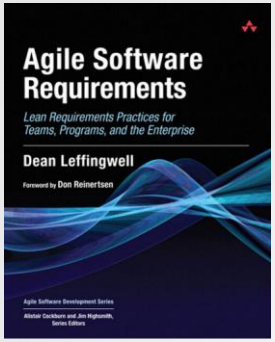


Agile Software Requirements

Software Requirements Engineering – 40688

Computer Engineering department

Sharif university of technology

Fall 402

Chapter 4:

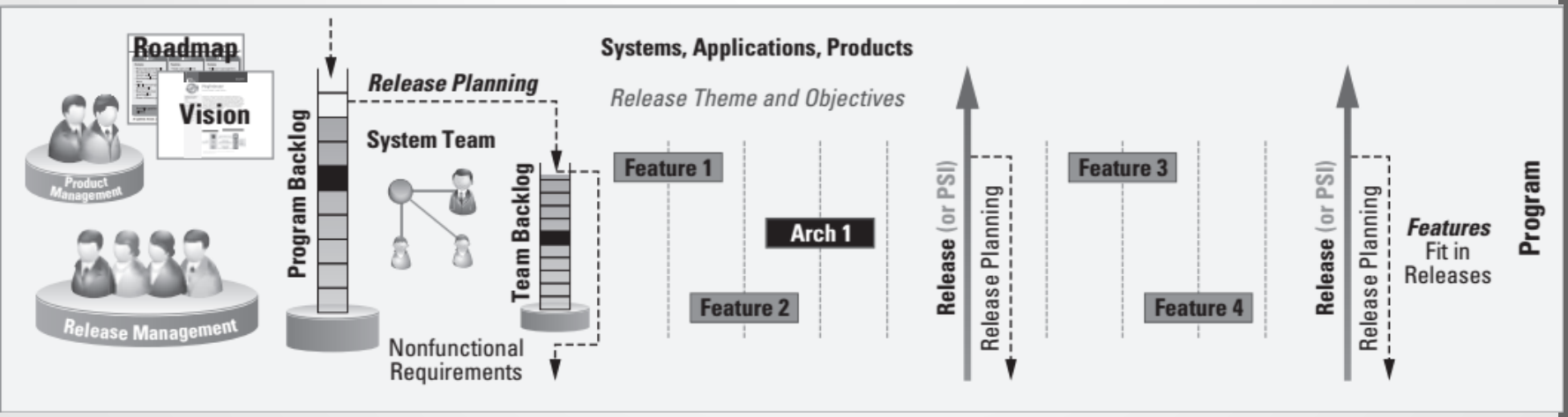
Agile Requirements for The Program

Contents (1)

Contents (2)

Program Level (1)

At the Program level, however, the problem changes, and the enterprise faces an additional set of challenges to successfully execute agility at this next level of scale.



Program Level (2)

The objectives at this level include the following:

1. **Maintaining Vision and Roadmap**
2. **Release management**
3. **Quality management**
4. **Deployment**
5. **Resource management Eliminating impediments**

Organizing agile teams at scale (1)

How to organize the agile teams in order **to optimize value delivery of requirements**?

- For the **smaller enterprise**, this is usually **no issue** at all. They will organize naturally around the few products or applications that reflect the mission.
- At scale, however, like most other things agile, the problem is different, and the challenge is to understand **who** works on **what** and **where**. Do we organize around features, components, product lines, services, or what?

Organizing agile teams at scale (2)

Although there is no easy answer to this question, the question must be explored because **so many agile practices** must be reflected in that decision; like:

- How many backlogs there are?
- And who manages them?
- How the vision and features are communicated to groups of teams?
- How the teams coordinate their activities to produce a larger solution?

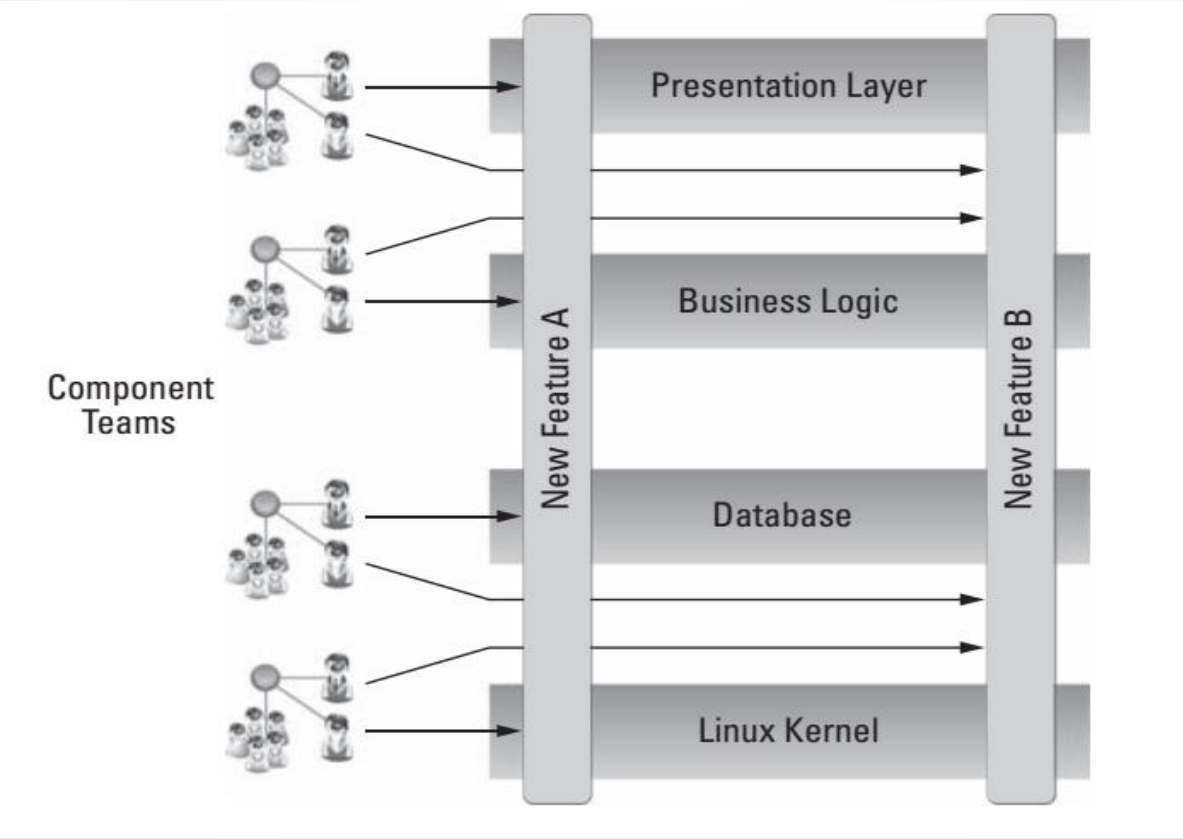
Component Teams (1)

