off an artifact to a tester. These various interim artifacts eventually become a piece of working software. Traditionally, the hand-off procedure is to "throw it over the wall" to the next functional silo in line. It's exactly the same as dropping a loaded gun on the floor and kicking it across the room,

> spinning in all sorts of unsafe directions and colliding unsafely with all sorts of unsafe obstacles.

With a safe hand-off, we do the same thing. The person handing off work explains in great detail everything about the work, based on the Definition of Done. The person receiving the work makes very sure they understand everything clearly, and their own Definition of Ready has been met.

Depending on circumstances, and particularly for a software development team comprising the roles suggested in this section, another mechanism for safe hand-offs is known as The Three Amigos. Created by Agile consultant George Dinwiddie, The Three Amigos brings together an analyst, tester, and programmer just at the point when the team pulls a work item (such as a User Story). They sit down together and make very sure everyone has clarity about what is to be done. It's a mechanism for the over-communication I mentioned.

29.4.1.5 | Step 5: From Hand-Offs to Joint Work

As people become familiar with what their team mates need from them for each particular hand-off, the over-communication involved in the "safe" hand-off procedure starts to feel overly formal and time-consuming.

At that point, people are probably ready to begin doing the work jointly, working in pairs or slightly larger groups. The analyst now contributes to test plans, and the tester participates in analysis. Their roles are beginning to collapse into a single role.

29.4.1.6 | Step 6: Repeat at the Next Level of Silos

For development teams, there may be two or more "levels" of functional silos that can be broken in this way. Now that the analysts and testers are merged, and the designers and programmers are

merged, we can collapse more siloes and cultivate more generalists on the team.

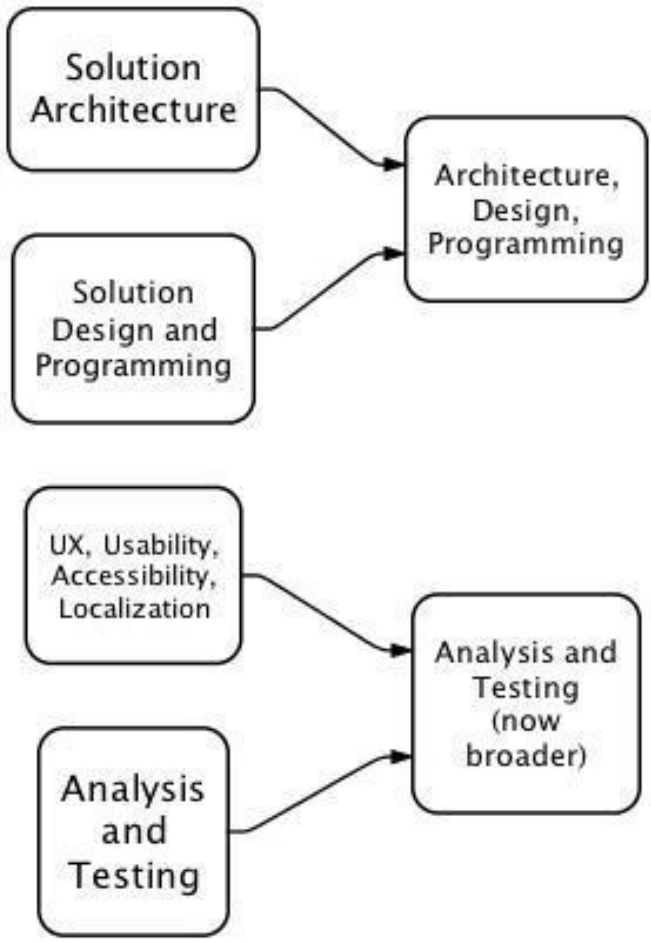

Figure 25.5: Role affinities in development (2)

This iterative process of merging previously-siloed roles on a

development team can be taken to whatever level makes sense in context to support the competitive capabilities that represent the business goals of the transformation.

29.4.1.7 | Step 7: Fold In Support for Builds & CI

For development teams that are doing well after Step 6, the next easiest baby step for silo-busting is to start migrating responsibility for builds and continuous integration support from the separate service team into the various development teams.

There is value to the organization in deciding on a single build/CI toolchain per technology stack, but there is no need for all activity around builds and CI to be isolated in a functional silo. Development teams are well able to handle this work on their own. This will eliminate another cross-team dependency with its attendant delays and inevitable miscommunications.

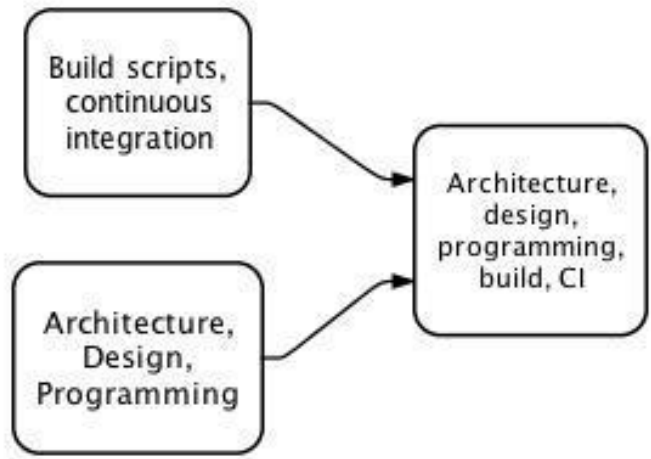

Figure 25.6: Role affinities in development (3)

Build scripts and CI servers are not difficult for product teams to manage, but there are other functions of IT operations that have traditionally been centralized, and for which a degree of central control and standardization is valuable. See section 29.5 below for more on the topic.

29.4.1.8 | Step 8: Collapse Programming and Testing Silos

The next logical step in collapsing silos within product development teams is to start to merge the programming-related and the testing-related activities. We can use the general formula we used before, starting with safe hand-offs and gradually moving toward more-effective methods of collaboration, and finally extending team member skill sets to cover both sets of activities.

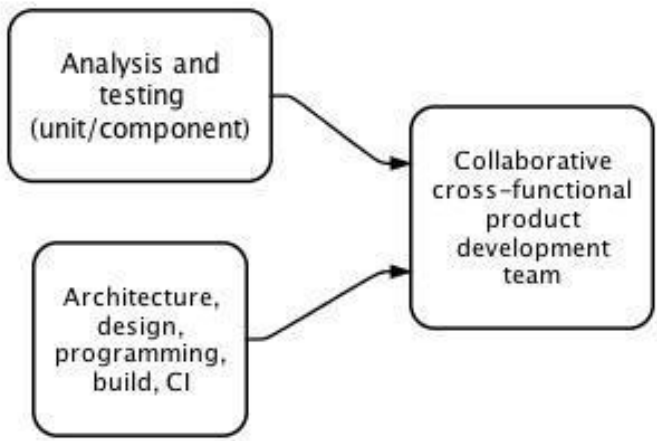

Figure 25.7: Role affinities in development (4)

This chapter doesn't go into improvements in process or technical practices, but the team structure that results from this collapsing of

functional silos can enable good practices such as

- specification by example
- acceptance test-driven development
- test-driven and behavior-driven development
- whole-team exploratory testing
- *more*

It can also give each team member a broader perspective on the work. A programmer who learns to wear a testing "hat" will approach solution design in a way that enables testing. That in itself tends to result in designs that are more robust, resilient, simple, and loosely-coupled, as well as source code that is better organized and more understandable, than when programmers aren't in the habit of thinking like testers.

Similarly, a tester who learns basic coding skills will be able to take advantage of the many test automation tools available, as well as using automation to support genuine testing activities (as opposed to validation or checking).

An analyst who learns something about testing and programming will be better equipped to write clear and meaningful requirements.

Similar benefits accrue across all the various skills that had been siloed previously.

29.4.2 | Deployment, Release, Operations, Support

Let's apply the same step-by-step approach to busting silos around another general category of work: deployment, release, operations, and production support.

Most organizations that I've seen or heard of have numerous specialized teams, each of which carries out just one type of

task. One team provisions environments. Another team moves applications from one environment to the next. Another team handles production releases and rollbacks. Another team monitors the production environment.

Many large organizations isolate the following activities in separate service teams that support all the development teams:

- migrating code through environments
- releasing applications into production
- production operations
- production support

There may be even further, fine-grained breakdown of the work. For instance, within the infrastructure engineering or platform engineering team, there may be specialized sub-teams that handle just one operating system; a sub-team for Linux, a sub-team for Windows, etc. There's also the same separation of teams per technology stack as we saw for software development activities.

Collapsing all these silos can take time and patience, especially when we consider that nearly all this work is probably being done manually, and nearly all of it will have to be automated or at least streamlined to achieve goals like *Respond to Change*, *Shape the Market*, and *Retain Customers* (see "Leverage: Where's the Fulcrum?").

The end state is likely to require a single team or small group of teams to support each product or product line from concept to operations. That means collapsing the broader development and operations areas. For now, we're concerned with the initial steps we can take to begin the process of collapsing silos. We're not ready to fold development into operations just yet.

29.4.2.1 | Step 1: Restructure Service Teams

To enable collaboration across previously-siloed specialists, we want to collect closely-related responsibilities together. Responsi-

bilities that have natural affinities may include:

- provisioning environments and moving code through environments
- operations and first-line production support

29.4.2.2 | Step 2a: Collapse Provisioning Environments and Migrating Code

We know that the end-state vision calls for creating environments dynamically as needed (to the extent possible). Ultimately, the work around provisioning environments has to be folded into the development process, picking things up where continuous integration leaves off. That may be too large a step to take at the outset.

A practical initial state may be to collapse the silos for *provisioning environments* and *moving code through environments*.

The *status quo ante* in many organizations is to dictate how certain "standard" environments will look, and requiring application designers to fit their solutions into that.

We want engineers to start thinking in an application-focused way rather than a platform-focused way, to encourage thinking about provisioning from the point of view of each application's needs.

Engineers responsible for provisioning environments tend to create a server instance and try to keep it stable and operational for as long as possible. This is traditional infrastructure management.

We want engineers to think of server instances as disposable resources that can be created or reset on demand, when applications need them. This is dynamic infrastructure management.

If the same team provisions environments and migrates application code through them, they will naturally start to think about these things in a holistic way. So, this is a baby step in the direction we need to go.

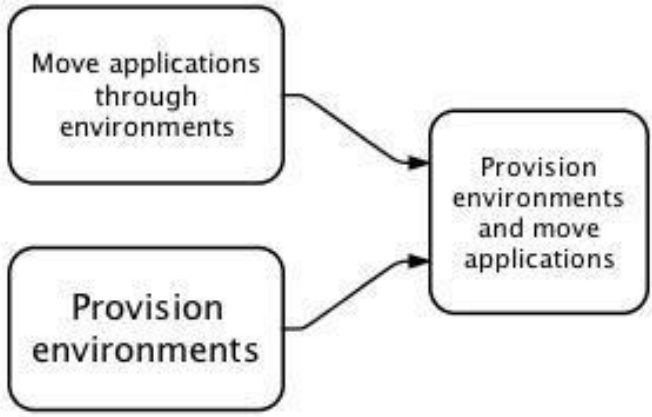

Figure 25.8: Role affinities in engineering

In conjunction with this structural change, we'll be introducing automation for provisioning server instances in a consistent way. Before long, we'll start maintaining the provisioning scripts under version control, moving toward *infrastructure as code* (Morris).

29.4.2.3 | Step 2b: Collapse Operations and First-Line Support

Ultimately, we want a product-aligned team to handle operations and production support, as well as development and deployment, to remove delays and waits from the process.

The *status quo ante* in many organizations is that these two functions are separated. One team monitors production systems, but mainly with the goal of keeping them up and running. Another team receives trouble reports from users, performs some degree of triage, and if they can't solve the problem they hand it off to second level support.

One mechanism to remove delays and waits from the process and

to help ensure the stability of production operations is to adopt the so-called *serverless* mentality. It means we're thinking mainly of the applications and their users, rather then the servers on which the applications run. In contrast with the past, an application is long-lived and the servers that support it may come and go. Standardization and automated provisioning of servers enables this by simplifying and rationalizing that work.

