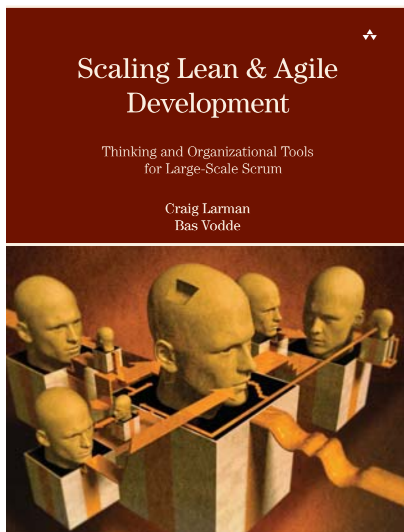
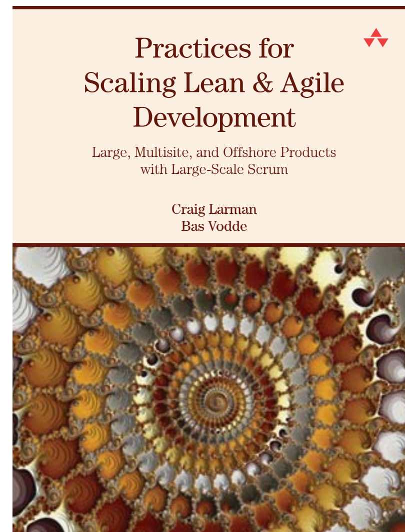
Chapter

- **Introduction to Feature Teams 154**
- **Avoid…Single-function teams 161**
- **Avoid…Component teams 161**
- **Try…Feature teams 180**
- **Transition 195**





Book

1 Introduction 2

Thinking Tools

2 Systems Thinking 10

3 Lean Thinking 40

4 Queueing Theory 94

5 False Dichotomies 126

6 *Be* Agile 140

Organizational Tools

7 Feature Teams 150

8 Teams 194

9 Requirement Areas 218

10 Organization 230

11 Large-Scale Scrum 288

Miscellany

12 Scrum Primer 310

13 Recommended Readings 332

14 Bibliography 338

Chapter 7

Feature Teams

Better to teach people and risk they leave, than not and risk they stay
—anonymous

Introduction to Feature Teams

Figure 7.1 shows a **feature team**—a long-lived,[1] cross-functional team that completes many end-to-end customer features, one by one.

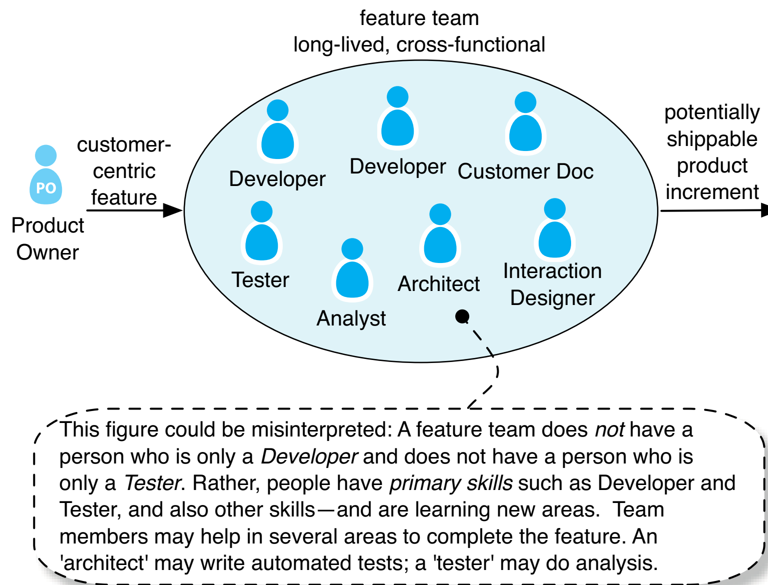


Figure 7.1 feature team—long-lived, cross-functional, learning-oriented, multi-skilled people

1. A misunderstanding is that new teams re-form for each feature. Not true. A great feature team may stay together for years.

Excerpt from *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, Larman & Vodde, Addison-Wesley. Copyright (c) 2009. All rights reserved.

See also the companion book *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*, Larman & Vodde, Addison-Wesley.

In Scrum and other agile methods the recommended team structure is to *organize teams by customer-centric features*. Jim Highsmith, in *Agile Project Management [Highsmith04]*, explains:

> *Feature-based delivery means that the engineering team builds [customer-centric] features of the final product.*

lean thinking wastes p. 59

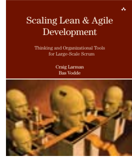
In lean thinking, minimizing the wastes of handoff, waiting, WIP, information scatter, and underutilized people is critical; cross-functional, cross-component feature teams are a powerful lean solution to reduce these wastes.

Why study the following in-depth analysis? Because feature teams are a key to accelerating time-to-market and to scaling agile development, but a *major* organizational change for most—changing team structure is slow work, involving learning, many stakeholders, and policy and mindset issues. If you're a change agent for large-scale agility, you need to really grasp the issues.

Figure 7.2 a long-lived feature team; developers, testers, and others create a complete customer feature

Scrum team p. 314

A proper **Scrum team** is by definition a feature team, able to do all the work to complete a Product Backlog item (a customer feature). Note that Scrum team (feature team) members have no special title other than "team member." There is not emphasis on 'developer' versus 'tester' titles. The goal is to encourage multi-skilled workers and "whole team does whole feature." Naturally people have primary specialities, yet may sometimes be able to help in less familiar areas to get the job done, such as an 'analyst' helping out with automated testing. The titles in Figure 7.1 should not be misinterpreted as promoting working-to-job-title, one of the wastes in lean thinking.



Feature teams are not a new or 'agile' idea; they have been applied to large software development for decades. They are a refinement of **cross-functional teams**, a well-researched proven practice to speed and improve development. The term and practice was popularized at Microsoft in the 1980s and discussed in *Microsoft Secrets* [CS95]. Jim McCarthy [McCarthy95], the former development lead of Visual C++, described feature teams:

cross-functional team p. 197

> *Feature teams are about empowerment, accountability, identity, consensus and balance…*
>
> ***Empowerment**—While it would be difficult to entrust one functional group or a single functional hierarchy, such as Development, for instance, with virtually absolute control over a particular technology area, it's a good idea to do that with a balanced multi-disciplinary team. The frontline experts are the people who know more than anyone else about their area, and it seems dumb not to find a way to let them have control over their area.*
>
> ***Accountability**—… If a balanced group of people are mutually accountable for all the aspects of design, development, debugging, QA, shipping, and so on, they will devise ways to share critical observations with one another. Because they are accountable, if they perceive it, they own it. They must pass the perception to the rest of the team.*
>
> ***Identity**—… With cross-functional feature teams, individuals gradually begin to identify with a part of the product rather than with a narrow specialized skill.*
>
> ***Consensus**—Consensus is the atmosphere of a feature team. Since the point of identification is the feature rather than the function, and since the accountability for the feature is mutual, a certain degree of openness is safe, even necessary. I have observed teams reorganizing themselves, creating visions, reallocating resources, changing schedules, all without sticky conflict.*
>
> ***Balance**—Balance on a feature team is about diverse skill sets, diverse assignments, and diverse points of view.*

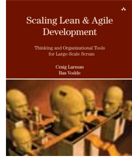
Feature teams are common in organizations learning to deliver faster and broaden their skills. Examples include Microsoft, Valtech (applied in their India center for agile offshore development), the Swedish software industry [OK99], Planon [Smeets07], and telecom industry giant Ericsson [KAL00]. The report on Ericsson's feature teams clarifies:

> *The feature is the natural unit of functionality that we develop and deliver to our customers, and thus it is the ideal task for a team. The feature team is responsible for getting the feature to the customer within a given time, quality and budget. A feature team needs to be cross functional as it needs to cover all phases of the development process from customer contact to system test, as well as all areas [cross component] of the system which is impacted by the feature.*

To improve development on large products (one sub-project may be one million person hours) in their GSM radio networks division, Ericsson applies several practices supporting agility, including feature teams and daily builds. It's no coincidence that *both* these practices were popularized by Microsoft in the 1990s; Ericsson also understands the synergy between them [KA01]:

> *Daily build can only be fully implemented in an organization with predominantly customer feature design responsibility.*
>
> *… The reasons why feature responsibility is a prerequisite for taking advantage of daily build is the amount of coordination and planning needed between those responsible for delivering consistent parts of each module that can be built. … In a feature team this coordination is handled within the team.*

In another book describing the successful practices needed for scaling agile development, Jutta Eckstein similarly recommends *"vertical teams, which are focused around business functionality"* [Eckstein04]. Feature teams do 'vertical' end-to-end customer features (GUI, application logic, database, …) rather than 'horizontal' components or layers. In her more recent scaling book she again emphasizes *"In order to always keep the business value of your customer in mind, there is only one solution: having feature teams in place"* [Eckstein09].