In Scaling Software Agility, we described a typical organizational model whereby many of the agile teams are organized around the **architecture of a larger-scale system**.

- There, they leverage their technical skills and interest and focus on building robust components, making them as reliable and extensible as possible, leveraging common technologies and usage models, and facilitating reuse.
- We even called the team's define/build/test "component" teams, which is (perhaps) an unfortunate label.

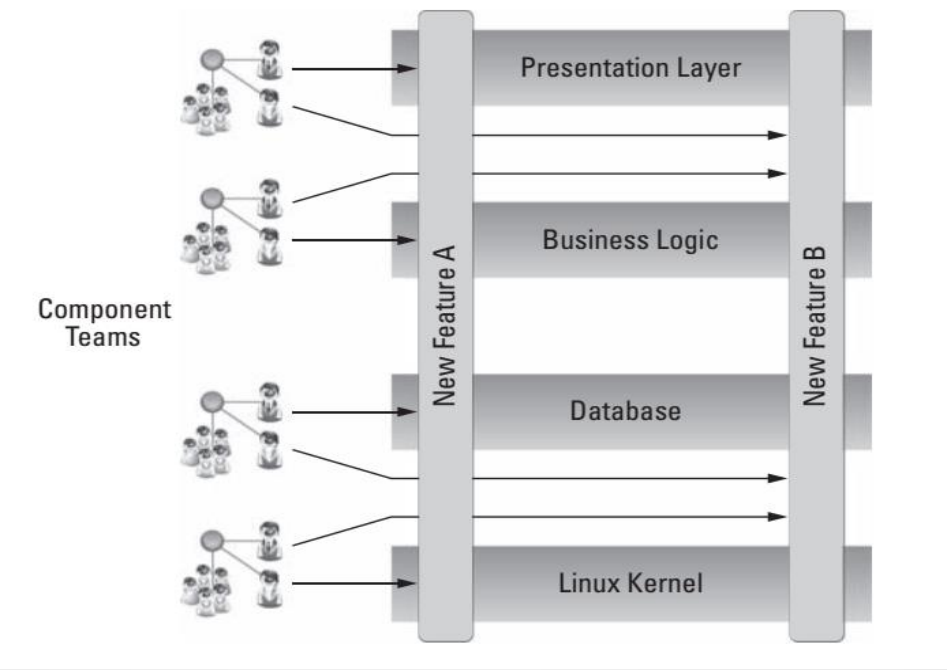
Component Teams (2)

In any case, in a component-based approach, the development of a new feature is implemented by the affected component teams.



Component Teams (3)

- A **new feature** requires the creation of **new backlog items** for each team that contributes to the feature.
- Component teams **minimize multiplexing** across features by implementing them **in series**, rather than parallel.



Component Teams (4)

Some **advantages** are obvious:

