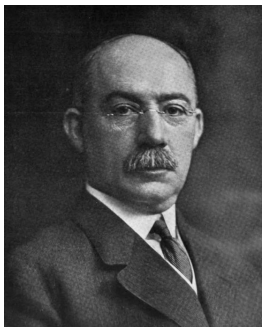
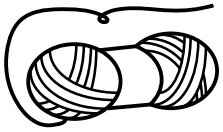
Large Organization Software Development Misconception #1: Are "Dependencies" In Knowledge Work Caused By Immutable Laws Of Physics?

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.

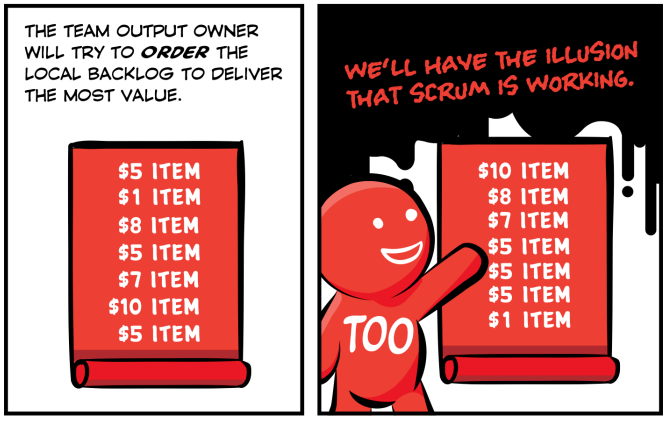
Once we internalize this realization, it leads to actions which were counterintuitive before – often the opposite actions that project management would advise. Unfortunately your organization’s structure and policies have actively encouraged these knowledge gaps.

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.

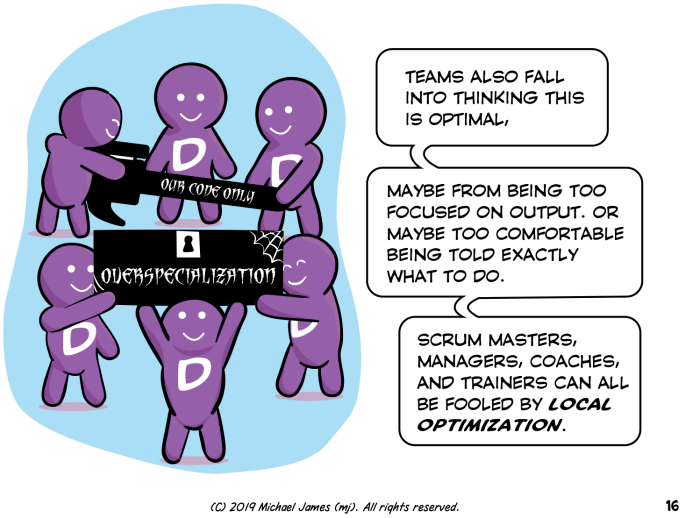
Japanese version (https://scrummaster.jp/misconception-1-dependencies-are-caused-by-immutable-laws-of-physics-jp/)

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?

