## You Won't Change Your Organization Without An Optimization Goal

A distinguishing feature of Craig Larman's work (e.g. *Scaling Lean & Agile Development*) is the explicit focus on a system optimization goal for a change initiative. Here's an example system optimization goal that we consider consistent with Agility:

> Increase an organization's ability to respond to change.

Craig Larman clarifies:

> *tis true that the org design from LeSS is consistent with broad adaptiveness, but… suggest that we coach that adaptiveness is not for its own sake. rather, adaptiveness is in the service of something else: learning towards the discovery and delivery of high value or high impact, in the context of a world in which it's usually difficult to a priori know for sure what is high value/impact, and a world in which competition does stuff, new technologies emerge, new trends emerge, etc.*

So, less succinctly:

> Increase an organization's ability to discover and deliver the highest value in a world where we don't know everything, and everything's changing.

I'm not seeing any consistent optimization goal in SAFe, Scrum@Scale, the "Spotify model" and the way some people explain basic Scrum.

### Why not just "Make Everything Better"?

*Better* in some ways is *worse* in others. For example, the goal of *increased customer satisfaction* could be inconsistent with *increased stock price this quarter*. Or another example, I heard a project manager turned Scrum trainer say "In my experience, Scrum of Scrums works great!" And I can see how that could be true, if we're optimizing for the sort of problems project managers are usually asked to solve. But if we're doing Scrum to increase agility, we'll want to consider some better alternatives.

## What happens without an optimization goal?

I spent a little some time with a company which I initially thought was a perfect candidate for an Agile adoption. They had less than 100 people in the company, all co-located on the same floor of their hip, modern office. Their management initially seemed quite gung ho. But as the discussions progressed, it became more clear that this management did not *want* to untangle the byzantine organizational structure and the overspecialization that had built up over the past 20 years, didn't think people could learn new skills, and really felt it was best to micromanage employees.

When the company attempted an Overall Retrospective (https://less.works/less/framework/overall-retrospective.html), their actions were to *increase* the organizational complexity that was at the root of their problems! Local optimization bias is so powerful that doing retrospectives blindly can actually make things worse if we are not clear about our optimization goal.

At another similarly-sized company I worked with, the CEO himself came to our mob programming training sessions to see the company's code for himself, and suggest ways of adding automated tests. (This is similar to Ahmad Fahmy's Gemba Sprint (https://www.infoq.com/articles/guide-gemba-sprint/)) This sent everyone a clear message that it's often appropriate to *stop and fix*, rather than continuing to add bugs by churning out crap code.

If management cannot express a clear optimization goal and act consistently with it, perhaps we're dealing with too low a level of management.

## Does the change initiative have consistent goals?

Here's a list of other optimization goals[1] that may exist in your organization. They may be explicit or implicit. Some may be consistent with each other in your situation, and others may not:

- increased release frequency
- fewer defects in each release
- predictability
- clarity about who does what
- resource utilization (keeping people busy)
- ideation (creating ideas)
- typing code faster
- secrecy[2]
- comfort

- employee satisfaction/retention (break this down into different categories of employees: coders, line managers, department heads, etc.)
- customer satisfaction
- appearance that the change initiative has succeeded
- increase top-line revenue
- maintaining lead or monopoly
- reduce cost per developer
- product versatility
- broaden/diversify customer base
- preserve status quo
- compliance with rules/regulations
- lead the market by disruption
- increase return on investment
- survival
- conformance to a plan
- rate of developing features immediately
- rate of developing features within this quarter
- rate of developing features next year


1. Adapted from a list Viktor Grgic assembled. ↩

2. Yes, I have seen this as an *implicit* goal that was surfaced during training. ↩

# How is Knowledge Work Different? (Why Johnny, Inc. Can't Code.)

## The fundamental constraint of knowledge work – such as software development – is our ability to discover and create knowledge.

Imagine that you had spent Monday through Thursday working on an essay, a song, or even a computer program. You tore up seven ways that you realized wouldn't work. You took a walk outside. You slept on it. You did research and found an eighth way that looked like it would work, until you showed it to a friend who pointed out a serious flaw with one part of your idea. By Thursday afternoon you had nearly finished a ninth way that combined the best of your previous approaches and looked to be nearly done. Then Thursday night your cat walked across the keyboard and somehow deleted everything you typed that week.


Would it take you another four days to get back to where you were? Of course not! By 10AM Friday you'd be ahead of where you were Thursday night. Therefore what was the actual constraint on that knowledge work? It wasn't the typing. It was the learning and creation of new knowledge.

---

## Why Johnny, Inc. Can't Write Software

Have you noticed that companies with lots of money and people can't seem to develop software that works very well? I know interns who can make better websites than certain airlines can. Those shops aren't built for learning. They're organized as if software development is just about typing code. When that fails, they try to fix it with harmful measures: adding incentives and "accountability", creating new roles and departments, hiring even more people to type code.

#### Hint: Johnny, Inc. Is Not "Understaffed"

I recently visited a big bureaucratic agency that had trouble doing things that are easy for a small co-located team using modern development practices. I almost fell over when I heard one of the dozen project managers say "We're understaffed." No … that's not the problem.

## Implications

What would be different about an organization that recognizes that better software is really about our ability to discover and create knowledge?

| Traditional Organization | Continuous Learning Organization |
|---|---|
| people work in separate cubicles, offices, or cities | each small team works at the same table |
| more people | fewer people |
| more roles | fewer roles |
| more departments | fewer departments |
| single-skilled workers | multi-skilled workers |
| fewer people learn from customers and end users (https://www.youtube.com/watch?v=RAY27NU1Jog) | many people learn from customers and end users |
| private code (my code/your code) practices or policies | internal open-source practices |
| people spend time reading/writing “tickets” | people spend time sharing screens and whiteboards |
| focus on execution, quantity, and velocity | focus on impact |
| learning happens only during brief training periods or on employee’s own time | learning happens every day on company time |
| mistakes are not survivable | experimentation is encouraged |
| retrospectives affect teams only | retrospectives cause improvements to company policies and structure (https://less.works/less/framework/overall-retrospective.html) |
| managers coordinate, reallocate, motivate, and mediate | managers are capability builders (https://less.works/less/management/role-of-manager.html) |
| extended overtime | adequate sleep |

Becoming a *learning organization* isn’t something that gets solved once, by a training program, a reorg, or a one shot “Agile transformation.” It doesn’t happen just by talking about organizational culture. It is a deep change that cannot ignore the policies and structure of the organization.