A baby step in that direction is to combine the operations team and the first-level support team. For now, we'll still have second- and third-level support handled by different teams (third-level support is probably the development team). Eventually, the product-aligned cross-functional team will handle everything, but we aren't there yet.

29.4.2.4 | Step *n*: Collapse Operations/Support and Development

This collapse will become feasible after the development area and engineering area have separately improved their structures, processes, practices, and tools to a certain extent. Merging these broad areas of responsibility into a single product-focused team is a big change, and one that requires considerable ground work to be laid before it can happen.

There may be several interim steps here, as test automation grows from the development side forward, and dynamic infrastructure management grows from the engineering side backward (where "forward" means "toward production"). I sometimes visualize this as two sets of tendrils reaching toward each other, gradually entangling, and eventually becoming One.

29.4.3 | Collapsing the Big Dolls

Client leadership may decide to stop collapsing silos at some point, when they're satisfied with the way the organization is operating. But if silo-busting is taken to the *n*th degree, we will eventually collapse the technology-based teams into a single, end-to-end, product-aligned team. That team will include all the skills needed to deliver product features and modifications from mobile through the backiest of back ends.

It boils down to a repetition of the process of

- putting people together
- encouraging them to collaborate directly
- cross-skilling through collaboration
- blurring the lines of specialization between them

guided by the Cycle of Change, to ensure we don't go off on a tangent and we remain focused on the business goal.

29.4.4 | Ongoing Iterative Improvement

It will almost always be feasible to find the next "level" of affinity between two roles or teams, and collapse them using very small changes controlled by the Cycle of Change, with clarity about the impact of each change and the ability to quantify the impact of the change. Affinities at "higher" levels depend on our having collapsed teams at "lower" levels of affinity, as suggested by the progression described in this section.

While the team structure in the engineering area may be more complicated than that in the development area, the changes in processes, practices, and tooling are likely to be more challenging for engineering than for development. So, there's more early transformation work to do in those areas than there is with respect to structure.

29.5 | Special Considerations When Decentralizing Responsibility

Changes in team structure that distribute responsibility for a function from a previously-centralized service team into numerous development teams introduce a new need: Maintaining consistency and simplicity in the technical infrastructure without creating a bottleneck for product delivery.

When we decentralize responsibility for things like build scripts, continuous integration servers, version control systems, provisioning of environments, and other functions, we need to avoid numerous different tools or versions of tools from proliferating in the environment.

We're pulling responsibility for something out of a central group and distributing it across multiple teams. Those teams will be focused on supporting different product lines or value streams. They won't automatically coordinate changes in the infrastructure or platforms they use.

Depending on the nature of the work, this can be managed in one way or another without reverting to 20th-century controls.

29.5.1 | Central Definition With Self-Service

Version control systems, platform engineering, dynamic creation and destruction of virtual machines and containers, and other needs can be defined by a centralized team and then presented to product teams via a self-service system. The self-service system limits choices to approved products and versions of products and assures product teams will make choices consistent with organizational standards.

A pitfall to be aware of is that when there is a centralized team, such as a platform engineering team, their work is often prioritized

below product-related work that has an obvious connection with customers. One way to mitigate this is to ensure every change to any application or to the technical infrastructure is justified on the basis of customer impact.

The naive view that a change has to have an immediate and obvious impact on customers leads people astray. An infrastructure change whose impact is many steps removed from any customer is still necessary to support customers properly. The exercise of justifying those changes in terms of customer impact leads everyone involved to a clearer and more-consistent understanding of customer needs.

It's generally best to provide self-service facilities to product teams to the extent possible. This will minimize cross-teams dependencies with the central team and help avoid delay, miscommunication, back-flows, and rework while making delivery forecasts reliable and predictable.

29.5.2 | Documented Standards With Definition of Done

Continuous integration servers, build tools, server configuration, API design standards, naming conventions and the like can have documented standards. Whenever a product team makes a change that is governed by the standards, they know where to go to to find the standards and possibly working samples that demonstrate how to apply the standard. With that in place, product teams must include checks for standards compliance in their Definition of Done for work items.

Beyond compliance, there will be many times when changes to a standard are necessary. The documented rules must include an appropriate procedure for changing the standards. The guiding "beacon" for the transformation program provides natural guidance for these rules: The key requirement is that changes will not destabilize production operations. (See "Leverage: Where's the Fulcrum?")

A second requirement for changing organization-wide standards is that existing solutions must not "break." Some kind of versioning scheme or other safeguards to assure backward compatibility (within reason) must be developed concurrently with the first change of this kind.

29.6 | Team Size

In 20th-century organizations, team size was determined by adding up the number of people who were assigned to a project, either full-time or part-time. The same individual might be counted multiple times, if they were assigned to multiple projects concurrently. This led to teams sizes that could range over the 100 mark.

But these were not "teams" in the sense Lencioni writes about. They were too large and dispersed to function cohesively as real "teams," and the members were not dedicated to any single project. We would consider them "work groups" today, rather than "teams." The work groups were formed when a project was formed, and disbanded when the project was closed.

29.6.1 | Practical Team Size for Collaboration

Today, the idea of a "team" is a little different. It's a cohesive group of people who interact frequently and collaborate closely to achieve common goals, all dedicated to a single initiative at a time. It has the characteristics of a good team as defined by Lencioni.

To that end, the size of a team has to be amenable to collaborative work, and preferably collocated work. In my experience, teams as small as 2 people have been effective, depending on the nature of the work they performed. Teams as large as 12 to 15 people have also been effective and able to function cohesively.

But once you reach a size of around 12 to 15 people, teams start to divide spontaneously into sub-teams. They generally split up along logical "seams" in the work. For a development team, that could mean people interested in front-end development would gravitate together, leaving people interested in back-end development to form a separate sub-team.

It's not wrong for teams to be divided, but it's best to divide them mindfully and with purpose than to allow it to happen at random and without visibility.

For instance, in the example just given, we probably wouldn't want the front-end developers separated from the back-end developers. If we thought it was appropriate to split the team, we'd create two cross-functional teams with responsibility for some distinct subset of the product, organized in such a way as to minimize cross-team dependencies. That's unlikely to result from random chance.

One challenge with larger team sizes is that with more people involved, there are more lines of communication. Here's an image found on StackOverflow that illustrates the point. I don't know the origin of the image.

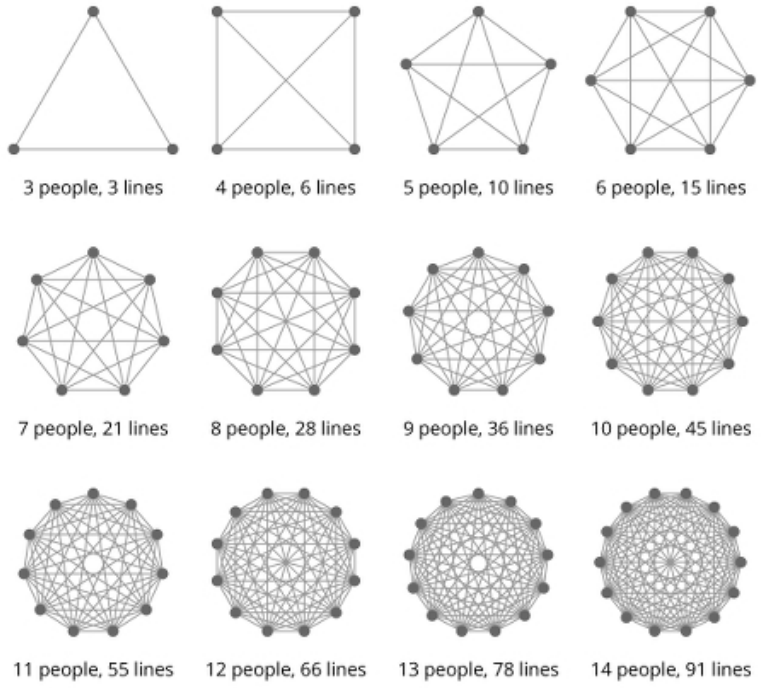

Figure 25.9: Team size and lines of communication

Intra-team communication overhead can overwhelm delivery effectiveness as team size grows beyond natural limits for collaboration.

On one engagement, I was the delivery lead as well as technical coach for a team. (That's not a good idea; the two roles have conflicting mandates! But stuff happens.) My manager approached me one day to remind me that the team was not on track to meet its commitments (this was an agile-in-name-only waterfall project).

He asked me how many more developers I needed to catch up with the delivery plan. I suggested we reduce the team from 8 to 6 developers. He frowned and asked whether some of the developers weren't pulling their weight. He was already reaching for his Firing

Pen.

I assured him everyone was pulling their weight. I wanted to reduce the communication overhead on the team so that they could move faster. He genuinely did not understand the concept. He agreed, but told me that if the idea didn't work, it would be on me.

The team of 6 increased the pace of delivery and completed all the required work before the deadline. The team went from 28 lines of communication down to 15, making it easier to get things done. The improvement in communication efficiency more than compensated for the smaller number of developers.

29.6.2 | Collapsing Silos Increases Team Size

As we collapse functional silos, we'll be increasing the number of individuals on each team. For example, if we combine a team of programmers with a team of analysts and a team of testers, we may end up with a pretty large group:

Original configuration

- Team A = 4 programmers
- Team B = 4 programmers
- Team B = 4 analysts
- Team C = 4 testers

Consolidated team:

- Team ABCD = 8 programmers, 4 analysts, 4 testers

That's 16 people, probably beyond the practical limit for a group to function cohesively as a team in the full sense of the word. It also may not be an ideal mix of skill sets for the work at hand. If we have aligned teams with products properly, this large team may also contain more people than necessary to fulfill its mandate.

29.6.3 | Multiple Teams Supporting One Product

It's possible we can identify a logical breakdown of the product the team supports such that two teams could operate with minimal dependencies on one another to support the product. We could then have:

- Team New-A = 4 programmers, 2 analysts, 2 testers
- Team New-B = 4 programmers, 2 analysts, 2 testers

This means coordination to manage multiple work streams that support the same product.

29.6.4 | Coordinate Changes in Structure, Process, and Practices

It's possible that with corresponding improvements in process and team member skills the same work can be done by fewer people, but initially the teams will be larger with each silo collapsed.

As we go forward with further silo consolidation, Team New-A may end up looking like this:

- Team New-A = 4 programmers, 2 analysts, 2 testers, 2 infrastructure/platform engineers, 1 DBA, 1 UX specialist

That's close to the practical limit for cohesive team collaboration. Improvements in process, team member skills, technical practices, tooling, and progress with automation can alleviate the need for large teams.

For that reason, changes in structure, process, and practices must be closely coordinated. One area must not be allowed to race ahead

of the others. The second level of silo-busting can only proceed when team members have settled into each succeeding level of collaboration and cross-skilling.

29.6.5 | Beware of Magic Numbers

As a technical coach, you may come into an organization that's already been damaged by a previous Agile transformation. You'll have to adapt to the client's actual starting point. Every company won't have the same situation as presented in this chapter.

David Anderson tells a story of a client that told him they had hired 400 new Product Owners, and didn't know what to do with them. When he asked why they had hired so many Product Owners, they explained their Agile coach had told them each team had to comprise 6 people, and each one needed its own Product Owner. Anderson asked whether they had 400 products. They did not.

You may have heard there's a magic number of "seven plus-or-minus two" team members for effective collaboration. That's a rule of thumb, not a rule carved in stone.

Your experience may differ, but I've seen teams ranging from 2 to 15 members working effectively together. It depends on the kind of work they're doing.

Furthermore, there's no "rule" that says each team requires any specific number of people in particular roles. The skillsets needed on a team depend on the kind of work they're doing.

In fact, there are situations in which a *work group* of up to 40 people can be effective, when the work they do doesn't particularly call for close collaboration. A "true" team isn't necessary in all situations.

I've been in large organizations that had 600 to 800 software development teams. When the coaches went out onto the floor to work with the teams, we discovered very few of them were

actually "software development" teams. They performed various tasks relative to the company's IT function.

Previous Agile coaches had advised management to create a large number of small teams and call them all "development" teams. Each of them was required to have a fixed number of analysts, testers, and programmers.

Therefore, the team members had those titles. None of that reflected the work they actually performed or the skills they actually possessed. The situation made it hard for us to determine how best to help the organization.

