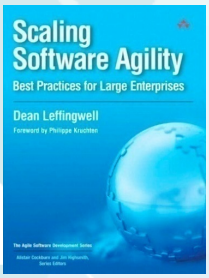
Scaling Software Agility: Rearchitecting Enterprise Class Systems

An Agile Enterprise Trifecta



By Dean Leffingwell

May, 2010

About Dean Leffingwell

Author

Coach

Agile Enterprise Coach
Continuing…medium size, large and very large software enterprises...

Executive Mentor
BMC Agile Transformation

Cofounder/Advisor
Ping Identity, Roving Planet, Rally Software

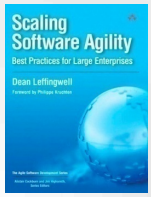
Executive

Founder and CEO
ProQuo, Inc., Internet identity

Senior VP
Rational Software
Responsible for Rational Unified Process (RUP) & Support of UML

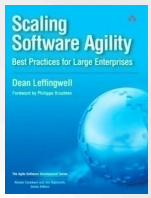
Founder/CEO
Requisite, Inc.
Makers of RequisitePro

More from Dean Leffingwell

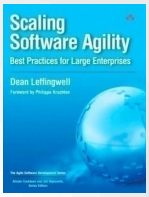


- Books
  - Coming Soon: *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*
    - (Note: much of this presentation is based on this upcoming book)
  - *Scaling Software Agility: Best Practices for Large Enterprises*
- Blog and Resources
  - www.scalingsoftwareagility.wordpress.com
- Reach me at DeanLeffingwell@gmail.com

Rearchitecting with Flow – An Agile Enterprise Trifecta



① Lean and Scalable Requirements Model
② The Agile Release Train
③ An Architectural Epic Kanban System



#1 – A LEAN AND SCALABLE REQUIREMENTS MODEL

- REASONING ABOUT SMALL AND BIG THINGS

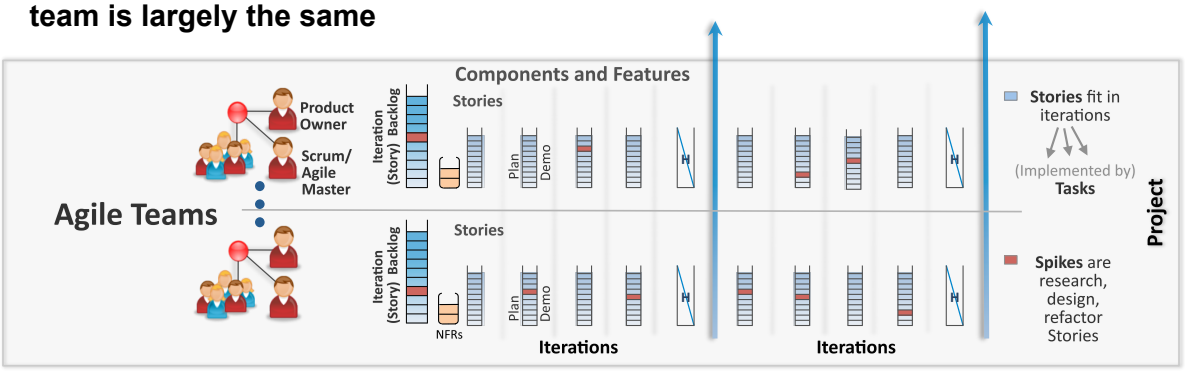
The Agile Team in The Enterprise

There can be a large number of teams in the enterprise

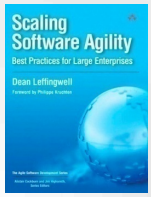
"pods" of 5-10 teams building a feature, component, or subsystem is not unusual

Some product lines require 30-40-50 teams to build

However, the structure of each team is largely the same



Their work is based on the user story

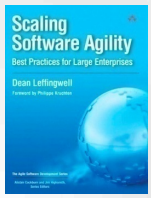


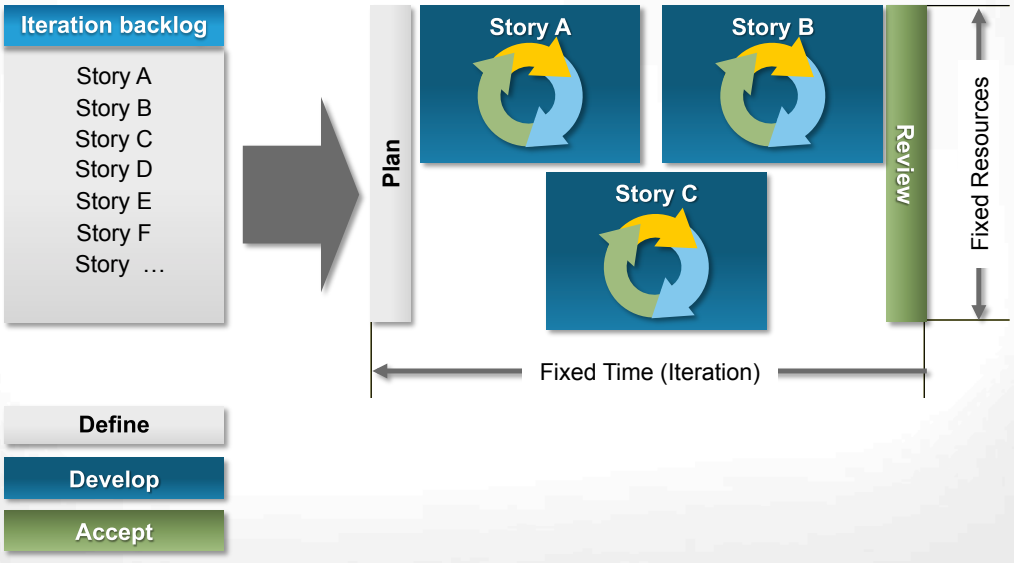
User Story

As a <role>
I can <activity>
So that <business value>

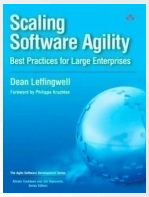
"As a Gmail user, I can select and highlight a conversation for further action"