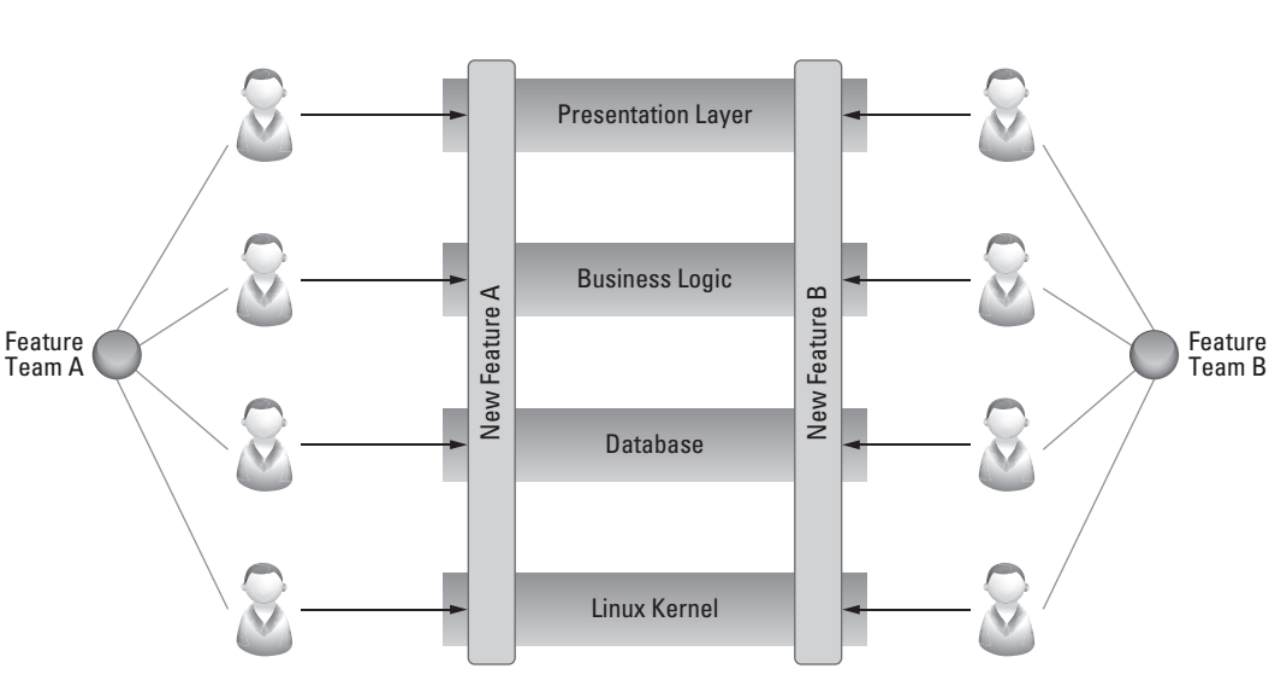
- Each team is able to aggregate the needs of multiple features into the architecture for their component and can focus on building the best-possible, long-lived component or service for their layer.
- Their component does not get "sliced and diced" by each new feature; rather, it evolves as a set of services to implement current and, ideally, future features.
- This approach may be reflective of an architecture-centric bias when building these largest-of-all-known software systems. That is because if you don't get the architecture reasonably right, you are unlikely to achieve the reliability, performance, and longer-term feature velocity delivery goals of the enterprise.

Component Teams (5)

- Based on its past successes, the enterprise may already organized that way, with specialists who know large-scale databases, web services, embedded operating systems, and the like, working together.
- These teams may already be co-located, simplifying communication and reducing the batch handoff of requirements, design, and test data.
- Technologies and programming languages may differ across components, making it difficult for feature teams to do pairing, collective ownership, continuous integration, test automation, and other factors.
- And finally, at scale, a single user feature can be an awfully big thing that could easily affect hundreds of practitioners.

Feature Teams (1)

The almost universally accepted approach for organizing agile teams is to organize around features.



Feature Teams (2)

The advantages to a feature team are:

- Teams build expertise in the actual domain and usage mode of the system and can typically accelerate value delivery of any one feature.
- There is less overhead, because teams don't have to pass backlog items back and forth to see that a feature is implemented
- There are far fewer interdependencies across teams.
- Planning and execution are leaner.

Sometimes the Line Is Blurry

- Even in light of this advice, we must also recognize that features and components are both abstractions, and the line is not so clear.
- For example, Trade Station Securities builds an online trading system where "charting" is a key capability for the trader. A few co-located agile teams work together on the charting function. On the surface, that looks like an excellent example of a feature team, because charting certainly is a major feature of the system.
- When new online trading capabilities are developed, such as "trading foreign exchange currencies (Forex)," new chart functionality must be added. However, driving this new chart functionality are major components such as streaming data, account management, and interfaces with Forex exchanges. Is the new feature value stream described as "trading Forex all the way through the specialty chart function?" If so, that would make an obvious vertical feature stream, and the teams might reorganize by taking some members of each component team and creating a new vertical feature team for Forex trading.

Lean Toward Feature Teams

But with agile's focus on immediate value delivery, there is an appropriate leaning toward feature teams. Mike Cottmeyer2 points out this:

> *"I tend to start with the feature team approach and only move toward components if I have to ...but the decision is situation- specifc."*

- To make this decision, you'll have to explore the diversity of your technology:

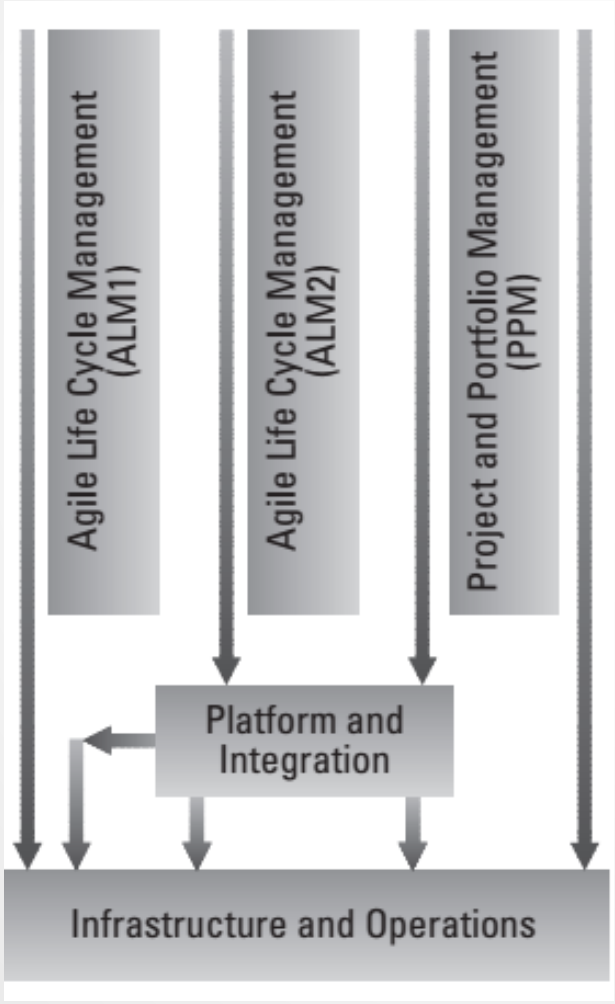
  how well your system is designed
  what tools you have to
  manage your code base
  the size and competence of your team
  how and where your teams are distributed and
  the quality of your infrastructure automation.