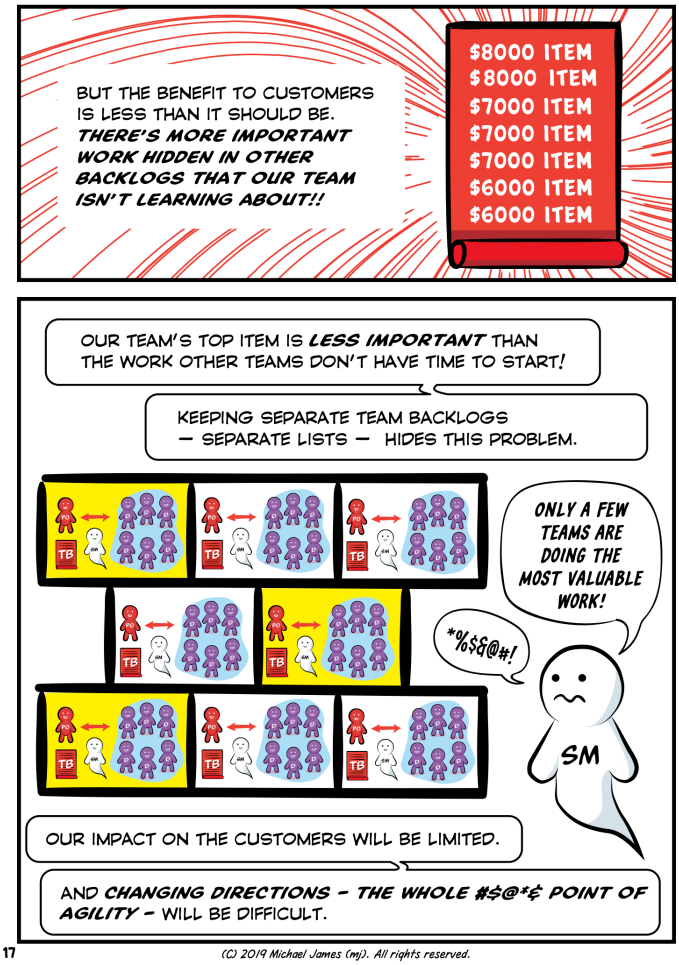


This is a great idea if your goal is to keep everyone busy. If your goal is to do the highest priority, highest value work, it could only be justified if several implausible things were true:



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.

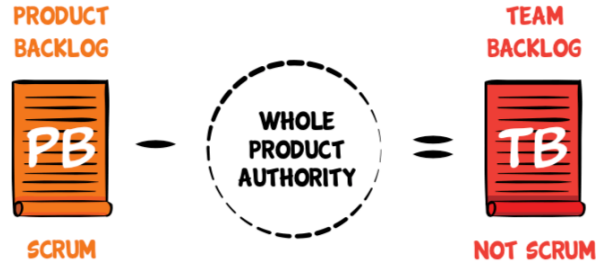
Japanese version: 大規模組織におけるソフトウェア開発の誤解その２:すべてのチームが等しい価値の業務に取り組んでいるか？ (https://scrummaster.jp/misconception-2-all-teams-are-working-on-equal-value-jp/)

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

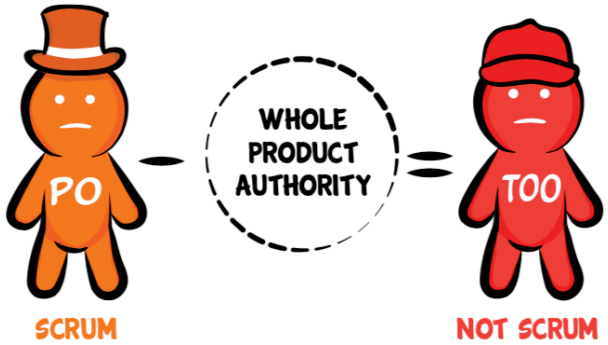
Small teams are great, and mandated by the definition of Scrum. Scrum also mandates a Product Backlog and Product Owner.

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.

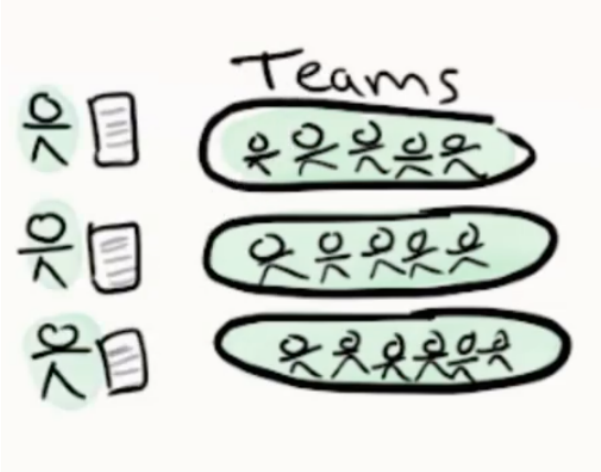
The next mistake is to give each team a "Product Backlog" which astute observers will notice is really only a *Team Backlog*.



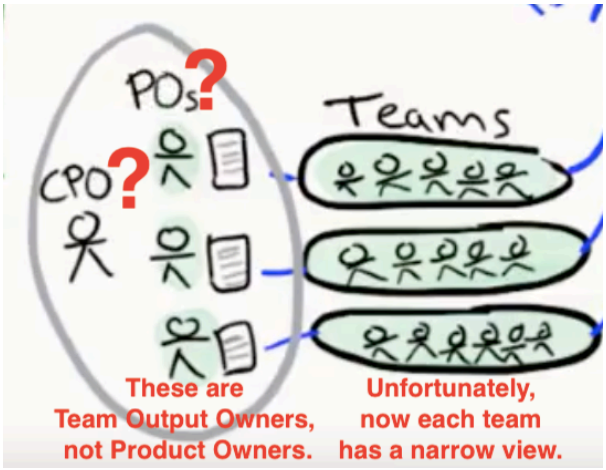
And then give each team a separate "Product Owner" who is really just a *Team Output Owner*.