Stories drive iterations





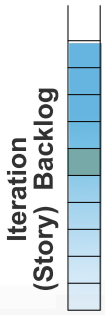
Stories are maintained in the teams backlog

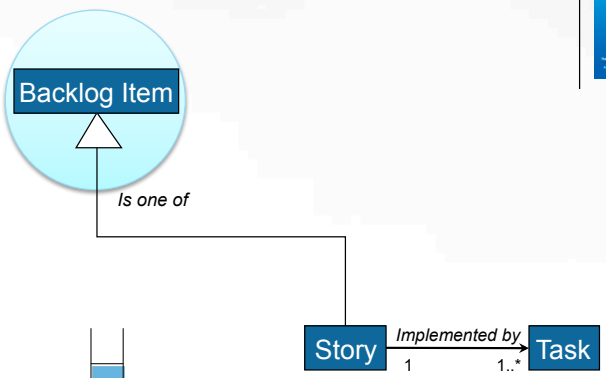


There is only one backlog for the team

All work comes from the backlog

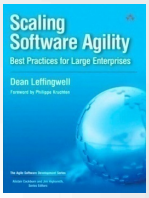
If isn't a user story (defect, etc) it still goes in the backlog





"If there isn't a story in the backlog, it ain't gonna happen"

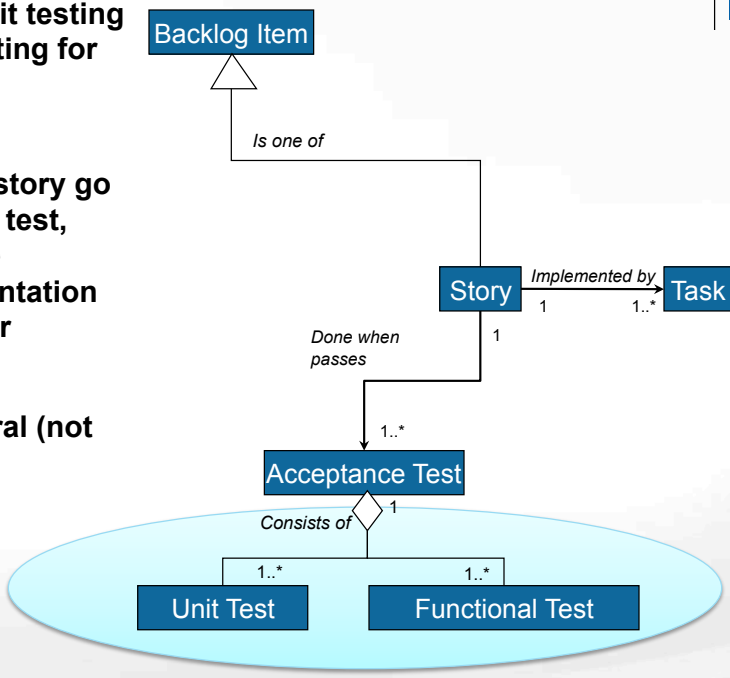
A test and quality-centric approach



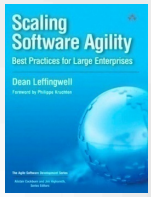
Teams perform unit testing <u>and</u> functional testing for every story

The details of the story go into the functional test, where they are the persistent representation of system behavior

Stories are temporal (not maintained after implementation)

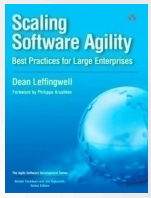


Scaling requires rethinking



- Assume a program requires
  - 200 practitioners, (25 agile teams) to deliver a product
  - The enterprise delivers software every 90 days in five, two week iterations.
  - Each team averages 15 stories per iteration.
  - Number of stories that must be elaborated and delivered to achieve the release objective = 25*5*15= **1,875!**
- How is an enterprise supposed to reason about things?
  - What is this new product going to actually do for our users?
  - If we have 900 stories complete, 50% done, what do we actually have working? How would we describe 900 things?
  - How will we plan a release than contains 1,875 things?
- And, what if it took 500 people?

And further

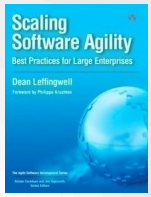


- And, even if I know 100 things that “as a <role> I can <activity> so that <business value>”, can do

  what *Features* does the system offer to its user and what *benefits* does it provide?

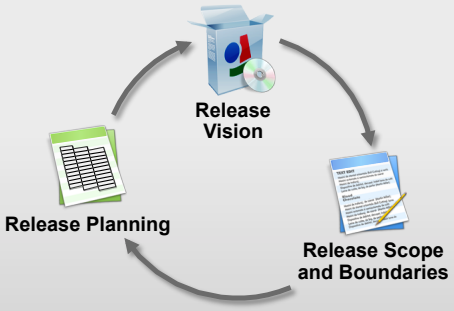
Feature	Benefit
Stars for conversations	Highlight conversations of special interests
Colored label categorization	Easy eye discrimination of different types of stories (folder like metaphor)
Smart phone client application	Faster and more facile use for phone users – ease adoption

