Engineering Our Teams

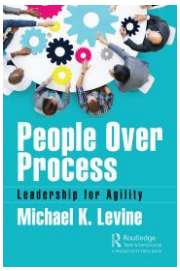
Michael K Levine

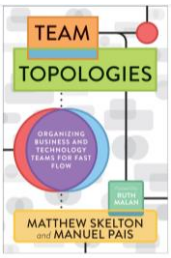
Location, Date

With material & ideas with permission from Team Topologies, Matthew Skelton & Manuel Pais, 2019

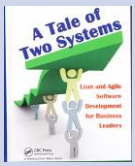
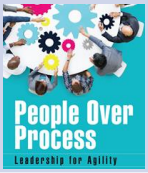
Agenda

- Team and organizational structures matter
- Teams: nature & limits
- Simple model: Silos & Bridges
- Team Topologies
  - Four types of teams
  - Three kinds of interactions
  - Putting it all together
- Some examples & implications
- What to do





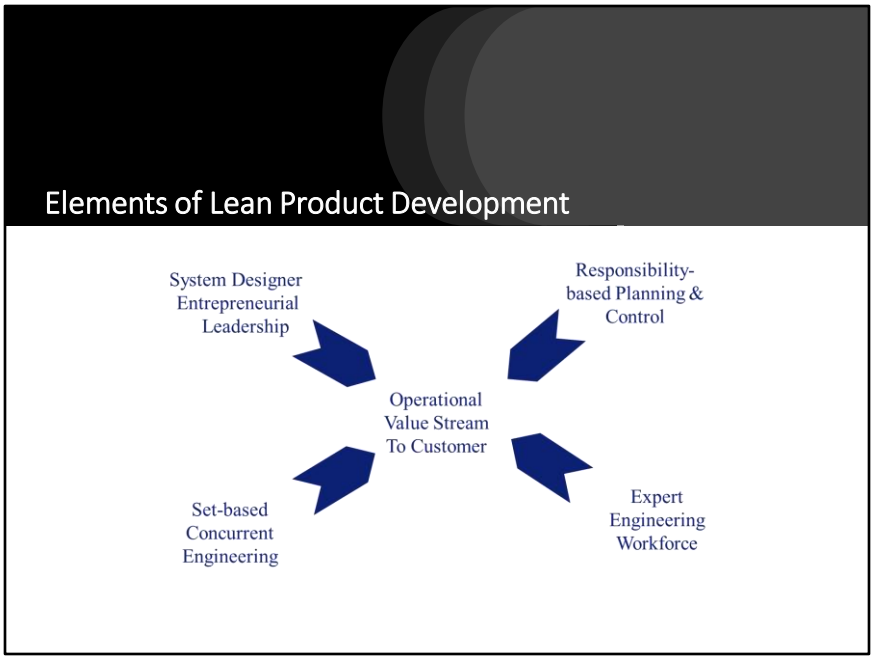
Michael's Lean / Agile Journey

	Prior to 2002	2002-2007	2007-2011	2011-2019
Career Focus	Financial Software	Scale mortgage operations & software; Wells Fargo CORE Failure	Leading Wells Fargo tech & process through mortgage crisis	Bringing agile to rigid "waterfall" US Bank. Revamp branch & consumer lending technology incl. digital
Lean / Agile Focus	**Agile before it had a name**	**Lean operations (LEI); Adopt Agile & Scrum; Lean Product Dev (U of Michigan)**	**Building & transforming teams to delivery quickly and well (be lean & agile)**	**Leadership, values & principles: "post-agile." Scrum <> agile!**
Message		***What is lean & agile & why do we do it?*** 	***How do we transform?*** 	***How do we sustain?*** 

Why do we care?

- Agile implemented as SCRUM
- SCRUM scaled as SAFE
- Mania around “product” over “project,” “dojos, ” micorservices vs. monoliths, APIs, etc.
- Focus on processes and tools over people and interactions, violating first Agile value
- Danger of “best practices”:
  - “The great danger in “best practices” is that the practice can get disconnected from its intent and its context and may acquire a ritual significance that is unrelated to its original purpose”
- So we want to go back to first principles: How do human beings work best? How does intellectually-heavy, by-neccssity-collaborative work get done?

Quote is from Don Reinertsen, Managing the Design Factory, 1997. He was a pioneer in thinking about accelerating product development.

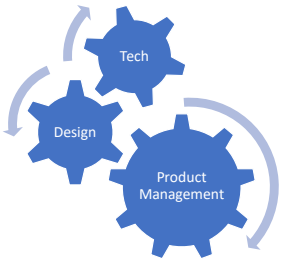


System Designer: The Chief Engineer. Fills a role that standard scrum / agile is often seriously missing.
Responsibility based planning and control. Includes idea of integrating events. A more general and flexible and people-oriented model that eg scrum of scrums, or consolidated release trains.
A very strong focus on expertise. If you want to build great systems, you have to hire and develop great people, and ensure that they become top experts and stay that way. Much more important than any methods.
Concurrent engineering. Since we are building knowledge, we often will want to have more than one approach going so we can learn and compare. Eg Coronavirus vaccines.

We'll touch on each of these in more detail.

Inspired

Marty Cagan, Silicon Valley veteran. “How to Create Tech Products Customers Love”

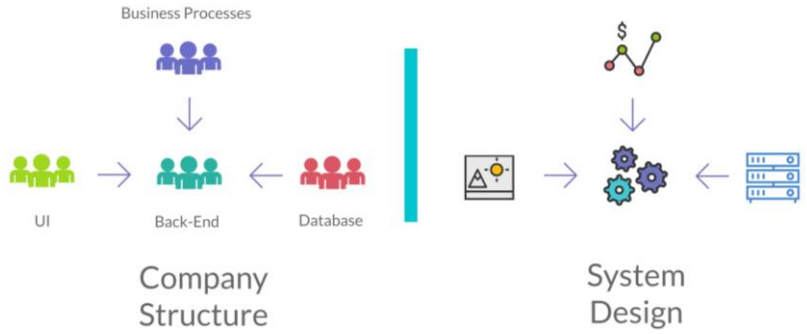


Three primary leadership dimensions

Design often organizationally aligned with Product

Conway's Law: "Organizations which design systems […] are constrained to produce designs which are copies of the communication structures of these organizations."