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/)

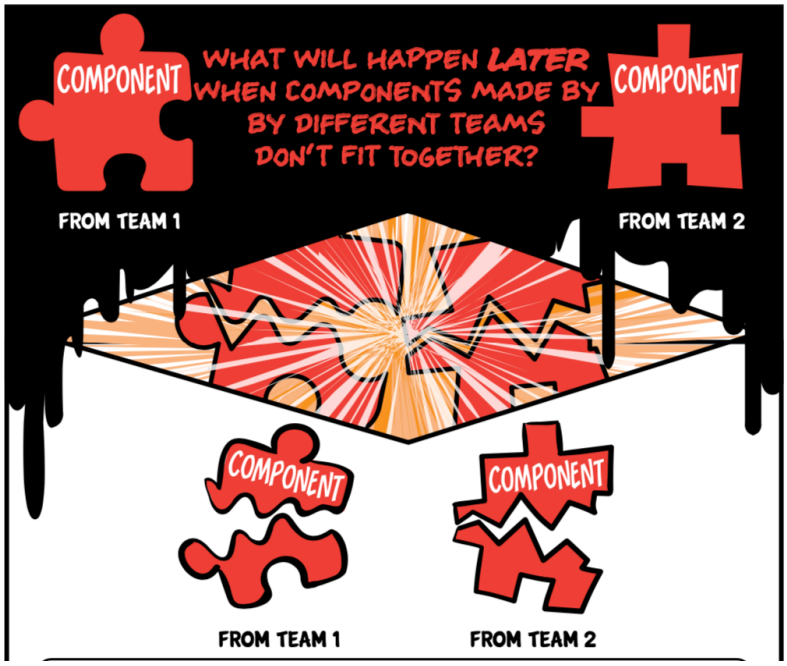
---

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?

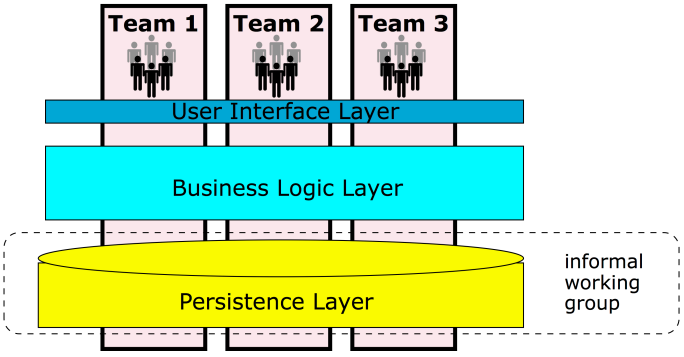
Misconception 4.1: Will Components From Different Teams Integrate Into A Feature Without Teams Working Together?

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?

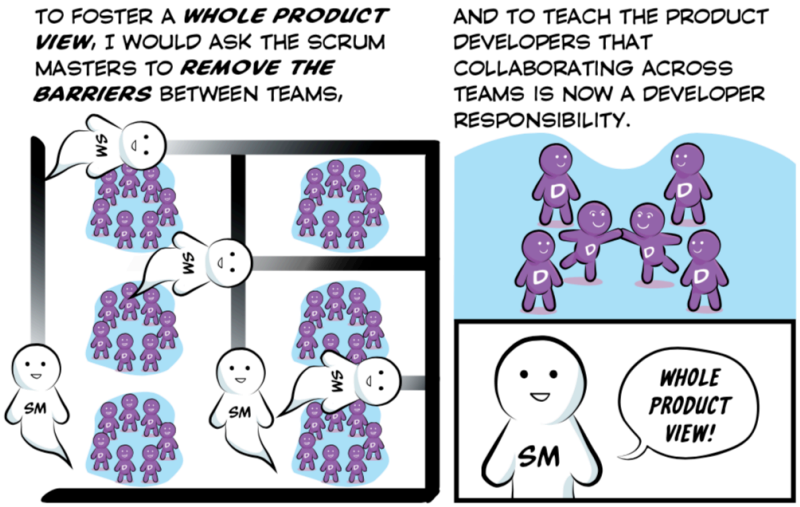
Misconception 4.2: Will Features From Different Teams Integrate Into A Product Without Teams Working Together?