So we need an additional level of planning



Features

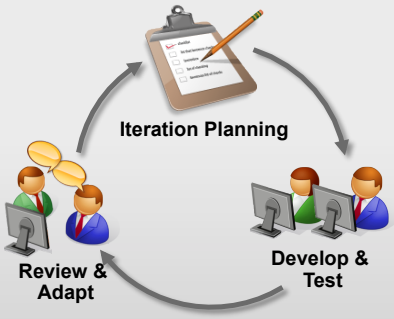
Product & Release Cycle



Drives

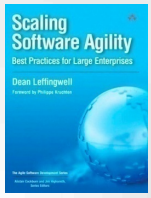
Stories

Iteration Cycle



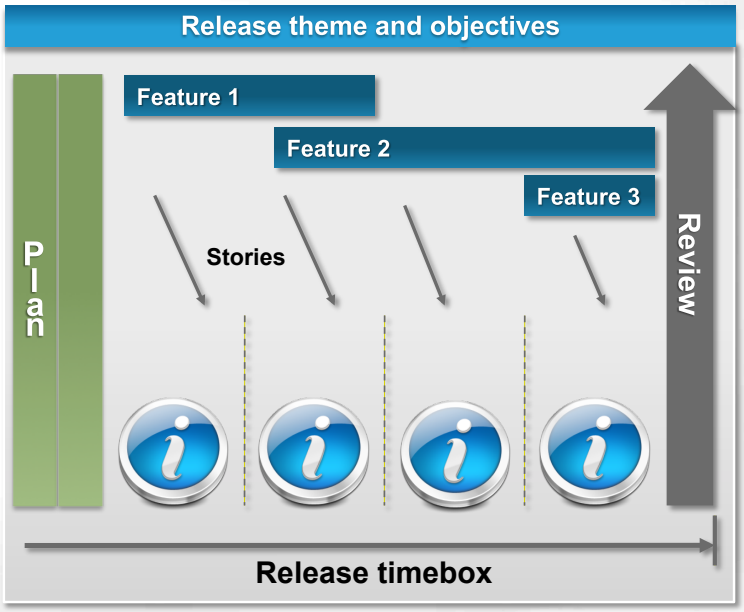
Feedback
- Adjust

Which creates an iteration and release pattern

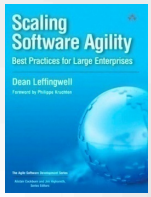


Prioritized Release (feature) Backlog

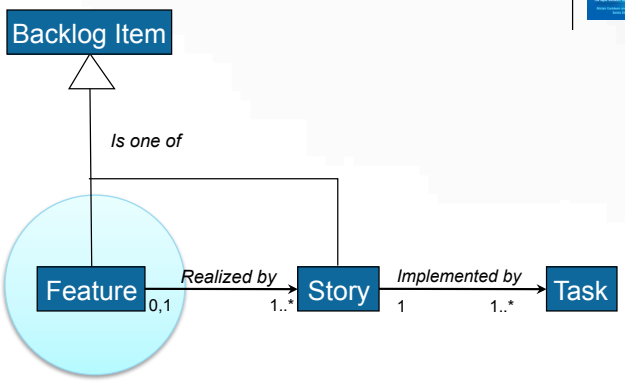
- Feature 1
- Feature 2
- Feature 3
- Feature 4



So we need to extend the information model



Features are another kind of Backlog Item

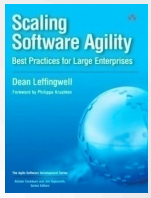


Introduce Gmail "Labels" as a "folder-like" conversation-organizing metaphor.

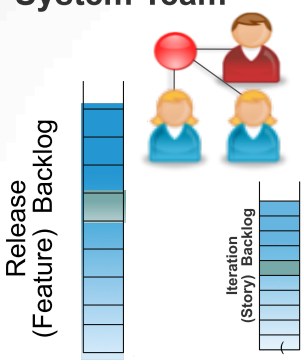
Or:

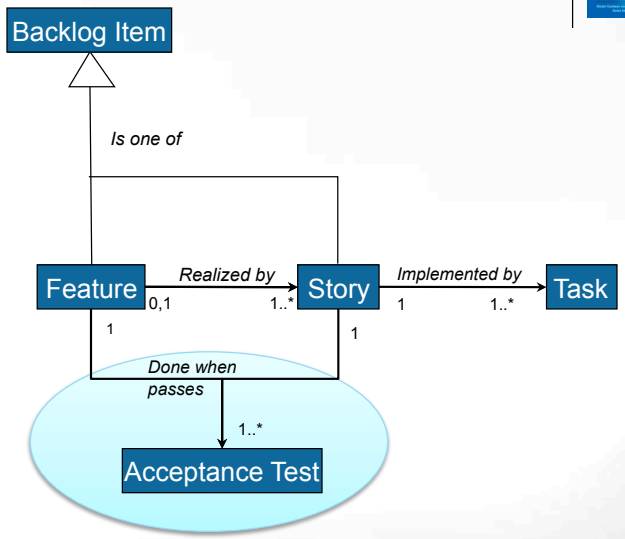
As a modestly skilled user, I can assign more than one colored label to a conversation so that I can see a conversation from multiple perspectives

Features also require testing



System Team



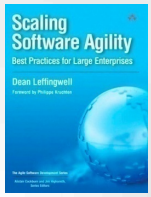


And maybe a new team …..

Features typically span <u>many</u> teams

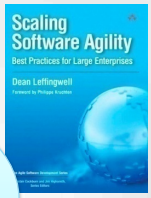
Sometimes, a special team is dedicated for the purpose of testing system level features

What about non-functional requirements?