- You need to take a hard look at what scale your feature teams WILL break down ...at some scale they WILL break down. Is scaling to this level something we need to address now or can it wait?

Likely a Mix

- In the larger enterprise where there are *many* teams and *many, many* features, one should consider the previous factors and then select the best strategy for your specific context. In most cases, as you can see, the answer will likely be a mix of feature teams *and* component teams.
- Indeed, even in the modest-sized agile shop, **a mix** is likely to be appropriate.

Ryan Martens
founder of Rally Software

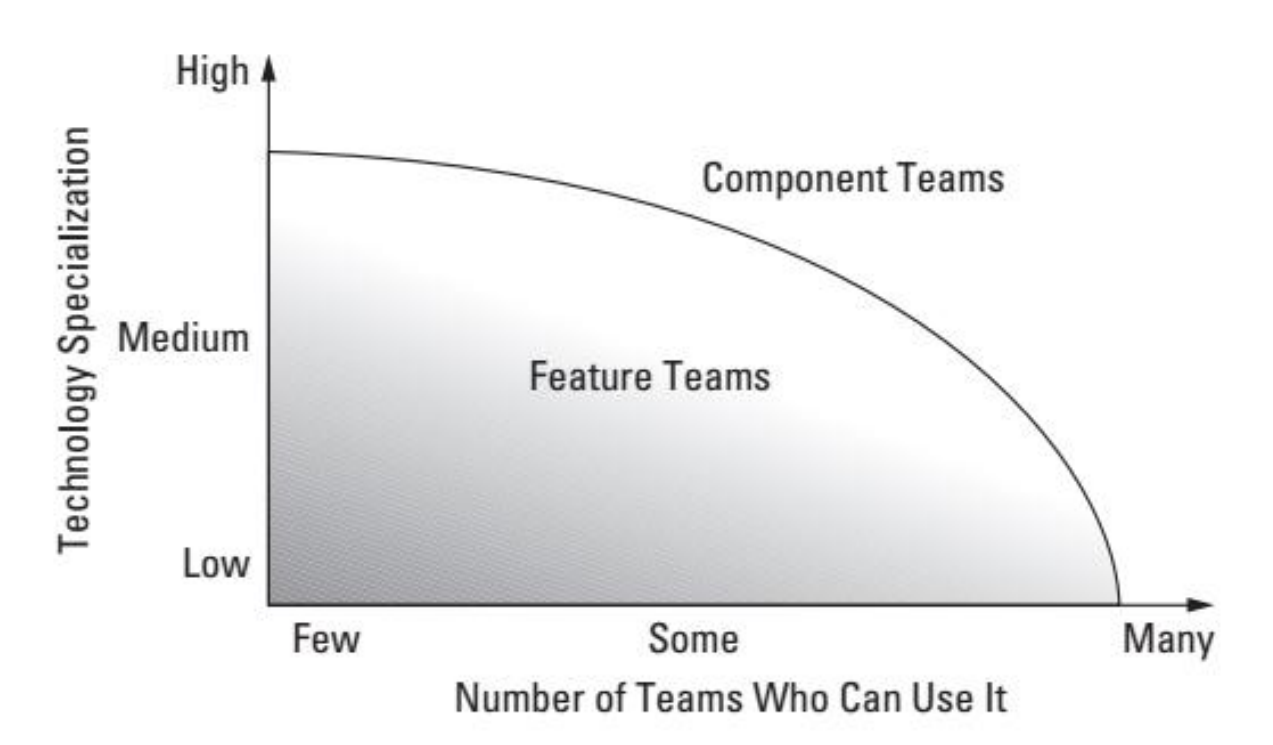


Ryan Martens
founder of Rally Software

While we don't think of it in these terms as such, three of these teams (ALM1, ALM2, and PPM in the top) would be readily identifiable as feature teams. One (I&O on the bottom) is clearly a component team. I don't know what you'd call the one (Platform and Integration) in the middle, because it sometimes originates its own features and sometimes is simply a supportive component for other features.

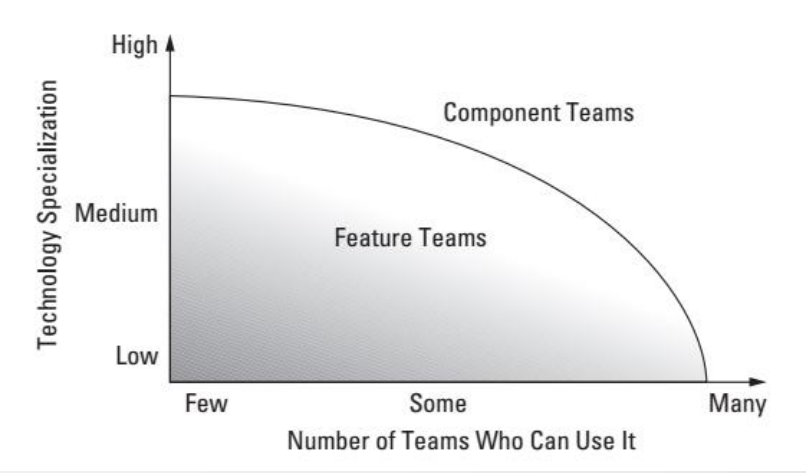
Two main factors that drive the mix

The practical limitation of the degree of specialization required and the economics of potential reuse.