A common misunderstanding is that each feature team member must know everything about the code base, or be a generalist. Not so. Rather, the team is composed of specialists in various software component areas and disciplines (such as database or testing). Only collectively do they have—or can learn—sufficient knowledge to complete an end-to-end customer feature. Through close collaboration they coordinate all feature tasks, while also—important point—learning new skills from each other, and from extra-team experts. In this way, the members are **generalizing specialists**, a theme in agile methods [KS93, Ambler03], and we reduce the waste of underutilized people (working only in one narrow speciality), a theme in lean thinking.

multi-skilled workers p. 205

To summarize the ideal **feature team**[2]:

Feature Team
❑ long-lived—the team stays together so they can 'jell' for higher performance; they take on new features over time
❑ cross-functional and cross-component
❑ co-located
❑ work on a complete customer-centric feature, across all components and disciplines (analysis, programming, testing, …)
❑ composed of generalizing specialists
❑ in Scrum, typically 7 ± 2 people

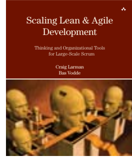
long-lived teams p. 200

Feature teams work independently by being empowered and given the responsibility for a whole feature. Advantages include:

work redesign p. 235

2. A Scrum feature team is typically stable, long-lived. The name "feature team" was first popularized by Microsoft, but is also used in the (relatively rare) method **Feature-Driven Development** (FDD). However, in FDD a "feature team" is only a short-term group brought together for one feature and then disbanded. Such groups have the productivity disadvantage of not being 'jelled'—a rather slow social process—and the disadvantage of not providing stable work relationships for people.

- **increased value throughput**—focus on delivering what the customer or market values most
- **increased learning**—individual and team learning increases because of broader responsibility and because of co-location with colleagues who are specialists in a *variety* of areas
  - critical for long-term improvement and acceleration; reduces the waste of underutilized people
- **simplified planning**—by giving a whole feature to the team, organizing and planning become easier
  - for example, it is no longer necessary to coordinate between single-specialist functional and component teams
- **reduced waste of handoff**—since the entire co-located feature team does all work (analysis, design, code, test), handoff is *dramatically* reduced
- **less waiting; faster cycle time**—the waste of waiting is reduced because handoff is eliminated and because completing a customer feature does not have to wait on multiple parties each doing part of the work serially
- **self-managing; improved cost and efficiency**—feature teams (and Scrum) do not require a project manager or matrix management for feature delivery, because coordination is trivial. The team has responsibility for end-to-end completion and for coordinating their work with others. Data shows an inverse relationship between the number of managers and development productivity, and also that teams with both an internal and external focus are more likely to be successful [AB07]. Feature teams are less expensive—there isn't the need for extra overhead such as project managers.
  - For example [Jones01]: "*The matrix structure tends to raise the management head count for larger projects. Because software productivity declines as the management count goes up, this form of organization can be hazardous for software.*"
- **better code/design quality**—multiple feature teams working on shared components creates pressure to keep the code clean, formatted to standards, constantly refactored, and surrounded by many unit tests—as otherwise it won't be possible to work



with. On the other hand, due to long familiarity, component teams live with obfuscated code only they can understand.

- **better motivation**—research [HO80, Hackman02] shows that if a team feels they have complete end-to-end responsibility for a work item, and when the goal is customer-directed, then there is higher motivation and job satisfaction—important factors in productivity and success.
- **simple interface and module coordination**—one person or team updates both sides of an interface (caller and called) and updates code in all modules; because the feature team works across all components; no need for inter-team coordination.
- **change is easier**—changes in requirements or design (*we know it's rare, but we heard it happened somewhere once*) are absorbed by one team; multi-team re-coordination and re-planning are not necessary.

AVOID…SINGLE-FUNCTION TEAMS

A Scrum feature team is cross-functional (cross-discipline), composed of testers, developers, analysts, and so on; they do all work to complete features. One person will contribute primary skills (for example, interaction deisgn or GUI programming) *and* also secondary skills. There is no separate specification team, architecture team, programming team, or testing team, and hence, much less waiting and handoff waste, plus increased multiskill learning.

cross-functional teams p. 197

AVOID…COMPONENT TEAMS

An old approach to organizing developers in a large product group is **component teams**—programmer groups formed around the architectural modules or components of the system, such as a single-speciality GUI team and component-X team. A customer-centric feature is decomposed so that each team does only the *partial* programming work for their component. The team owns and maintains their component—single points of specialization success or failure.

In contrast, feature teams are not organized around specific components; the goal is a cross-component team that can work in all modules to complete a feature.

Components (layer, class, …) still exist, and we strive to create good components, but we do not organize teams by these.

What About Conway's Law?

Long ago, Mel Conway [Conway68] observed that

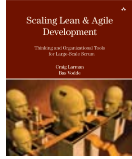
> *[…] there is a very close relationship between the structure of a system and the structure of the organization which designed it.*
>
> *… Any organization that designs a system […] will inevitably produce a design whose structure is a copy of the organization's communication structure.*[3]

That is, once we define an organization of people to design something, that structure *strongly* influences the subsequent design—typically in a one-to-one homomorphism. A striking example Conway gave was

> *[An] organization had eight people who were to produce a COBOL and an ALGOL compiler. After some initial estimates of difficulty and time, five people were assigned to the COBOL job and three to the ALGOL job. The resulting COBOL compiler ran in five phases, the ALG0L compiler ran in three.*

Why raise this topic? Because "Conway's Law" has—strangely—been incorrectly used by some to promote component teams, as if Conway were recommending them. But his point was very different: It was an *observation of how team structure limits design, not a recommendation.* Cognizant of the negative impact, he cautioned:

3. In [Brooks75] this was coined **Conway's Law**.



To the extent that an organization is not completely flexible in its communication structure, that organization will stamp out an image of itself in every design it produces.

… Because the design that occurs first is almost never the best possible, the prevailing system concept [the design] may need to change. Therefore, flexibility of organization is important to effective design. Ways must be found to reward design managers for keeping their organizations lean and flexible.

In this way, Conway underlines a motivation for feature teams.

In *Microsoft Secrets* [CS95], Brad Silverberg, senior VP for Windows and Office, explained their emphasis on feature teams, motivated by the desire to avoid the effects of “Conway’s Law”:

The software tends to mirror the structure of the organization that built it. If you have a big, slow organization, you tend to build big, slow software.

Disadvantages

It is extraordinary the amount of delay, overhead, unnecessary management, handoff, bad code, duplication, and coordination complexity that is introduced in large groups who organize into component teams, primarily driven by two assumptions or fears: 1) people can’t or shouldn’t learn new skills (other components, testing, …); and 2) code can’t be effectively shared and integrated between people. The first assumption is fortunately not so, and the second, more true in the 1970s, has been resolved with agile engineering practices such as continuous integration and test-driven development (TDD).

Component teams seemed a logical structure for 1960s or 1970s sequential life cycle development with its fragile version control, delayed integration, and weak testing tools and practices because the *apparent* advantages included:

- people developed narrow specialized skill, leading to apparently faster work when viewed locally rather than in terms of overall systems throughput of customer-valued features, and when viewed short-term rather than long-term

- those specialists were less likely to break their code
- there were no conflicting code changes from other teams

Fortunately, there has been much innovation since the 1960s. New life cycle and team structures have been discovered, as have powerful new version-control, integration, and testing practices.

Systems and lean thinking invite us to ask, "Does a practice globally optimize value throughput with ever-faster concept-to-cash cycle time, or locally optimize for a secondary goal?" From that perspective, let's examine the disadvantages of a component team…

Promotes Sequential Life Cycle Development and Mindset

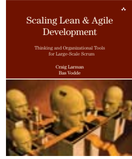
Customer features don't usually map to a single component nor, therefore, to a single component team; they typically span many modules. This influences organization of work.

Who is going to do requirements analysis? If several component teams will be involved, it is not clear that any particular one of them should be responsible for analysis. So, a separate analyst or analyst team does specification in a first step.

Who is going to do high-level design and planning? Again, someone *before* the component teams will have to do high-level design and plan a decomposition of the feature to component-level tasks. She is usually titled an architect or systems engineer; in [Leffingwell07] this role is called requirements architect. In this case, one usually sees a planning spreadsheet similar to the following:

Feature	Component A	B	C	D	E	…
Feature 1	x	x			x	
Feature 2	x	x	x			
…						

Who is going to test the end-to-end feature? This responsibility doesn't belong to any one component team, who only do part of the work. So testing is assigned to a separate system-test team, and they start high-level testing *after* development has finished—some-



times long after, as they need the work of multiple component teams and these teams seldom finish their work at the same time. Plus, they have a backlog of other features to test.

Now what do we have?