Japanese version: 知識創造を行う仕事は、どうしてこうも異なるのか？（なぜあの会社はコーディングができないのか） (https://scrummaster.jp/how-is-knowledge-work-different-jp/)

## My Code, Your Code. Private code policies considered harmful to agility.

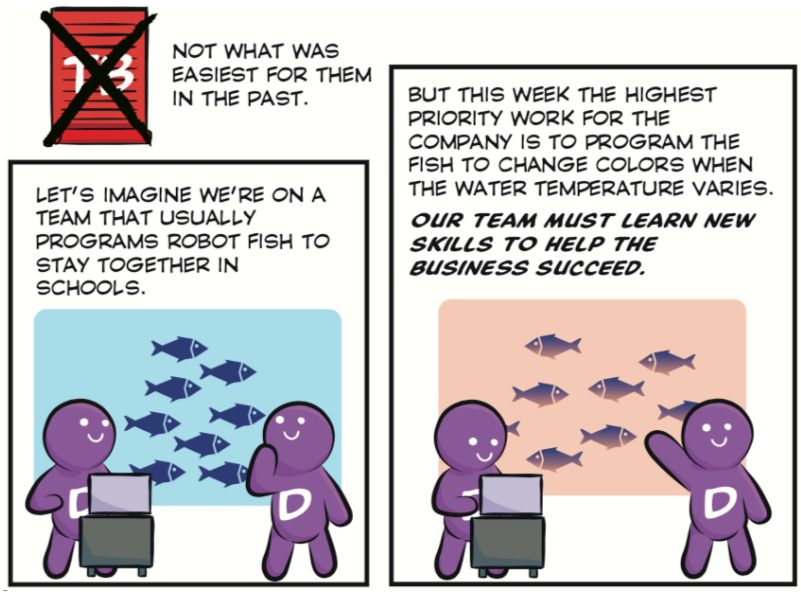

Watch out for *private code policies and practices*. Tying particular people or teams to particular parts of the code has several negative impacts:

1. Teams don't work in priority order. Some teams or individuals do lower valued work, creating *inventory*. Other teams or individuals become bottlenecks, causing *Work In Progress* (WIP) queues and delaying end-to-end cycle time.

2. In dark corners, our code starts to stink. Fewer people see it, write tests for it, and refactor[1] it. A lack of automated tests makes *continuous integration* more difficult and *continuous delivery* impossible. The lack of automated tests makes it even scarier for other people to change the code (for fear of introducing regression failures). The lack of refactoring makes the code difficult to understand and change. If sunlight is the best disinfectant, private code practices are the best incubator for code rot. Taken together, these things discourage other people from working on the code, which makes it stink even more.
3. Our knowledge and skills do not improve as much as they would when we see other parts of the system. In our little bubble, we have no idea how much else there is to learn about software development.
4. Ultimately the customer suffers and our company becomes less competitive.

## Possible Remedies

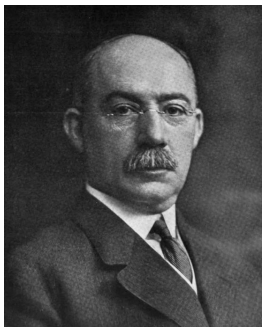
Craig Larman and Bas Vodde have written the most comprehensive list of remedies to the My Code, Your Code problem that I've seen: https://less.works/less/technical-excellence/architecture-design.html#Behavior-OrientedTips (https://less.works/less/technical-excellence/architecture-design.html#Behavior-OrientedTips).



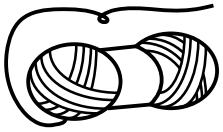

1. The purpose of source code is not only to communicate with the computer running the program. Source code should also communicate with the people working on it, now and in the future. As properly defined by Martin Fowler, to *refactor* means to improve the design of existing code, generally in tiny incremental steps, to keep it understandable and maintainable, without changing behavior. Refactoring is not practical unless we're writing automated tests as we write the code. ↩

# Large Organization Software Development Misconception #1: Are "Dependencies" In Knowledge Work Caused By Immutable Laws Of Physics?

## Background



Just before World War I, a colleague of Frederick Taylor named HL Gantt promoted the use of a project schedule chart highlighting activities that depend on each other. This can be useful in manufacturing, construction, logistics, and other fields where the dependencies are imposed by physical reality. (Interesting trivia: the Soviet Union tried to use a Gantt chart for it's first five-year plan.)



Unfortunately, people who haven't discovered that knowledge work is different from traditional work misapply these techniques to software development. Software development companies steeped in the project management mindset sometimes try to plan around perceived dependencies with traditional Gantt charts, or sometimes by modeling them with red yarn. The yarn method is only superficially different than a 100 year old Gantt chart, yet it's still being sold as an "Agile" technique today!

## Physical analogies for software development (and other knowledge work) are mental traps

The following real quote reveals the mindset of a project manager with tools to manage dependencies in physical work – where they actually exist – but has not written software in a modern way:

> *People who think that dependencies are just figments of old-style thinking should convince me by first putting on a shoe and tying the laces, then putting on a sock that goes between their bare foot and the tied shoe without removing the shoe.*

## In software development, asynchronous “dependencies” are not caused by immutable laws of physics!

The illusion that we need to plan around “dependencies” through multiple Sprints is a symptom of obsolete organizational structure, policy, and skills. We gain agility by directly addressing those problems rather than institutionalizing them with Gantt charts or red yarn. The socks and shoes argument inadvertently demonstrates that “dependencies” in software development *are* figments of old-style thinking.

Ten years ago I’d usually hear the same misconception expressed as a *house analogy*: “If you want to build a house, you have to build the foundation first. You can’t start with the roof.” And this may be true … about houses.

## The fundamental constraint of knowledge work – such as software development – is our ability to discover and create knowledge.

Imagine that you had spent Monday through Thursday working on an essay, a song, or even a computer program. You tore up seven ways that you realized wouldn’t work. You took a walk outside. You slept on it. You did research and found an eighth way that looked like it would work, until you showed it to a friend who pointed out a serious flaw with one part of your idea. By Thursday afternoon you had nearly finished a ninth way that combined the best of your previous approaches and looked to be nearly done.

Then Thursday night your cat walked across the keyboard and erased everything you typed that week.

Would it take you another four days to get back to where you were? Of course not! By 10AM Friday you’d be ahead of where you were Thursday night. Therefore what was the actual constraint on that knowledge work? It wasn’t the typing. It was the learning and creation of new knowledge.

## Perceived “dependencies” in knowledge work are caused by knowledge gaps.

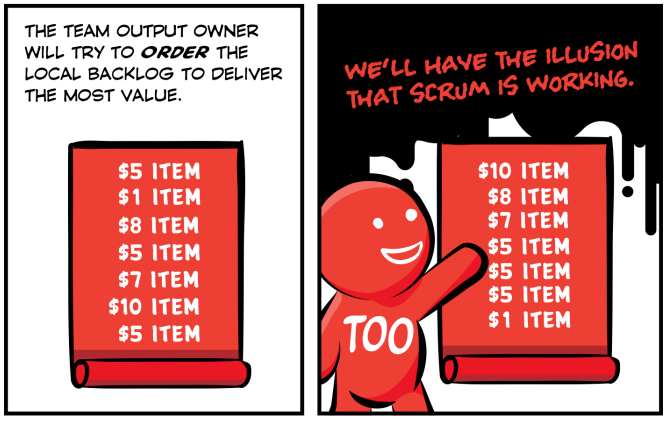
Once we internalize this realization, it leads to actions which were counterintuitive before – often the opposite actions that project management would advise. Unfortunately your organization’s structure and policies have actively encouraged these knowledge gaps.