Co-location

- If it still isn't obvious which way you might want to organize, then you may want to optimize around **co location**, because the communication, team dynamics, and velocity benefits of co-location may well exceed the benefits of the perfect component or feature organization.
- First, apply feature teams if *they* are, or can readily become, co-located. If not, and you find yourselves on the outer edges of the curve in the Figure, then apply component teams, especially since it is likely that the teams are already organized that way.



System-Level Testing

- Ideally, each team would have the ability to test all the features at the system level.
- Many feature teams do have such capabilities, and that's one of the reasons why feature teams work so well.
- However, the fact is that it is often not practical for an individual feature or component team to be able to test a feature in its full system context.
- Many teams may not have the local resources (test bed, hardware configuration items, other applications, access to third-party data feeds, production simulation environment) necessary to test a full system.

System Quality Assurance

- Similarly, many teams do not have the specialty skills and resources necessary to test some of the nonfunctional and other quality requirements for the system.
- This may include **load** and **performance testing**, **reliability testing**, conformance to industry compliance standards, and so on.
- Indeed, simply running a full validation suite on a large-scale system may even require a small but dedicated team who constantly updates the full system verification and test platforms and runs the validation.

Release Management Team (1)

There is no standard convention for its name, but it takes on a release management team or steering committee function. Members of this team may include key stakeholders of the Program level of the enterprise:

- Line-of-business owners and product managers
- Senior representatives from sales and marketing
- Senior line managers who have responsibility for the teams and are typically ultimately accountable for developing the solution for the marketplace.
- Internal IT and production deployment resources.
- Senior and system-level QA personnel who are responsible for the final assessment.
- System architects, CTOs, and others who oversee architectural integrity.

Release Management Team (2)

Weekly meeting of the team for addressing:

- Do the teams still clearly understand their mission?
- Do we understand what they are building?
- What is the status of the current release?
- What impediments must we address to facilitate progress?
- Are we likely to meet the release schedule, and if not, how do we adjust the scope to assure that we can meet the release dates?

Vision (1)

The Vision addresses the larger questions, including the following:

1) What is the **strategic intent** of this program?
2) What **problem** will the application, product, or system solve?
3) What **features** and **benefits** will it provide?
4) **For whom** does it provide it?
5) What **performance**, **reliability**, and so on, will it deliver?
6) What platforms, standards, applications, and so on, will it **support**?

Vision (2)
Requirements specification documents

Agile teams take a variety of approaches to communicating the Vision. These include the following:

- Vision document
- Draft press release
- Preliminary data sheet
- Backlog and Vision briefing

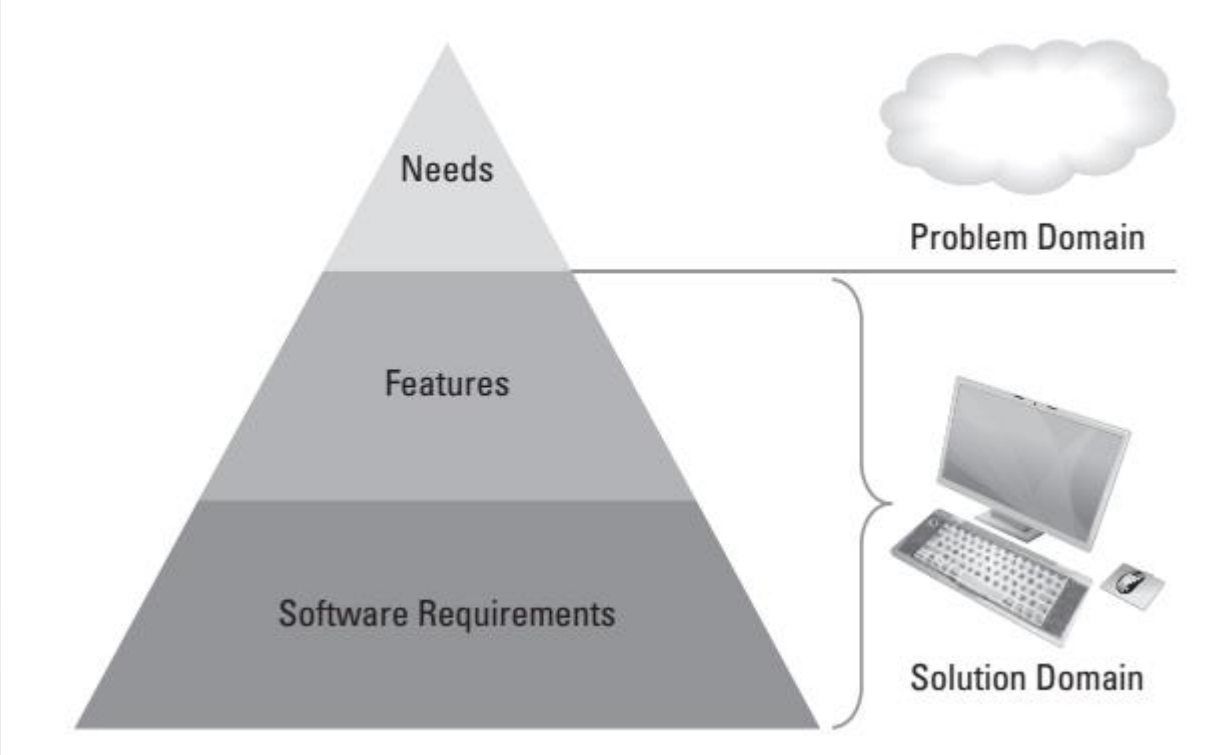
Features (1)

- In Leffingwell [2003], features were described in the following way:

✓ *Features are services provided by the system that fulfill stakeholder needs.*

- Features live at a level above software requirements and **bridge the gap from the problem domain** (understanding the needs of the users and stakeholders in the target market) **to the solution domain**.