Your colleagues on the Consulting Team who work with higher levels of management may not easily relate to this situation. Most people in that sort of role don't know what the technical work really looks like. Calling everyone a "developer" seems fine to them. So, you can't necessarily count on the results of the consultancy's initial assessment to understand the client's real starting point.

29.7 | Security

If we look at IT operations strictly from a Lean point of view, we would want to incorporate security seamlessly. But for legal and regulatory compliance reasons as well as to minimize the risk of collusion to commit fraud, it makes sense to separate the security function from the rest of IT. This can go so far as to have a separate management hierarchy just for security, that doesn't report to the CIO, and doesn't depend on the IT department's budget.

This can be done in a way that doesn't introduce excessive delay or bureaucratic overhead for the delivery pipeline. Many security issues are "known" and it's feasible to build checks for them into the delivery pipeline.

Other security-related issues pertain to application design and server configuration. Those matters can be documented, included

in teams' Definition of Done, and verified as part of automated test suites.

The separate security group would be responsible for defining those things and for helping development and engineering teams understand their role in maintaining security. They would also be responsible for proactive security-related activities and testing.

The security group will have a great deal to say about network perimeter configuration, among other things. But they will never become part of a product-focused cross-functional team.

29.8 | Summary

In this chapter, we've started to see a series of steps for *beginning* to collaborate. So far, in the present section, the steps have been:

1. Establish cross-functional teams. That puts people with different specialties together. It's possible they have never occupied the same physical space at work. Initially, they may continue to work separately and pass work to one another. But just the fact they're in the same space will help them learn a little about each other's work, and about what each is looking for from the other.
2. Create Definition of Done and Definition of Ready for each type of task the specialists on the team carry out. The purpose is to make it explicit just what each specialist requires to begin their work, and just what has to be added to the work item for it to be considered "done" by that specialist. (These formalities can be dropped later in the transformation when teams are collaborating smoothly.)
3. Encourage collaboration across certain roles. Some roles have natural affinities that make it easy for practitioners to relate to one another's work, such as analysts and testers. Initially when they sit together they may watch one another work,

or work independently but near enough to see each other's work and talk about it. It's pretty automatic that they will start to ask what the other person needs, and begin to tailor their work to make the other person's job easier.

4. Encourage the people to collaborate directly together. A baby step in that direction is to shift from hard hand-offs to "safe" hand-offs. That's a semi-formal process for passing work back and forth that involves "over-communicating" details about the work. The people must share a lot of information in the process of performing a safe hand-off, and they begin to learn more about each other's work.
5. Once safe hand-offs have become routine and feel "too formal" to the people involved, a natural next step is for them to *do* the work of both roles jointly. Rather than just sitting next to each other as in Step 1, they now sit next to each other and jointly complete tasks. For example, if they are an analyst and a tester, then at this stage of cultivating collaboration the analyst is helping create test plans and the tester is contributing to analysis.
6. Keep improvements in structure, process, and practices in sync. If one area goes faster or slower than the others, there will be friction.

The same pattern applies to all the specialized silos that may exist in the IT organization.

Part 5: Coaching

> Coaching, so you are no longer needed.
> — Steve Chandler (*Hands Off Manager*)

Coaching is an important part of guiding an organizational transformation. With particular emphasis on technical coaching, this part deals with the working definition of "coaching," the skill sets needed by a coach, and the reasons for the severe shortage of qualified technical coaches.

I also describe a practical approach to technical coaching that is becoming more widely used *circa* 2019, and that avoids some of the issues the technical community has experienced with ineffective or non-sticky technical coaching in the past.

In addition, I mention specific communication frameworks, collaboration techniques, and "internalities" to help coaches deal with stress and maintain focus during engagements.

34 | Coaching Skills

> A good coach can change a game. A great coach can change a life.
> — John Wooden

People who work in the organizational transformation field mention the words "coach" and "coaching" quite a bit. So do I, throughout this book. What does it actually mean?

In my view, to function effectively as change agents we need to cultivate skills in the following general categories:

- Guiding teams and individuals
- Facilitating events and meetings
- Skills in practical application (process and technical)
- Skills in effecting change as such

34.1 | Coaching Competencies

The *Agile Coaching Competency Framework* developed by Lyssa Adkins and Michael Spayd provides a comprehensive model of skills relevant to helping people change and improve and for guiding organizational transformation. Although the framework has the word "agile" in its name, it's broadly applicable.

They list several important areas of skill development for people who want to work in this sort of role. Here they are, correlated with the categories I listed above (those categories are not part of the framework):

- Coaching (Guiding)

- Mentoring (Guiding)
- Teaching (Guiding)
- Facilitating (Facilitating)
- Technical (Application)
- Business (Application)
- Transformation (Change)

Most of the following definitions aren't taken directly from the framework, but represent my understanding of the terms. The meanings are consistent with those of the framework.

34.1.1 | The Guiding Competencies

Coaching is helping others discover their strengths and find their own way forward. It's important that the coach not impose any personal growth objectives on the coachee. The coach's role is to help the coachee progress toward their own objectives.

That's a general definition. In the context of a transformation program, the coach must guide coachees toward the defined goals of the transformation, so it isn't solely about their personal growth goals. This can involve helping people recognize how their personal goals may align with the organization's goals.

The word *coach* is overused in our field. In the context of working with technical teams, people usually use the word *coach* when all they really mean is someone who has a higher level of skill than the rest of their team, and can show them how to do things. The majority of people who work as "technical coaches" have no awareness of *coaching* skills or techniques as such.

Mentoring is showing people how to do specific things; for example, demonstrating how to do refactoring, and working side-by-side with someone as they learn it.

Teaching consists of explaining concepts about a topic, to provide foundational knowledge. This may be done individually or in groups, formally or informally.

Coaching, mentoring, and teaching are often interleaved in the day-to-day activities of a team-level technical coach. I think of them as part of the *guiding* function of a coach.

34.1.2 | The Facilitation Competency

Facilitation is a *guiding* skill, too, but is sufficiently distinct from the other guiding skills that it warrants its own section.

Adkins and Spayd define a *facilitator* as a

> Neutral process holder that guides the individual's, team's, or organization's process of discovery, holding to their purpose and definition of success.

I often use facilitation skills when guiding teams in process-related improvements, and sometimes in connection with technical practice improvement as well; particularly in a group setting such as a mob programming session or randori-style dojo.

The facilitator gets a session started and allows participants to run it, only speaking up when necessary to keep things on track. Sometimes participants aren't sure what to do, or the session veers away from the goal, and requires some correction.

A facilitator never "owns" the meeting and is not the "boss." At the team level, the coach may facilitate team activities such as the *team coordination meeting* or *daily stand-up*, gradually relinquishing control until the team learns to handle these events on their own.

Under normal circumstances, the need for a facilitator fades away naturally as teams learn to handle routine activities independently.

However, when there is a conflict, there may be a need for an outsider to facilitate a discussion to help resolve the conflict.

Eventually, client personnel should be able to handle conflict situations without an outside facilitator, as well. This may require more learning curve time than normal activities.

34.1.3 | The Application Competencies

The ability to *apply* the skills in which we're guiding client teams is especially important in the technical coaching role, as technical people have little confidence in coaches who can't "walk the walk."

Technical coaches typically guide teams in both technical practices and process activities. This calls upon both the *technical* and *business* application skills, as well as sufficient domain knowledge to be able to function in context.

The particular applied skills necessary will vary based on the kinds of work each team performs.

For a software development team, technical skills could include things like software testing, test automation, requirements analysis, programming, design, architecture, database, continuous integration, build scripts, and so forth.

For an infrastructure support team, technical skills could include shell scripting, server provisioning, working with virtual machines and containers, system administration skills, operations skills, and so forth.

For a team that supports a third-party product such as a Business Rules Engine, ETL product, Data Warehouse product, application platform (Oracle, PeopleSoft, Siebel, SAP, etc.), technical skills include general knowledge of the category of functionality the product supports and, possibly, some experience working with the particular product (although that may not be necessary).

On the process side, strong understanding of Lean principles will help the coach guide teams in focusing on value, maintaining continuous flow, and removing waste, regardless of any specific methods or frameworks in use.

Practical working knowledge of any specific methods or frameworks the team must use will be helpful for guiding them on the process side.

In the case when the technical coach is engaged as part of a larger program that's led by a prime contractor, and the prime contractor is introducing a specific model or framework for the client, then the technical coach has a professional obligation to support that model or framework and to provide advice consistent with that of the prime contractor. If for philosophical or ethical reasons the technical coach is opposed, then they should withdraw from the engagement.

34.1.4 | The Transformation Competencies

Transformation skills pertain to specific techniques for driving *change* in the organization, on a team, or by an individual. They also involve coordination to ensure consistent messaging when more than one consultant or coach is engaged.

Many people working in the role of technical coach can demonstrate techniques such as test-driven development, design by contract, etc. That is *mentoring*. Mentoring does not automatically translate into "sticky" change.

The coach may explain the value proposition, underlying philosophy, mechanics, and tooling around a particular technical practice; say, trunk-based development, continuous integration, or deployment automation. That is *teaching*. Teaching does not automatically translate into "sticky" change.

The coach may encourage and support individual team members'

progress in learning to apply various technical skills. That is *coaching*. Coaching does not automatically translate into "sticky" change.

Transforming or changing people's habits is a different matter from guiding them in learning new technical practices. It requires a range of "soft" skills and a certain amount of field experience, including making many mistakes, to cultivate transformation skills.

"Coaching Approach," touches on some of these skills. For the moment, it's sufficient to mention that this is a *bona fide* skill area for technical coaches.

Another transformation competency is the ability to convey a consistent message to coachees. In large-scale transformation programs, team-level technical coaches often work as part of a consulting team that is engaged at multiple levels in the organization's management hierarchy.

In that case, the technical coach must coordinate with consultants and coaches who interact with various levels in the organization to ensure a consistent message and direction are provided *vertically*. They also must coordinate with other team-level coaches to ensure consistent messaging *horizontally* .

Many transformation programs have been compromised when different client personnel received, or *thought* they received, conflicting messages from different consultants or coaches.

Sometimes just using a different word for the same concept can lead to misunderstanding and friction. Remember that everyone is under stress due to the changes happening all around them. It doesn't take much to set off an emotional response.

34.2 | Self-Awareness and Empathy

An important aspect of coaching skill is *self-awareness*, or being conscious of how you are perceived and how your guidance is

received by coachees. Mindful use of communication models can be helpful here, as can the "internalities."

Practice in growing your EQ, or Emotional Intelligence Quotient, is also helpful for cultivating the ability to notice how you are perceived as well as detecting how others are responding on an emotional level.

Part of the job is to ensure psychological safety for coachees, so they will feel confident in trying new things and making mistakes. People must feel safe before they will adopt unfamiliar practices.

Some consultancies and some client companies place a high value on standard personality tests. They may conclude that individuals are "hard wired" to behave in certain ways, and cannot learn to behave differently.

In my view, this is incorrect. Everyone can learn self-awareness and empathy. People's underlying tendencies might make that sort of learning easier or harder for different individuals, but that is not the same as assuming people literally can't learn or improve. You can.

Another factor that seems related to this topic is language. In different organizations, different words may be "trigger" words that cause people to shut down or to oppose the advice we offer. Many coaches are fond of particular words, and spend time (often, weeks or longer) trying to make client personnel adopt their understanding of a particular word.

In my experience, it's more effective to adapt our language to whatever "works" in the client organization's culture. There's more than one way to express an idea, and at the end of the day it doesn't really matter what labels people apply to various concepts, provided they achieve the business goals of the transformation.

34.3 | Course of Least Resistance

My observation has been that people tend to do whatever is *easiest*. They don't necessarily do what is *best*.

34.3.1 | Case 1

A story comes to mind about a company where the software development manager and most of the people on development teams said they were very interested in adopting pair programming as a standard practice.

Management set up pair programming stations at the end of each row of cubicles in the software development area, where team members could go and pair up whenever they wished. No one ever used them. Yet, in every team retrospective, every team stated they really, really wanted to start pair programming.

What was the problem? It was easy enough to pair. Just find a partner and to sit at the end of the cubicle row.