George Conway, "How do Committees Invent?," 1967.



Some examples

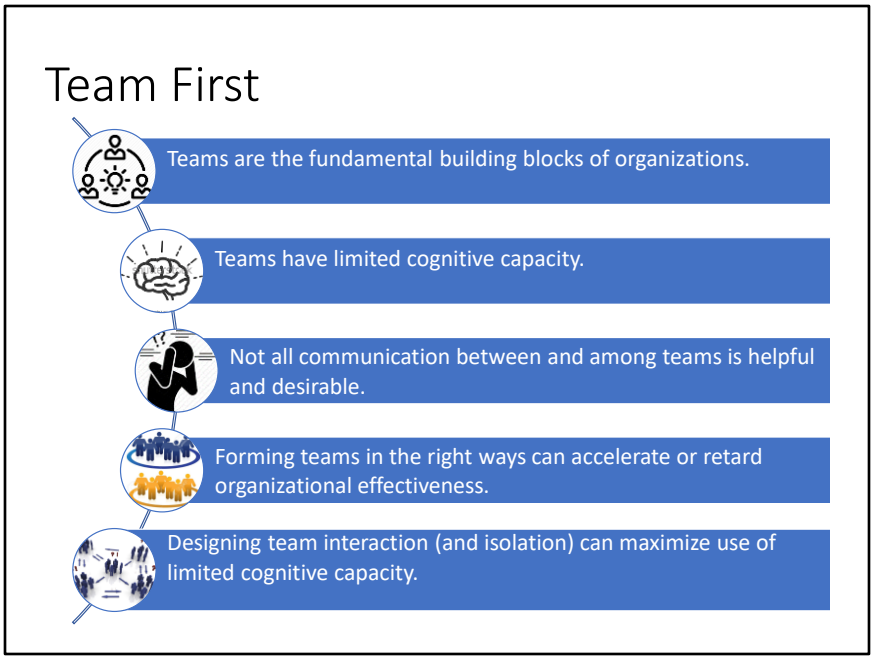
If your organization has a web/mobile team, an API team, an application team, and a database team, the structure of the software will be in those layers. And deployed in those layers. And tested in those layer.

We can use this idea to use organization structure in a positive way, to design our organizations to get the software architectures we want. This is called "the inverse Conway maneuver."

Many companies are doing this through what they call "Product Orientation." That is, to organize full-stack team around whatever it is they call "Products." This is potentially a positive step, recognizing the centrality of teams, but it is an over simplistic approach that ignores one more fundamental idea that Skelton & Pais point out: the inherent human limitations to think and remember things.

They called this "cognitive load."

Let's talk about teams next.



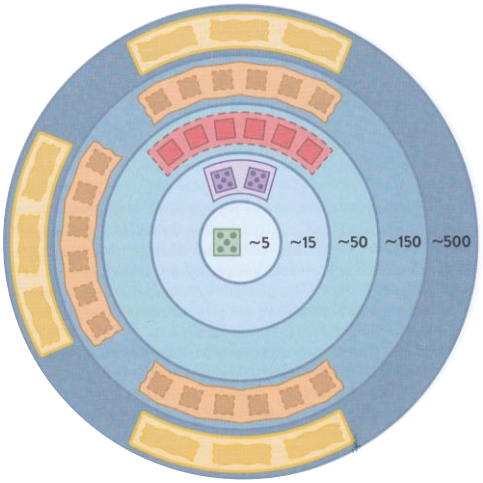
Bldg blocks…we all know this. 1st agile principle, people over process…when we need something done, we first ask, who do we need to help?

Cognitive capacity…I'll cover more on this on next slide, but basic idea is that each of us can only handle so much information at one time.

Overload..related to capacity. Jack Welsh's "eliminate all the silos" is not a good idea.

Given above, clear that as leaders how we form teams, and how we design their interaction, is crucial to the functioning of our enterprises

Limits to Trust



5	Close personal relationships
15	Deep trust
50	Mutual knowledge and trust
150	Can remember each other's capabilities

Source: Team Topologies by Matthew Skelton and Manuel Pais

This is from Team Topologies

Common team sizes tend to be 7-9, from software teams, to baseball/football, to the smallest units in armies since antiquity.

Research by anthropogists (Dunbar most famously, Dunbar's number is 150) and others (Snowden, Karlgaard and Malone, cited in TT) suggest a rough scaling as shown in the diagram.

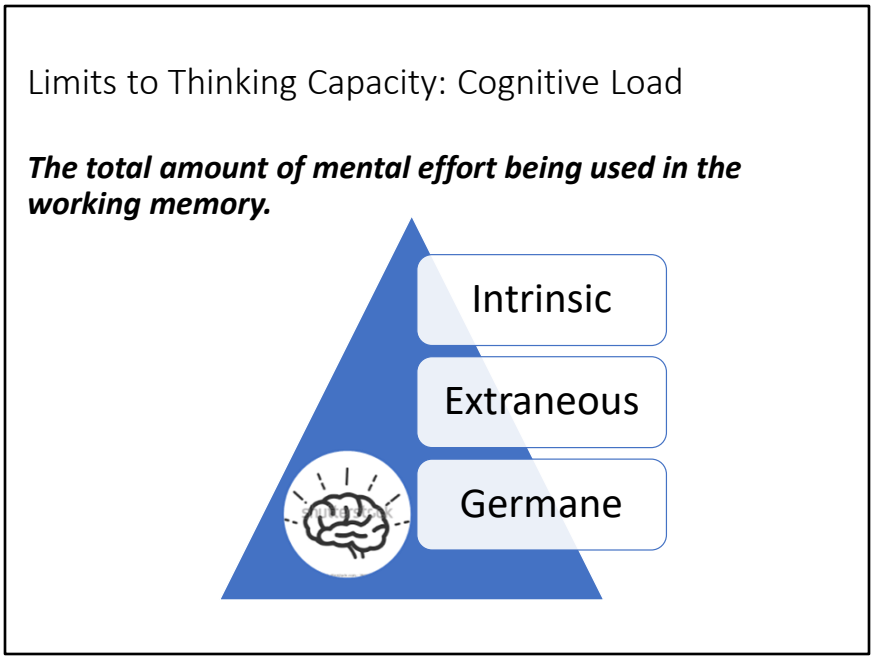
We can quibble about the numbers, but I think we all know the basic truth of this scaling.