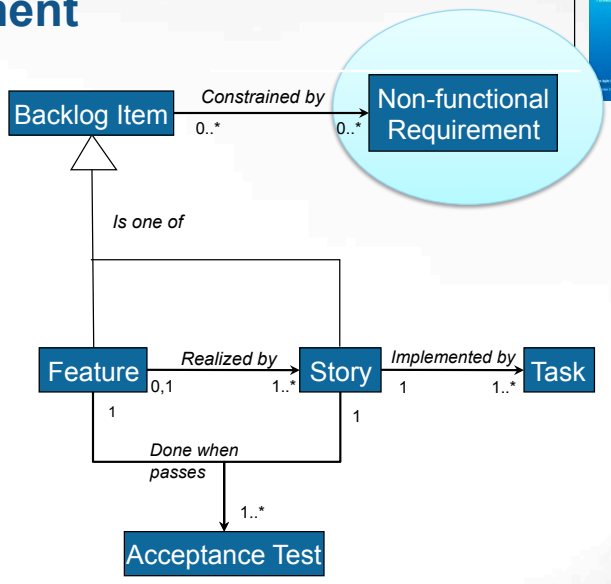


- Features and user stories express functional requirements
- But other requirements (NFRs) determine system quality as well:
  - Performance, reliability and security requirements
  - Industry and Regulatory Standards
  - Design constraints, such as those that provide common behavior across like components
- Typically, these system level qualities
  - Span multiple components/products/applications/ services/subsystems
  - Can often only be tested at the system level

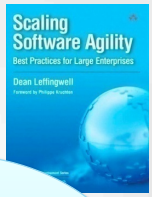
NFRs can be considered as constraints on new development



"When we add labels to conversations, we still have to meet the accessibility standards."

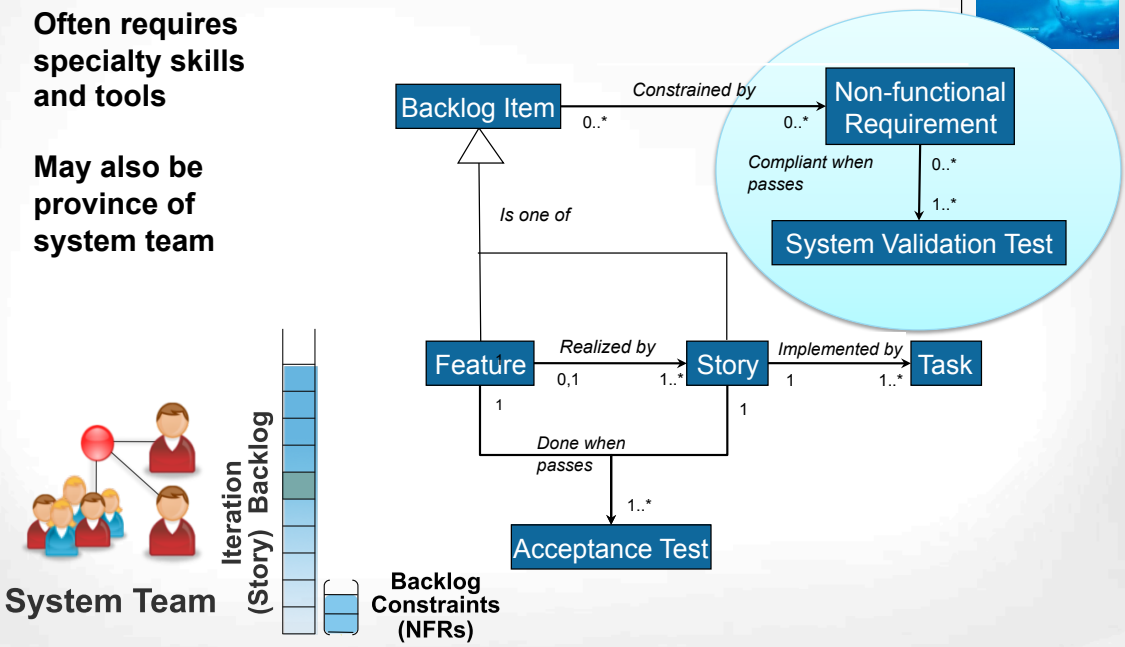


Which must also be tested

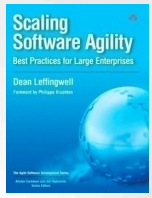


Often requires specialty skills and tools

May also be province of system team

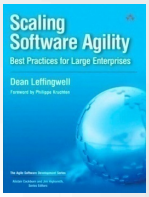


At the enterprise portfolio level, even system features are too fine grained



- There may be dozens of concurrent programs
- Each delivering dozens of features to market
- How do portfolio managers and system architects communicate the sweeping, larger scale initiatives that drive those programs?
- We use the word “Epic” to describe this content type

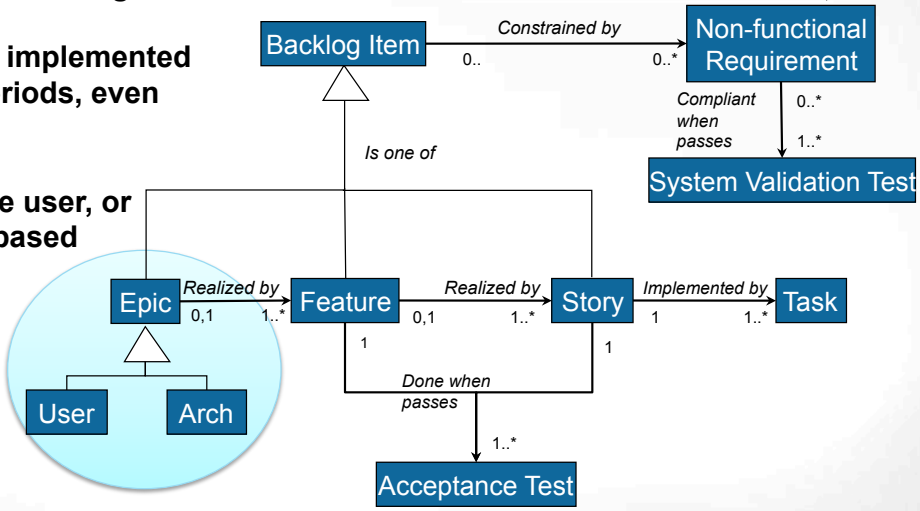
Epics drive programs with features



Epics are key value propositions that create competitive advantage

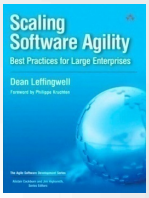
Epic may be implemented over long periods, even years

Epics may be user, or technology based



Big, abstract, high level, visionary

Architectural Epics

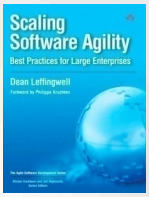


Large, technology development initiatives, cutting across dimensions:

***Time** – affecting multiple releases of products, systems, services or solutions*
***Scope** – affecting multiple products, systems, services, or solutions*
***Organization** – affecting multiple teams, programs, business units*

Examples

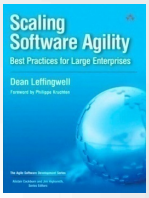
- UI framework for porting existing apps to mobile devices
- Common installer and licensing mechanism
- Industry security standard to lower data purchasing costs
- Support 64 bit back office servers



#2 – THE AGILE RELEASE TRAIN

- DRIVING STRATEGIC ALIGNMENT
- IMPLEMENTING ENTERPRISE PRODUCT DEVELOPMENT FLOW

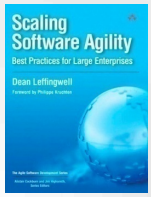
Flow Principles Drive the Release Train



1. Take an economic view
2. Actively manage queues
3. Understand and exploit variability
4. Reduce batch sizes
5. Apply WIP constraints
6. Control flow under uncertainty - cadence and synchronization
7. Get feedback as fast as possible
8. Decentralize control

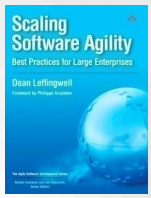
Reinertsen, *Principles of Product Development Flow, 2009.*

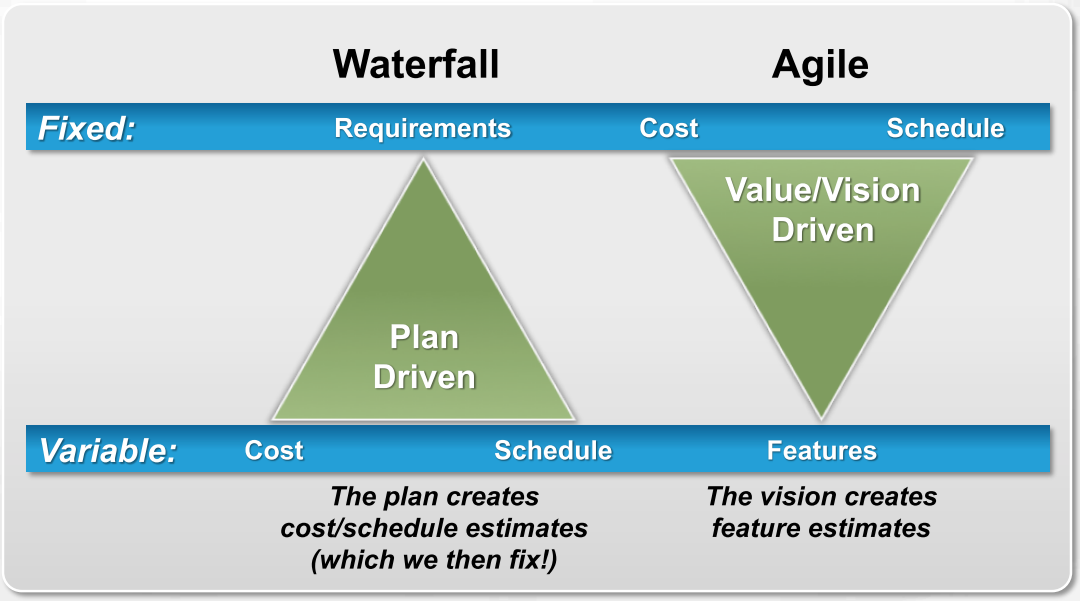
Agile Principles Drive the Release Train



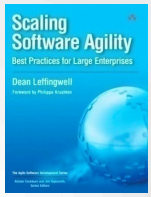
- Incremental build and delivery of value
- Fixed (date, quality, resources) vs. variable (scope) parameters.
- Smaller and more frequent releases (smaller batch sizes)
- Decentralized planning
- Continuous, system-level integration

Fixed: Cost, Quality, Schedule
Variable: Scope



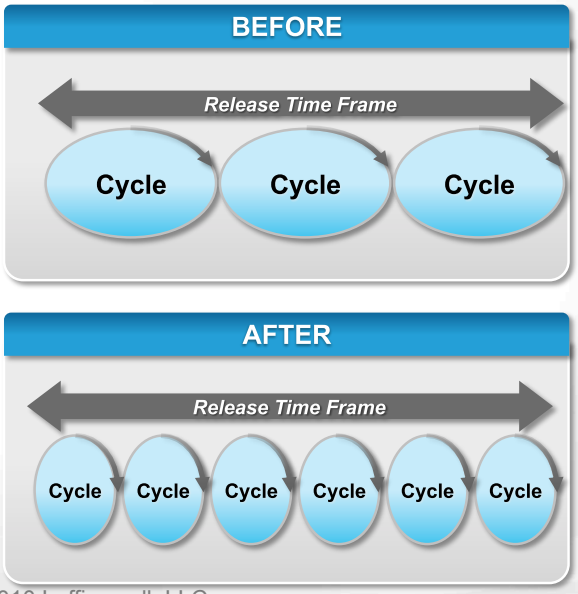


Regular Cadence - Smaller, More Frequent Releases

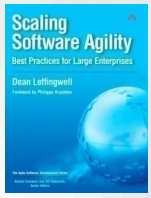


We have to figure out a way to deliver software so fast that our customers won't have time to change their minds.
—Poppendiecks - Implementing Lean Software Development

- Faster value delivery and faster feedback
  - 60-120 days
- Less Work in Process
- Predictable delivery
  - Date, theme, planned feature set, quality
- Scope is the variable
  - Release date and quality are fixed