The thing is, "easy" isn't sufficient. If you want people to do something different, it has to be *the single easiest thing they could possibly do*. It has to be the thing they will *do by default* because doing anything else would require effort.

34.3.2 | Case 2

In another company where a team kept saying they wanted to try pair programming, the eight developers would come in each day and sit around the big table in the middle of the team space and work solo at one of the eight workstations.

One day after hours, their ScrumMaster and I remained behind and removed four of the workstations.

The next morning, as soon as team member #5 showed up, they started pair programming. With four workstations and eight people, it would have been more difficult to *avoid* pairing than just to do it.

34.4 | Make Things Visible

Many coaches spend a lot of words trying to explain concepts in great depth. Sometimes this is effective and sometimes it leads to confusion, boredom, or disengagement.

I've found that it's often sufficient to make a key thing visible to the team. Once they see it, they tend to take effective action on their own initiative, even if they haven't received any sort of formal introduction to the underlying concepts.

The *coaching skill* in this case is the ability to recognize *which things* to make visible, in order to shine a light on the problem you want the team to address.

34.4.1 | Case 3

On an engagement at a large client, I wanted to capture Cycle Time and Process Cycle Efficiency metrics for four teams that I was coaching. I wanted to do it in a way that would not be too intrusive on their normal work flow.

I set up a Lego plate for each team, and asked them to place bricks in a line corresponding to each User Story. The hours of the day were represented across the top of the plate.

I asked them to place a green 1x2 brick under each hour when they spent *any time at all* moving that User Story forward, and a red brick otherwise. When they completed a User Story, they placed a white 1x1 brick to show that they hadn't simply forgotten to place a brick for that hour.

I didn't worry about getting down to minutes for the PCE data. A crude level of detail was sufficient for the purpose. So, if a team paid attention to a User Story for five minutes out of an hour, that hour got a green brick. The PCE numbers looked a little high, but that was not a problem in context.

At the end of each day, I copied the data from the Lego plate into a spreadsheet and cleared the plate. Here's how the plates and spreadsheet looked for a team called "Cleaners" for the first couple of days:

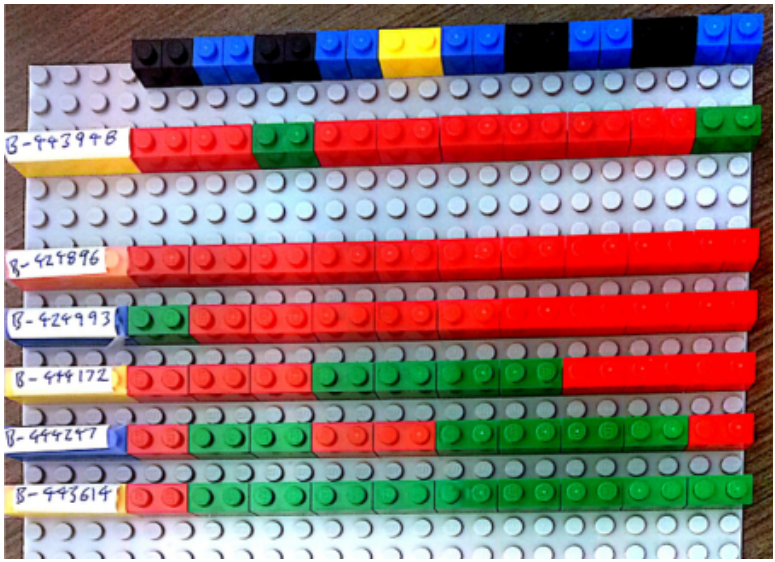

Figure 16.1: Raw data as Lego bricks, day 1

	A	B	C	D	E	F	G
1	Updated: 6/18/14						
2	Story	Team	External Dependency?	Value add hours	Total hours	PCE	Done
3	B-443948	Cleaners		1	8	12.50%	
4	B-424896	Cleaners		0	8	0.00%	
5	B-242993	Cleaners		0	8	0.00%	
6	B-444172	Cleaners		4	8	50.00%	
7	B-444247	Cleaners		2	8	25.00%	
8	B-443614	Cleaners		9	10	90.00%	
9							

Figure 16.2: Collected CT and PCE data, day 1

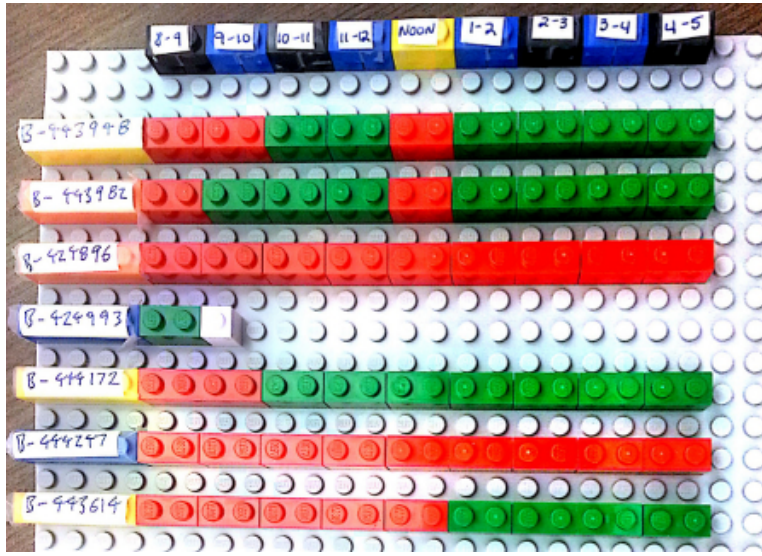

Figure 16.3: Raw data as Lego bricks, day 2

	A	B	C	D	E	F	G
1	Updated: 6/19/14						
2	Story	Team	External Dependency?	Value add hours	Total hours	PCE	Done
3	B-443948	Cleaners		7	17	41.18%	
4	B-424896	Cleaners		0	16	0.00%	
5	B-242993	Cleaners		1	9	11.11%	Yes
6	B-444172	Cleaners		11	17	64.71%	
7	B-444247	Cleaners		2	16	12.50%	
8	B-443614	Cleaners		13	19	68.42%	
9	B-443982	Cleaners		7	9	77.78%	

Figure 16.4: Collected CT and PCE data, day 2

Don't be alarmed that the team appeared to be working 10 solid hours a day. Different team members arrived and left early or late, and some liked to work through lunch or take lunch at different times of day. So, there was activity throughout the day.

In two of the four teams, I noticed a response to the Lego plates that I had not anticipated. Here are excerpts from my blog post describing the observed behaviors (Nicolette).

> I was surprised to see the natural response when a team member reaches for a red brick. Others on the team immediately ask what the impediment is and how they can help. These were already practicing "agile" teams, so they are already stable teams working in team spaces, and collaboration was not a new concept at the start of the engagement. However, the immediacy of their reaction to seeing a red brick is a radical improvement in the speed with which teams respond to emerging issues.
>
> A second natural reaction to the boards is that when a team notices a large swath of red on their board, they start exploring the reasons why. Without any formal training in Lean concepts, they quickly conclude that they have too many User Stories in play. They limit their WIP, even without knowing that term. Before

> the impact of high WIP was visible, team members often said they did not understand the "big deal" about pulling just one or two User Stories at a time.
>
> Nearly all teams in the organization have the classic "hockey stick" burndown chart, and a CFD that looks like a single block of color representing "in progress." The teams that have started to notice the impact of high WIP thanks to their Lego boards are already starting to show burndowns that look like actual burndowns. They are pulling User Stories through to completion rather than starting everything on the first day of the sprint. Within days, it became a sort of game to see if the team could eliminate all the red bricks from their board.

Just making visible the difference between value-add time and non-value-add time triggered a natural response in two of the four teams, even with no further explanation or training, or any verbal encouragement to do anything differently.

I trust you noticed that the effect did not occur naturally in the *other* two teams. It goes to show that no technique or approach will be universally effective. We need to build up a toolkit of coaching methods so that we have more than one way to try and drive change.

34.4.2 | Case 4

On another engagement, a team I was coaching expressed the desire to raise their unit test coverage numbers. Coverage is a negative metric; that is, if it seems too low, it's a negative indicator, but there's no "magic number" that tells us things are going well. In fact, when we see 100% coverage, it almost always means the team is gaming the metrics. So, coverage is a metric to be used with care.

But this team had unit test line coverage of 5%. They understood that was not good, and they wanted to get into a more reasonable range, like 70% or 80%, so they could feel as if they were taking the codebase in a good direction. They weren't fixated on a number, as if it were a target. That was a healthy way to think of it, in my view.

Yet, day after day they didn't extend their unit test suite. They were in the habit of working in a certain way, and habits can be hard to change even when we want to change them.

I started to post the unit test coverage percentage on the wall next to the door, just *outside* the team room. I plotted dots to create a line chart that showed the day-by-day change in line coverage, based on the final build of the day.

Without any further discussion of the matter, the team began to add unit test cases to their suite incrementally in the course of completing User Stories. Just having the statistics visible was sufficient incentive for them to cultivate new working habits.

34.5 | Manipulation

"Manipulation" is quite rightly seen as a negative thing, as masters of manipulation tend to be found mostly on the Dark Side. But sometimes coaches have to use some of the same techniques to get things moving in a transformation program.

On the bright side, many manipulation techniques are really the same as general social interaction skills. People think of them as "manipulation" only when they're used for nefarious purposes.

34.5.1 | Connect Organizational and Personal Goals

People involved in the transformation program may feel uneasy about changing anything about their work. They aren't certain the transformation will really happen. If they've worked in a large company for very long, then they've already experienced and witnessed attempted transformations that were abandoned or reversed.

One way to encourage people to try new things is to help them see how it could benefit them personally. Arguing logically from the point of view of what benefits the company may not resonate with everyone on a personal level.

The coach has to cultivate a positive working relationship with each coachee. In a private conversation, or a series of them, find out about the person's goals, fears, and any other concerns they may have that are blocking them from moving forward with new practices.

Then think of ways the proposed organizational changes will alleviate their worries and/or advance their personal agenda. I've often found this an effective way to turn people around from hardline refuseniks into valuable allies in the transformation effort.

This really isn't an "evil" use of manipulation. It helps the other party see what's in it for them; it helps them achieve their own personal goals.

34.5.2 | Build Relationships

Some of the world's most effective technical coaches make a point of having lunch with different client team members, as well as various individuals external to the team who can affect the team's work, each day. During these times, they explicitly don't talk about work.

The connection with learning new skills and being willing to try new things under the stressful conditions of a transformation program may not be obvious, and yet it's a powerful way to encourage change. When people feel a personal connection, they're more willing to accept suggestions and take (small) risks.

We're still well clear of the Dark Side. We aren't *pretending* to get to know the people; we're *actually* getting to know them. And we don't know whether we're making temporary working relationships or long-term friendships. Time will tell.

34.5.3 | Mirroring

Mirroring is a way to make the other party feel at ease during any sort of discussion. It's often recommended to people for job interviews and business negotiations.

If the other person crosses their legs, then so do you. If they lean forward when they speak, then so do you. If they position their hands or arms in a certain way, then so do you.

The effect is thought to be that subconsciously the other party identifies with you because they see themselves reflected in you, to an extent.

The manipulative aspect of mirroring comes when you gradually shift who is mirroring whom. You start by mirroring the other person, and once they're subconsciously doing it, you start guiding them into mirroring you instead.

You can do this verbally, as well. When the coachee expresses reasons or fears about changing some aspect of their work, you agree with them and reflect their own language back to them. Gradually build their connection with you by agreeing with them and even extending their argument with additional points.

Then, using the same words, start to guide the conversation toward positive aspects of the proposed change, and negative aspects of the

status quo.

You're definitely trying to manipulate them at this stage. You may choose to *omit* negative aspects of the proposed change and positive aspects of the *status quo* as a way to heighten the perceived benefit of the proposed change.

As the proposed change is deemed to be "good" in the context of the transformation program, this form of manipulation used in this context doesn't qualify as "evil."

34.5.4 | Playing Dumb

This is a way of guiding people's thinking by asking leading questions. It's useful when you want the other party to reason through a situation and arrive at a good answer. It can be useful when the coachee is unwilling to try a given technical practice.