A skillful software development organization allows the Product Owner to *prioritize* the Product Backlog from a business perspective, not just “order” it. Put away the red yarn.

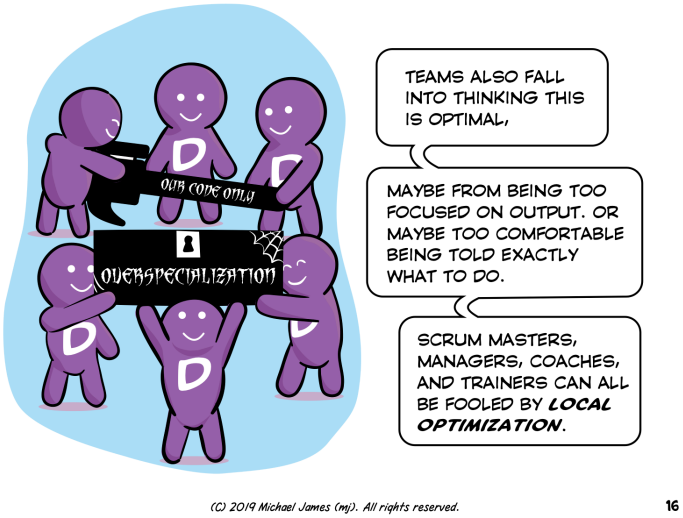
Japanese version (https://scrummaster.jp/misconception-1-dependencies-are-caused-by-immutable-laws-of-physics-jp/)

## Large Organization Software Development Misconception #2: Are All Teams Working On Equal Value Stuff?

If someone in an seven-team company had only a superficial understanding of Scrum, they might think each team should have its own segregated list of stuff to do. That keeps them "productive," right?

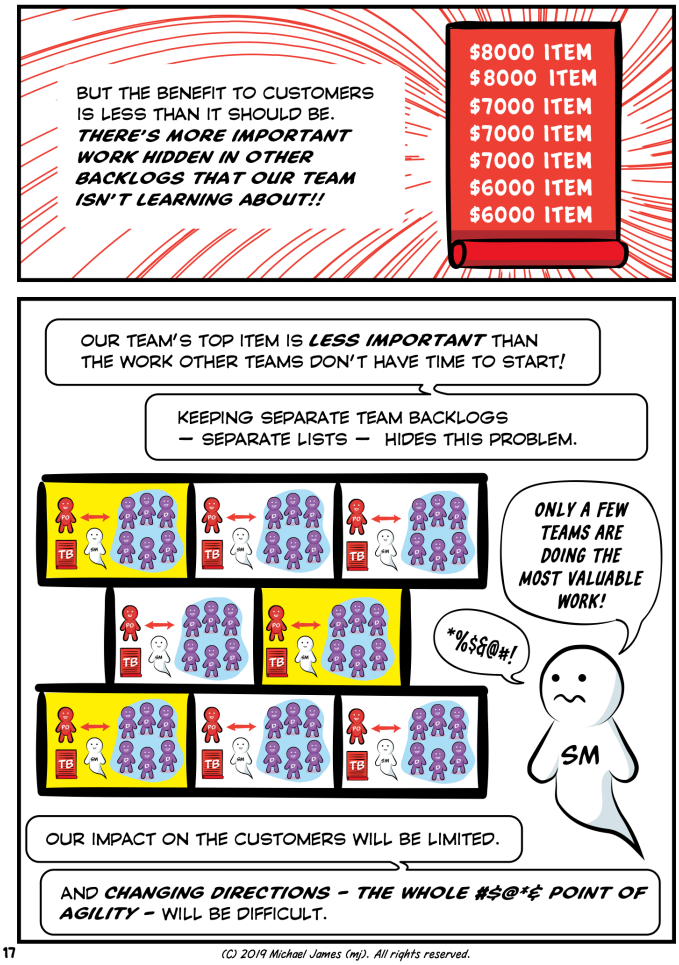



This is a great idea if your goal is to keep everyone busy. If your goal is to do the highest priority, highest value work, it could only be justified if several implausible things were true:




1. Each team's backlog would need to contain the *same priority work*, work of equal value. Team A's #1 item (A-1) would need to have the same customer impact as Team G's #1 (G-1) item. If you've spent much time with your eyes open in real organizations, you already know that one team's work will be more important in the big picture than another team's work. In real life it's not hard to imagine the value ratio exceeding 1000 to 1. The problem will be masked if we have single-function teams (e.g. design only, test only) or *component teams* (e.g. back end only) rather than feature teams (https://less.works/less/structure/feature-teams.html).
2. Each team's sub-list would have to *stay* the same priority over time. Even as we learned more about customer needs, we would not be able to reprioritize beyond the limited context of each team's list. We'd have to decide up-front, once and for all, the value of each team's type of work. This type of "Scrum" would work if we didn't need agility. Of course we could cheat and move items between Team C and Team D's lists, but that would be admitting that it's really just one list we've artificially segregated. (The more common form of crypto-reprioritization is to yank key people from one thing to another: *resource management*. This habit is so widespread, it should be a clue that superficial-Scrum actually *reduces* agility, and managers need a way around it!)

3. Each team would have to complete every item at the same rate. If Team A started #A-3 before Team B started #B-2, they would be working on the stuff we'd declared lower value at the expense of higher valued work. Have you ever gotten in line at a grocery store and noticed that latecomers in a different line finished before you?




These are the points that people who advocate the so-called "Spotify Model" (aka. Shiny Happy People Holding Hands) are usually missing: there's a difference between *feeling* productive, and actually doing the highest value work. Employee satisfaction is a potentially useful indicator. But by itself, employee satisfaction isn't evidence that the product development organization can learn and adapt to reality as it's discovered. A happy company that cannot pivot is not Agile.

There may be other justifications for keeping separate lists. Maybe they really really really are unrelated products, though we should be skeptical of this (https://less.works/less/framework/product.html). Maybe we're bound by overspecialization because the skills and knowledge gaps seem insurmountable. Maybe we've made a management decision to keep our less capable developers on Team F. Maybe Teams X and Y are on the other side of the world and we're not willing to give them anything important. Let's use Ken Schwaber's term for these things: *organizational impediments*.

Gentle reader, I have no idea what you should do in your own situation. I hope I've at least helped you gain clarity about why Scrum has only one Product Backlog per product, regardless of the number of teams.