Of the 5-15, we can know their capabilities, know what to delegate and how to monitor, know how to accept questions and requests, and know what to expect

At 50, we know enough of them that have some idea of what they can and cannot do, and know whether we can trust them or not

At 150, we least know who they are and roughly what they can do

At 500, we have vague ideas.



We need to help our team members avoid information overload. We only have so much capacity to handle information in our working memories. This definition is from John Sweller. Cognitive load theory was developed in the late 1980s out of a study of problem solving by John Sweller.[1] Sweller argued that instructional design can be used to reduce cognitive load in learners. Much later, other researchers developed a way to measure perceived mental effort which is indicative of cognitive load.[2][3] Task-invoked pupillary response is a reliable and sensitive measurement of cognitive load that is directly related to working memory.[4] Information may only be stored in long term memory after first being attended to, and processed by, working memory. Working memory, however, is extremely limited in both capacity and duration. These limitations will, under some conditions, impede learning. **Heavy cognitive load can have negative effects on task completion**, and it is important to note that the experience of cognitive load is not the same in everyone. The elderly, students, and children experience different, and more often higher, amounts of cognitive load.

This is from Jo Pearce, a developer in the UK, "Hacking Your Head" presentation. Draws on Sweller.

Intrinsic Load. Imposed by the inherent complexity. So, juggle 8 balls harder than 3…this is the stuff our team members need to know to do the task, such as how to define a class in java, how to check out code from github, how to write and run an automated test. We can manage this by breaking large tasks into smaller ones, automating multistep tasks, and moving less-frequent or specialized tasks off the team.

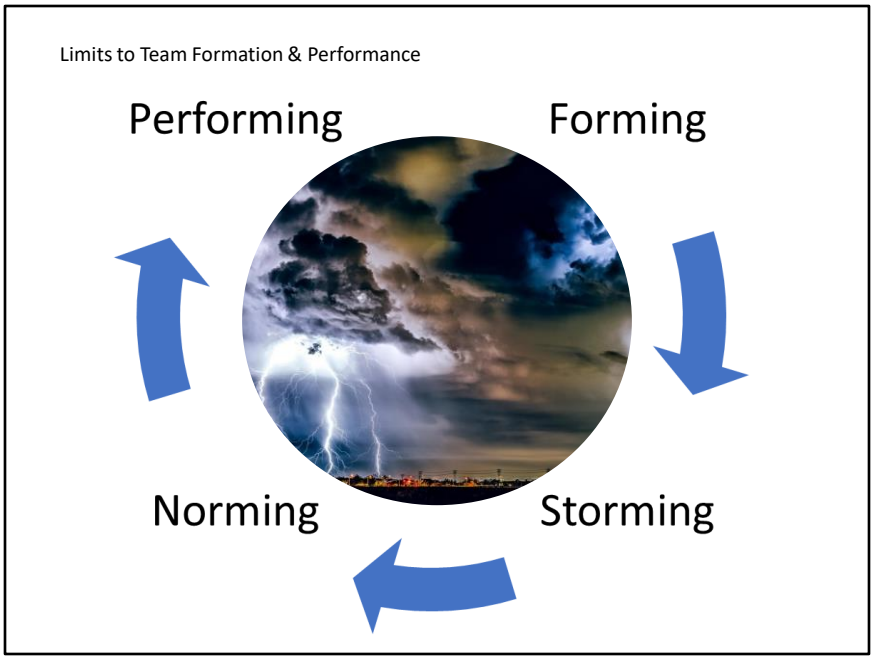
Extraneous (Irrelevant) Load. Imposed by distractions or information irrelevant or tangential to the goal. Loud environment, unreadable code, information not needed. Difficult multistep tasks done infrequently. Too many tools, too much change, too much interaction with others. From Don Reinertsen, the classic expert on product development : Whenever we see an intense need for communication it is typically a sign that the system has been incorrectly partioned"

One example of distractions: Open Offices

The impact of the 'open' workspace on human collaboration Ethan S. Bernstein1 and Stephen Turban2 1 Harvard Business School, Boston, MA, USA 2 Harvard University, Cambridge MA, USA. In two intervention-based field studies of corporate headquarters transitioning to more open office spaces, we empirically examined—using digital data from advanced wearable devices and from electronic communication servers—the effect of open office architectures on employees' face-to-face, email and instant messaging (IM) interaction patterns. Contrary to common belief, the volume of face-to-face interaction decreased significantly (approx. 70%) in both cases, with an associated increase in electronic interaction. In short, rather than prompting increasingly vibrant face-to-face collaboration, open architecture appeared to trigger a natural human response to socially withdraw from officemates and interact instead over email and IM. This is the first study to empirically measure both face-to-face and electronic interaction before and after the adoption of open office architecture. The results inform our understanding of the impact on human behaviour

Germane (Relevant) Load. Beneficial load imposed by information relevant to overall goal. These are things that help task accomplishments such as automation, repetitive patterns. **Includes knowledge of the business context, makes other tasks easier.**

Our goal in designing teams and interactions is to manage intrinsic load, reduce irrelevant load, and increase relevant load.



We all know that it takes time for a team to form.

This is a classic formulation of what teams do . The assemble, get to know each other, establish roles, routines, accommodations, agree on processes, communication. We all know the drill.