1. (before development) requirements analysis by a separate analyst
2. (before) high-level design and component-level task planning by a separate designer
3. (during) implementation by multiple interdependent component teams that have to coordinate partially completed work
4. (after) system testing of the feature

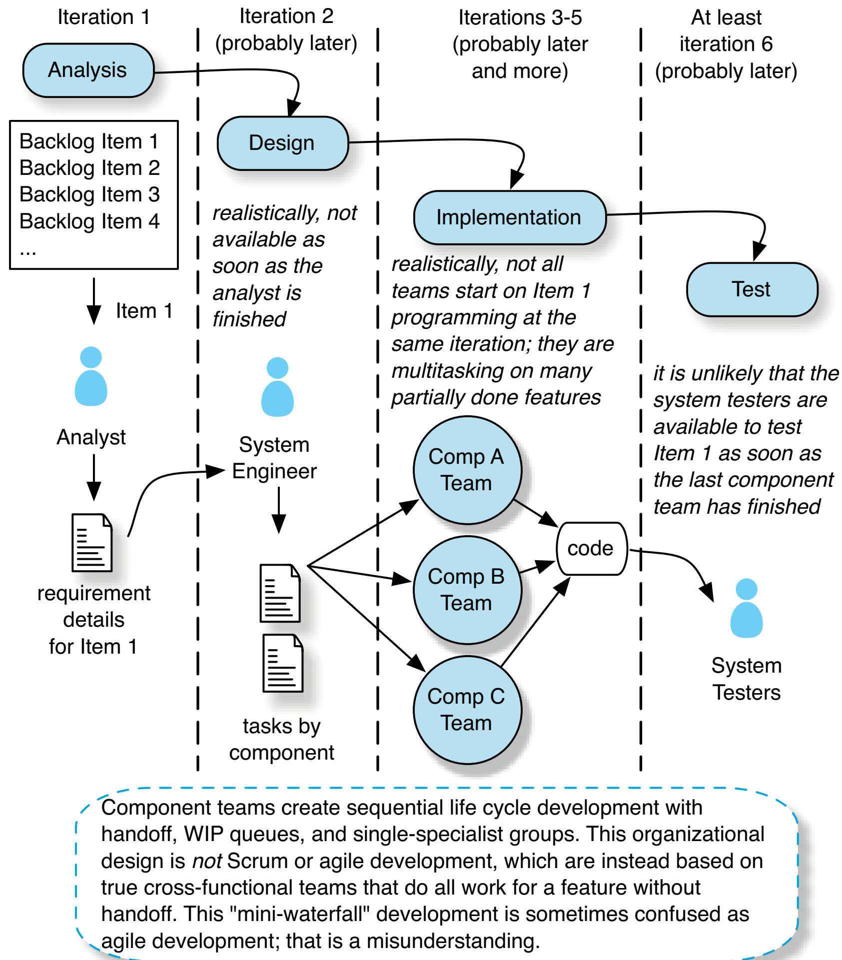
Back to a waterfall! There is massive handoff waste in the system and plenty of delay. This is traditional sequential life cycle development and mindset, even though—ironically—people may incorrectly think they are doing Scrum or agile development simply because they are doing mini-waterfalls in a shorter and iterative cycle (Figure 7.3). *But mini-waterfalls are not lean and agile development*; rather, we want real concurrent engineering.

Completing one non-trivial feature now typically takes *at least* five or six iterations instead of one.[4] And it gets worse: For very large systems the organization adds a subsystem layer with a subsystem architect and subsystem testing—each specialized and each adding another phase delay before delivering customer functionality.

> Component team structures and
> sequential life cycle development are directly linked.

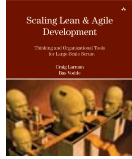
4. Five or six iterations is optimistic. With multiple component teams, the handoff, waiting, and overhead coordination delays implementation over many iterations.

Figure 7.3 component teams lead to sequential life cycle



Limits Learning

Consider this thought experiment, although it will never be achieved: Option 1—Everyone working on the product can do everything well. Option 2—Every person can do one (and only one) small task extremely well, but nothing else. Which option allows faster feature throughput? Which option has more bottlenecks? Which



offers more adaptability? Although the perfection vision of option-1 isn't possible, viewed along a continuum of desirability, we want to encourage a learning organization that is moving in that direction—reducing bottlenecks, people learning one area well, then two, …

Observations:

- Developing multi-skilled people takes plenty of learning opportunities and close work with different kinds of experts.
- More specifically, developing *programmers* who can help in several components requires a variety of experiences and mentors.
- Data shows an extraordinary variance in individual programmer productivity—studies suggest an average of four times faster in the top versus bottom quartile [Prechelt00].

There's a strong link in software development between what you *know* and what you can *do well*—software is the quintessential knowledge-sensitive profession. In short: There are great business benefits if we have skilled developers who are constantly learning.

This learning has preconditions, of management responsibility:

- slack[5]
- a *structure* to support continual learning
  - but there's a systemic flaw in component teams…

How do developers become skilled in their craft and broadly knowledgeable about their product? We asked Pekka Laukkanen—an experienced developer and creator of the Robot test framework [Laukkanen06, Robot08]—a question: "How do you become a great developer?" He thought about it carefully and answered: "Practice—and this means not just writing lots of code, but reflecting on it. *And reading others' code because that's where you learn to reflect on your own*."

Yet, in traditional large-product groups with component teams, most developers know only a narrow fragment of the system, and most salient, they don't see or learn much that is new.

5. See *Slack* [DeMarco01] on the need for slack to get better.

And on the other hand, there are always a few wonderful people who know a lot about the system—the people you would go to for help on an inexplicable bug. Now, when you ask how that's possible, a common answer will be, "*He knows everything since he always reads everybody's code*." Or, "*He's worked on a lot of different code*." Interestingly, such people are more common in large open source products; there is a culture and value of "*Use the source, Luke*" [Raymond] that promotes reading and sharing knowledge via code.

Why does this matter? Because component teams inhibit developers from reading and learning new areas of the code base, and more broadly, from learning new things.

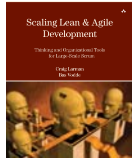
Contrast the organizational mindset that creates such a structure of limited learning with the advice of the seminal *The Fifth Discipline* [Senge94] in which MIT's Peter Senge summarizes the focus and culture of great long-lived companies: *learning* organizations. *Lean Process and Product Development* [Ward06] also stresses this theme; it summarizes the insight of Toyota's new product development success: *It's about creating lots of knowledge, and about continual learning*. And *Toyota Talent* [LM07] asks the question: "How does Toyota continue to be successful through good times and bad?" and answers "The answer is simple: great people," and

> *It is the knowledge and capability of people that distinguishes any organization from another. For the most part, organizations have access to the same technology, machinery, raw material, and even the same pool of potential employees as Toyota. The automaker's success lies partially in these areas, but the full benefit is from the people at Toyota who cultivate their success.*

Isao Kato, one of the students of Taichii Ohno (father of the Toyota Production System), said:

> *In Toyota we had a saying, "Mono zukuri wa hito zukuri", which mean "Making things is about making people." [Kato06]*

Yet what is the journey of the software developer in many large product groups? After graduating from university, a young developer joins a large company and is assigned to a new or existing component. She writes the original code or evolves it, becoming the specialist. There she stays for years—apparently so that the organization



can "go faster" by exploiting her one specialty—becoming a single point of success or failure, a bottleneck, and learning only a few new things. The university did not teach her good design, so where did she learn good from bad? How can she see lots of different code? How can she see opportunities for reusable code? How can she help elsewhere when there's a need?

Note that the problem is not specialization; it is *single*-specialization, bottlenecks, and team structures that inhibit learning in new areas. To create a learning organization, we want a structure where developers can eventually become skilled in two areas—or more. Component teams inhibit that.

Component team (and single-function team) organizations gradually incur a **learning debt**—learning that should have occurred but didn't because of narrowly focused specialists, short-term quick-fix fire fighting, lack of reflection, and not keeping up with modern developments. When the product is young, the pain of this debt isn't really felt. As it ages and the number of single-specialized teams—the number of bottlenecks—expands from 5 to 35, this debt feels heavier and heavier. Those of you involved in old large products know what we mean.

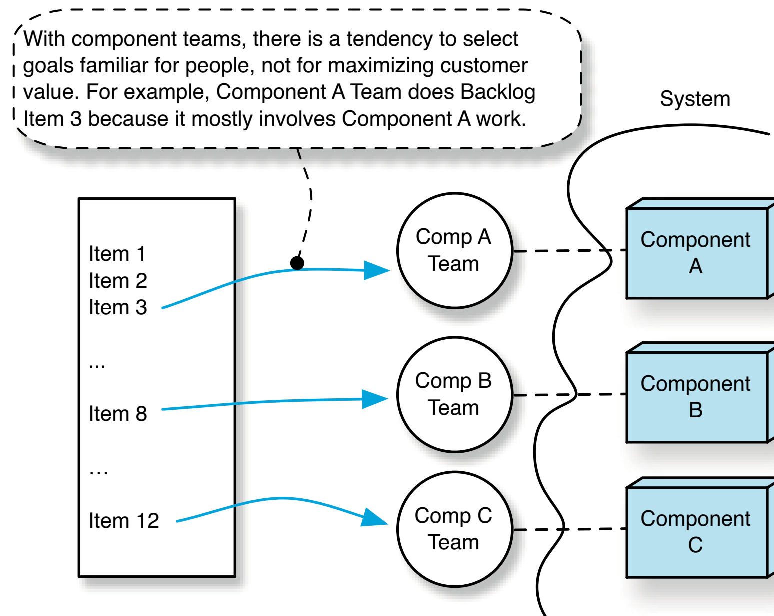
Encourages Delivery of Easier Work, not More Value

Component specialists, like other *single*-specialists, create an organizational constraint or bottleneck. This leads to a fascinating sub-optimization: *Work is often selected based on specialty rather than customer value.*

Component teams are faster at developing customer features that primarily involve their single-speciality component—if such single-component customer features can be found (not always true). For that reason, when people are sitting in a room deciding what to do next, features are often selected according to what available component teams can do best or quickest. This tends to maximize the amount of code generated, but does not maximize the value delivered.[6] Therefore, component teams are optimized for *quickly developing features (or parts of features) that are easiest to do, rather than*

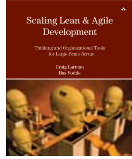
of highest value. We once saw a component team assigned to code their small part of a low-priority customer feature that was not due for more than 18 months in the future, simply because it was easier to plan that way.