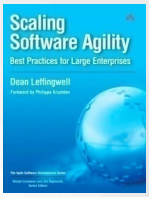


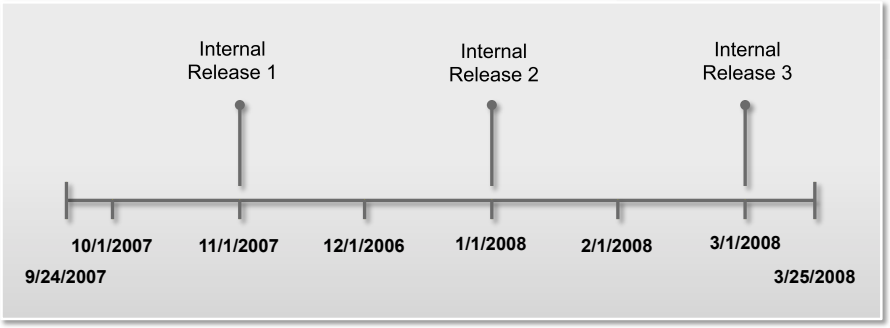
Benefits



- Rapid customer feedback reduces waste
- Earlier value delivery against customer's highest needs
- Frequent, forced system integration improves quality and lowers risk
- Low cost to change
  - Accepts new, important customer features
    - Reprioritize backlog at every iteration & release
  - Reduced patching headaches
    - "It's only X days the next release, that feature can wait"
    - Or easy, high-confidence patching
- Smaller batches for higher productivity
  - Leaner flow through the entire organization to customer

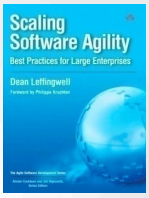
Achieving Cadence: Fix Dates & Quality - Float the Features





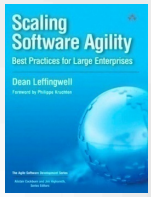
- Teams learn that dates **MATTER**
- Product owners learn that priorities **MATTER**
- Agile teams **MEET** their commitments
- Floating features provides the capacity reserve to meet deadlines

Managing Large-Scale Development Requires Intense, Systemic Cooperation



- Align all teams to the enterprise mission
- Scaling agile requires managing interdependencies amongst distributed agile teams
- Teams themselves must understand and manage *their* dependencies
- Requires coordinated planning and synchronized development activities
- This is facilitated by an “agile release train” delivery model

Principles of the Agile Release Train

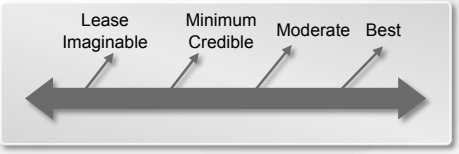


- Release dates for the solution are fixed
- Intermediate, global integration milestones are established and enforced
- Therefore, component functionality must flex

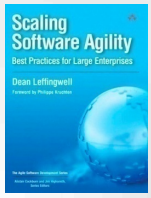
"Time pressures will drive extreme use of simultaneous engineering"

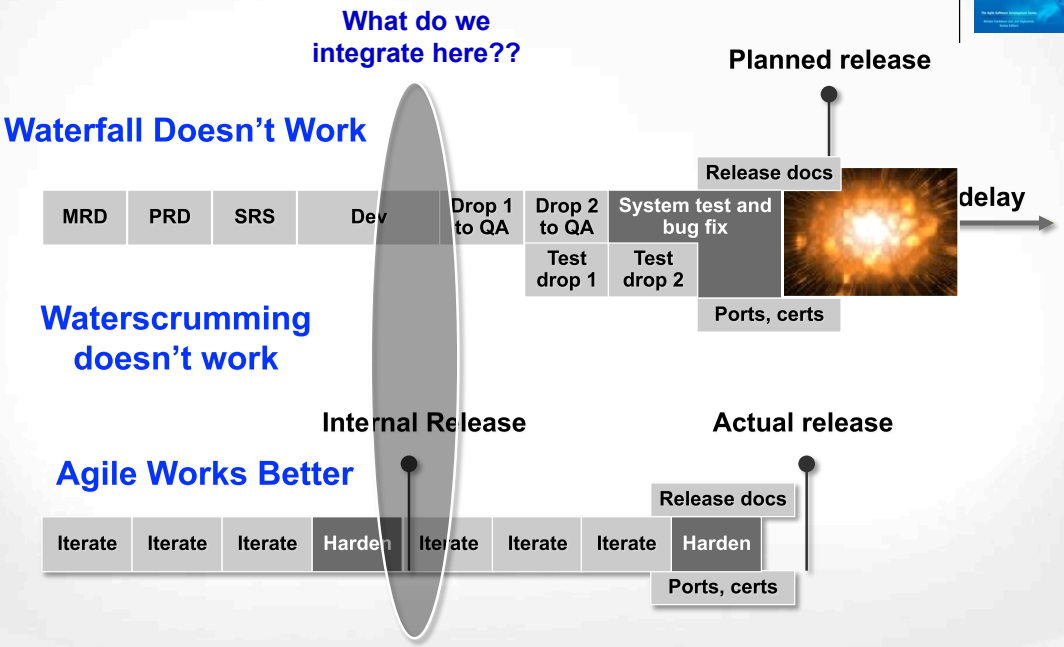
-- The New, New Product Development Game
Harvard Business Review, 1986

- Teams evolve to a flexible model:
  - Design spectrum for new functionality
  - Backup plan to ship existing assets if necessary