See also: https://www.scrumwithstyle.com/ing-improves-on-the-so-called-spotify-model-using-less/ (https://www.scrumwithstyle.com/ing-improves-on-the-so-called-spotify-model-using-less/)

Japanese version: 大規模組織におけるソフトウェア開発の誤解その２:すべてのチームが等しい価値の業務に取り組んでいるか？ (https://scrummaster.jp/misconception-2-all-teams-are-working-on-equal-value-jp/)

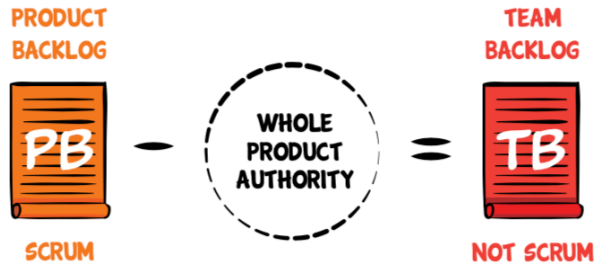
 

## Large Organization Software Development Misconception #3: Should Our Team See A Narrow View Or A Whole Product View?

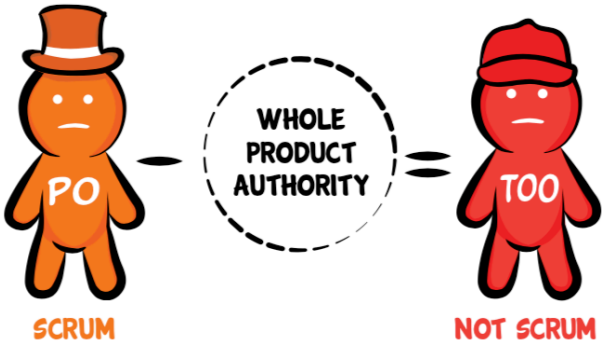
Small teams are great, and mandated by the definition of Scrum. Scrum also mandates a Product Backlog and Product Owner.

Most companies have more than 11 people working on any given product. *Just to make Scrum fit* (a local optimization), they will redefine "Product" to mean something narrower than the whole product. This is the first step into a house of mirrors where words mean the opposite of their original intent.

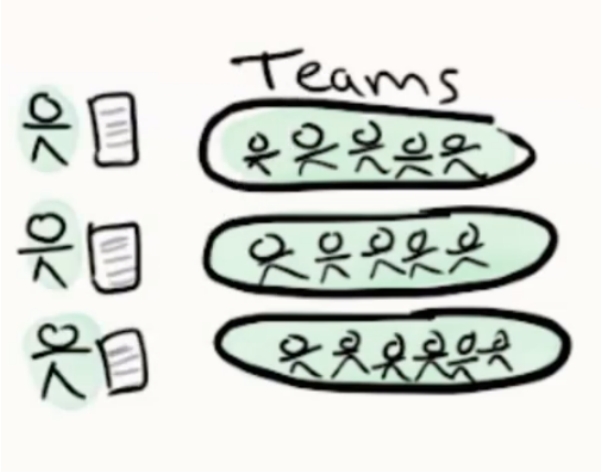
The next mistake is to give each team a "Product Backlog" which astute observers will notice is really only a *Team Backlog*.



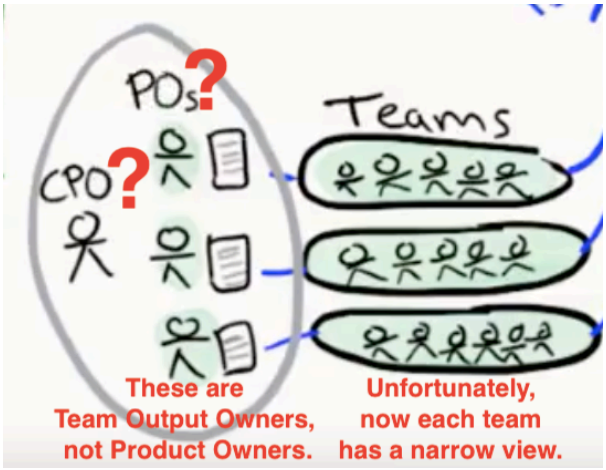

And then give each team a separate "Product Owner" who is really just a *Team Output Owner*.




This leaves us with multiple separate "Scrum Teams," each with a narrow view of the real product. Unfortunately this is recommended at the end of Henrik Kniberg's extremely popular Product Owner video (a great video up until the few minutes at the end).[1]




Managers worry these teams would run around like chickens with their heads cut off unless we pulled them together somehow. The traditional way of managing lots of people, dating back at least as far as Julius Caesar's army, is to add hierarchical layers. Humans have trouble thinking of alternatives to hierarchical layers. So now we get "Chief Product Owner" or "Business Owner" or some other way of filling the gap above the multiple Team Output Owners.




Calling someone a "Chief Product Owner" or "Business Owner" is a symptom of an unnecessary intermediate layer. The Scrum and LeSS term for the real business decision maker is simply Product Owner (https://less.works/less/framework/product-owner.html).

A decendant of the failed Rational Unified Process (RUP) approach called Scaled Agile Framework (SAFe) (http://www.lafable.com/) adds a multi-layer stack of traditional management on top of our "Scrum Teams." It sells because it's really just renaming all the traditional roles to Agile-sounding roles (https://fansofless.com).

These traditional/Agile hybrid approaches leave the biggest problems in the traditional space rather than the Scrum space of single-list prioritization and team self organization. With a narrow view of the product, only our toes wind up being Agile. The rest of our body is still hidebound by traditional management.

Scrum Masters (and management, if you have it) should help eradicate narrow views and encourage a Whole Product Focus (https://less.works/less/principles/whole-product-focus.html) instead.

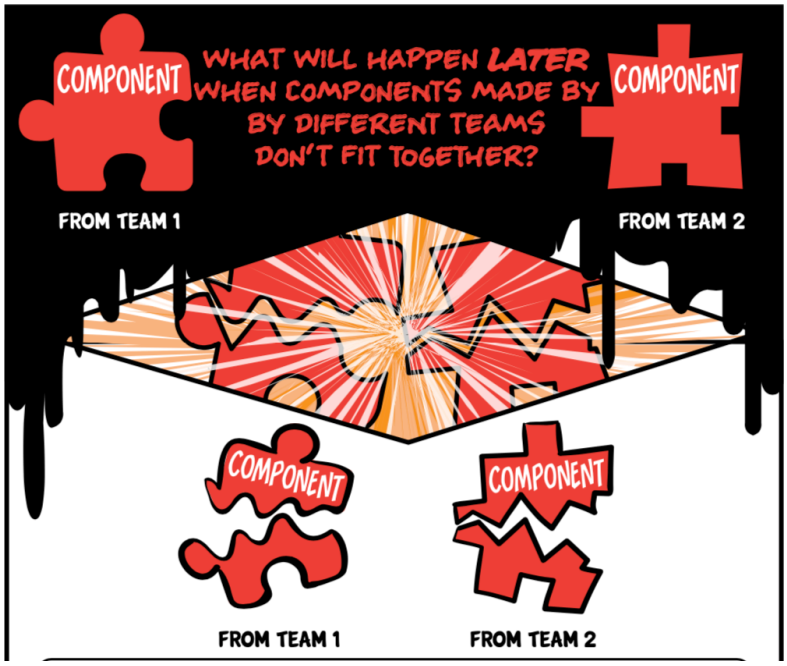
Japanese version: 大規模組織におけるソフトウェア開発の誤解その３：チームの視野は狭くて良いか、プロダクト全体を見るべきか? (https://scrummaster.jp/misconception-3-should-our-team-see-a-narrow-view-or-a-whole-product-view-jp/)