On one engagement I was working with a team to show them various practices from Extreme Programming. The team lead had a strongly negative opinion about test-driven development (TDD). The rest of the team respected him highly and followed his lead.

One day, I approached him at his desk and asked if I could sit with him. I didn't call it "pair programming," as he had a strongly negative opinion about *that*, as well. I told him I wanted to watch him work as a way to get familiar with the code base. That made sense to him, and he agreed.

We picked up the next ticket to work on, and I pretended I didn't understand the requirement. He explained it to me patiently a couple of times, but I just didn't get it.

Acting as if I had just thought of it out of the blue, I asked whether he thought it would be a good idea to write some client code that would interact with the planned code, to get an idea of what it would look like to call that code and get return values from it. He thought that was a reasonable thing to try.

We wrote what was, in effect, a kind of "unit test," although the scope was larger than we would normally prefer, and we didn't use any sort of testing framework. Even so, when he saw how easily and clearly the exercise defined the requirements for calling the code, dealing with return values, and handling possible exceptions, he really liked it.

Almost immediately, he became a strong proponent of TDD and encouraged his team mates to try it. Best of all, it was his idea. He gathered the team members together and explained the benefits of TDD to them, without asking me first. I didn't have to convince each team member individually. It was a good day for all.

34.5.5 | Nouns Over Verbs

We're often advised to focus on *behaviors* rather than criticizing *people*. But to encourage people to move outside their comfort zone, it's sometimes more effective to do the opposite.

It can be done in a subtle way. It seems that people tend to think about *behaviors* when they hear verbs, and to think about their *self-identities* when they hear nouns.

When describing people who don't do the things you're recommending as a coach, structure your statements around nouns that have mildly negative connotations. When describing people who already use the techniques or practices you're recommending, use nouns that have mildly positive connotations.

Avoid extreme words, as these will cause people to snap to attention. You want to lull them along, so they don't think too deeply but just allow the subconscious response to happen.

Now we're moving into less benign manipulative territory, but not "evil" just yet.

34.5.6 | Don't Allow Time for Arguments

From this point forward, we're drifting ever closer to the Dark Side. These last three manipulation techniques are to be used advisedly, and only if nothing else has worked. They will probably work no more than once per team, if indeed they work at all.

This is a variation on a standard verbal manipulation technique, adapted for technical coaching situations. There are two aspects to it.

First, you must have established a positive working relationship with the coachees. This will cause them to want to please you, or at least not want to disappoint you.

The second aspect is a variation on the fast-talking used car salesperson technique. You're encouraging the team to try an unfamiliar technical practice, and they're hesitant. They want to debate it.

Instead, you quickly take the keyboard and "just do it," with a slight air of annoyance. This works for small things, like beginning to refactor a long method, for instance.

They see the result of doing what you were recommending before they have time to formulate verbal arguments against trying it. They're also a little unhappy that they've disappointed you.

Afterwards, they will be more willing to try it themselves; in part, because they've seen the benefit, and in part to win back your approval.

Important note: If you haven't established a relationship that causes them to care about your opinion, they'll just watch you type with the same degree of engagement as a house-cat watching a car crash.

34.5.7 | Fatigue As a Motivator

Sometimes, when people are tired and want to get out of the office at the end of the day, they'll be more willing to try new things than

they are in the middle of the day. They don't want to waste time arguing, and they'll do what you suggest just to get you out of their hair so they can leave.

This works with *some* people *some* of the time for *some* new skills. You have to know the people well enough to judge whether it's likely to be effective.

The point is to get a person try the new thing at least *once*, to give themselves a fair chance to experience the benefit of it.

Now you're just a tiny bit evil. You deliberately waited until they were ready to leave work before you sat them down to do one more task.

34.5.8 | Fear As a Motivator

A last resort manipulation technique is to prey on people's fears. Trying the new skill might be slightly less frightening than something else, such as a negative performance review or a tongue-lashing by their manager.

Some fears are benign while others are emotionally significant.

Some people might care enough about code quality to fear introducing defects into production. Some people might feel competitive, and will respond to the suggestion that another team will outperform their own team.

You know your teams, so you know what different individuals might respond to. But don't overuse this technique. It's pretty evil. And the second time around, it's totally obvious, too.

34.6 | Acting

This might seem an unusual skill to cite for technical coaches, but it actually applies in a couple of different ways.

34.6.1 | Improv Techniques Applied to Coaching

Techniques from *improv* are useful both to increase our self-awareness, empathy, and deep listening skills, and to help client personnel break down internal barriers of fear or nerves to reach a point where they're willing to try new things (Fidei). The latter is more appropriate to a workshop setting than *in-situ* team coaching.

If you have seen or used the popular exercise, "Yes, but vs. Yes, and," then you've had a taste of how improv techniques help us with listening and empathy. You can take it much further by practicing improv techniques guided by a skilled facilitator. If you haven't participated in this sort of workshop at a conference or coach camp or other event, I recommend it.

34.6.2 | When You Can't Be Yourself

The second application of acting skills to coaching is to "present" in a way that resonates with coachees and may help influence them to change in desired ways.

The *manipulation* techniques "Mirroring," "Playing Dumb," "Don't Allow Time for Arguments," and "Fear As a Motivator" (and possibly others) require us to pretend. In addition, when an introverted coach must present as an extravert (or *vice versa*) in order to interact effectively with coachees, some basic acting skills can be helpful.

34.7 | Removing Organizational Constraints

In most cases, the project manager, team lead, ScrumMaster, or some other person has primary responsibility to remove orga-

nizational constraints that affect their team's ability to deliver, and to get any necessary resources, system access rights, or other requirements the team needs to get their work done.

During an organizational transformation program, the technical coach may have to take on the responsibility for this, at least initially. Be prepared to go to client management and/or collaborate with other consultants on the consulting team to help teams with these issues.

In the meantime, mentor and coach team members to take on this responsibility going forward.

Elsewhere I shared the story of a client where I was coaching a single team that had five bosses. I stepped outside the formal boundaries of my contract to ask those five people to collaborate to manage the team's backlog, prioritizing requests according to how they supported the company's official list of business goals for the year. That's an example of acting to remove organizational constraints that interfere with a team's ability to deliver.

But I didn't approach them out of the blue. I had established at least minimal working relationships with each of them before then, without asking them for anything. Otherwise, they probably wouldn't have met with me at all.

34.8 | Let the Team Stumble

A coach's role is to help people improve. Sometimes, people will listen with an open mind when you warn them away from problematic practices and steer them toward effective practices. Sometimes, they need to experience the natural consequences of a practice in order to understand its effects.

There are times when the best thing a coach can do to help a team is to let them experience problems, and then help them connect the dots that led to the problem (either with immediate feedback or via

a facilitated retrospective). A key point is not to let them crash and burn entirely. That would have negative ramifications for you, the team, and the organization.

But the coach is not there to support *delivery*; they're there to drive *improvement*. Improvement sometimes involves deeply understanding how the dominoes are likely to fall, depending on which one you push first. Nothing beats direct experience for that.

There is a special consideration here. You or other consultants on your team have to prepare client management for this technique. Management often expects (or *hopes*) improvements can be made with little or no downside impact on current delivery performance. That's not realistic, but many managers will balk at coaching methods that slow teams down, even temporarily. This has to be socialized and agreed vertically in the organization, or it will likely backfire.

34.9 | Be the Rock

At the risk of belaboring the point, I'll repeat that everyone involved in a transformation program is under stress nearly all the time. I've noticed an interesting phenomenon when coaching teams in this context. When everyone around me is getting stressed out, if I remain calm (even if only outwardly), it tends to defuse the situation.

Occasionally, a team member has told me after the fact that they appreciated my calmness. One day, a coachee said they wouldn't have made it through the day had I not been there with the team, even though I said very little that day. In the midst of some serious challenges at a large client, a colleague from our consulting firm said, in front of the others, that it seemed to him that stress didn't affect me.

Well, stress affects me plenty. But I think it's part of the job to

handle it. We have to provide psychological safety for our coachees. We can't do that very well if we, ourselves, are obviously stressed. We have to guide our coachees through challenging days when things aren't going well. We can't do that very well if we, ourselves, are visibly affected by the challenges of the day.

It surprised me the first few times it happened, but clearly if one member of the consulting team can maintain calm, it gives a form of non-verbal emotional support to the other consultants. This can make the difference between a day that erodes client relationships and a day that strengthens them.

For those reasons, I include this as a coaching skill we should all cultivate and practice. You may find the "internalities" to be useful for this purpose.

34.10 | Conflict Resolution

Friction and conflict will be frequent in any transformation program. Part of our role as coaches is to resolve conflicts among client staff members and, if necessary, among our own colleagues on the consulting team.

To do so, we need to bring to bear a range of different skills, including:

- relationship-building;
- facilitation skills;
- specific conflict resolution models;
- various communication models; and
- the "internalities."

In addition, it's within the scope of our role as coaches to teach and mentor the same capability with our coachees, so that they can learn to resolve conflicts themselves.

Details of specific techniques are out of scope for this book, but in summary they include:

- Forcing - imposing one party's will on the other
- Collaborating - seeking a win-win solution
- Compromising - finding a less-than-perfect solution that is acceptable to both parties
- Withdrawing or Avoiding - agreeing to set aside the conflict temporarily or permanently, and living with any leftover friction
- Smoothing - accommodating the other party's wishes as a way to avoid further conflict (possibly temporary)

Each of these, including the ones that *sound* completely negative, have advantages and disadvantages and might be selected based on context.

See (separately) Bolton, Human Metrics, and Weeks in *References* for more information.

You might think a technical coach can comfortably remain within the warm embrace of technology, and not have to deal with conflict. But in fact it's a problem that comes up quite often.

34.10.1 | Case 5

There was a case many years ago when a company needed to connect two disparate systems - a DEC VAX and an IBM mainframe. (I *told* you it was many years ago.) Applications, user documentation, and everything else was ready to roll, but the two systems were still not talking after months of delay.

Long story short, the engineer on the DEC side and the engineer on the IBM side sat in cubicles about 3 meters apart. They never spoke to one another. It turned out we needed to make simple configuration settings on both systems, and then they could talk.

I had to convince the two of them that it was in their personal interest to get it done. We ended up with a *collaborating* or *win-win* solution.

The two engineers were not hostile to one another. Strangely enough, they *didn't know about each other*, and neither of them knew enough about the other system to know how to connect them.

34.10.2 | Case 6

In a similar story at a different company, a Java application needed to talk to a CICS application. Again, there had been months of delay. The two engineers sat in the cubicles right next to each other, and never spoke. Each insisted that the other system had to be configured to the liking of their own system.

This time, by the way, the two engineers *were* hostile to one another.

I managed to convince the Java guy to "be the bigger man" and configure his side to conform with the IBM engineer's preferences. It was a *smoothing* solution, in which one side concedes to the other. I wouldn't have guessed at that outcome based on the initial conversations. You never know how these things will turn out.

34.10.3 | Case 7

More recently, I was visiting colleagues in another company and one of them invited me to shadow him as he worked with a client. I was not formally engaged there; I was a guest and observer only.

Toward the end of a working session with a certain team, there was a blow-up between team members. The coach was taken by surprise; he hadn't been aware of tension between them before. He suggested an "intervention." We all went to another room to work things out.

Unfortunately, the people involved in the conflict pointed to me and said, "We want to know what *that guy* thinks. He hasn't been here before, so he's objective."

Well, maybe, for some definition of "objective." It was an uncomfortable position for me. I had to use *facilitation* skills and *active listening* to try and draw out people's real thoughts and feelings about the situation.

Their coach had described one of the individuals as "sad," and I could see her shut down. When an opportunity arose, I remarked that I didn't see her as "sad," but rather as "angry." She immediately lit up and vented her frustrations for several minutes. Using the correct word, even if it wasn't a "soft" word, broke the impasse. Difficult but necessary conversations followed.

Ultimately the team agreed to a *compromising* solution they felt would be good enough to get them through the next month of working together, and then they would see how things felt at that point.