Figure 7.4 lower-value work chosen



Interestingly, this sub-optimization is often invisible because 1) there isn't prioritization based on a customer-value calculation or the prioritization scheme consists of bizarre super-coarse-grained variants such as "mandatory" versus "absolutely mandatory"; 2) component teams tend to be busy fixing bugs related to their component; and, 3) there is plenty of internal local-improvement work. Everyone appears very busy—they must be doing valuable work!

6. Not only do more lines of code (LOC) *not* imply more value, more code can make things worse. Why? Because there is a relationship between LOC and defect rates and evolution effort. More code equals more problems and effort. Great development groups strive to *reduce* their LOC while creating new features, not increase it. Since component teams have a narrow view of the code base, they don't see reuse or duplication issues.



The sub-optimization becomes clear when we create a real Product Backlog, sorted by a priority that includes value (Figure 7.4).

The Resource Pool and Resource Manager Quick Fix

One quick-fix way that traditional resource management tackles the priority problem is by creating projects according to which specialists are required and available [McGrath04]. Project managers select people from a specialist **resource pool** and release them back when finished. This gives rise to **project groups** or **feature projects**, usually with matrix management. In such organizations one hears people called 'resources' as though they were machine parts and human or team dynamics had little importance on productivity or motivation (which is not the case).

project versus product p. 239

Thus, with a resource pool, management twists the organization around single-specialist constraints. It seems to work well on paper or in a project management tool. But people are not machine parts—they can learn, be inspired or de-motivated, gain or lose focus, etc. In practice, resource pool and feature project management has disadvantages:

- **lower productivity due to non-jelled project groups**—there is clear evidence that short-lived groups of people brought together for a project—a "project group"—are correlated with lower productivity [KS93].
- **lower motivation and job satisfaction**—I often lead a "love/hate" exercise with many people in an enterprise to learn what they, well… *hate*. In large groups focused around resource pools and project groups, "we hate being part of a resource pool thrown into multiple short-term groups" is always at or near the top.

work redesign p. 235

- **less learning**—more single-specialization as people seldom work/learn outside their area.
- **lower productivity due to multitasking**—with resource pool management it is common to create partial 'resource' allocations where a person is 20% allocated to project-A, 20% to project-B, and so forth.[7] This implies increasing multitasking

and—key point—lots of multitasking reduces productivity in product development, it does not improve it [DeMarco01].

- **lower productivity and throughput due to increased handoff and delay waste**—the people in the temporary group are often multitasking on many projects. If that's the case, it leads to another productivity/throughput impact: Since they are not working together at the same time on the same goal, there is delay and handoff between the members.
- **lower productivity and increased handoff and delay due to physical dispersion**—the project group is rarely co-located in the same room; members may be in different offices, buildings, or even cities (and time zones), and have little or no relationship with each other; physical and time zone dispersion of a task group impacts productivity [OO00].
- **lower productivity and higher costs due to more managers**—if each temporary project group has a project manager (usually in a matrix management structure), costs are higher and productivity lower because of the inverse relationship between management count and software productivity.

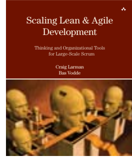
Go See p. 53

Observe the relationship between the lean "Go See" attitude and the belief that it is skillful to have resource pools that optimize around single-specialist constraints. People that do not spend regular time physically close to the real value-add workers may believe in resource pools and short-lived project groups because it appears on paper—as with machine parts—to be flexible and efficient. Yet those frequently involved in the real work directly see the subtle (but non-trivial) problems.

Promotes Some Teams to Do "Artificial Work"

A corollary of the disadvantage of *encourages delivery of easier work, not more value* is illustrated by an example: Assume the market wants ten features that primarily involve components A–T and thus (in the simplest case) component teams A–T. What do component teams U–Z do during the next release? The market is not calling for high-value features involving their components, and there may even

7. Or worse. We've even seen 10% partial project allocations!



be *no* requests involving their components. In the best case, they are working on lower-value features—because that is all they can do. In the worst case, there is an explicit or more frequently a subtle implicit creation of artificial work for these teams so that component team U can keep busy doing component-U programming, even though there is no market driver for the work.

With component teams and large product groups there is often a resource manager who tries to keep the existing teams busy ("100% allocated") by choosing and assigning this low-value or artificial work, or by asking the underutilized teams for advice. Their focus is the *local optimization* of "everyone doing their best"—generating code according to what people know, rather than generating the most value. And the work is sometimes 'redesign': If we don't have anything new, we'll redo what we did before.[8]

More Code Duplication and Hence Developers

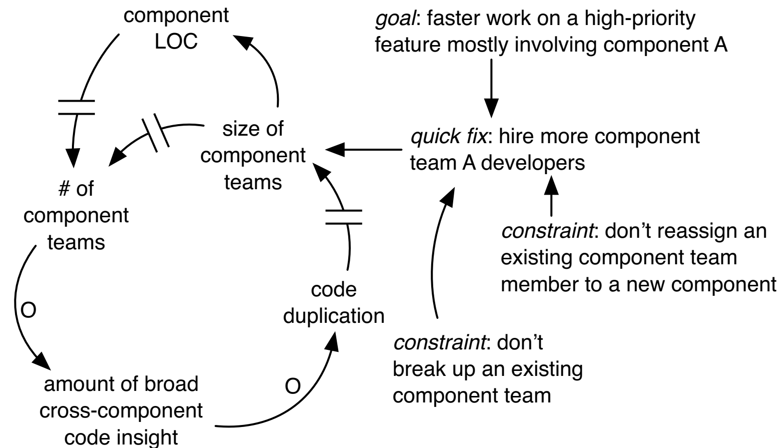
We once visited a client with many component teams and discussed the link between this structure and code duplication. The client asked, rhetorically, "Do you know how many XML parsers we have?"

Consider duplication: Good code is free of it, and great developers strive to create *less* code as they build new features, through constant refactoring. It's difficult to see duplication or opportunities for reuse in a code base with single-component specialists, because one never looks broadly. Single-component specialists increase duplication. And so the code base grows ever larger than necessary, which in turn demands more single-component specialists…[9]

see Legacy Code in companion book

8. Improving existing code is a good thing; our point is different.
9. Code-cloning statistics based on (imperfect) automated analysis of large systems shows around 15% duplicated code [Baker95], but this is probably an underrepresentation because such tools don't robustly find "implicit duplication" of different-looking code that does the same thing. *Anecdote*: I've done refactoring (to remove duplication) on large systems built by component teams, removing explicit and implicit duplication; reduction averaged around 30%.

Figure 7.5 system dynamics of component teams and number of developers



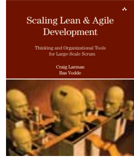
Ever-Growing Number of Developers

Component teams create several forces to increase the number of developers. One reason examined previously is the increased code bulk due to duplication. A second reason involves the mismatch between the existing component teams and the high-priority work, as explained next and summarized in the system dynamics diagram, Figure 7.5.

Component teams become boxes in the formal organization structure, each with its own manager. Several component teams form a subsystem group or department with a second-level manager. This leads to an interesting dynamic…

Example: **Current release**—A high-priority goal involves mostly work in component or subsystem A, and therefore component-A or subsystem-A groups work on it. They hire more people in the belief it will make them go faster. Component-C team has lower-priority goals and does not need or get more people. **Next release**—A high-priority goal involves primarily work for component C. Now, they are viewed as the bottleneck and so hire more people (see Figure 7.6).

We could have moved people from one component team to another, and gradually taught them (through pair programming) to help,



instead of hiring more people. But this rarely happens. The other component team already has work chosen for the release, so they won't wish to lose people. And there is a fear it will take too long to learn anything to be helpful. Also, the mindset is that "it would be a waste of our specialist to move her to another component team." Finally, moving people is inhibited by political and management status problems—many managers don't want to have a smaller group (another force of local optimization). Conway formulated this well:

> *Parkinson's law [Parkinson57] plays an important role… As long as the manager's prestige and power are tied to the size of his budget, he will be motivated to expand his organization. [Conway68]*

Thus, the component-A team will grow, as will the component itself. It may even eventually split into two new components, and hence two new teams. The people will specialize on the new components. In this way large product organizations tend to grow even larger.