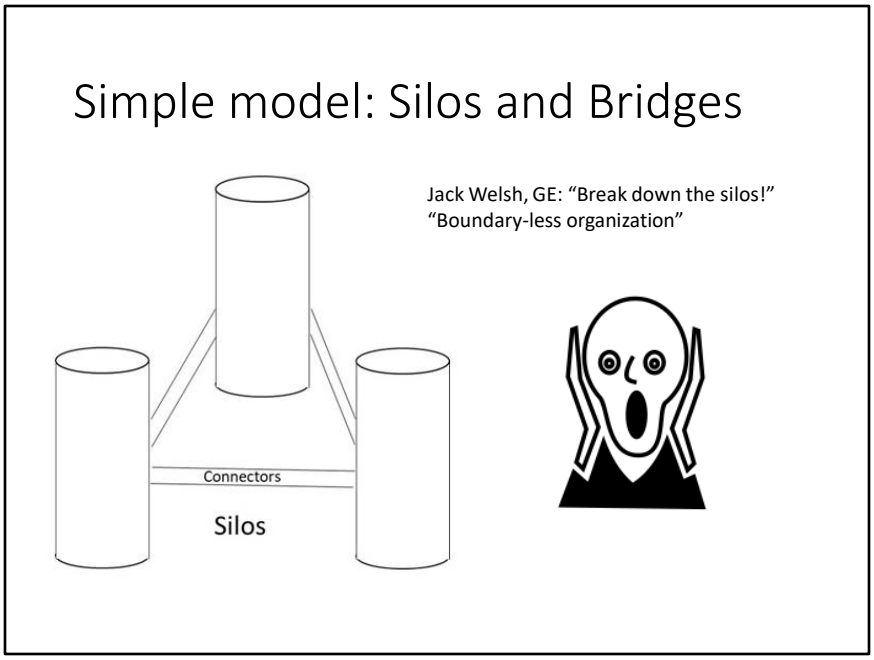
The classic formulation is linear, ending at Performing..which some teams do get to, eventually. But we all know that good teams keep storming forever, and wind up reforming and renorming to stay or regain or get to performing.

This means that once a team is well-formed, we should be treat it as a valuable asset.

This is some of what is behind the “project to product” model – having persistent dedicated teams for our most important work.

There is another idea, which I don’t believe works so well, of continually building and reshuffling teams based around project work. The idea of the itinerant gig worker, the business analyst who can work on anything, the project manager or scrum master who is primarily that role and not deeply into a persistent topic. We know that isn’t

very effective.



Amazon: Two pizza teams focused on a goal.
Scrum: 7-9 people focused on a code delivery for a system.

Now, what do we do when we need to scale? There are two cases – multi-team common goal, and multi-goal but intertwined enterprise.

In this simple case, we won't go deeply into different kinds of teams and the different interactions. We'll leave that for the next section, when I recap the great work by Matthew Skelton and Manuel Pais in Team Topologies. Here I'll just touch on some common bridging mechanisms that we know work.

Multiple teams, single goal

- Clear, repetitive leadership communication on shared goals and schedules.
- Shared physical space
- Scrum of scrum / leadership team meetings
- Common backlog in a shared tool.
- Common sprint schedule, and shared demos at sprint end.
- Common development and testing environments.
- Shared communication tooling, including visuals in the rooms, conferencing and

texting, and documentation storage.

- Master test strategy developed and communicated across teams.
- Formal management hierarchies. Many of multiple participants from a single discipline (e.g., Sales, Development) reported to common managers, who stayed involved and helped with communication and governance. Also essential to deal with team composition and performance issues.

Multiple goals, single enterprise

A large enterprise needs to connect it program or product teams, so they have the right information flowing in and out for good decision-making; and can react to change or problems, gain efficient contribution from those who can help, and ensure adoption.

- **Formal management reporting hierarchy**: as in the team context, a powerful bridge. For example, the devops team members — who remain part of the broader enterprise tech ops department, bring new capabilities to the team, and raise team problems to the department.
- **Governance structures**: reducing excessive hierarchical control is a worthy goal but we can't to eliminate it entirely. A formal governance group for major initiatives allocates senior leadership accountability, providing a routine forum for cross-functional collaboration and connection-making.
- **Informal relationships**: just as important as formal relationships. Team members and their managers should reach out to their peers and partners, establishing routine relationships across functions and even companies. In tough times these relationships unlock keys to rigor, alignment, and efficiency.
- **Statements of Work**: this concise, somewhat standard document lays out the what, why, when, how, and who of each major initiative, and provides a framework to establish rigor and efficiently enable engagement.
- **Status reports**: a major goal of the two-pizza team is empowerment, but in practice, and if done well, "trust but verify" improves rigor and alignment.
- **Transparency**: each team's work should be visible, as touted at Slack, where channels are typically open to review. But along with transparency, there still needs to be outgoing outward communication.
- **Inherent structural factors**: large traditional companies often have an enterprise release for widely-used shared systems. Control mechanisms for these releases can help teams align, much as the very existence of these periodic large release events can impede agility. On the other end of the spectrum, a microservice API-centric technology architecture like Amazon's can minimize the need for cross-team communication and coordination.

Going Deeper: Team Topologies

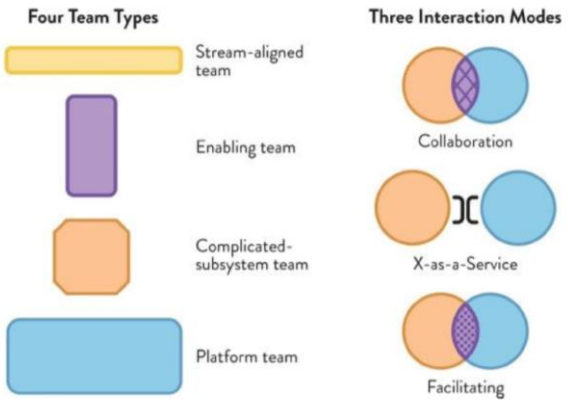


Figure 0.1: The Four Team Types and Three Interaction Modes

Source: Team Topologies

Matthew Skelton and Manuel Pais have been working on organizational design for clients globally, and documented their wisdom in their book published in late 2019. Matthew is based in London, Manuel in Madrid. Both are software engineers, both have been focused on development, devops, continual delivery and such.

Topology is defined ast “the way in which constituent parts are interrelated or arranged.”

Skelton & Pais have dug into more details on types of teams and types of interactions. Whereas I show in Silos and Bridges just a generic team and the generic connection mechanisms, they go deeper, which I found very valuable.

Let’s do a rough overview on this slide, and then I’ll take you deeper into the team types and the interaction modes. We’ll wrap by bringing them all together with some examples.