A lesson from this is that we mustn't let friction continue; we need to confront it as soon as we notice it. I say so because in this case the team members had "checked out" and buried themselves in their individual tasks for months. They all had made unverified assumptions about the others, and tension had steadily mounted until it reached a breaking point.

The team had suffered a significant loss of trust and was not functioning cohesively. As coaches we need to be aware of these things and prevent them from reaching that stage.

Under the stressful conditions of an organizational transformation program, conflict is normal and frequent. It isn't something any coach can avoid; not even a technical coach. We must cultivate the skills to handle it.

34.11 | Principles-Based Adaptation

While most organizations and teams experience broadly-similar issues, it's also true that every team is unique. Each team has its own work to do, likely different from the work of other teams, and its own organizational constraints, similar to other teams insofar as management style and organizational culture are concerned, but otherwise unique, usually based on what cross-team dependencies or resource limitations they have.

With that in mind, one coaching skill is the ability to apply *principles* to a wide range of different situations, to help determine effective ways to improve performance. As this is written (2019), the industry is flooded with freshly-minted ScrumMasters who have little to no practical experience in IT, and whom client leaders think are qualified as team-level coaches because they hold a certification.

Many of these coaches know how to apply exactly one model or framework in a rote way. When they are trying to help a team whose unique circumstances don't quite fit that model, they're at a loss.

34.11.1 | Case 8

I was engaged as a team-level process/technical coach as part of a IT-wide transformation program. It was one of those "agile adoption" programs, this time based on the client's internally-defined Agile scaling framework. It was as good as any of the branded frameworks that are sold commercially. Maybe better. At least it was crafted around their business needs, rather than a "canned" model.

One of the other team-level coaches, who was a process coach and not a technical coach, came to me and asked if I would visit her team to try and see why they were rejecting all her advice.

We all met in a conference room. I told the team I wasn't familiar with their work or their situation, and asked them to fill me in. They explained what they did and how their team interacted with other teams in the organization.

It became clear they were not a "conventional" software development team. They did various kinds of work, some of which was of a "technical" nature, but they didn't fit the canonical "development team" pattern. Their coach was advising them to do things like "daily stand-up" and "two-week iterations" and "slice stories to a small size." The usual.

They said "agile" wouldn't work for them. I said, okay, tell me what it is about "agile" that won't work for you. They took turns, and I made notes. It was obvious they needed to vent, so I didn't interrupt them. If anything, I asked clarifying questions to draw out even more of their frustrations.

Then I affinity-grouped their comments, and it boiled down to six specific things about "agile" that didn't fit their situation. I don't recall what the six things were, but that's not relevant to the coaching pattern.

I told them how I had grouped their comments, and they agreed with the categorization. Then I went through the items one by one, and told them I agreed in each case that canonical "agile" didn't support any of them. Their coach grew increasingly nervous during this part of the process. The team had seemed tense and adversarial up to that point, and now they were relaxed and open.

Then I suggested we think about the guiding principles of "agile" and see if we could get any value from them as applied to their own work. We were able to think of at least one thing in each category they believed would help them. They came away feeling cautiously optimistic about "agile" and how to apply it in context.

34.11.2 | Case 9

I've had a couple of engagements where the Siebel platform was involved in a team's work. It's a "legacy" product designed to host multiple applications. There are other products like it. That was a good architecture in the day when centralized systems were the way to go, and when there was less need for systems to be ready to interact with new clients that were unknown at design time. It was also an era when solutions were delivered in big-bang fashion with long lead times (compared with today's expectations).

Owing to those design considerations, a Siebel instance is fully defined in a single file, known as an SRF, or Siebel Repository File. It's a binary file that contains the "compressed" or "compiled" definition of all the applications hosted by the particular instance. A perfectly good solution in its day.

Today, we generally expect each application to be independently deployable on a much shorter cadence than was normal in those days. Teams that support this type of platform have challenges in adopting lightweight methods.

In one situation, the organization had six applications running on a single Siebel instance. They had three Siebel environments - development, test, and production. They wanted to "go agile," and that meant each of the six teams had to be able to define User Stories and deploy software changes independently of the other five. In fact, there were no *bona fide* dependencies between the applications; the only thing they had in common was that they shared a single Siebel instance.

None of the six teams could deliver any faster than the time required to push changes through all three environments for all six teams. In that case, it was around 12 weeks. They were stressed out because they were being told they had to deliver a production-ready solution increment in 2 weeks.

For a little context, this is just the sort of situation Sean Dunn

reports in his article, "Eating Your Own Dogfood: From Enterprise Agile Coach to Team Developer," in part 6 of the article - "Sometimes stories cannot be split." (Dunn)

The area manager asked for a working session. We stepped through all the issues and challenges, and I made a recommendation: Set up a separate Siebel instance for each application. That meant buying 18 Siebel licenses - development, test, and production times six.

The manager balked. He worried about the cost of the licenses. I walked him through a quick-and-dirty analysis of the costs of the current setup. An hour later, he understood that the way they were currently doing things was costing far more than the price of the additional licenses, due to delayed delivery and the impact of bug-fixes for any one application on the release schedules of the other five applications.

34.11.3 | Case 10

On another large-scale transformation program (also in the nature of an "agile" implementation), I was asked to work with a particular team the other coaches felt was "impossible." When I asked why they said so, they looked at one another with knowing smiles and replied, "You'll see."

I joined that team during one of their daily stand-up meetings. They gathered in a room that was just a bit small for them. The usual "agile" things were taped up on the walls, placed there by their former coach. The team basically ignored all that stuff. They were extremely cynical.

I asked them to explain what work they did and how they interacted with the rest of the organization. It turned out they were on a long-term project to upgrade all the company's communications equipment at 160 locations in several countries.

The work involved physically inspecting each site to inventory what was actually installed there, rather than what the official

records said was *supposed* to be there. Those were two different things.

Then they would return to each site to install and configure the new equipment. They could monitor the entire network from the home office, but the information was not very useful, as the external contractors who installed the gear and wiring didn't follow specifications, so things like MAC addresses weren't what they were supposed to be. Contractors would also take short-cuts like using indoor CAT-5 for outdoor applications, and pinching or twisting the cables such that they didn't perform as CAT-5 anymore.

Several details were different from conventional "agile" software development teams. For one, the pace of change in their project was much slower than on a software development project. They were frustrated with the daily stand-up, because nothing changed day to day. I clarified the *purpose* of the stand-up - briefly, to ensure everyone knew what was going on, and could ask for and offer help as needed - and asked them how frequently they felt they should touch base as a team to achieve that purpose, in their context.

They settled on twice a week. I actually thought that was more than they needed, but if they were comfortable with it, that was fine. I reminded them they could adjust that schedule at any time, if it wasn't serving their needs. That pleased them, as they had been led to believe nothing in "agile" was flexible.

They were immediately more relaxed and open, the moment an option was offered to them. Their first coach had imposed canonical "agile" on them without collaborating with them. That was one cause of their frustration. From that point on, it became much easier to work with that team, because they were solving their own problems. I only offered guidance and suggestions as needed. They were genuine engineers as opposed to "software engineers," so I was the least intelligent person in the room by a long shot.

Some of my suggestions weren't "agile" things, particularly, but I tried to tie them in with "agile" principles if possible, as that

was the overarching purpose of the transformation program. For example, I used the idea of "test-driving" to suggest they clearly specify requirements for the external contractors, and pay them only if they met the requirements; not just because they showed up and drove a truck around a site.

The MAC addresses had to be per spec and the CAT-5 cables had to respond as CAT-5 when tested. Otherwise, no check-ee. They liked that one a lot. It saved them considerable post-installation time and effort.

Coaches need to be able to adapt whatever model they're teaching to different contexts based on fundamentals, as every team will have its own context, and sometimes that context will be very different from software development.

34.12 | Having Multiple Ways to Explain Things

In the first installment of Cutter Consortium's *Cutting-Edge Agile* series, Agile Manifesto author Alistair Cockburn notes:

> We are now in the "post-Agile" age.
>
> Post-something means that we are in an in-between period, where one something has been incorporated into our culture and the next something has not yet fully formed to the point of naming.

As of this writing (2019), Cockburn is quite right: There's no name for what we think we ought to be doing and showing other people how to do. What most coaches do is continue to use the old "agile" words and phrases.

By now, pretty much everyone has experienced some flavor of "agile," or something labeled as such. Many of those people did not like the taste.

When freshly-minted ScrumMasters and the like come bounding into the room full of energy and enthusiasm, they're often taken aback by the lukewarm (or cooler) reception they get.

The thing is, the various "agile" and "lean" buzzwords and phrases have become *triggers* for people out in the world. When we first visit a client organization, we don't know which words are triggers there. They might not be the same words as the ones that are triggers at the company across the street.

For that reason, I consider it a basic coaching skill to be able to describe the benefit and the mechanics of any given method or practice in more than one way. If we can describe a practice without recourse to buzzwords, and still convey the essence of that practice, then we'll avoid tripping any verbal landmines.

Why worry about verbal landmines? Remember the high stress level. Remember that people are on the alert for reasons to avoid changing their habits. When we trip a verbal landmine, what follows may be days or weeks of explaining and re-explaining to try and recover to the position where we started.

Try explaining the value and/or mechanics of these things, and any others you can think of, without using any of the usual buzzwords:

- sprint
- pairing or pair programming
- mobbing or mob programming
- test-driven development
- test automation
- hold accountable
- limiting work in process
- batch size

- buffer management
- feedback
- retrospective
- emergent design
- technical debt
- Scrum
- agile

It may be challenging at first, but once you get going you'll find you can describe these things using words that won't trigger a negative reaction. Different clients have different triggers, so you'll need two or three different ways to express these ideas without buzzwords. I suggest practicing your explanations as if rehearsing for a play.

34.13 | Awareness of Context

There are two general scenarios for clients to engage team-level technical coaches:

- technical coach engaged by a team manager (or area manager responsible for a handful of teams), possibly to introduce good practices with the aim of general improvement in skills, product quality, and/or delivery effectiveness; or
- technical coach engaged as part of an organizational transformation program; one member of a consulting team that interacts with the organization at multiple levels to provide cohesive and comprehensive guidance toward one or more business goals.

There is a popular notion among team-level coaches that one must be "invited" by a team to offer coaching advice to them. The coach must earn the trust of the team before he/she will be welcome to offer advice. This is appropriate for the first case.

When we are engaged as part of an organizational transformation program, we may not be able to wait for each team to come to trust us enough to invite us personally to coach them. Our "invitation" was already sent, from the CEO or CIO or other client leader who called for the transformation.

In that case, our job is to *make change happen*, as opposed to waiting until team members are good and ready to consider changing. As it's ineffective to *tell* people what to do, that may call for a wide range of different ways to influence people.

It's still good if teams trust our advice - and we're dead in the water if they *distrust* us - but it isn't necessary for them to "invite" us in quite the same way as when we're engaged to help a single team. Context matters.

34.14 | Knowing When To Quit

No matter how beneficial everyone believes the recommended technical practices may be, there will always be a few individuals who just don't want to work that way.

As coaches, I believe our responsibility to these individuals is to ensure they understand the trade-offs they may be making so that they can make an informed professional judgment about whether to adopt the recommended practice.

We need to cultivate in ourselves the "internalities" that I call *Eye of the Hurricane* and *Non-Attachment*. The former gives us the inner peace to accept the fact other people are in charge of their own lives, and need not accept our advice. The latter reminds us that we don't "own" our clients' outcomes; we can shine a light on the path, but we can't walk for them.

34.14.1 | Case 11

In my experience, one of the most fundamental software development practices is *test-driven development* (TDD). It always surprises me when good developers want to argue against it.

At one client, the senior-most developer on a team was dead-set against TDD. No amount of explaining or demonstrating would move him.

One day we were planning a learning event for developers in which we would run a code dojo around TDD. He volunteered to facilitate the event. With no training or preparation from any of the coaches, he ran the event like an expert. He understood as much about TDD as any of the coaches did, and he could explain it well and demonstrate it effectively. He helped 24 colleagues learn and practice the technique.

There was no doubt that he knew TDD well. Yet, his personal choice was not to develop code that way.

As a coach, I thought that was fine. He made an informed professional judgment. It was different from my own professional judgment, but it was a completely valid choice on his part.