Features (2)


Needs
Features
Software Requirements
Problem Domain
Solution Domain

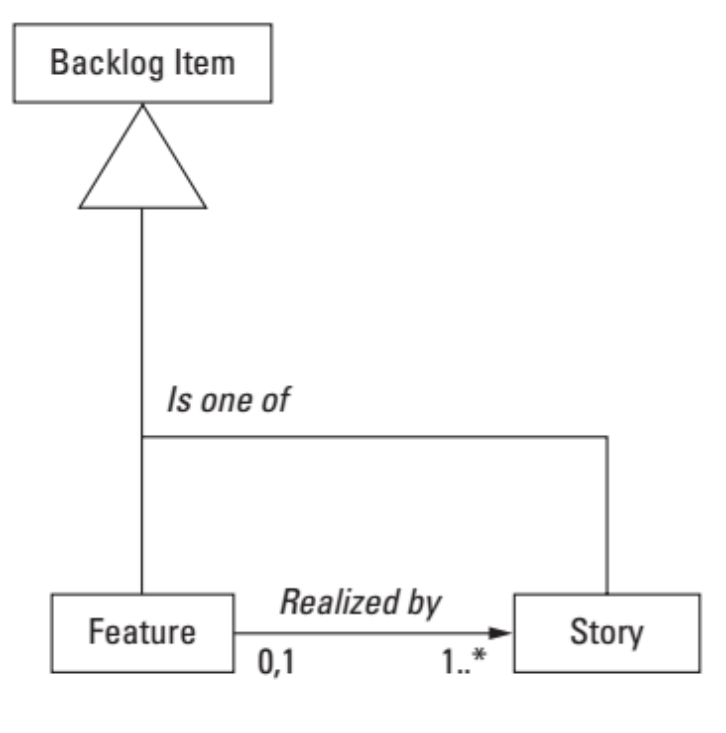
Program Backlog (1)

- Features, then, are first-class citizens of our agile requirements model.
- Features are realized by **stories**.
- At release planning time, features are decomposed into stories, which the teams use to implement the functionality of the feature.
- Features are typically expressed in **bullet form** or, at most, **in a sentence or two**.

Program Backlog (2)

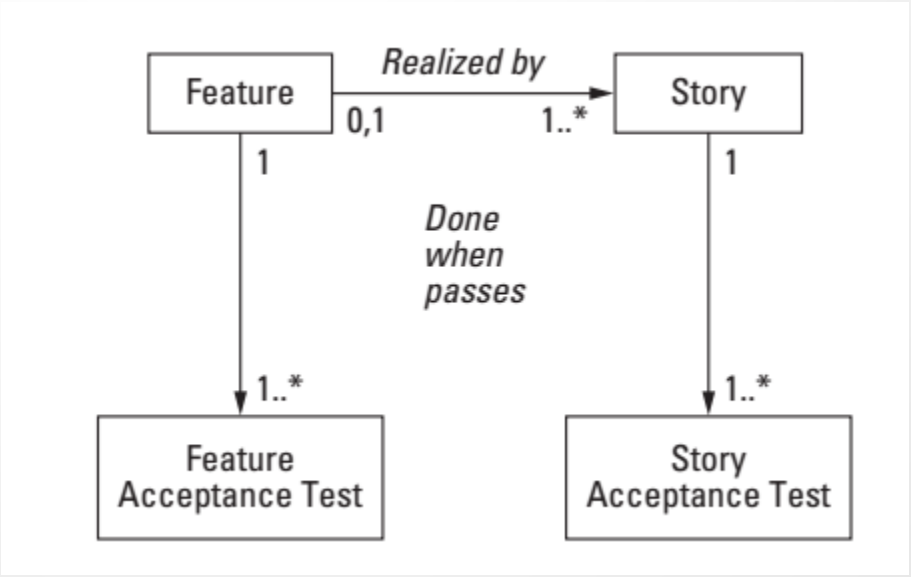
Provide "Stars" for special conversations or messages, as a visual reminder that you need to follow up on a message or conversation later.

Introduce "Labels" as a "folder-like" conversation-organizing metaphor.



Testing Features

- All code is tested code.
- At the Program level, the question arises as to whether features also deserve (or require) acceptance tests.
- Every feature requires one or more acceptance tests, and a **feature can also not be considered DONE until it passes**.



Nonfunctional Requirements

From a requirements perspective so far, we've used the feature and user story forms of expression to describe the functional requirements of the system.

- Traditionally, these were often described as the **system qualities**.
- **quality**, **reliability**, **scalability**, and so on—and they are critical elements of system behavior.

Nonfunctional Requirements as Backlog Constraints (1)

From a requirements modeling perspective, we could just throw the NFRs into the program backlog, but they tend to behave a little differently.

- New **features** tend to enter the backlog, get implemented and tested, and then are simply deleted.
- **NFRs** *constrain* new development, thereby eliminating some degree of design freedom that the teams might otherwise have.