Walk through team types:
Primary team type is the “Steam aligned.” This is the team that is aligned with the flow of software from business need to production, whether new systems, routine

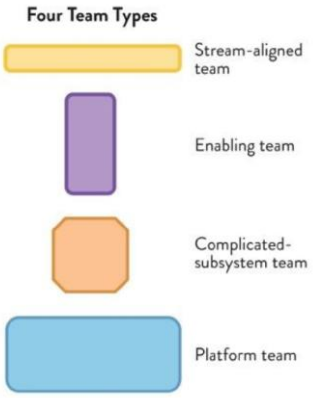
incremental updates, or whatever.

The other three kinds of team provide support in various ways.

The interaction modes include:
Collaboration, Facilitation, and providing a service.

Lets dig into these.

The Four Team Types



- **Primary team type.**
- **Aligned to business unit or organizational capability.**
- **Closest to customers / businesses.**

- **Specialists in technical / product domains.**
- **Examples: tools, frameworks, practices, ecosystem choices.**

- **Manage system elements that depend heavily on specialized knowledge**
- **Keep internals hidden while enabling stream teams to use**

- **Provide the technical foundation that enables stream-aligned teams to deliver**

Source: Team Topologies

Stream aligned teams. Can be aligned to a specific customer or customer segment, a business area, a geography, a product, a user-persona, or even compliance. Typically persistent, funded over time, portfolio of work not just as a fleeting project. Contrast with a project-focused organization, which would typically involve funding a piece or work (or collection) for several existing teams who each need to fit the new work into their existing backlogs.

Common example: Amazon teams, which are cross-functional from mange/design/build/test & operate, connected via APIs. (exception is testing roles which are highly coordinated across teams). Squads at Spotify.

Behavior: deliver business value. Do as much as possible within the team. From requirements through dev / deploy / production support.

Roles discussion. Who people report to discussion.

Enabling teams. Also called "technical consulting teams." Help stream aligned teams stay current, evolve over time, without having to take on all that work. Need to

avoid becoming “ivory towers”.. Enabling teams could include UX, architecture, testing; but also build engineering, continuing delivery, deployment, test automation. EG provide a deployment skeleton, or test automation framework. Could act as proxy for vendor or external service too complicated for each stream-aligned team to engage directly with.

I think about central testing team managing test data across teams; common enterprise release; etc.

How different than a community of practice? Full time vs part; specific leadership-defined goals and budgets.

Behavior: Increase the autonomy of stream-aligned teams, not cripple them with requirements and demands. Not a permanent ongoing dependency; episodic

Complicated Sub-System Teams. Some examples: risk reporting system for financial transactions; face recognition algorithym; real-time trade reconciliation mechanism. Customer info shared system, a wire system for a bank.

The focus in NOT reuse. That is secondary. Focus is limiting team cognitive loads. These complicated sub systems are needed by several stream teams, typically, but having the knowledge distributed directly to them is not really possible.

Different than traditional component teams. Those tended to be driven by a management desire to share out of efficiency. But they often created interfaces for many teams that were not worth the trouble. So, traditional component teams (eg, DBA or middleware) often should be liquidated with the expertise distributed to the flow teams that need the service (keeping an enabling team to help / simplify / make self service), or if more of a “platform” made that. There are definitely cases for component teams becoming sub-system teams, but be careful..organization shouldn’t be stuffed with these.

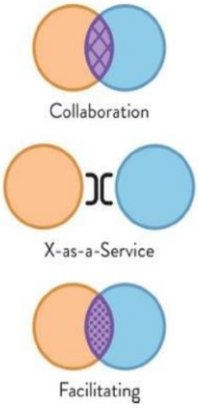
Behavior: Know the needs to steam-aligned teams and serve them. Manage the complicated system life cycle well.

Platform Team. Could be a “thick” platform comprising multiple inner platform teams with multitude of services, or “thin” team providing a layer on top of commercial solution (e.g., Salesforce, Amazon). Thick team can itself have all four team types. Goal: Thinnest Viable Platform (TVP). Some other historical platforms: Windows, JVM, .Net, Kubernetes.

Behavior: Value is what they provide to stream-aligned teams. Ease of use. Reduction in cognitive load of stream teams. Feedback loops to manage the platform. Evolution.

Good example: A central team building out internal cloud…providing for flow teams tools such as Kubernetes, Docker, Jenkins pipeline, Mongo, multi-side always on, etc.

Three Interaction Modes



- **Either working very closely on core common goal (almost one team), or pooling accountability for specific set of things**
- **Best when time-limited**

- **Ideal when possible as it limits interaction to minimum, standardized**
- **Good for components, discreet items used by multiple stream teams**
- **Downside: variation, change, innovation**

- **Manage system elements that depend heavily on specialized knowledge**
- **Keep internals hidden while enabling stream teams to use**

Source: Team Topologies

Collaboration. Its like becoming one team for a while, in pursuit of rapid innovation and discovery. Beware of becoming too many people (over 15). Adds cognitive load.

Don't do with more than one other team at a time.

Typical uses: Stream with complicated sub-system; stream with platform; complicated with platform

Behavior: high interaction, mutual respect.

X-as-a-Service. Eg amazon, vendor-provided web services such as Fannie Mae DU, credit bureaus, etc. When done well limits cognitive load and enables rapid provision of value.

Do with multiple teams! Beware of reduced flow if the boundary / api isn't effective, or if changes are needed to support flow.

Typical uses: stream and complicated subsystem teams consume platform as service from platform team, and consume components, services, libraries

from complicated subsystem teams.

Comment. Documents as a service for mortgage…very complicated service, contractual issues, set up and customization. For provider, this their whole business and they have multiple teams of each kind.