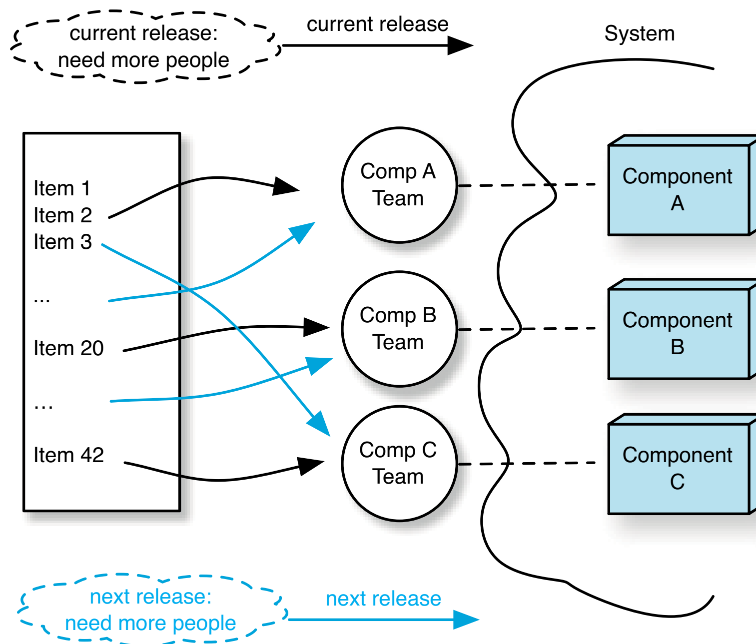


Figure 7.6 ever-growing size with component teams

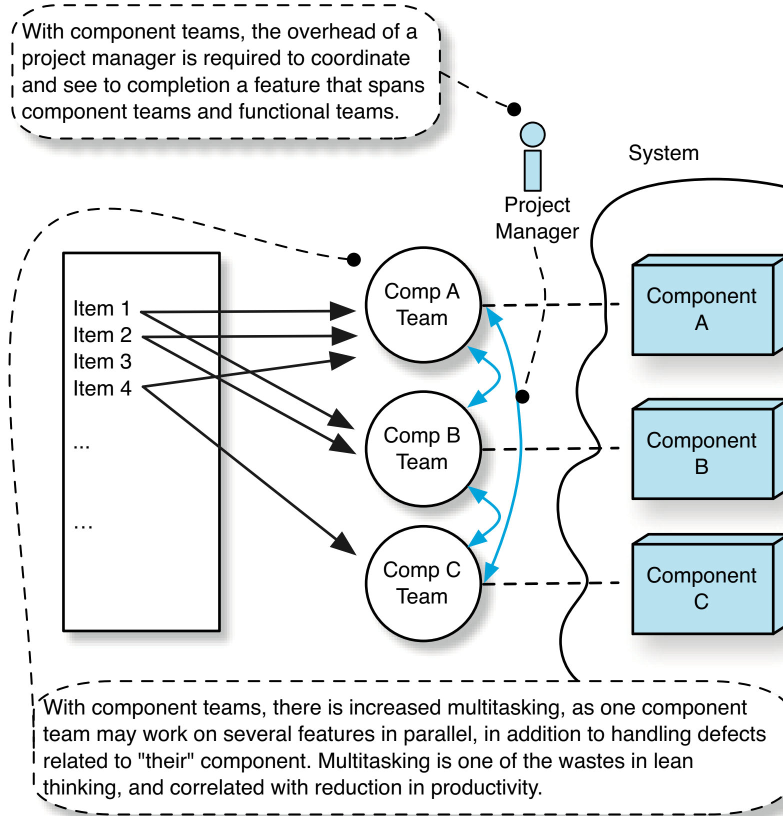
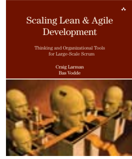


Figure 7.7 challenges in planning—coordination

Problems in Planning and Coordination

Scrum (and other agile methods) strive for an integrated product at the end of every iteration with demonstrable customer functionality. For most features this involves multiple component teams and therefore complicates planning and coordination between teams.

Example: In the next iteration the goal is to do Product Backlog items 1, 2, 3, and 4. Backlog item 1 (customer feature 1) requires changes in component A and B. Item 2 requires changes in component A, B, and C, and so forth. All teams depend on one another in the iteration planning and need to synchronize their work during



the iteration (see Figure 7.7)—a task that is often handled by a separate project manager. Even if we successfully plan the complex interdependencies for this iteration, a delay in one team will have a ripple effect through all component teams, often across several iterations.

Delays Delivery of Value

Value can be delivered only when the work of multiple component teams is integrated and tested. Figure 7.3 illustrates how component teams promote sequential life cycle. So what? With a component team organization, the work-in-progress (WIP) from a team usually waits several iterations before it can be combined into a valuable feature. This WIP, like all inventory, is one of the wastes in lean thinking; it hides defects, locks up an investment, reduces flexibility, and slows down the delivery of value. And in addition to the straightforward sequential life cycle reasons already discussed, component teams delay delivery as follows…

Example:

1. Item 1 in the Product Backlog involves component A. Component team A will work on their part of item 1 next iteration.
2. Item 4 involves components A and C. Since component team A is busy with item 1, they do not work on item 4.
3. Item 4 is the highest goal involving component C. Component team C therefore works on their part of item 4 next iteration.

- *First problem:* Not every team is working on highest value.
- *Second problem*: After the iteration, item 4 (which needs code in components A and C) can't yet be integrated, tested, and delivered, because of the missing component A code. Item 4 delivery has to wait for component team A.

Organizations try to solve this problem by the quick fix of creating a role, called **project manager** or **feature manager**, for coordinating the work across teams and/or by creating temporary **project groups** whose far-flung members multitask across multiple concurrent feature goals. Such tactics will never fundamentally resolve the

problem or support rapid development, since the problem is structural—baked into the organization, built into the system.

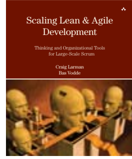
More Poor Code/Design

see Design in companion

Perhaps the greatest irony of component teams is this: *A mistaken belief behind their creation is that they yield components with good code/design*. Yet, over the years we have looked closely at the code across many large products, in many companies, and it is easy to see that the opposite is true—the code in a component maintained by a single-component team is often quite poor.[10] For example, we will sit to start pair programming with a component team member (who knows we'll be looking for great code), and with a slightly apologetically grin the programmer will say, "Yeah, we know it's messy, but *we* understand it." What is going on?

- **limited learning**—as discussed above, developers are not exposed to vast amounts of different code; this limits their learning of good design.
- **familiarity breeds obfuscation**—when I stare at the same complicated, obfuscated 10,000 lines of code month after month it starts to be familiar and 'clear'; I can no longer see how complicated it is, nor does it especially bother me, because of long exposure—so I am not motivated to deeply improve it.
- **obfuscation and duplication-heavy large code bases breed job security**—some *do* think like this, especially in groups where line management are not master programmers, not looking at the code, not encouraging great, refactored code.
- **no outside pressure to clarify, refactor, or provide many unit tests for the code**—no one other than the team of five component developers (who are long familiar with the complicated code) works on it; thus there is no pressure to continually refactor it, reduce coupling, and surround it with many unit tests so that it is clear and robustly testable for other people to work on.

10. New developers joining an existing component team (i.e., component) also report this observation.



The perpetuation of belief that component teams create great code is an indicator of a lack of "Go See" behavior by first-level management. If they were master developers (lean principle "my manager can do my job better than me") and regularly looking in depth across the code base, they would see that on average, *more—not less—fresh eyes on the code makes it better.*

Summary of Disadvantages

- promotes sequential life cycle development and mindset
- limits learning by people working only on the same components for a long time—the waste of underutilized people
- encourages doing easier work rather than most valuable work
- promotes some component teams to do "artificial work"
- causes long delays due to major waiting and handoff wastes
- encourages code duplication
- unnecessarily promotes an ever-growing number of developers
- complicates planning and synchronization
- increases bottlenecks—single points of success are also single points of failure
- fosters more poor code/design

Platform Groups—Large-Scale Component Groups

In large product organizations, there often exist one or more lower-level platform groups distinct from higher-level product groups. For example, in one client's radio networks division a platform group of hundreds of people provides a common platform to several market-visible products (each involving hundreds of people). Note that the platform group and a higher-level product group that uses it are essentially two very large component groups. There is no absolute constraint that a separate platform group must exist; for example, the software technologies and deployment environment are the same in both layers. A higher-level developer could in theory modify code in the lower-level 'platform' code—the boundary is arbitrary.

So, the long-term organizational change toward feature teams, large-scale Scrum, and less handoff waste implies that an artificially constructed platform group may merge into the customer-product

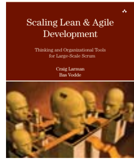
groups, with feature teams that work across all code. This is a multi-year learning journey.

TRY...FEATURE TEAMS

Most drawbacks of component teams can be resolved with feature teams (defined starting on p. 150). They enable us to put the requirements analysis, interaction design, planning, high-level design, programming, and system test responsibilities within the team[11], since they now have a whole end-to-end customer-feature focus. Planning, coordinating, and doing the work are greatly simplified. Handoff and delay wastes are dramatically reduced, leading to faster cycle time. Learning increases, and the organization can focus on truly high-priority market-valued features. And because multiple feature teams will work on shared components, sometimes at the same time, it is essential that the code is clean, constantly refactored, continually integrated, and surrounded by unit tests—as otherwise it won't be possible to work with.

> **Note a key insight**: Feature teams shift the *coordination challenge* between teams *away* from upfront requirements, design, and inter-team project management and *toward* coordination at the code level. To see this, compare Figure 7.7 and Figure 7.8. And with modern agile practices and tools, *coordinating at the code level is relatively easy.* Naturally, developers and managers unfamiliar with these practices don't know this, and so continue with upfront responses to the coordination challenge.