1. Images from *Product Owner In A Nutshell* by Henrik Kniberg. ↩

# Large Organization Software Development Misconception #4: Will Parts Made By Different Teams Fit Together By Magic?

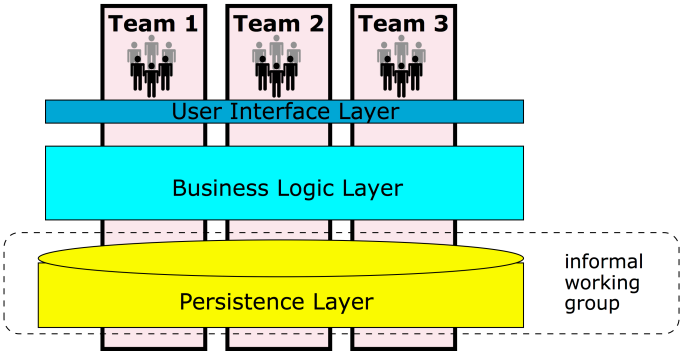
## Misconception 4.1: Will Components From Different Teams Integrate Into A Feature Without Teams Working Together?

By "components" I'm usually referring to architectural layers within the software, such as front end, back end, "platform," a device driver, etc. Too many large organizations have teams that can only work on components.




*Avoid Component Teams and Delayed Integration*

For Agility, we will usually prefer feature teams (https://less.works/less/structure/feature-teams.html) who span multiple components and can develop end-to-end, customer centric features in a shared codebase, rather than *component teams* who don't and can't.




*Example Feature Teams*

If we're using modern programming practices (TDD, Continuous Integration, trunk-based development, etc.), transitioning to feature teams *may* reduce the coupling between teams a bit. But not to zero. In fact, we should *not* try to reduce team-to-team coupling to zero unless our goal is team "productivity" instead of Product Development agility.

> *We should not try to reduce team-to-team coupling to zero unless our goal is team "productivity" instead Product Development agility.*

#### Misconception 4.1.1: Doesn't XYZ Technical Approach Eliminate The Need For Teams To Work Together?

No.

I guess people are getting this idea by the way the Microservices approach is sometimes pitched, or maybe after watching a Henrik Kniberg video about how Spotify used Chromium Embedded Framework to reduce team interdependencies. Only two months after publishing this article I heard someone propose "DDD, CQRS, Event Sourcing, Reactive Systems, Actor model" as ways to eliminate the need for teams to talk to each other. Remember when people thought RPC, XML, SOAP, etc. were going to eliminate the need for developers to collaborate? Look how many years people have sought alternatives to working together! All these things may be wonderful inventions, but successful Agile organizations value individuals and interactions over processes and tools (https://agilemanifesto.org).

#### Misconception 4.1.2: Don't Well Defined Interfaces, Open Standards, etc. Eliminate The Need For Teams To Work Together?

Preexisting APIs share types of information that have been shared before. For example, Traffic Message Channel (https://en.wikipedia.org/wiki/Traffic_message_channel) allows anyone to make devices that are aware of motor vehicle traffic. Preexisting APIs allow old things to be used in new ways.

But companies also want their product to do newer things than that, where *both sides* of the interaction are still changing and old APIs may not suffice. If developing a new capability requires sharing new types of information between different parts of software, somehow developers will need to agree how to do it, and what the new types of information mean.

Teams will need to talk to each other. Remember what the Agile Manifesto (https://agilemanifesto.org/principles.html) has to say about the most effective way of communicating?

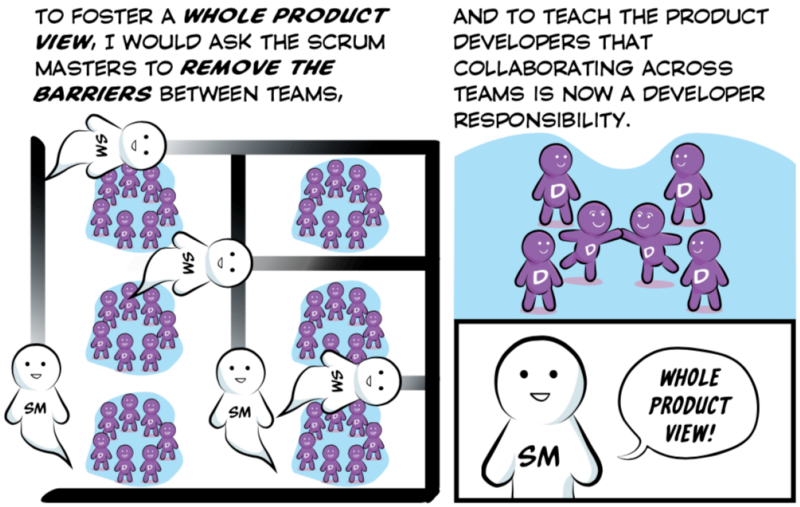
## Misconception 4.2: Will Features From Different Teams Integrate Into A Product Without Teams Working Together?

Let's say you've taken the wise step of switching from component teams to feature teams (https://less.works/less/structure/feature-teams.html). Your feature teams will *still* need to collaborate with each other to develop a cohesive product inside and out.

As we continuously integrate on the trunk into our shared code base, how will we keep the internal architecture and shared data from turning into a mess (https://less.works/less/technical-excellence/architecture-design.html)? (If your answer included the word "Microservices," please re-read the previous sections.) How will we cross-pollinate the learnings across multiple teams? How will everyone develop a *Whole Product View*?

The examples most obvious to a non-programmer may be User Interface and User Experience. Have you ever used a product that felt like each part was made by a different designer? If we want to make a cohesive product, we'll want team members involved in UI/UX to collaborate across team boundaries also.

So Scrum Masters should be encouraging collaboration across teams, just as they encourage collaboration within teams. Fortunately there are many better ways to do this than "Scrum of Scrums" (https://less.works/less/framework/coordination-and-integration.html).



## Misconception 4.3: Will Products Sold Separately *Or* As Parts Of A Suite Integrate Without Teams Working Together?

It would be a bizarre business strategy for a 10-team product company to simultaneously, with the same intensity, develop 10 products that had nothing to do with each other. (I'm focusing on a real *product company*, not a project-services company making bespoke software for a variety of clients.) Companies don't gain competitive advantage by making products that don't work together. (And we might have a few products from the past that we only maintain occasionally. As discussed in Misconception #2: Are All Teams Working On Equal Value Stuff?, they won't all warrant the same amount of attention.)

Except in trivial cases where we aren't doing anything new, engineering effort will be required to ensure the products work together properly. Take Adobe products (please): The value of the various Adobe products I use is related to how well the integration works (or doesn't work) between them.

In some cases I can substitute incompatible products in my workflow by using the lowest-common-denominator interface to the otherwise-incompatible tool. But that usually adds hassle and reduces functionality.

For example, I can move a picture of a cartoon character from a drawing program into an incompatible e-learning editor. But it takes extra steps to export a PNG file from one program and import it into the other. And due to the loss of layering information in the PNG file format, I won't be able to change her facial expression from the e-learning editor.

Companies won't improve their product suite by ignoring integration amongst their products.