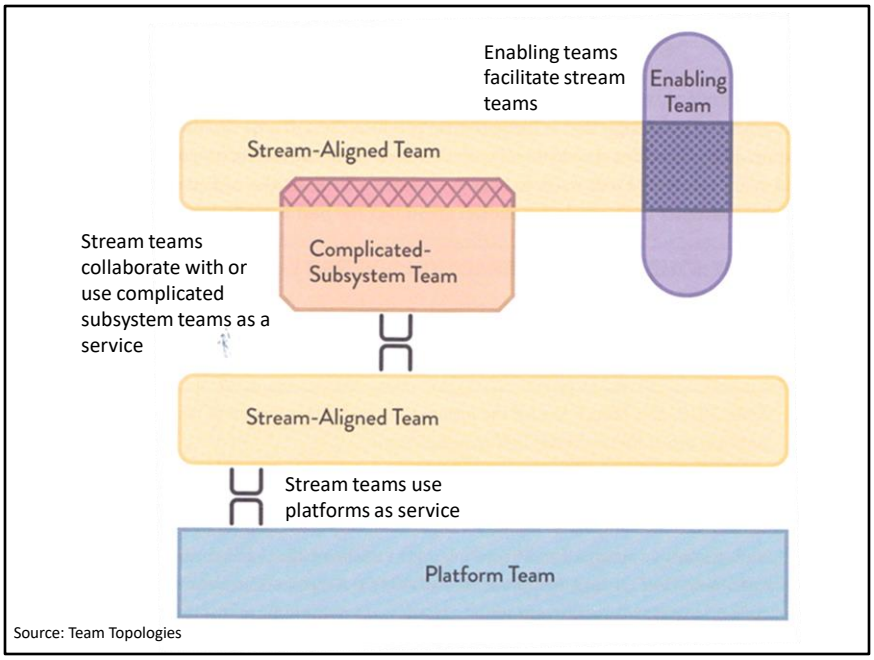
Behavior – emphasize ease of use. So stream teams don't need a lot of collaboration.

Facilitating. Goal is to sense and reduce gaps in capabilities. Typically help many stream or complicated subsystem teams. Don't actually build, but find ways to help.

Unblock stream aligned teams to increase flow. Detect gas and fill.

Typical uses: Think about this as coaching, not coding. Stream, platform, and complicated subsystem teams will likely have several of these interactions going at all times. These kind of teams can have these relationships with each other as well.

Some examples: Centralized test automation team. Agile coaching (don't overdo this!). Devops tool chain usage.



These are the typical interactions, see from perspective of stream aligned teams (which are the primary delivery model for any organization)

Lets take as an example a maturing startup I'm familiar with, which has built their product on top of the Amazon platform. This is quite similar to another one I'm close to, which has built on Salesforce as platform.

Each has multiple stream aligned teams, but lets look at from perspective of one team.

This core product development team depends on a complicated subsystem team – a team that manages customer-specific integrations. Many features that it builds requires such integration, and cannot be released into productive use without their support. So, as new features are built, the stream aligned team needs to collaborate closely with the complicated subsystem team. Both need to build new capabilities simultaneously to enable progress.

At the same time, other stream-aligned teams that are not building new features that require new integration capabilities are using the integration team as a service. They

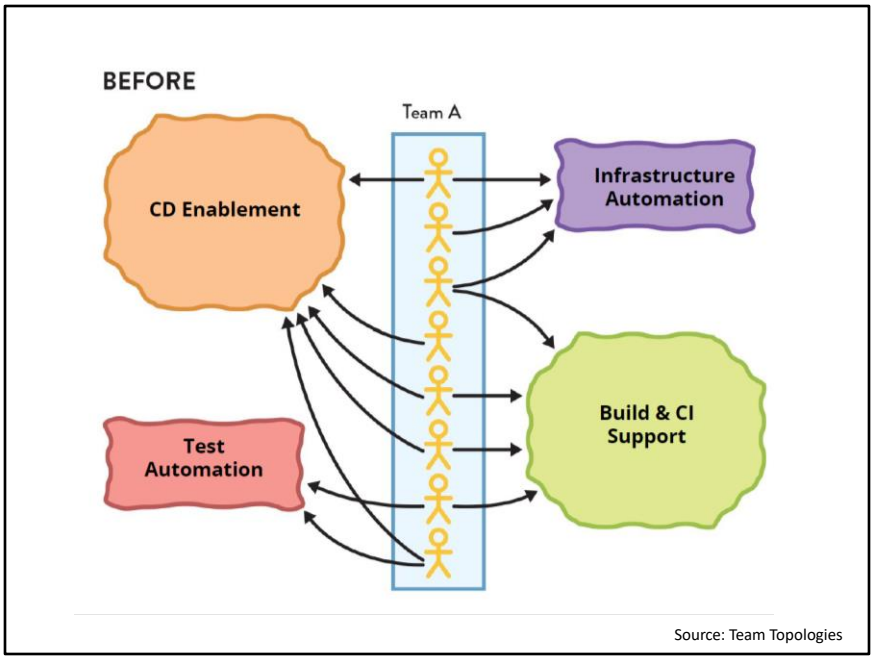
can connect their new features, and regression test their existing features, using the well-documented APIs, test environments, and test data beds that the integration team has built as enablement.

Over time, complicated subsystem teams should work towards becoming services, as we see here.

Our core product development team will also have periodic engagements with various enabling teams. In this example, our team is being helped by the internal user experience team, as they need some support to interview customers and prospects and observe and query users as they work on a new feature. The stream team has some of this capability embedded on the team in the form of business / system analysts, but for this new feature they need some additional expertise and need to be in context with other stream teams.