We have to accept the fact that not everyone will choose to do things the way we recommend, and that's okay.

What we want to avoid is to let people dismiss unfamiliar techniques just because they don't understand them or because they've heard negative stories. We need to help people understand the choices they're making. As long as that's true, there's no "rule" that says they have to make the same choices we would make.

36 | Technical Coaching Approach

> In order to change an existing paradigm you do not struggle to try and change the problematic model. You create a new model and make the old one obsolete.
> — R. Buckminster Fuller

It seems to me the majority of transformation models focus on training and coaching executive leadership, management, and team leads in a given framework or method, and training delivery teams on their role in the method, and then "a miracle occurs" with respect to technical infrastructure, technical practices, automation, and so forth.

36.1 | Problems With the *Status Quo*

There are several reasons why I think technical coaching has to take a more central role in IT transformation programs, and why the approach to technical coaching has to be well thought-out. These are listed more-or-less in order of importance, as I see it.

36.1.1 | Misunderstanding the Importance of the Role

A failure mode in transformation programs is that both client management and consultancies who hire technical coaches under-value the role, underestimate the difficulty of becoming qualified to perform the role, overestimate the availability of qualified coaches

in the job market, and assume the work the coaches perform is in the nature of "staff augmentation" or a low-end "commodity" service.

Almost invariably, transformation programs are started with too few technical coaches, underqualified technical coaches, and insufficient time planned to produce "sticky" improvement in client teams.

36.1.2 | The Fulcrum Is In the Technical Area

According to the general case analysis in "Leverage: Where's the Fulcrum?" the focus of the transformation resides within the IT sphere rather than the business sphere: *stable production operations*.

Without appropriate guidance in the technical area, all the great ideas at the executive and management levels can never be executed.

36.1.3 | The Key Need Lies in an Emerging Area

Approaches, team composition, tooling, and practices for production operations and support are currently undergoing significant and rapid evolution.

Existing staff skills don't align with the emerging practices, and focused coaching is needed in that area to a greater extent than the usual areas of programming, testing, and delivery pipeline automation.

This exacerbates the shortage of qualified technical coaches, as most of them specialize in either application programming or testing rather than in production operations.

36.1.4 | Global Shortage of Technical Coaches

There is a severe shortage of truly *qualified* technical coaches worldwide. As an industry, we can't afford to pile multiple coaches on each delivery team on a full-time basis because there simply are not enough of them to meet demand.

We need to find practical ways to enable fewer technical coaches to support more teams while maintaining effectiveness.

36.1.5 | Technical Learnings Don't "Stick"

As far as I have been able to learn from experience reports, comments from staff at companies that have undergone transformations, colleagues in the technical coaching arena, blogs, social media posts, and my own direct field experience, the technical practices that were taught and coached have not "stuck" for very long after the coaches left, except in a vanishingly small number of cases.

The fact this continues to be an industry-wide problem lends credence to the observation that we haven't solved it. Logically, then, something about conventional approaches to technical coaching hasn't been working. We need to understand how to change our own practices to achieve better results.

36.1.6 | Coaches Lack Agency (or Believe So)

Another failure mode for technical coaching is that many coaches lack skills in communicating and collaborating with non-technical stakeholders to gain assistance removing organizational constraints limiting the impact of their technical coaching.

Many act as if they were helpless leaves in the wind, with no agency to function as true consultants. When client personnel say

a recommended action would be difficult or impossible, the coach stops working for the change and tries to work around the problem in some way rather than solving it.

36.1.7 | Coaches "Go Native"

One failure mode for technical coaching is that when coaches remain with the same team for an extended time, they start to feel like members of the team rather than coaches.

They "take ownership" of the team's work instead of their true work of guiding the team forward.

36.1.8 | Teams Grow Bored

Another failure mode for technical coaching is that when coaches remain with the same team for an extended time, the team either grows tired of them or starts to take them for granted, and no longer listens to their advice.

36.1.9 | Coaches Become Part of the Background

When coaches are always available (assigned to the same team full-time), team members feel no sense of urgency to try the things the coaches suggest, as they "know" the coach will still be around tomorrow and the next day and the next.

36.1.10 | Teams Become Dependent on Coaches

Another failure mode is that when coaches remain with a team full-time, the team has no opportunity to think about, stumble through

and learn from, and finally internalize any of the new practices the coach shows them. When the coaching engagement ends, the teams are still not self-sufficient.

36.1.11 | Coaches As *De Facto* Team Leads

Most technical coaches enjoy the technical work themselves, and they like to solve problems. There's a tendency for them to become too actively involved in team tasks. Sometimes this results in teams looking to the coach as a kind of technical lead. They don't learn to make choices or take chances on their own initiative, but turn to the coach for leadership.

When visiting teams to observe or to coach the coach, I've often noticed non-verbal exchanges between team members and their coach in the nature of "asking permission." It blocks teams from becoming self-sufficient.

36.2 | Addressing the Problems

Let's consider some changes we might make in the way we approach technical coaching to address these problems.

36.2.1 | Balance Business and Technical Improvements

I hear many circular debates among consultants along the following lines:

- It's important to determine the *right things to build* to deliver business value to customers. There's no value in being able to build and deliver software effectively if we don't know what to build. You end up with a lot of "stuff" nobody wants.

- It's important to establish the capability to build and deliver software effectively. There's no value in understanding the right things to build if there's no way to execute. You end up with a list of pretty things you'd like to have, but can't obtain.

There may be a couple of reasons for this type of debate.

First, people are human. They tend to focus on the subjects they understand best, and to de-emphasize other subjects. This may be particularly true for people who see themselves (or portray themselves) as "experts." They don't want to spend a lot of time talking about things they don't understand well, especially in front of clients or prospects.

Second, there's been a sort of "pendulum" effect in our line of work over the years. People focus on technical excellence for a few years, they begin to neglect customer value, and there's a backlash. The backlash leads to the opposite situation: People focus on business excellence for a few years, they begin to neglect technical excellence, and there's backlash in the other direction.

I suggest this type of debate is not useful. Both arguments are right, and both are wrong. They're right in that it's important to determine the right things to build, *and* to be able to build them. They're wrong in that there is no case when the "other" capability is secondary. The two considerations are of *equal* importance.

As consultants, we need to structure our proposals for organizational transformation in a way that places equal emphasis on the business and technical areas. This may require some education for consultants as well as for prospects.

36.2.2 | Focus On the Key Dependency: Leadership Challenges

One concept in my approach that differs from conventional wisdom is that I distinguish between the *business driver* for the transfor-

mation and the *focus for improvement* during the transformation program.

The key dependency that enables *every* market-facing competitive capability is *stable production operations*. The business driver for the transformation may be one or more of those competitive capabilities. The primary focus for improvement efforts has to be the key dependency.

Given the typical approach to transformations by most consultancies today, the key dependency in our model would receive short shrift. It falls into the bubble on the flowchart labeled, "And then a miracle occurs."

Instead, the key dependency has to be front and center. It's the "beacon on the hill" that lights the way to achieving the business goals of the transformation.

36.2.3 | Focus on the Key Dependency: Coaching Challenges

If we can get consultants and client leaders to recognize the centrality of *stable production operations* to the success of the transformation program, that's a good start. Then we run into a couple of interesting challenges.

First, the global shortage of qualified technical coaches makes it hard to staff up a transformation program appropriately to address this need. I needn't belabor the details again.

Second, the key dependency happens to fall in an area that's currently undergoing rapid change. Most people working in operations aren't well-versed in emerging solutions. They are still operating in a reactive mode, responding to support tickets as they are opened. Most technical coaches, scarce as they are, know little to nothing about proactive approaches to supporting applications in production.

Most technical coaches have strong expertise in application software development; their skills are primarily in areas like programming, testing, and continuous integration. They aren't prepared to guide teams in applying new methods of managing production operations.

The general solution will require a shift in emphasis throughout the technical coaching industry. Coaches need to learn about new methods of monitoring and supporting operations, merging development and operations teams, building observability into application designs, and designing work flows for teams that take these considerations into account. This is a significant learning curve. It's not a question of reading an article or following a five-minute tutorial.

Development practices such as trunk-based development, continuous integration, test automation, and so forth are dependencies of the key dependency. That is, stable production operations require all the underlying technical practices to be applied in ways that ensure deployments will not break production. Production, rather than development, becomes the focus of our thinking regarding technical practices.

A focus on the health of production operations has to become core to coaching all technical teams, regardless of their particular roles or activities.

36.2.4 | Mitigating the Technical Coach Shortage

The long-term solution is to shift the technical coaching industry toward a more-comprehensive and meaningful definition of the range of skills a technical coach needs. This is connected to the information in "Coaching Skills."

Based on a broad consensus among technical coaches regarding the specific skill sets involved, we (as an industry or within individual

consultancies) can develop training programs for coaches to build the missing skill sets.

There are a couple of shorter-term solutions that can mitigate the shortage.

First, an approach to technical coaching developed by Llewellyn Falco, and now used by quite a few of the top technical coaches in the industry, enables one coach to work with up to six teams concurrently.

That is not the way Falco applies his approach, but there is potential to use it to gain some leverage from a relatively small number of technical coaches on a transformation engagement. It promises to be a practical approach for medium-scale transformation programs involving tens of teams.

The approach is described in more depth later in this chapter.

Second, we can train a group of client personnel to an entry-level technical coaching capability. The external coaches can then rotate among them to make course corrections and "coach the coaches" throughout the transformation program.

This less-than-ideal approach may be a practical workaround for the coach shortage for large-scale transformation programs involving hundreds of teams.

36.2.6 | Building Coaching Competencies

The remaining problems listed in section 33.1 boil down to a lack of coaching competencies. Technical coaches today have strong technical skills and some ability to mentor people in effective practices, but generally they lack a broad coaching skill set.

Stickiness is reduced when coaches take too active a hands-on role in a team's work, and when they sit with a team full-time such that the team has no "alone time" for new learnings to sink in.

In my view, this reflects a lack of clarity about where the coach's responsibilities end and the team's responsibilities begin.

Coaches who tend to lead or drive team events such as retrospectives and stand-ups effectively take ownership of the team's outcomes. Team members never have a chance to learn to take ownership of their work. This reflects a lack of *facilitation* skills, which is a key area of coaching competence as defined in the Agile Coaching Competency Framework. This is another factor that affects stickiness.

Coaches who behave like "victims" of current organizational constraints in the client's environment may be unclear about the things a consultant normally does to drive change. It seems to me this reflects the mentality that the coach is "just another team member." This is related to building and maintaining influence and trust with the client.

The other issues mentioned in section 33.1 can occur when coaches lack skills in interpersonal interactions, such as communication models, conflict resolution, and ways of encouraging or nudging people to change their habits.

As an industry - or perhaps within forward-looking consultancies to create a competitive advantage - we need to build these skills among technical coaches in an intentional way.

36.3 | Team-Level Technical Coaching Model

In this section, I'll describe a pattern for effective technical coaching. It addresses the problem of "stickiness" of new learnings, and also enables one coach to influence more than one team at a time.

Independent technical coach Llewellyn Falco arrived at this model as a way to improve the effectiveness of his own coaching work,

after noting that many teams he had helped did not continue their improvement on their own after he left. Others who do technical coaching subsequently found the method useful, and it's gaining momentum.

36.3.1 | A Scalable Approach

In my view, an additional benefit of the model is that we can scale a small number of technical coaches to support a medium-scale transformation program. This offers a practical mitigation strategy for the general shortage of qualified technical coaches.

Do we need to "scale" coaching? I can share an anecdote from experience. A certain consultancy did good work on a small transformation engagement for a large client. On the strength of their success, they won a larger contract from the same client.

The second contract included technical coaching for 30 development teams (among other things at the management and executive levels). The consultancy did what they all do: They scrambled to hire some technical coaches at the last minute, to avoid making commitments to people before they had a signed contract in hand.

They were able to find three people who had reasonably strong technical skills, but none of the broader coaching skills, and very little experience in coaching as opposed to direct delivery work.

Using conventional coaching methods, that provided the consultancy with sufficient coaching capacity to support three client teams out of 30, and the level of coaching skill was not high. This was not a Good Thing. One of the consultancy's senior coaches played a "coach the coaches" role, and expressed frustration to me privately.