One of my clients is in a domain largely devoid of prior standards and APIs because (unlike Adobe) no one else has done what they're doing. They have a few hundred developers working on what used to be a dozen distinct products. But they know their real competitive advantage will come when customers see the whole suite as one integrated product.

These products weren't initially developed to work together. For them it's worth the millions of dollars of effort they're expending to make them work together inside and outside. It would be harmful to send them a typical Agile coach or Scrum trainer who is going to focus them on individual team productivity – like I used to 15 years ago – rather than the whole product.

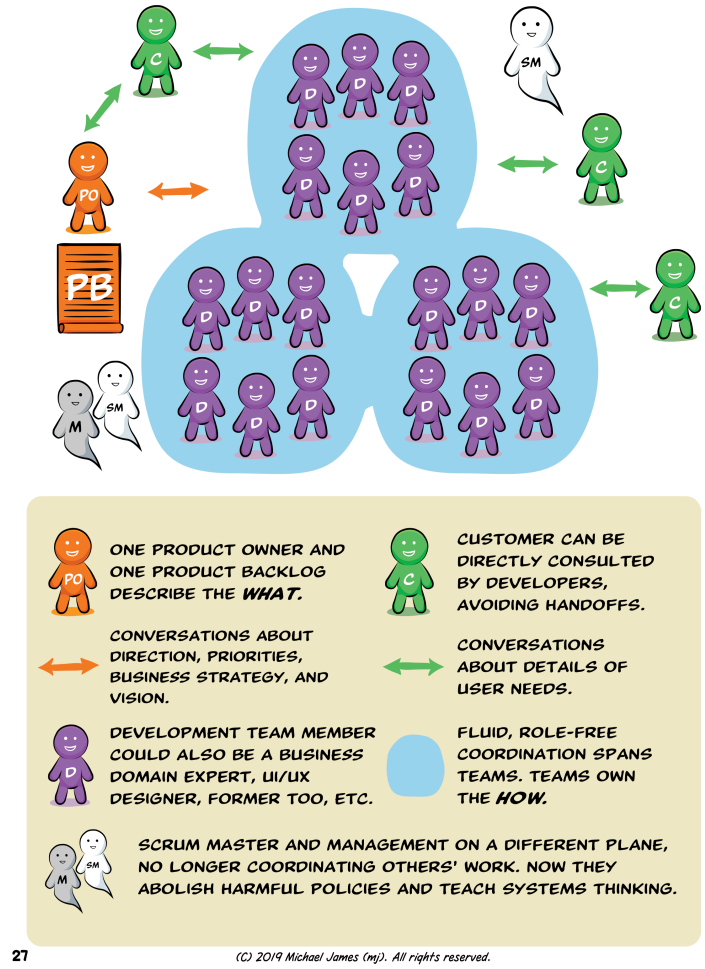
 

## History Of Attempts To Eradicate This Misconception

Here are NATO conference notes from 1968 pointing out how silly it is to believe the following:

> *Interfacing this system to the rest of the software is trivial and can be easily worked out later.*

If we have more than 12 people in a product company, we want teams working with teams (and customers).



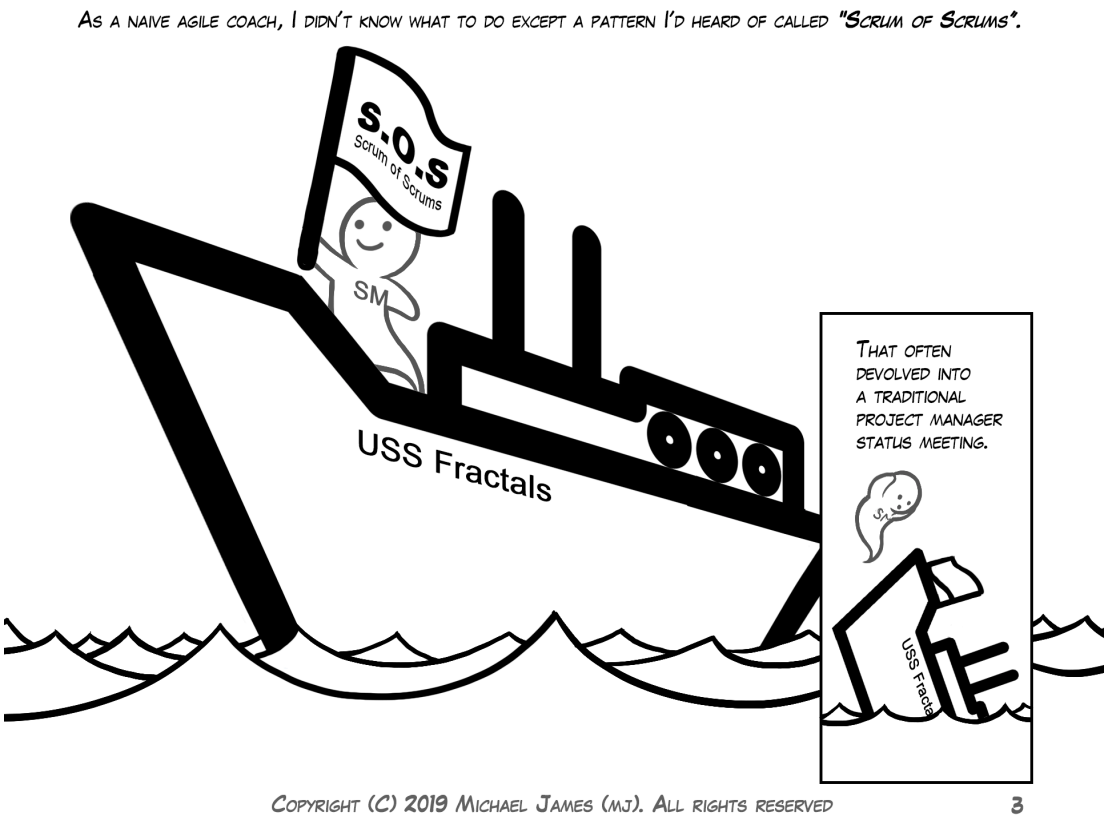

Japanese version: 大規模組織におけるソフトウェア開発の誤解その４：各チームが作ったパーツは、魔法の力で組み合わさるのか？ (https://scrummaster.jp/misconception-4-will-parts-made-by-different-teams-fit-together-by-magic-jp/)

## Seven Alternatives To The Dreaded Scrum Of Scrums

Scrum Master Peter Woolley writes:

> *I have only seen Scrum of Scrums (SOS) work once, when SMs and POs met 3 times a week to share their premeditated conversations and working agreements at a feature level, frequently it descends into a round robin project update or a mind numbingly boring task level detailed update meeting where PMs confuse and confound each other.*

I'll start with the most practical link in this article. If you're in a hurry, just go to the overview of the seven alternatives here: https://less.works/less/framework/coordination-and-integration.html (https://less.works/less/framework/coordination-and-integration.html)



## Back Story of Scrum of Scrums

### The Black Book

Ken Schwaber and Mike Beedle published the first Scrum book in 2001. We called it "The Black Book" because of its obnoxious cover. They had a lot of abstract theory and just a bit of practical experience how to make Scrum work in a small context, but much less to say about multi-team endeavors. In what strikes me as almost an afterthought, The Black Book contained the idea of Scrum Masters from multiple teams meeting with each other regularly to coordinate. It had the catchy name "Scrum of Scrums," reflecting the seductive fractal symmetry of the idea.