11. Ideally, customer documentation is also put within the team.



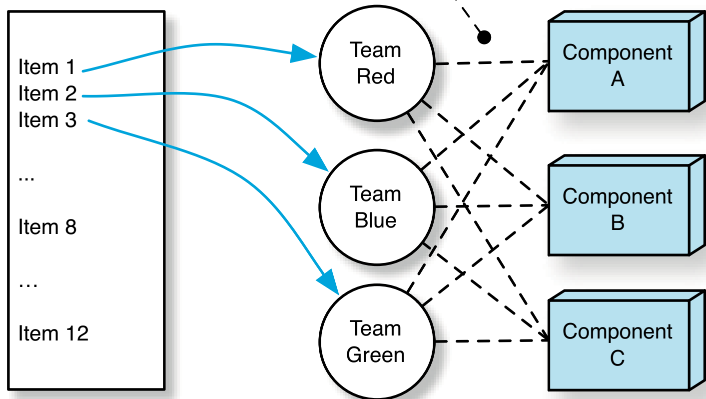


Figure 7.8 feature teams shift the coordination problem to shared code

As the shift to shared code coordination illustrates, a feature team organization introduces new issues. In traditional development these seemed difficult to solve. Fortunately, there are now solutions.

The following sections analyze these challenges and illustrate how modern agile development practices ameliorate them, thus enabling feature teams. Challenges or issues of feature teams include:

- broader skills and product knowledge
- concurrent access to code
- shared responsibility for design
- different mechanism to ensure product stability
- reuse and infrastructure work
- difficult-to-learn skills
- development and coordination of common functional (for example, test) skills that span members of many feature teams
- organizational structure
- defect handling

Feature Teams versus Feature Projects

Feature teams are not feature projects. A **feature project** is organized around one feature. At the start, the needed specialists (usually developers from component teams or a resource pool) are identified and organized into a short-lived group—a virtual **project group**. The specialists are usually allocated a percentage of their time to work for the feature project. Feature teams and feature projects have important differences:

Long-life teams—A feature team, unlike a project group, may stay together for several years. The *team* has an opportunity to jell and learn to work together. A well-working jelled team leads to higher performance [KS93].

Shared ownership—In a feature team, the whole team is responsible for the whole feature. This leads to shared code ownership and cross-learning, which in the long run increases degrees of freedom and reduces bottlenecks. In feature projects, developers only update their particular single-specialty section of code.

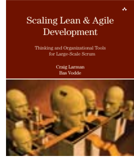
Stable, simple organizational structure—Feature teams offer a simple structure; they are the stable organizational units. Traditional project teams are ever-shifting and result in matrix organizations, which degrades productivity.

Self-managing; improved cost and efficiency—Feature teams (and Scrum) do not require overhead project managers, because coordination is trivial.

Broader Skills and Product Knowledge

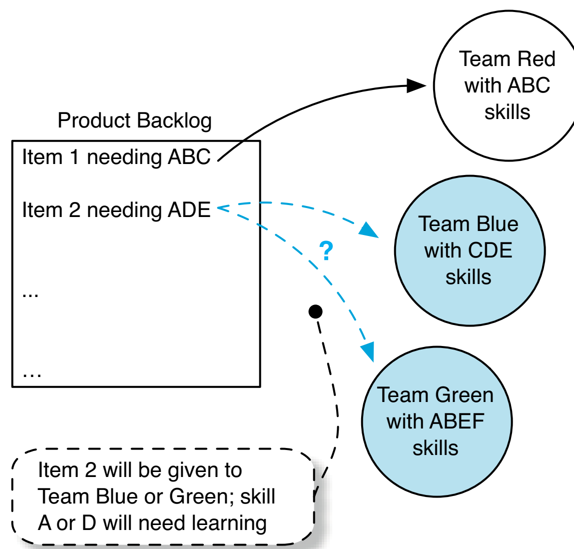
This is the opposite of the *limits learning* problem of component teams. The feature team needs to make changes in any part of the system when they are working on a customer feature.

First, not all people need to know the whole system and all skills. The Product Owner and teams usually select a qualified feature team for a feature, unless they have the time and desire for a 'virgin' team to invest in some deep learning in less familiar territory. In the common case, the team members together need to already know enough—or be able to learn enough without herculean effort—to complete a customer-centric feature. Notice that feature teams *do*



have specialized knowledge—that's good. And, since learning is possible, they are slowly extending their specializations over time as they take on features that require moderate new learning, strengthening the *system* of development over time (see Figure 7.9). This is enhanced by more pair-work and group-work in a team with various skills. We move beyond false dichotomies such as "specialization good, learning new areas bad" and "generalists good, specialists bad."

Figure 7.9 specialization is good, learning is good



Learning new areas of the code base is not a profound problem for "moderately large" products, but beyond some tipping point[12] it starts to be a challenge.

One solution is **requirement areas**. In traditional large product development, component teams are usually grouped within a major subsystem department. Similarly, when scaling the feature team organization, we can group feature teams within a **requirement area**—a broad area of related customer requirements such as "network performance monitoring and tuning" or "PDF features." To

requirement areas p. 218

12. It depends on size, quality of code, and unit tests, …

clarify: A requirement area is not a subsystem or architectural module; it is a domain of related requirements from the customer perspective.

What's the advantage? Most often, a requirement-area feature team will *not* need to know the entire code base, since features in one area usually focus across a semi-predictable subset of the code base. Not always, but enough to reduce the scope of learning. Requirement-area feature teams provide the advantage of feature teams without the overwhelming learning challenge of a massive code base.[13]

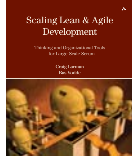
But stepping back from the 'problem' of requiring broader knowledge: Is it a problem to avoid, or an opportunity to go faster?

A traditional assumption underlying this issue is the notion that assigning the existing best specialist for a task leads to better performance. Yet this is an example of *local optimization* thinking—no doubt locally and short-term it seems faster for code generation, but does it increase *long-term* systems improvement and throughput of highest market-valued features? In addition to the obvious bottlenecking it promotes (thus slowing throughput of a complete feature), does it make the organization as a whole speed up over time? As previously explored in the section on the disadvantages of component teams:

> *Product groups that repeatedly rely on single-skill specialists are limiting learning, reducing degrees of freedom, increasing bottlenecks, and creating single points of success—and failure. That does not improve long-term system throughput of highest market-valued features or the ability to change quickly.*

There is an assumption underlying concerns about broader product knowledge: The assumption is that it will take a *really* long time for a developer to learn a new area of the code base. And yet, in large product groups, it is not uncommon for an existing developer to move to a different component team and within four or five months be comfortable—even shorter if the code is clean. It isn't trivial, but neither is it a herculean feat. Programmers regularly learn to work

13. A requirement-area feature team may eventually move to a new area; we haven't seen that yet.



on new code and in new domains all the time; indeed, it's an emphasis of their university education.

Still, to dig deeper: Why is it hard to learn new areas of the code base? Usually, the symptom is incredibly messy code and bad design—a lack of good abstraction, encapsulation, constant refactoring, automated unit tests, and so forth. That should be a warning sign to increase refactoring, unit tests, and learning, not to avoid new people touching fragile code—to strengthen rather than to live with a weakness.

Learning new code and changing an existing code base is indeed a *learnable skill*. It takes practice to become good, and people in feature teams get that practice and learn this skill.

potential skills p. 207

Returning to the apparent quick-fix, short-term performance advantage of choosing the best existing specialist for a task, this "common sense" has also been questioned in a study [Belshee05a].

Development ran with one-week iterations. Each iteration the team experimented with new practices. One experiment involved task selection. A traditional approach may be called *most qualified implementer*—the specialist who knows most about a task works on it. The team experimented with a task selection method called *least qualified implementer*—everyone selects the task they know least about. Also, task selection was combined with frequent pair switching, called **promiscuous pairing**, each 90 minutes. First, the initial velocity did not drop significantly. Second, after two iterations (two weeks) the velocity increased *above their previous level*. The benefit of increased learning eventually paid off.

Belshee explains the above result with a concept called *beginner's mind*. "*Beginner's Mind happens when the thinker is unsure of his boundaries. The thinker opens himself up and thoroughly tests his environment… The whole mind just opens up to learning.*" [Belshee05b]

An experience report from Microsoft related to these practices:

> *The principles laid out in Belshee's paper are not for the faint of heart. They require dedication, commitment and courage. Dedication is required of each team member to strive for self*

improvement. Commitment is needed for each team member to ensure the values and principles will be followed and the team will hold itself accountable. Courage, because the emotions that Promiscuous Pairing invites will be not unlike the most fun and scariest roller-coaster ever experienced. [Lacey06]

The studies illustrates the potential for acceleration when an organization invests in broadening learning and skill, rather than limiting it through dependence on bottlenecks of single-specialists.

Concurrent Access to Code

As illustrated in Figure 7.8, one important difference between component teams and feature teams is that the dependency and coordination between teams shifts from requirements and design to code. Several people may concurrently edit the same source code file, typically a set of C functions, or a class in C++ or Java.