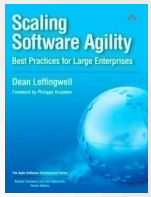


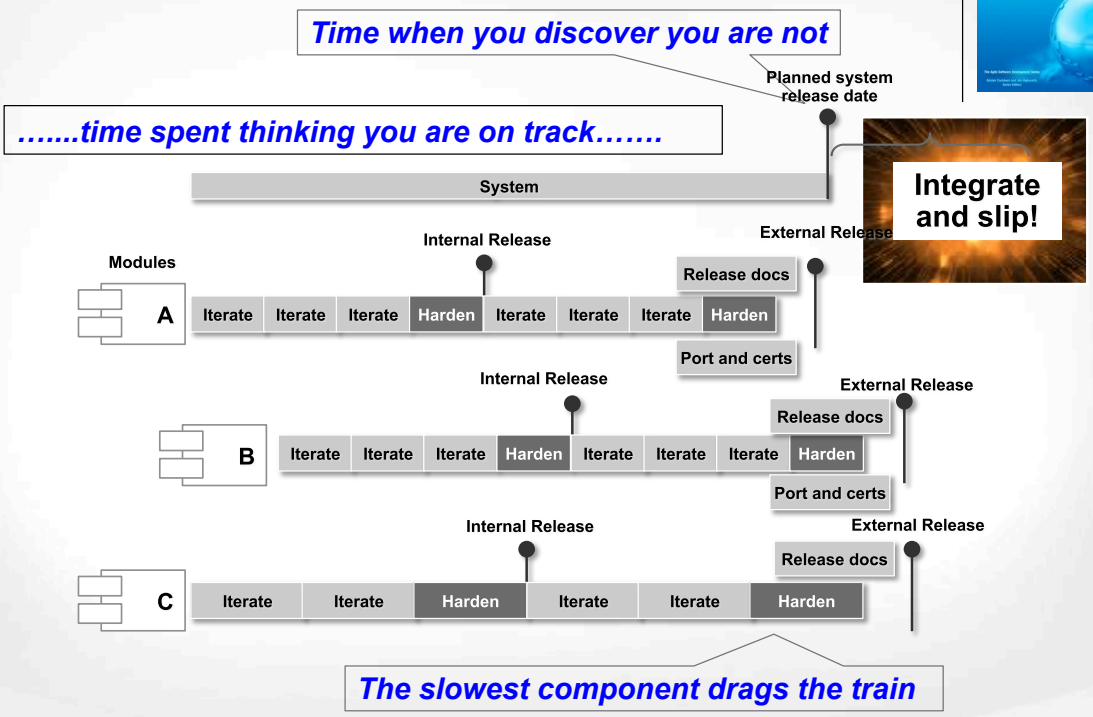
Everybody Must Be on the Train



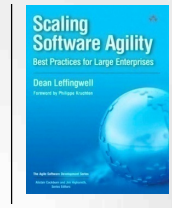


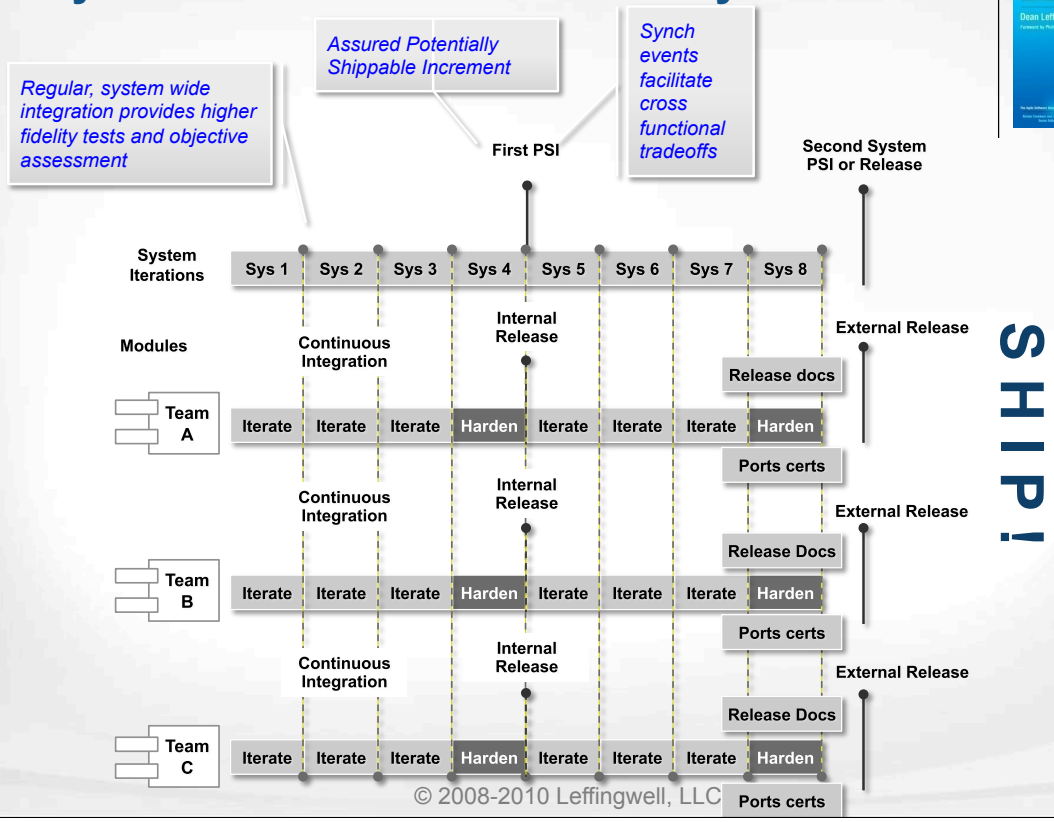
Cadence Alone is not Enough



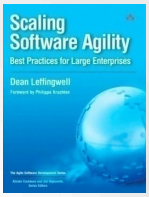


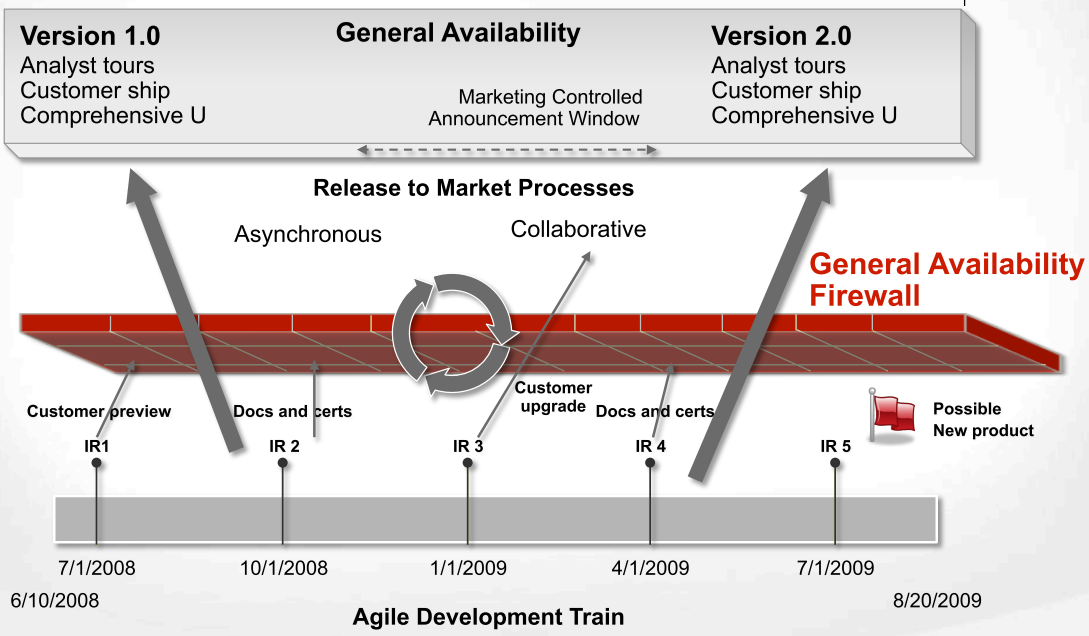
Synchronize to Assure Delivery



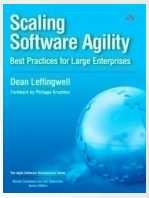


Separate Development Concerns from Release Concerns



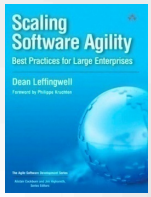


Systems Engineering Benefits



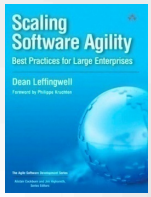
- Continuous, Objective Status
  - Status (working code) and quality measures at iteration and release boundaries
- Availability
  - Forces availability of Potentially Shippable Increment at least at (internal) release cadence
- Quality
  - Continuous integration at each iteration boundary
  - Platform for concurrent system level feature/epic testing
  - Forces holistic, feature maturity at release boundaries
  - Hardening iterations provide “guard band” for full validation and reduction of technical debt

Release Planning – The Pacemaker



Global alignment. Local prioritization.

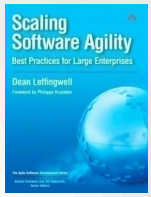
- A full day or two for every release (every 90 days typical)
- Decentralized planning: the plan is owned *by the teams*
- Co-location - most everyone attends in person
- Product/Solution Managers own feature priorities
- The team builds the plan from the vision
- Development team owns planning and high-level estimates
- Adequate logistics and facilitation
- Architects work as intermediaries for technical governance, interfaces and dependencies



#3 – AN ARCHITECTURAL EPIC KANBAN SYSTEM

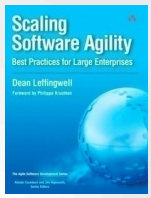
- IMPLEMENTING REALLY BIG THINGS, INCREMENTALLY

Motivation



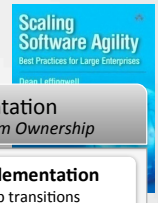
- Drive agile, incrementalism in architectural refactoring
- Make architectural work in process (AWIP) visible