### The Grey Book

Ken Schwaber wrote his best book ever in 2004: *Agile Project Management With Scrum*, a.k.a. "The Grey Book." Please skip The Black Book, but you don't really understand the modern *Scrum Guide* if you haven't read The Grey Book. Unfortunately the grey book still included Scrum of Scrums, but fortunately had the improvement of sending team representatives (explicitly "engineers" in Ken's 2007 book) instead of Scrum Masters. Ken's last page (before the Appendices) contains this fascinating self critique:

> *When I [Schwaber] presented these case studies at a meeting of ScrumMasters in Milan in June 2003, Mel Pullen pointed out that he felt that the Scrum of Scrums practice was contrary to the Scrum practice of self-organization and self-management. Hierarchical structures are management impositions, Mel asserted, and are not optimally derived by those who are actually doing the work. Why not let each Team figure out which other Teams it has to cooperate and coordinate with and where the couplings are? Either the ScrumMaster can point out the dependency to the Team, or the Team can come across the dependency in the course of development. When a Team stumbles over the dependency, it can send people to serve as "chickens" [uncommitted participants] on the Daily Scrum of the other Team working on the dependency. If no such other Team exists, the Team with the unaddressed dependency can request that a high-priority Product Backlog item be created to address it. The ScrumMaster can then either let the initial Team tackle the dependency or form another Team to do so.*

> *This was an interesting observation. Scrum relies on self-organization as well as simple, guiding rules. Which is more applicable to coordinate and scale projects? I've tried both and found that the proper solution depends on the complexity involved. When the complexity is so great that self-organization doesn't occur quickly enough, simple rules help the organization reach a timely resolution. If self-organization occurs in a timely*

> *manner, I prefer to rely on it because management is unlikely to devise adaptations as frequently or well as the Team can. Sometimes the ScrumMaster can aid self-organization by devising a few simple rules, but it is easier for the ScrumMaster to overdo it than not do enough.*

Ken's CSM training materials and online discourse from this time on reflect his growing realization that the Scrum Master should have no authority over the team.

### The Disavowal

At a Scrum Gathering (probably Stockholm in late 2007), Ken publicly disavowed Scrum of Scrums. Nearly every slide about it disappeared from his training deck. In a discussion group many years later, Mike Beedle expressed regret for the Black Book's recommendation of using Scrum Masters as team representatives.

In 2008, Craig Larman and Bas Vodde wrote in *Scaling Lean & Agile Development*:

> *The most common misconception regarding the SoS is the assumption that it is the best or only way to hold a coordination meeting in Scrum. The SoS seemed a reasonable idea when first proposed (based on limited experiments), but there are alternatives that people now realize may work better…*

### SoS Meeting Recommendation Morphs Into A Structural Recommendation

Bas Vodde sent me this comment:

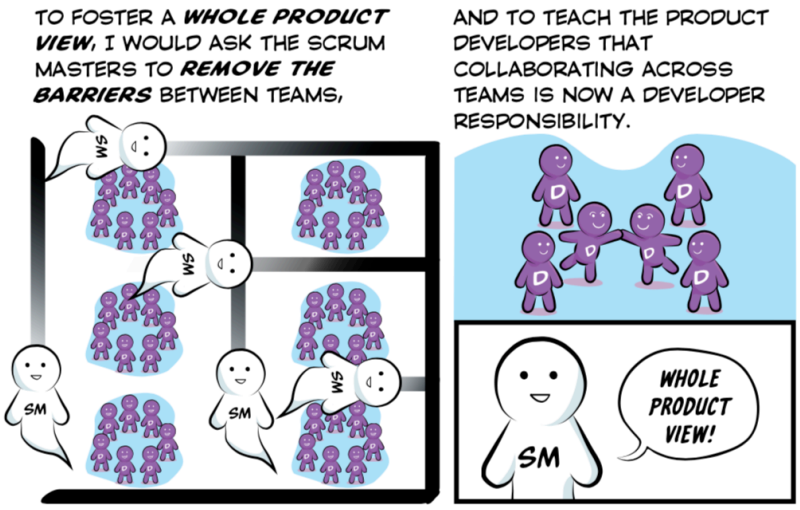
> *I think you probably want to distinguish two different Scrum of Scrums concepts:*
>
> *1) The Scrum of Scrums meeting. Which seems to be in most of Ken's writing and to be associated with SAFe. Most of the article is about that.*
>
> *2) The "Scrum of Scrums" structural idea. This is what Cesario was referring about and is called "the Nexus" in Nexus. You 'could' think of an Requirement Area in LeSS of that (I do not, but can understand the argument).*
>
> *My interpretation of history is that (1) morphed into (2) over time…*
>
> *Bas*

### Why Scrum of Scrums Continues 16 Years After The Problems Are Known And 10 Years After Ken Disowned it

My theory is that Scrum of Scrums is like a bad song we can't get out of our heads. I think other influential people (including Mike Cohn and Jeff Sutherland, who became a Scrum trainer in 2008) kept promoting it because we had so few agreed answers for organizations with more than 11 people. Scrum of Scrums is not in the *Scrum Guide*, but it is built in the two top-marketed "big box" approaches to scaling (http://www.lafable.com). Scrum of Scrums is semi-discouraged in LeSS (the less marketed approach to scaling) because we keep finding alternatives that are better at increasing an organization's ability to learn and adapt.

---

## Coordination & Integration: What To Do Instead




All excerpts from https://less.works/less/framework/coordination-and-integration.html (https://less.works/less/framework/coordination-and-integration.html)

> *Scrum values decentralized, self-organized coordination and integration over the centralized, controlled coordination found in traditional project management. LeSS coordination and integration concentrates on how to support decentralized coordination while providing enough boundaries and structure to avoid chaos.*

> *1. Just Talk*
>
> *Probably the best way to coordinate between teams is to simply coordinate between the teams. Any member of a self-managing team would be able and expected to reach out to another team if there is an issue to be discussed. When there is a need for coordination then "just talk" by going to the other team, picking up the phone or, in the worst case, dropping them an email. You do not need a formal, official, usually slow coordination mechanism in order to coordinate. Get up and talk to people.*

How do we know they'll want to talk to us? In LeSS we try to remove the usual obstacles to that, such as teams are working for different Team Output Owners. Multiple teams work in one Sprint, toward one Sprint Review (https://less.works/less/framework/sprint-review.html) of one potentially shippable product increment (https://less.works/less/framework/potentially-shippable-product-increment.html). Scrum Masters should encourage inter-team collaboration instead of focusing on individual team output.