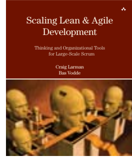
With weak or complex version-control tools and practices, common in the 1980s and still promoted by companies such as IBM, this was a concern. Fortunately, it isn't an issue with modern free tools and agile practices.

see Continuous Integration in companion

Old-generation and complex (and costly) version control systems such as ClearCase defaulted to **strict locking** in which the person making a change locked the source file so that no one else could change it. Much worse, vendors promoted a culture of avoiding concurrent access, delaying integration, complex integration processes involving manual steps, integration managers, and tool administrators. This increased costs, complexity, bottlenecks, waiting, and reinforced single-team component ownership.

On the other hand, the practices and tools in agile and in open source development are faster and simpler. Free open source tools such as Subversion[14] default to **optimistic locking** (no locking),

14. Subversion is likely the most popular version control tool worldwide, and a de facto standard among agile organizations. *Tip*: It is no longer necessary to pay for tools for robust large-scale development; for example, we've seen Subversion used successfully on a 500-person multisite product that spanned Asia and Europe.



and more deeply, have always encouraged—through teaching and features—a culture of simplicity, shared code ownership, and concurrent access [Mason05]. With optimistic locking anyone can change a source file concurrently. When a developer integrates her code, Subversion automatically highlights and merges non-conflicting changes, and detects if conflicts exist. If so, the tool easily allows developers to see, merge, and resolve them.

An optimistic-locking, fast, simple tool and process are required when working in an agile development environment and are a key in eliminating problems related to concurrent access to code.

Optimistic locking could in theory lead to developers spending inordinate time merging difficult changes and resolving large conflicts. But this is resolved with **continuous integration** and **test-driven development**, key practices in scaling agile and lean development.

When developers practice **continuous integration** (CI) they integrate their code frequently—*at least* twice a day. Integrations are small and frequent (for example, five lines of code every two hours) rather than hundreds or thousands of lines of code merged after days or weeks. The chance of conflict is lower, as is the effort to resolve. Developers do *not* normally keep code on separate "developer branches" or "feature branches"; rather, they frequently integrate all code on the 'trunk' of the version-control system, and *minimize* branching. Furthermore, CI includes an automated build environment in which *all* code from the hundreds of developers is endlessly, relentless compiled, linked, and validated against thousands of automated tests; this happens many times each day.[15]

see Continuous Integration in companion

parallel releases p. 210

In **test-driven development** every function has many automated micro-level unit tests, and all programming starts with writing a new unit test before writing the code to be tested. Further, every feature has dozens or hundreds of automated high-level tests. This leads to thousands of automated tests that the developer can rerun locally after each merge step—in addition to their continual execution in the automated build system.

see Test in companion

In lean thinking terminology, CI replaces big batches and long cycle times of integration (the practice of traditional configuration man-

15. Note that this implies driving down the build time of a large system.

agement) with *small batches and short cycles* of integration—a repeating lean theme.

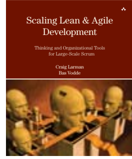
Shared Responsibility for Design

In a traditional component team structure, each component has an owner who is responsible for its design and ongoing "conceptual integrity." On the other hand, feature teams result in shared ownership. This could—without the practices used in agile methods—lead to a degradation of integrity. All that said, it must be stressed that in reality, code/design degradation happens in many groups anyway, regardless of structure; recall the reasons component teams ironically often live with obfuscated code (p. 173).

Continuous integration (CI) implies *growing* a system in small steps—each meant to improve the system a little. In addition to integration of all code on the trunk multiple times daily and non-stop automated builds running thousands of automated tests, CI with ongoing design improvement is supported by other practices:

see Design in companion

- **evolutionary design culture**—since (as Conway points out) the initial design vision is rarely great, and in any event since software is ever-changing, encourage a culture in which people *view the design or architecture as a living thing* that needs never-ending incremental refinement
  - a sequential life cycle with a single upfront architectural or design phase gives the false message that the design is something we define and build *once*, rather then continually refine every day for the life of the system
- **test-driven development**—drive code development with automated micro-unit tests and higher-level tests; each test drives a small increment of functionality
  - this leads to hundreds of thousands of automated tests
- **refactoring**; a *key step*—after each micro-change of a new unit test and related solution code, perform a small refactoring step to improve the code/design quality (remove duplication, increase encapsulation, …)



– refactoring implies always leaving the code a little better than we found it

– note that *design quality means code quality*; there is no real 'design' in software other than the source code [Reeves92]

These CI practices support continuous design improvement with feature teams, and the 9th agile principle: *Continuous attention to technical excellence and good design enhances agility.* Plus, there are strong connections between these agile practices and the lean principles *Stop and Fix*, *Continuous Improvement*, and the kaizen practice of endless and relentless small steps of improvement—in this case, "*kaizen in code.*"

be agile p. 140

Successfully moving from solo to shared code ownership supported by agile practices doesn't happen overnight. The practice of **component guardians** can help. Super-fragile components (for which there is concern[16]) have a component guardian whose role is to teach others about the component ensures that the changes in it are skillful, and help remove the fragility. She is *not the owner* of the component; changes are made by feature team members. A novice person (or team) to the component asks the component guardian to teach him and help make changes, probably in design workshops and through pair programming. The guardian can also code-review all changes using a 'diff' tool that automatically sends her e-mail of changes. This role is somewhat similar to the **committer** role in open source development.[17] It is another example of the lean practices of regular mentoring from seniors and of increasing learning.

Another possible practice is establishing an **architecture code police** [OK99]; to quote, "*The architecture police is responsible for keeping a close check on the architecture.*" Note that since the only real design is in the code, architecture code police are responsible for continually looking at the *code* (not at documents), identifying weaknesses, and coaching others while programming—they are master-

16. A typical reason for concern about delicate components is that the code is not clean, well refactored, and surrounded by many unit tests. The solution is to clean it up ("Stop and Fix"), after which a component guardian may not be necessary.
17. But the roles are not identical. Guardians (or 'stewards') do more teaching and pair programming, and allow commits at any time. Committers also teach, but less so, and control the commit of code.

programmer teachers. Architecture code police are a variant of component guardians; they are responsible for overall code quality. But no single person is responsible for a specific component. Warning: This practice could devolve into a separate "PowerPoint architects" group that is not focussed on the code, and not teaching through pair work.

A related practice is used at Planon, a Dutch company building workplace management solutions. The co-creator of Scrum, Jeff Sutherland, wrote: *"We have another Scrum company that has hit Gartner Group's magic [leaders]"* [Sutherland07]. They have multiple feature teams, each consisting of an architect, developers, testers, and documentation people. There is also one lead architect, but he is *not* responsible for defining the architecture and handing it over to the team. Instead, he is "*the initiator of a professional circle, that includes all architects, to keep the cross-team communication going.*" Planon's term *professional circle* is a **community of practice**, in which people with similar interest form a community to share experiences, guide, and learn from each other [Wenger98, WMS02]. At Planon, they have a community of practice for different specialists such as architects, testers, and ScrumMasters [Smeets07].

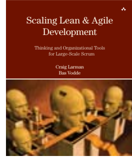
community of practice p. 253

see Design in companion

Another practice to foster successful shared design is the **design workshop**. Each iteration, perhaps multiple times, the feature team gets together for between "two hours and two days" around giant whiteboard spaces. They do collaborative **agile modeling**, sketching on the walls in a creative design conversation. If there are component guardians or other technical leaders (that are not part of the feature team) who can help guide and review the agile modeling, they ideally also participate. See Figure 7.10.

For broad architectural issues **joint design workshops** (held repeatedly) can help. Interested representatives from different feature teams (not restricted to 'official' architects) spend time together at the whiteboards for large-scale and common infrastructure design.[18] Participants return to their feature team, teaching joint insights in their local workshops and while pair programming.

18. Solutions for multisite joint design workshops are explored in the *Design* chapter.



Handoff and partially done work (such as design specifications) are wastes in lean thinking. To reduce this and to encourage a culture of teaching, it is desirable that design leaders not be members of a separate group that create specifications, but rather be full-time members on a feature team who also participate in joint design workshops as a *part-time architectural community of practice*.

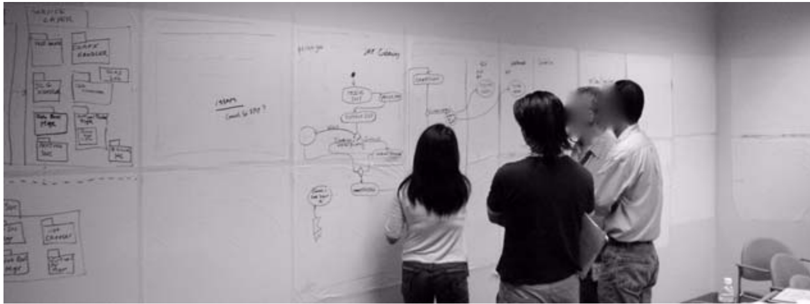


Figure 7.10 design workshop with agile modeling

New Mechanisms for Code Stability

Code stability in a component team organization is attempted with component owners. They implement their portion of a customer feature in their component, hopefully keeping it stable. Note that stability is an ideal rather than an assured consequence of this approach. It is common to find large product groups where the build frequently breaks—often as a consequence of the many coordination problems inherent to and between component teams.[19]

With feature teams, new—and just plain better—stability techniques are used. Massive test automation with continuous integration (CI) is a key practice. When developers implement new functionality, they write automated tests that are added to the CI system and run constantly. When a test breaks:

see Test and Continuous Integration in companion

1. The CI system automatically (for example, via e-mail or SMS) informs the set of people who might have broken the build.