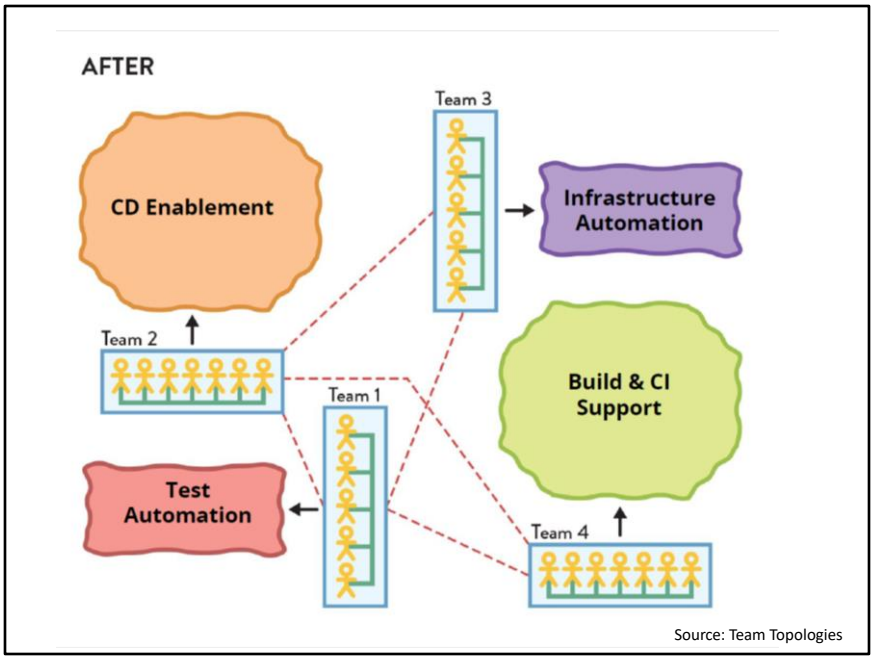
Finally, our stream team, and all likely all of the stream teams, is consuming Amazon as a service. There is an internal platform team which works to standardize use of Amazon, ie which of the multiple tools and approaches they prefer, how to control costs, etc., and sometimes the stream teams needs some facilitation / consulting as well.

Of course this is dramatically simplified, but you get the message. The stream team is in the center of everything. The other teams needs to sense what the stream teams need, get ahead of that if they can, and make it easy to consume interactions with them.



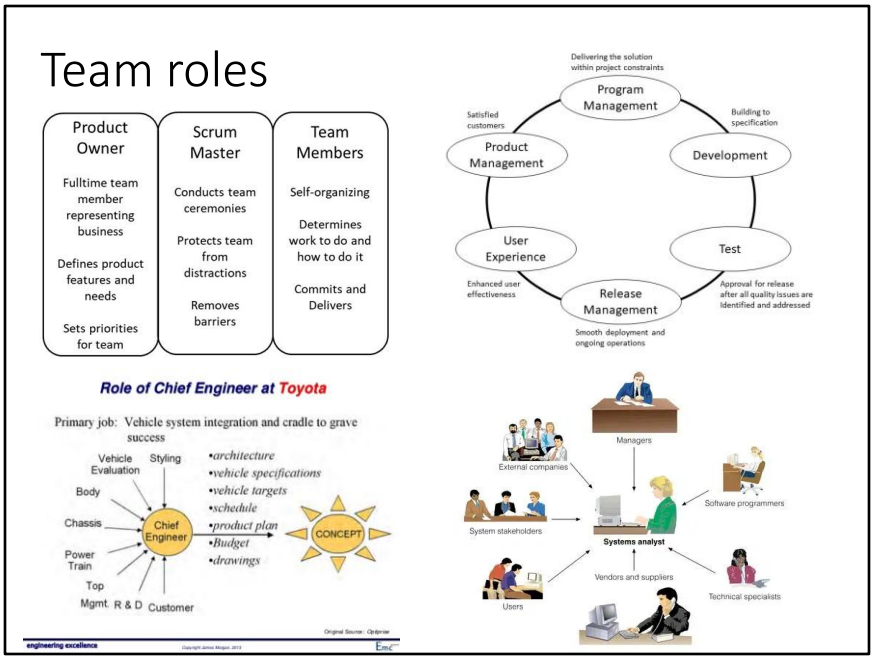
Of course, teams are not static. We are all evolving from where we are to where we would like to be. And as things change, we need to adjust. Lets look at one example from Outsystems that Manuel Pais talked about in a presentation.

Here, a stream-aligned team is supported by several different supporting teams, using some combination of APIs and facilitation. This diagram shows various team members on Team A interacting with one or more of these supportive teams. Lets postulate that Team has grown from its initial size of say 8 people to more than 20. The cognitive loan on Team A has grown, and its time to reduce it. Their ideas teaming blew.



Here Team A is split into four teams. They were able to do so in a way that created limited regular collaboration among the four teams in limited channels (reducing cognitive load), significantly limited intra-team communication, and limiting the interaction of each team with a few number of supporting teams (further reducing cognitive load). This is ideal, but very hard to do. How do you take one full life cycle stack stream aligned team and limit each's interactions as you split it?

Autonomy, mastery, & purpose. People were happier. Most of times people could work independently. Sometimes they needed to collaborate with the parallel stream teams.



From Don Reinertsen, a pioneer in product development. "If everyone is working on everything they need to communicate constantly. The team spends most of its time in meetings." "Common characteristic of organizations with poor role definitions"

I'd like to spend a little time on the team members themselves, because it does matter. I've made this chart a bit confusing and busy on purpose because this topic is that – confusing and actually quite religious to some. I posted a link to a podcast on the show "Master Business Analysis" with an introduction that "Although Scrum doesn't have a specific role for business/systems analysts, I am a big believer in the value of these (and other) specialized skills integrated properly onto delivery teams." Drew the expected comments from scrum lovers, that we shouldn't have such "intermediaries," that developer should work directly with users and product owners. Here, I'd like to give you my views.

Scrum roles: The only 3. Comment on the product owner. Why it was created in first place. How dreamy it is. That if it works we are back to waterfall. And it doesn't work. Scrum master – what is it…team manager (as at Sam's)? A low to midlevel conductor? Do they have domain knowledge? Then team members – are they undifferentiated and multi-talented, as Scrum and my critics preach? They just work it

out themselves? The gaps here are so broad! We can do better than this for our teams.

So lets look at another view, from Microsoft Solution framework back when Microsoft ruled the tech world. Here, the roles are needed but can combined for small teams, broken apart for larger ones. The key idea is the roles all overlap, and all are accountable for successful delivery, but we allocate focus areas to folks with specific skills, connections, knowledge, experience.

Product Mgt is kind of like product owner, but we split out user experience explicitly.

Program Mgt is kind of like scrum master, but here there is specific accountability to keep eagle eye on delivering within constraints. Interestingly, in this model the program manager owns the functional spec (precisely what to do) because here is where tradeoffs are made. Tell story of Pankaj on mortgage instant decision, and Jerry on small business loan instant approvals.

Development, we know. It says build to spec, but don't take that too literally.