> *2. Communicate in Code*
>
> *LeSS groups adopt continuous integration (https://less.works/less/technical-excellence/continuous-integration), which means that everyone has all their code checked in to the central repository mainline (branches are an unnecessary complication that should be avoided). Everyone in the product teams synchronizes with the repository several times a day and will get all the changes that other people have made.*
>
> *So, when you update, spend two minutes going over the changes that were made by others and see if they relate to what you are working on now. If so, feel free to get up and "just talk" in order to synchronize your work with the others!*

Branching is *delayed integration*! Merge problems increase exponentially over time.

What else makes *Communicate in Code* hard? The my code, your code practices and policies many companies have.

> *3. Send Observers to the Daily Scrum*
>
> *A simple coordination method for teams is to send a representative —not the Scrum Master—as a silent observer to the Daily Scrum of other teams doing related work. The observers then report back to their teams so they can take further action.*

Unfortunately, some locally optimizing Scrum Masters try to prevent this! A Scrum Master with a Whole Product Focus would do everything possible to make this easy.

> *4. Component communities and mentors*
>
> *People working on the same components at the same time need to know of each other so they can ask questions and review each other's code. Do this by creating component Communities of Practice (CoPs) (https://less.works/less/structure/communities.html), which should communicate via mailing lists, chat, occasional meetings, and other remote collaboration channels.*
>
> *These communities are often organized by a "component mentor" who is usually a member of a feature team who has taken some additional responsibilities such as (1) being the teacher of how a component works, (2) monitoring the long-term health of a component, (3) organizing a component community, (4) organizing design workshops, and (5) reviewing code.*
>
> *A component mentor doesn't review code for approval before it's committed. He's a teacher and steward of the component, not a gate.*
>
> *A component may have several mentors who share the work and thereby reduce key-person dependency.*
>
> *Important!... Besides helping coordination, these practices help maintain or improve the code/design quality of a component, and increase learning.*

One place I'm working with invited a component mentor from another team to help them with some code they weren't familiar with yet, with the agreement that the mentor would not touch the keyboard! This might seem "inefficient" to someone who doesn't yet realize the full cost of knowledge gaps.

> *5. Scrum of Scrums*
>
> *A Scrum of Scrums meeting is a Daily-Scrum-like meeting between teams, typically held two or three times per week.*
>
> *Usually the format is three questions, similar to a Daily Scrum:*
>
> 1. *What did my team do that is relevant to other teams?*
> 2. *What will my team do that is relevant?*

3. *What are my team's obstacles that other teams should know about or can help with?*

*Naturally, use and evolve whatever questions your group finds useful.*

*A caution: The desire to hold a Scrum of Scrums can be a sign of unnecessary dependency or coordination problems caused by single-function groups and component teams, or by teams not able or willing to identify and do shared work.*

*Scrum of Scrums isn't a part of LeSS and as a more formal centralized coordination technique, it is also not recommended. That said, if it is working, please don't stop doing it… yet don't feel you must do it because of adopting LeSS.*

If you decide to do Scrum of Scrums meetings anyway, remember to rotate team members. And of course you wouldn't send Scrum Masters, since they have no authority to coordinate teams.

#### *6. Multi-team meetings*

*It is common for some teams to need to work closely together whereas others may not feel that need. For teams that work closely together, it is common to have multi-team LeSS meetings. Some examples would be:*

- *Multi-team Product Backlog Refinement (https://less.works/less/framework/product-backlog-refinement.html)*
- *Multi-team Sprint Planning Two (https://less.works/less/framework/sprint-planning_two.html)*
- *Multi-team Design Workshops (https://less.works/less/technical-excellence/architecture-design.html)*
- *Exchange members in Daily Scrum.*

Are you seeing the theme here? Only have the meetings you need. If you've been to an Open Space Conference (https://www.amazon.com/dp/B005XOOKOY/), you know The Law Of Two Feet requires you to use your two feet whenever you're not learning or contributing.

#### *7. Travelers to exploit and break bottlenecks and create skill*

*Sometimes a product group relies on a couple of experienced technical experts. How can the knowledge of these (scarce) experts be kept available to all teams? They can become travelers. Each Sprint they join a different team, coaching via pairing, workshops, and teaching sessions.*

> *Although travelers are not specifically created for coordination, by joining different teams they create or strengthen a broad network, which is exactly what is needed for informal coordination channels. And they increase the consistency of some knowledge or practice across teams, realizing a coordination goal.*

Once when I helped teams reorganize (using a team-self design workshop (https://www.ahmadfahmy.com/blog/2013/12/5/the-rise-of-the-team)) I was surprised that several specialists did not want to join teams and opted to call themselves travelers instead. But a few months later, they had settled into teams.

> *8. Leading Team*
>
> *In some domains, features are monstrously large. When you split these giants into smaller Product Backlog Items, it can require many teams working closely together, each separately on its own PBI, to create a single monster feature.*
>
> *Another technique for coordinating teams working together on the split items of a big related feature is a leading team. A leading team is just a regular feature team that takes a leading role for the overall giant feature. In addition to doing development work, they are responsible for keeping track of what the other teams are doing and helping them synchronize. In short, they organize cross-team coordination related to the giant in addition to themselves doing development.*
>
> *Sometimes several teams start implementing the giant at the same time. At other times, the leading team starts alone to focus the early knowledge transfer and simplify the creation of a cohesive design. After a few Sprints, more teams join. In this case the lead team also has a teaching responsibility to help the incoming teams learn what they already know.*

Do not underestimate the power of a single team in an optimized environment! The development director of one place I worked with in local government told me that as he introduced Scrum he realized he really only had one capable team out of his 40 employees. In another famous story, over 100 contractors for the FBI were unable to get anything done. The solution was to use a much smaller group in the basement of the FBI Headquarters.

Japanese version: 面倒なスクラム・オブ・スクラムに取って替わる７つの方法 (https://scrummaster.jp/seven-alternatives-to-scrum-of-scrums-jp/)

## Why We Do Not Recommend Trying To Break Products Into Small Independent Pieces For Small Teams

Just today I heard two people give the same naive scaling advice: divide products into independent pieces for different teams to work on. This advice is so common that it's become a cliché. It illustrates a widespread blind spot in the Agile coaching and training community.

Pre-dividing products would appear to make life easier for Scrum Teams. But in the big picture, it actually *reduces* the product development organization's agility:

- As you may know, agile developers try to avoid *big design up front*. Designing an organization around which teams will work on which parts is *big prioritization up front*. It adds friction to future re-prioritization, resulting in some teams doing work out of priority order. Isn't Agile supposed to make it *easier* to re-prioritize? Please see Large Organization Software Development Misconception #2: Are All Teams Working On Equal Value Stuff?.
- When teams only see a narrow view, we put the important problems in the space of traditional management methods rather than team self organization. Please see Large Organization Software Development Misconception #3: Should Our Team See A Narrow View Or A Whole Product View?.
- Customers want integrated products and solutions, not pieces that don't fit together. Please see Large Organization Software Development Misconception #4: Will Parts Made By Different Teams Fit Together By Magic?
- It reduces team learning about the world outside their "own code." Please see My Code, Your Code. Private code policies considered harmful to agility.