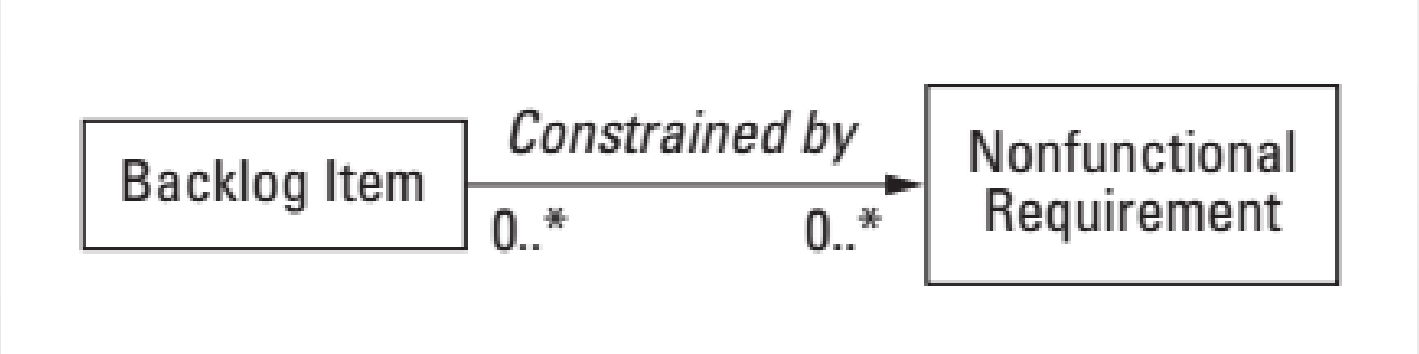
For partner compatibility, implement SAML-based single-sign-on (NFR) for all products in the suite.

Nonfunctional Requirements as Backlog Constraints (2)

In other cases when new features are implemented, existing NFRs must be revisited, and system tests that were previously adequate may need to be extended.

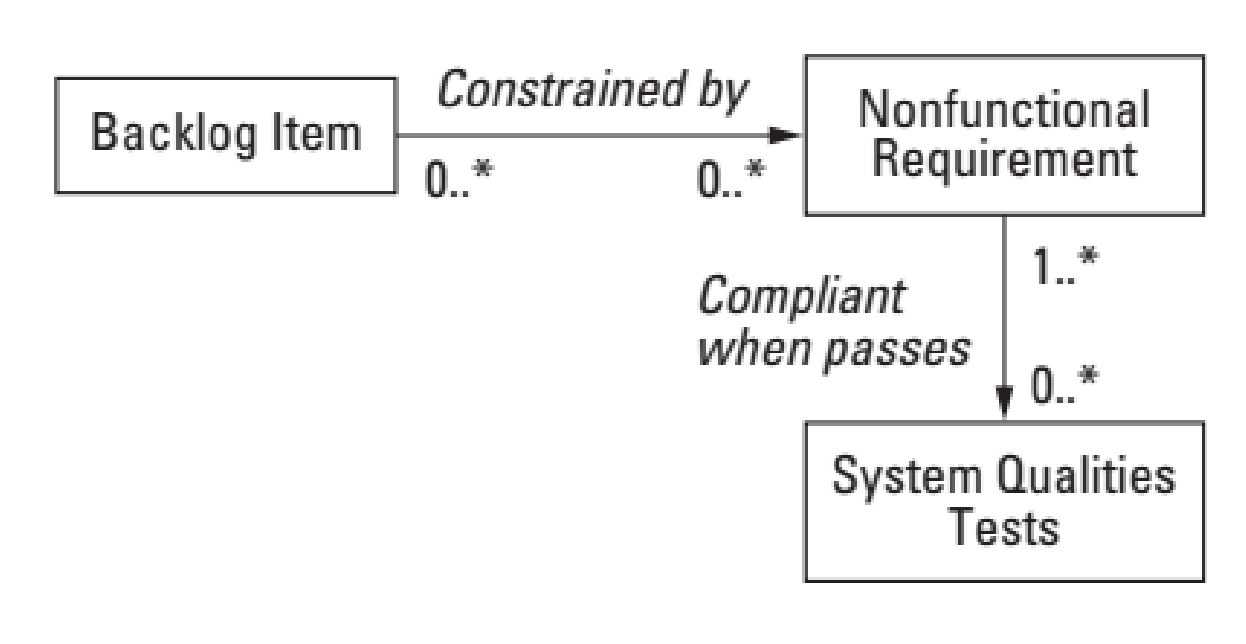
- ✓ *The new touch UI (new feature) must still meet our accessibility standards (NFR).*

So, in the requirements model, we modeled NFRs as backlog constraints.



Testing Nonfunctional Requirements

These types of requirement like **usability**, **reliability**, **performance**, **supportability**, and so on are often described as the "**ilities**" or **qualities** of a system.



Releases and Potentially Shippable Increments

- The development of system functionality is accomplished via multiple teams in a synchronized **Agile Release Train (ART);** A standard cadence of timeboxed iterations and milestones that are date- and quality fixed but scope variable.
- The ART produces **releases or potentially shippable increments (PSIs)** at frequent, typically fixed, 60- to 120-day time boundaries.
- The **PSI** is to the enterprise what iterations are to the team, in other words, the basic iterative and incremental cadence and delivery mechanism for the program (an "**ubersprint**").

Release Planning (1)

Release planning is the **periodic program activity** that aligns the teams to a **common mission**. During release planning, teams **translate the Vision into the features and stories** they will need to accomplish the objectives.