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.

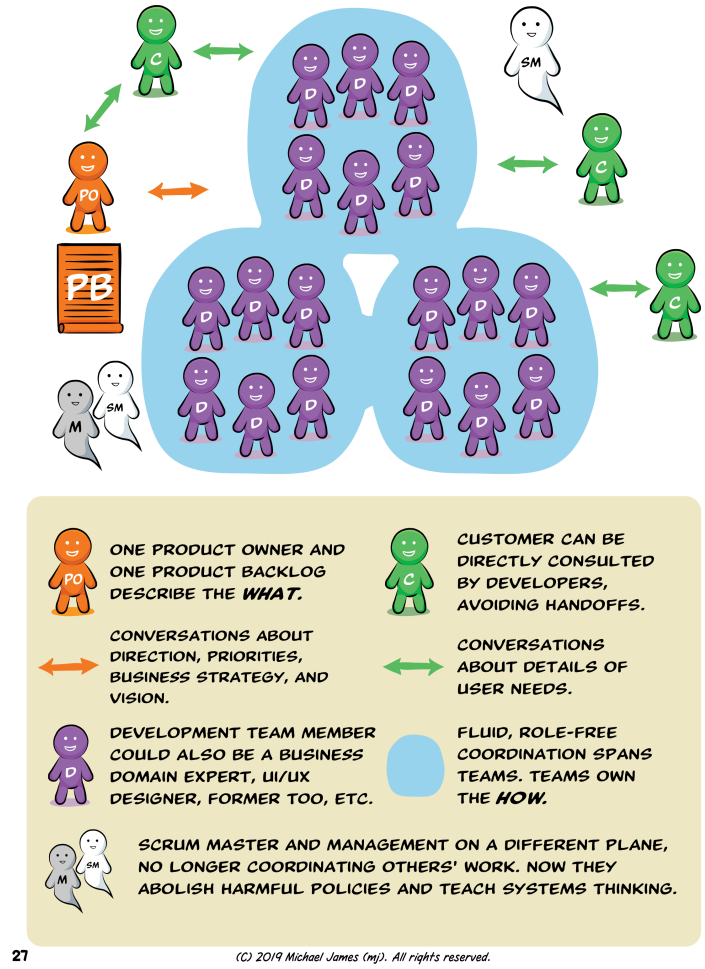
These products weren't initially developed to work together. For them it's worth the millions of dollars of effort they're expending to make them work together inside and outside. It would be harmful to send them a typical Agile coach or Scrum trainer who is going to focus them on individual team productivity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

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

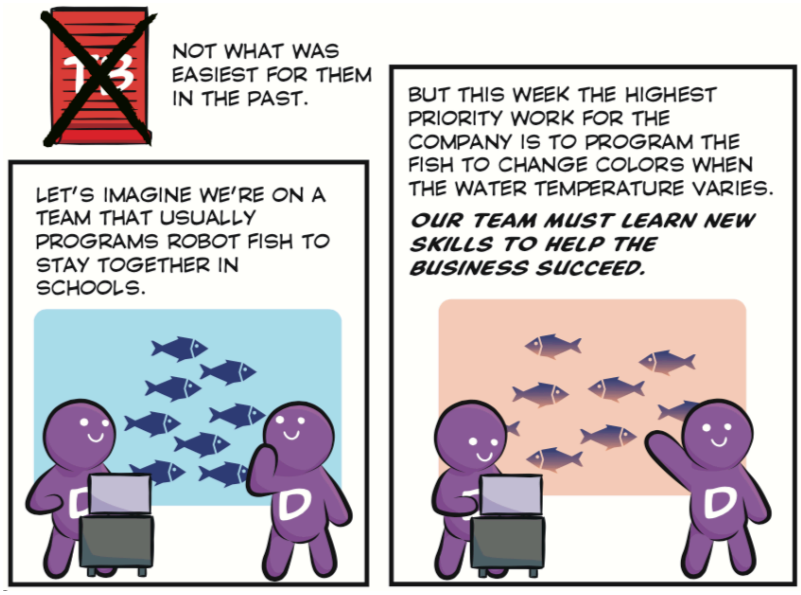
Watch out for *private code policies and practices*. Tying particular people or teams to particular parts of the code has several negative impacts:

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. ↩

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.