- Establish AWIP limits to control queue sizes, limit global WIP and help assure product development flow
- Drive an effective collaboration with the development teams

Principles of Agile System Architecture

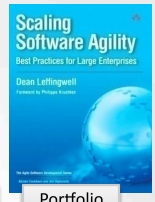


- Principle # 1 – The teams that code the system design the system.
- Principle # 2 – Build the simplest architecture that can possibly work.
- Principle # 3 – When in doubt, code it (or model it) out.
- Principle # 4 – They build it, they test it.
- Principle # 5 – The bigger the system, the longer the runway.
- Principle # 6 – System architecture is a role collaboration.
- Principle # 7 – There is no monopoly on innovation.
- Principle # 8 – Implement architectural flow

Architectural Epic Kanban System

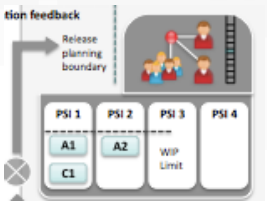


State Diagram



Queue	Activities to transition	Transition criteria	Next	Authority
Funnel	▸ Estimate value ▸ Estimate effort ▸ Test against investment themes	1. Rank >threshold 2. WHEN Slot available 3. Fails criteria	→Backlog →Trash	Architectural Authority
Backlog	▸ Assign Cost of Delay ▸ Effort estimate refined ▸ Establish Relative rank	Ranked relative to other items Highest ranked item pulled When age of item> limit	→Analysis →Escalate or Trash	Pull system Architectural Authority
Analysis	▸ Workshops, modeling, design alternatives ▸ Development collaboration and cost estimates ▸ Dev design spikes ▸ Product/Solution management review ▸ Implementation options ▸ Market validation of value ▸ Business case	Business case with GO/NO GO recommendation GO -> implementation NO GO 1-> more elaboration needed No GO 2 - reject	→Impl. → Stay in queue →Trash	Product/ Technology council

Splitting Epics for Implementation in the Release Train



Partition by subsystem, product or service	Major/Minor effort
System qualities	Simple/Complex
Incremental functionality	Variations in data
Build scaffolding first	Break out a spike

The Agile Enterprise Big Picture

For discussion, see www.scalingsoftwareagility.wordpress.com