19. We have seen many examples of a three-month or worse ‘stabilization’ phase in traditional large products that used component teams.

2. Upon notification, one or more of these people stop, investigate, and bring the build back to stability.
   - this CI attitude illustrates the lean principle of *Stop and Fix*

Infrastructure and Reuse Work

In a component team organization, goals such as a reusable framework or improving test automation are usually met by formation of a temporary project group or with an existing component team.

In a feature team organization with Scrum, these major goals are added to the Product Backlog—an exception to the guideline to focus on adding customer-feature items, since these goals span all features.

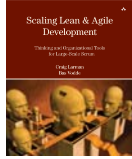
This backlog infrastructure work is prioritized by the Product Owner in collaboration with the teams. Then the infrastructure work is *given to an existing feature team*, as any other backlog item. This team works on infrastructure for a few iterations (delivering incremental results each iteration) and thus may be called an **infrastructure team**, a temporary role until they return to normal feature team responsibility.

Difficult-to-Learn Skills

potential skill p. 207

A feature team may not have mastery of all skills needed to finish a feature. This is a solvable problem if there is the **potential skill** [KS01]. On the other hand, some skills are really tough to learn, such as graphic art or specialized color mathematics. Solutions:

- **fixed specialist for the iteration**—This creates a constraint in the iteration planning; all work related to that skill needs to be done by the feature team with the specialist (who may be a permanent or temporary visiting member).
  - A good Stop and Fix approach to working with the specialist is that he is a teacher and reviewer, not a doer
- **roaming specialist**—During the iteration planning several teams request help from a specialist; she schedules which teams she will work with (and coach) and roams between them.



- **visit the specialist at her primary team**—the specialist physically stays with one feature team that needs her most (for the iteration) and invites other people to visit her for mini-design workshops, review, and consultation.

Solo specialists are bottlenecks; avoid these solutions unless team learning is not an option. Encourage specialists to coach, not do.

Coordinating Functional Skills: Communities of Practice

An old issue in cross-functional teams is the development and coordination of functional skills and issues across the teams, such as testing skills or architectural issues. The classic solution, previously introduced, is to support **communities of practice** (COP) [Wenger98, WMS02]. For example, there can be a COP leader for the test discipline that coordinates education and resolution of common issues across the testers who are full-time members of different feature teams and part-time members of a common testing COP.

communities of practice p. 253

Organizational Structure

In a component- and functional-team (for example, test team) organization, members typically report to component and functional managers (for example, the "testing manager"). What is the management structure in an agile-oriented enterprise of cross-functional, cross-component feature teams?

In an agile enterprise, several feature teams can report to a common feature team's line manager. The developers and testers on the team report to the same person. Note that this person is not a project manager, because in Scrum and other agile methods, teams are self-managing with respect to project work (11th agile principle).

organizational structure p. 242

Handling Defects

In a traditional component team structure, the team is usually given responsibility for handling defects related to their component. Note that this inhibits long-term systems improvement and throughput by increasing interrupt-driven multitasking (reducing productivity)

for the team, and by avoiding learning and reinforcing the weakness and bottleneck of depending upon single points of success or failure.

On a large product with (for example) 50 feature teams, an alternative that our clients have found useful is to have a rotating maintenance (defect) group. Each iteration, a certain number of feature teams move into the role of maintenance group. At the end of the two or three iterations, they revert to feature teams doing new features, and other feature teams move into maintenance. Lingering defects that aren't resolved by the timebox boundary are carried back to the feature team role and wrapped up before new feature work is done.

As an additional learning mechanism, consider adding the practice of handling defects with pair programming, pairing someone who knows more and someone who knows less, to increase skills transfer.

Transition

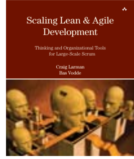
In his report on feature teams in Ericsson [KAL00], Karlsson observed, *"Implementing daily build and feature teams in an organization with a strong [traditional] development process, distributed development and a tradition of module [single component] responsibility is not an easy task."* It takes hard work and management commitment.

There are several tactics for transitioning to feature teams:

- reorganize into broad cross-component feature teams
- gradually expand team responsibility

Reorganize into Broad Cross-Component Feature Teams

One change tactic is to reorganize so that, collectively, the new teams have knowledge of most of the system. How? By grouping different specialists from most component areas (Figure 7.11).



A variation is that a new team is formed more narrowly with specialists from the *subset* of most components typically used in one (customer) **requirements area**, such as “PDF printing.” This approach exploits the fact that there is a semi-predictable subset of components related to one requirements area. It is simpler to achieve and reduces the learning burden on team members.

requirement areas p. 218

When one product at Xerox made the transition to feature teams, it started out by forming larger (eleven- or twelve-member) teams than the recommended Scrum average of seven. The advantage was that a sufficiently broad cross-section of specialists was brought together into feature teams capable of handling most features. The disadvantage was that twelve members is an unwieldy size for creating a single jelled team with common purpose.

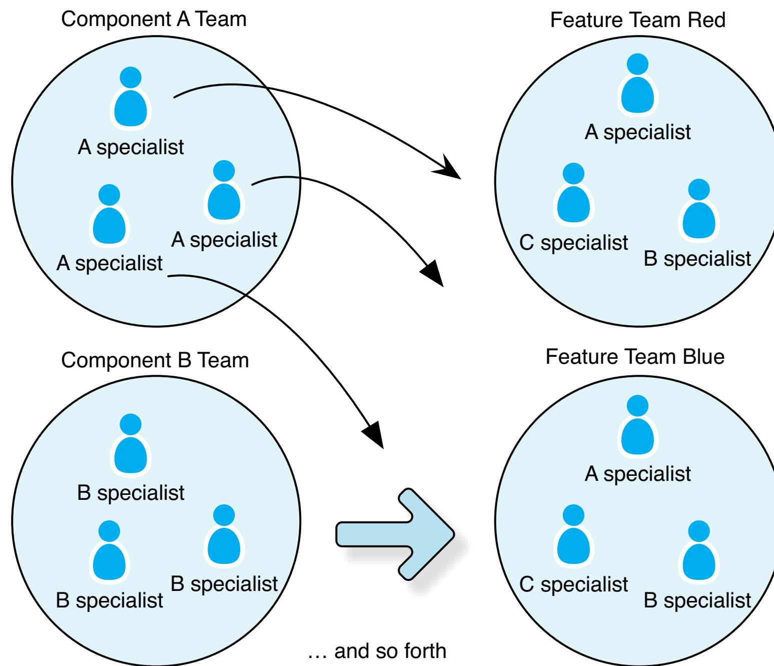


Figure 7.11 moving to feature teams

Gradually Expand Teams' Responsibility

For some, reorganizing to full-feature teams is considered too difficult, although in fact the impediments are often mindset and political will. As an alternative, take smaller steps to gradually expand teams' responsibility from component to "multi-component" teams to true feature teams.

Simplified example: Suppose an organization has four component teams A, B, C, and D. Create two AB teams and two CD teams from the original four groups, slowly broadening the responsibilities of the teams, and increasing cross-component learning. A customer feature will still need to be split across more flexible "multi-component" teams, but things are a little better. Eight months later, the two AB and two CD teams can be reformed into four ABCD teams… and so on.

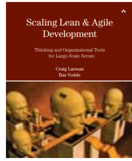
One Nokia product took this path, and formed AB teams based on the guideline of combining from closely interacting components; that is, they chose A and B components (and thus teams) that directly interacted with each other. Consequently, the original team A and team B developers already had some familiarity with each other's components, at least in terms of interfaces and responsibilities.

Conclusion

Why a detailed justification toward feature teams and away from single-function teams and component teams? The latter approach is endemic in large-product development. The transition from component to feature teams is a profound shift in structure and mindset, yet of vital importance to scaling agile methods, increasing learning, being agile, and improving competitiveness and time to market.

Recommended Readings

- *Dynamics of SoftwareDevelopment* by Jim McCarthy. Originally published in 1995 but republished in 2008. Jim's book is a true classic on software development. Already in 1995 it



emphasized feature teams. The rest of the book is stuffed with insightful tips related to software development.

- "XP and Large Distributed Software Projects" by Karlsson and Andersson. This early large-scale agile development article is published in *Extreme Programming Perspectives*. It is a insightful and much under-appreciated article describing the strong relationship between feature teams and continuous integration.
- "How Do Committees Invent?" by Mel Conway. This 40-year article is as insightful today as it was 40 years ago. It is available via the authors website at www.melconway.com.
- *Agile Software Development in the Large* by Jutta Eckstein. This is the first book published on the topic of scaling agile development. It describes the experience of a medium-sized (around 100 people) project and stresses the importance of feature teams in large-scale development.
- "Promiscuous Pairing and Beginner's Mind" by Arlo Belshee. This article is not directly related to feature teams or large-scale development but it does contain some facinating experiments that question some of the assumptions behind specialization.