Test. Love this one. Makes no sense to me that no one in scrum has this accountability. Aren't there specific skills and experiences and knowledge and connections needed? Of course there are. This is framed "approval for release after all quality issues are identified and addressed."

Release management. Again, scrum relegates this to just an unspecified dimension. In practice teams I've seen mostly have this specialized. We don't it total handoff, but we don't want lack of focus either.
Note: MSF is very similar to Marty Cagan's views in INSPIRED. Product, UX, & Tech are his three primary, then adds in things like test.

Ok, lets move on the systems analyst case. I know of several organizations that eliminated this role and called it a "junior product owner" or a "technical product owner." I don't see this role being eliminated in practice just hidden. Explain this vs a programmer…personalities, preferences, focus areas, skills, domain knowledge. A lot of programmers don't like talking to everyone, figuring out what is needed, documenting it, getting it through compliance and legal…they'd rather have a partner to do that and let them code.

Finally, lets talk about the chief engineer. On a scrum team, who is accountable for technical leadership? "The team." If everyone is accountable, no one is. Toyota role..Esteemed. Give examples of Mark, Jerry, James, Teresa.

Wrap the slide up. Its not one size fits all. But lack of specialized knowledge can equal ignorance. Create and use and evolve the right roles for your team and your organization.

My take on good roles: developer (various levels & skills); systems / business analyst; product manager (levels); project manager (as needed); user experience (not just UX, but includes training, completeness, etc); devops specialists (code mgt, release mgt, auto test integration); test management (strategies, environments, data, coordination, bug mgt, performance); chief engineer.

Formal Organizational Structures Matter.....A lot!

- Avoid over-centralization / over- standardization functionally
- Non-engineers should manage engineers only at top of business units
- Similarly, highly-skilled functional specialists need to be managed by someone in that field (e.g. legal, finance)
- Members of flow teams should report to as few managers as possible, given above constraints.
- Ensure you have delivery leadership: Chief Engineer (lean) or Tech Lead (Cagan).

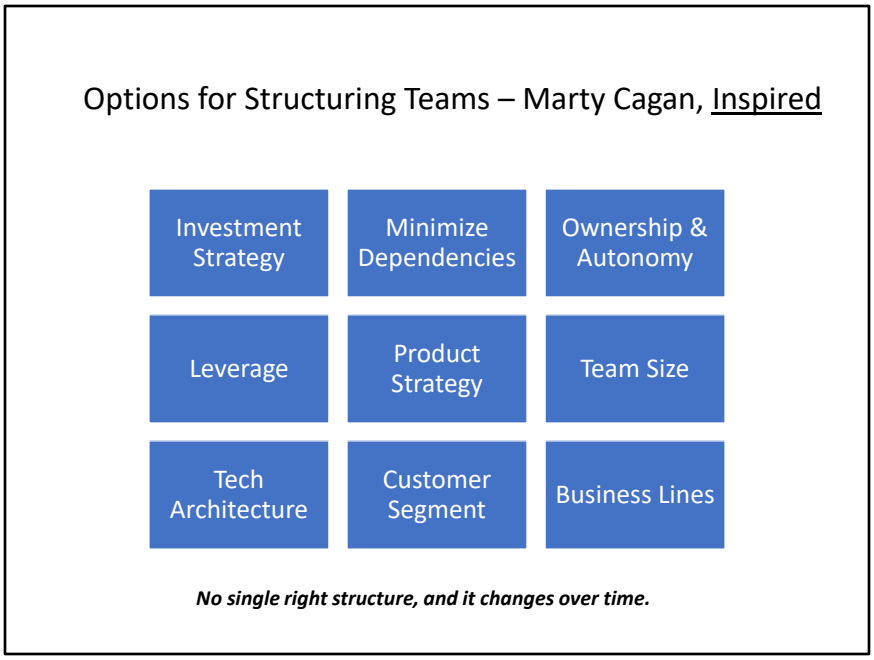
Formal organizational hierarchies
On teams, who reports to whom? Who hires? Who assigns? Who evaluates? Who allocates resources and priorities? Who makes deals, chooses partners, etc?

First, it matters!
Second, some rules:

1) Centralize functions as little as possible. EG a centralized UX dept staffing all the teams is not a good idea. A centralized facilitating team can be ok depending on commonalities (eg many business units sharing common customers). But definitely NOT if its about just sharing expertise or saving money
2) Don't have engineers report non-engineers at too low a level. At CIO level, of course; division CIO, sure. Much less than that be very careful! Certainly NOT at team level.
3) To extent you can, flow teams need to report MOSTLY to one manager. Some actually have scrum master as manager. Not advocating that, but it can work well. Something like engineers (dev, test, pm, devops) to one is good, then product mgt, sales, compliance, risk, etc. can be to another in business line(s) if fits.
4) Expertise really does matter. So having functions like compliance, Legal, HR, etc.

reporting functionally and allocated is ok.

5) When you have functions reporting at low level, eg to team leads or one above, you need to care for expertise and sharing. Spotify guilds. Toyota functional alignments. Product Dev at Toyota vs. Chrysler, etc.
6) Everyone needs a CTO. Cagan calls these tech leads. They have to be aligned with product/business leadership.



Investment Strategy: You need to have one, and teams need to reflect that. What are trying to do from business perspective?

Dependencies. We've talked about that in Team Topologies work.

Ownership. Want missionaries not mercenaries.

Leverage. Two edge sword. Definitely don't want teams reinventing wheels, but don't want too many interdependencies. Over time this grows. Common needs across teams. Eg in multiproduct line company, tech dealing with customer admin should be shared.

Product Vision. Similar to investment strategy. Vision is direction, strategy is milestones.

Team Size…dealt with

Architecture. Remember lean product dev goal of TOWERING TECHNICAL EXPERTISE. This matters esp for engineering. Idea of full stack engineer is cool but often not practical. Similar to the idea of Leverage.

Customer Segment. Also Value Stream is similar idea. EG in bank, consumer vs wholesale. Or Uber, driver vs rider.

Business Lines. If tech is independent, treat as separate companies, just sharing admin systems. If we

share common components, eg in bank, DDA is shared across everyone, as is customer and online foundation, then we need to mix structures. Might have product teams for shared, and flow teams for business lines.