These events gather the stakeholders to address **the following objectives**:

1) Build and share a common Vision.
2) Communicate market expectations, features, and relative priorities for the next release.
3) Plan and commit to the content of the next release.
4) Adjust resources to match current program priorities.
5) Evolve the product Roadmap
6) Reflect and apply lessons learned from prior releases.

Release Planning (2)

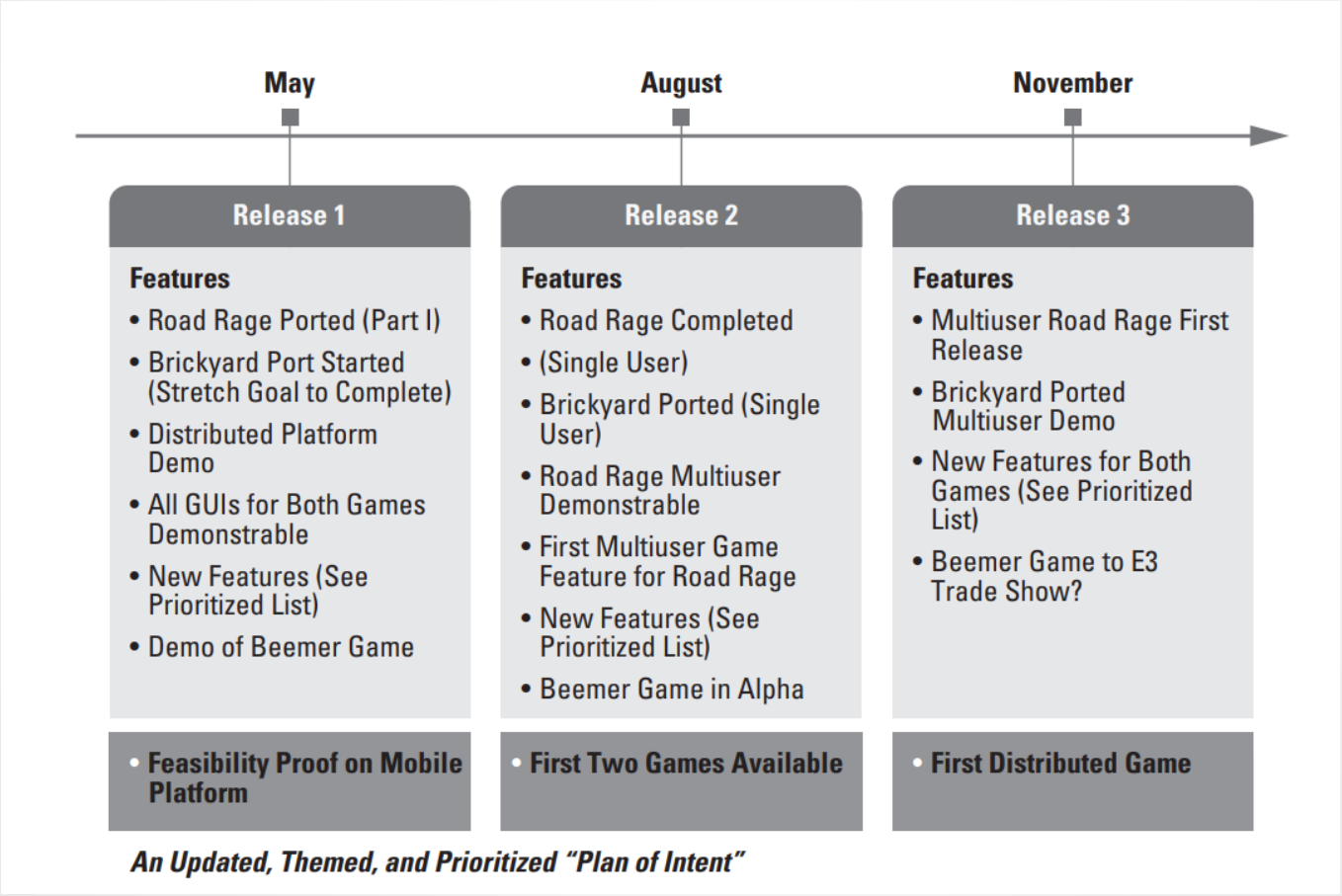
The frequency of the event depends upon the company's required responsiveness to market conditions and the iteration and release cadence it has adopted. In most enterprises, it occurs **every 60** to **120** days with a **90-day cadence** being typical.

Road Map (1)

- When we described the **Vision**, it was presented as **time-independent**; in other words, the Vision describes the **objectives of the product or system** without any binding to time.
- The **Roadmap** consists of **a series of planned release dates**, each of which has a theme and a prioritized feature set.

Road Map (2)

Example product roadmap for a hypothetical gaming company



An Updated, Themed, and Prioritized "Plan of Intent"

Summary

- ✓ In this chapter, we introduced new requirements roles, artifacts, and processes that are necessary to apply agile development in programs that require many teams.
- ✓ We described how to organize the teams to optimize value delivery.
- ✓ We introduced a number of new requirements artifacts—Vision, features, nonfunctional requirements, and Roadmap—and described how teams use these artifacts to communicate the larger purpose of the product, system, or application they are developing.
- ✓ We also described how teams aggregate a series of iterations to build PSIs, or incremental releases, via an Agile Release Train, which incrementally delivers value to the users and customers. In the next and final chapter of Part I, we'll increase our level of abstraction one last time and introduce a set of requirements practices suited to building a portfolio of products and services suitable to the needs of the larger enterprise.