Had they applied Falco's approach, the three coaches could have supported a maximum of 18 client teams; far short of the 30 teams included in the transformation program, and probably not enough

to smooth over cross-team dependencies in order to get anything of significance done.

This underscores the severity of the shortage as well as the inadequacy of conventional methods of coaching.

36.3.2 | Foundations of the Coaching Model

There are several foundational ideas underlying Falco's approach, including:

- consistent schedule for activities
- consistent locations for activities
- alternating touch time and alone time for teams
- blend of coaching, mentoring, facilitating, and teaching
- preference for experiential learning
- training in short, focused segments rather than lengthy classes
- use of a strict form of mob programming and strong-style pairing for coaching sessions
- a slow pace of change that gives people time to work through the S-curve for new practices, and to internalize new learnings
- one new thing at a time, to enable *fluency*
- frequent small retrospectives with action taken on feedback
- explicit measures to prevent teams growing bored with coaches
- explicit measures to reach beyond the team to address organizational constraints
- assess progress when team is absent and team is working independently

Having a consistent schedule for activities that involve the coach helps teams manage their time. Without this, coaches are at the mercy of the client's existing schedule. Typically, in organizations that are in need of change, that meeting schedule is at the root of

many problems. To be effective, this is one of the first things a coach has to break.

The daily routine Falco describes is as follows. This is from the point of view of a technical coach.

Start Time	End Time	Event
9:00	11:00	Coaching Team A
11:00	12:00	Learning Hour
12:00	1:30	Lunch
1:30	3:00	Coaching Team B
3:00	5:00	Coaching Team C

Having a consistent location for recurring activities makes it easier for team members to remember when and where the events will take place and to make room in their schedules to participate. Without this, recurring events occur in different locations based on the availability of conference rooms and other facilities. Client leadership may have to exercise formal authority to make this happen.

Alternating touch time and alone time gives teams the opportunity to take ownership of their own continuous improvement, to practice and stumble with new practices, and to internalize new learnings. Ultimately the teams can become self-sufficient and independent of the coach.

This alternation also helps teams avoid taking the coach for granted, as they are not always around, and lengthens the time it takes for teams to become bored with the coach.

If the coach has a tendency to "go native," being away from the team periodically also helps with that problem.

In Falco's model, there are two cadences of alternation between coach touch time and team alone time.

Each day, the coach works with any given team for an hour in a mobbing session and team members participate in the Learning

Hour. The rest of the day, the team works without the coach present (Zuill).

During the mobbing session, the coach acts as facilitator. One team member has the keyboard and one other is the navigator. Roles are switched at frequent intervals, such as 3 to 5 minutes (Falco).

The coach works with up to three teams for two weeks, and then leaves the teams alone for two weeks. The purpose is to give teams time to internalize and gain fluency with new skills (Larsen).

In a nutshell, when a person has a *fluent* skill, they can perform that skill without having to think about how to do it. When they are working under stress, they can perform the skill effectively. When they lack fluency in a skill, they have to think about how to do it while they're doing it.

The model recognizes that a human can deal with one source of stress at a time. The effort to learn a new skill is a source of stress. The implication is people can't effectively gain fluency in a new skill if they're already under stress from some other source.

In the context of an organizational transformation program, the program itself is the primary source of stress. That means a crucial part of the transformation engagement is to alleviate the stress arising from the transformation. Otherwise, people will be unable to gain fluency in new skills, as their minds will be occupied with handling the stress of change.

When the sole source of stress is the new skill being learned, then people can practice the skill until they gain fluency with it. Then they can move on to learning another new skill.

The pace of learning in this model takes these factors into account. Progress is slow and deliberate, so that people have time to gain fluency with each new skill.

During the day, the coach balances coaching, mentoring, facilitating, and teaching as needed to help each team. The differences among these skills are described in "Coaching Skills."

When in teaching mode, the coach exploits experiential learning techniques. This approach is thought to be more effective for inculcating practical skills than the conventional lecture-then-practice approach (Kolb).

Each event concludes with a brief retrospective lasting between about 5 and 10 minutes. The scope is kept small so that action items can be addressed without delay and everyone can see progress.

For any single team, the model looks like this:

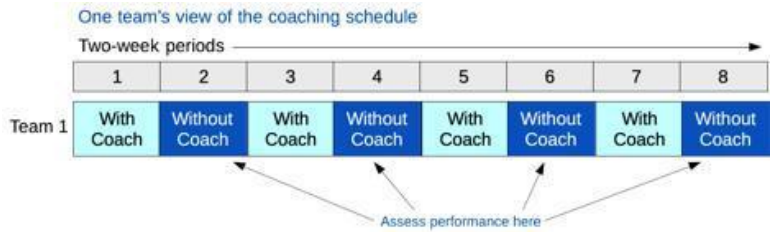

Figure 33.1: Single team's view of coaches

To enable the coach to address organizational constraints and obtain cooperation from team stakeholders, Falco makes a point of having lunch with one individual each day. These are not members of the teams he is coaching, but they are close to the teams in the organization's universe.

Lunchtime conversations are not usually about work. There are no immediate tactical goals for the conversations. The objective is to build positive working relationships. As challenges arise in the course of the engagement, these relationships can open doors for resolving issues.

Finally, it's important to measure team performance at the times when the coach is absent. When the coach is present, teams tend to do what the coach asks or to do things they believe the coach would like to see. This skews the results of any observations we might make about the team's level of fluency or internalization of new practices.

36.3.3 | What Happens in the Mobbing Sessions

During the regularly-scheduled mobbing sessions, the technical coach works with a single team. The team works on actual items from their work queue or backlog.

Work items are not cherry-picked to illustrate any particular technical practices or techniques. Techniques are introduced as the work demands them. That keeps all new learning in a real-world context, and limits the number of new concepts that are introduced at any one time.

The room is set up in the usual way for Mob Programming, with a single workstation at the front of the room, connected to a projector. The driver stands there.

Another team member is designated the navigator. They usually stand near the front of the room, possibly off to one side, so they can see the screen and everyone else can see and hear them.

Many variations are possible for Mob Programming. In this context, work is done using a strong-style pairing model in which the person at the keyboard isn't supposed to think. The only reason they type anything is that the navigator guides them. Everyone else is an observer.

Roles are switched every 3 to 5 minutes, so that everyone on the team has multiple opportunities to be the navigator and the driver, as well as ample opportunity to observe and learn.

The coach may act as a facilitator most of the time and as a teacher, mentor, or coach part of the time as appropriate.

Toward the end of the session, work stops and the group has a brief retrospective.

An important point is that the goal of the mobbing sessions is not to get work finished quickly. The goal is to learn. To that end, progress

is deliberate and the rationale and details of every new technique is explained carefully.

36.3.4 | What Happens in the Learning Hour

The Learning Hour is set aside for presenting specific topics in a more formal way than the Mob Programming sessions. Topics are short, so they will fit into the one-hour time slot. The lessons are designed around the experiential learning pattern to maximize learning effectiveness.

Some of the sample topics Falco lists in his "day in the life" video are:

- Koans
- Katas
- TDD
- Fake it till you make it
- Approval tests
- Combinatorial Testing
- Git
- Peel and slice
- User story writing
- Sparrow decks
- *many more*

As you can see, the topics are narrowly focused and learning goals are of limited scope so that people can get value within an hour.

Falco's observation from field work is that one hour of focused learning per day yields significant improvement over time. In a presentation, he suggests 5x improvement can be achieved in one year's time in this manner.

On the other hand, he doesn't mention how that is measured. For me, the key point is that this approach is more effective than the

"heavy" classroom approach. Exactly how many X's it yields is less interesting, and probably highly variable from case to case.

Three-to-five-day formal classes in which participants attend 8 hours a day are far less effective. It's been known for many years that when new skills aren't applied in practice within a couple of weeks of training, people forget what they saw in class. Besides that, people get worn out from sitting in a class all day long, day after day, and lose focus.

36.3.5 | Avoiding Burnout and Boredom

Falco's model includes a mechanism to hand teams off to new coaches. Advantages include avoiding team boredom, avoiding coach burnout, and bringing additional perspectives and coaching styles to bear.

The mechanism is called the Guest Coach Program. A new technical coach joins the current one for a couple of weeks. First, the new coach shadows the original one. Then, the new coach facilitates team events while the original one observes and guides. Finally, the new coach takes over, following the same schedule and routine as the original coach did.

Technical coach Emily Bache describes this in her talk at SCLConf 2018. At the time of this writing, she is also preparing an ebook on LeanPub entitled *Technical Agile Coaching*, that covers Falco's approach (Bache).

36.4 | Scaling the Coaching Model

As any single technical coach works with three teams in the course of a day and alternates two weeks on and two weeks off with the teams, the model offers a straightforward way to scale. A coach

can work with three teams for two weeks and with three different teams the following two weeks.

For example, two technical coaches could support up to 12 teams in this way:

Coaching Schedule: 12 teams, 2 coaches

	Two-week periods →							
	1	2	3	4	5	6	7	8
Coach A	Teams 1,2,3	Teams 4,5,6	Teams 1,2,3	Teams 4,5,6	Teams 1,2,3	Teams 4,5,6	Teams 1,2,3	Teams 4,5,6
Coach B	Teams 7,8,9	Teams 10,11,12	Teams 7,8,9	Teams 10,11,12	Teams 7,8,9	Teams 10,11,12	Teams 7,8,9	Teams 10,11,12

Figure 33.2: 12 teams, 2 coaches

Depending on how many qualified technical coaches are available, this approach could scale linearly to an arbitrarily large transformation program. Given the current shortage of qualified technical coaches, it is probably feasible for medium-scale engagements involving no more than some tens of teams. For large-scale programs involving hundreds of teams or more, there are simply not enough qualified technical coaches available.

We have to find creative workarounds for that problem, at least in the short to medium term.

36.5 | Developing Internal Technical Coaches

As a workaround to the shortage, to support large-scale transformation programs we can train client personnel to serve as entry-level or apprentice technical coaches under the guidance of qualified technical coaches.

The downsides are:

- longer timelines to effect meaningful improvement in large-scale transformation programs;
- impact of shifting some of the best technical people in the client organization away from business as usual to become apprentice technical coaches;
- overall coaching effectiveness will not be as high as it could be if all the technical coaches had senior-level coaching competencies; and
- as the key dependency lies within the technical sphere (stable production operations), it will be the bottleneck for achieving the overall transformation goals (competitive capabilities).

The qualified technical coaches will perform the training. Afterwards, the apprentice coaches will play the role described above with one to three technical teams.

The qualified coaches will rotate around the organization, visiting each mobbing session and Learning Hour and consulting with the individual apprentice coaches to ensure they are continuing to hone their skills and teams are making progress.

This is clearly not an ideal solution, but may be a practical way to scale technical coaching in the face of the current shortage of qualified coaches.

36.6 | Barriers to Adoption

We are years away from having a strong pool of qualified technical coaches in the job market. Until we can build up the community in this area, this level of technical coaching expertise will remain a "boutique" offering.

Adoption of Falco's model is progressing very slowly among practitioners. Just as client personnel are reluctant to move outside their comfort zones, technical coaches are reluctant to move outside

their own. The idea of such a light touch with teams doesn't seem practical to many of them.

They may also be impatient to introduce a great deal of information quickly, thinking this will help the teams pick things up faster. In my view, if that were the case it would have happened by now, after all these years of conventional coaching methods.

A couple of other market trends may be contributing to the shortage of qualified technical coaches. For one, team-level coaching is seen as a low-end commodity staff augmentation service rather than as a genuine coaching service. Large consultancies are offering such coaching on that basis and at low rates. Client leaders tend not to understand the nature and importance of this level of coaching, and aren't interested in paying for it.

For another, most technical people are unaware of the importance of cultivating the other coaching-related skill sets. There is a common assumption that strong technical skills are sufficient to provide useful guidance to client teams.

The next point may be more a *trend* than a *barrier*, but it does affect the shortage of technical coaches. Understandably, many of the more-senior technical coaches feel as if they can add more value for clients if they focus higher in the organization. They transition from team-level technical coaching to mid-management or executive coaches. That makes the hole at the team level even bigger.