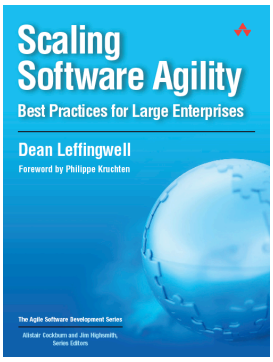


Leffingwell, LLC.

Whitepaper

The Big Picture of Enterprise Agility

By Dean Leffingwell

Abstract:
This whitepaper describes an organizational, process and requirements model for implementing agile methods at enterprise scale. While fully scalable to all levels of the project, program and portfolio, the foundation of the model is a quintessentially lean and agile subset in support of the agile project teams that write and test all the code. This whitepaper is based on the content of the author's book, *Scaling Software Agility: Best Practices for Large Enterprises*, from Addison Wesley, and from the companion blog's "Big Picture" series, which can be found at http://scalingsoftwareagility.wordpress.com/category/big-picture/.

Comments are welcome and may be submitted to the author through the blog.

Contents

The Big Picture of Enterprise Agility

As I described in my book *Scaling Software Agility: Best Practices for Large Enterprises*, [Leffingwell 2007] and the blog [www.scalingsoftwarereagility.wordpress.com], effectively implementing software agility at the enterprise level is no small feat. Even for the fully committed department or enterprise, it can take six months to a year to introduce and master many of the basic agile practices and a number of additional years to achieve the productivity and quality results that fully warrant the effort of such a significant enterprise-wide transformation. As challenging as this process may be however, a number of large organizations have made the transition before us and patterns for success have started to emerge.

In my discussions with teams, managers, and executives during this period, I often struggled to find a language for discussion, along with a set of abstractions and an appropriate graphic that I could use to quickly describe "what your enterprise would look like after such an agile transformation".

In doing so, I would need to be able to describe the new software development and delivery process mechanics, the new teams and organizational units, and some of the roles key individuals play in the new agile paradigm. In addition, any such *Big Picture* should highlight the requirements practices of the enterprise agile model, because those artifacts uniquely carry the value stream to the customer.

Eventually, and with help from others[1], I arrived at something that worked reasonably well for its purpose. I call it the *Agile Enterprise Big Picture* and it appears in Figure 1 below.

[1] Special thanks to Matthew Balchin and Juha-Markus Aalto of Nokia Corporation, and Mauricio Zamora of CSG Systems, Inc.

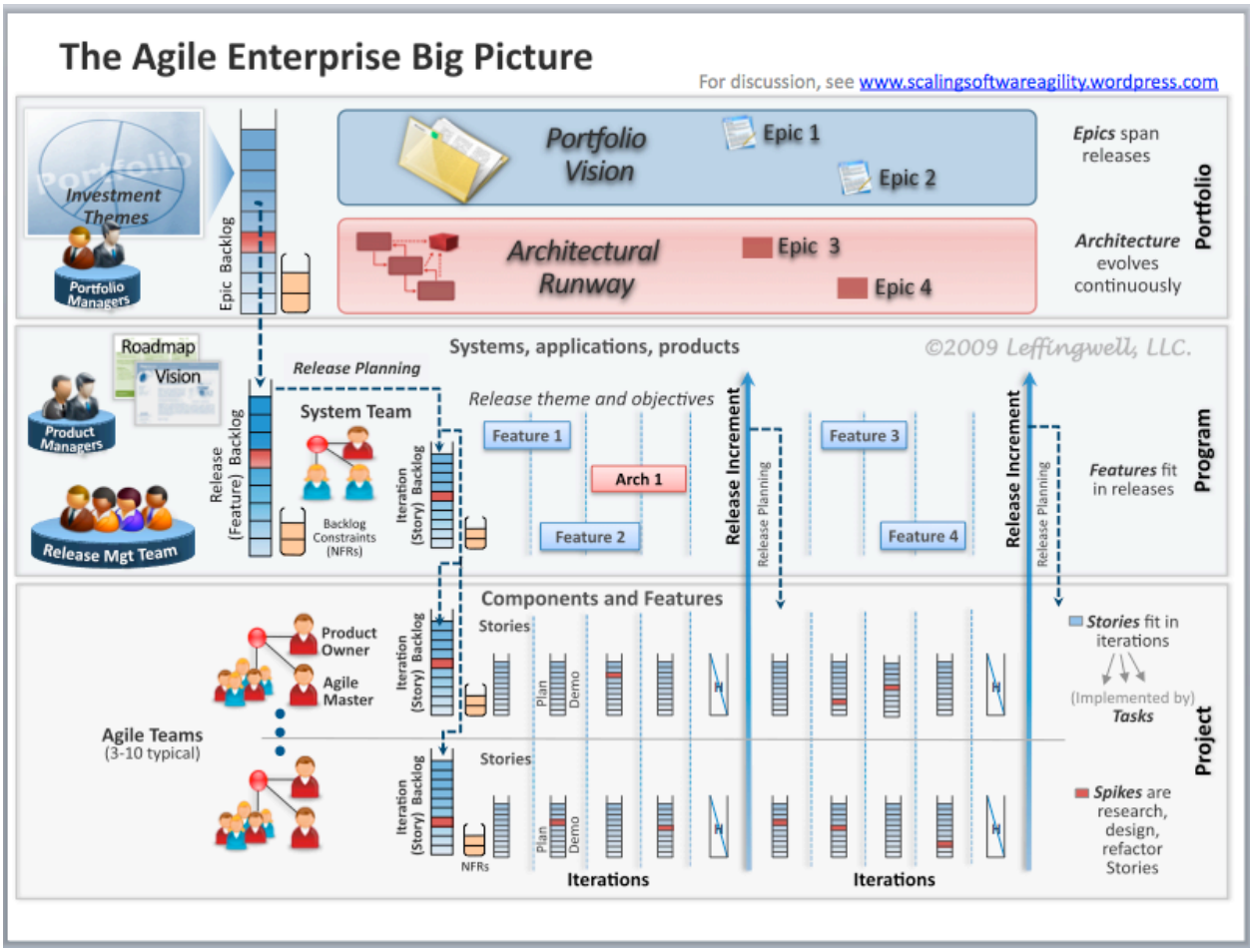


Figure 1 - The Big Picture of Enterprise Agility

Over the last year, I've described the Big Picture fairly extensively in a length series of blog posts under the series http://scalingsoftwareagility.wordpress.com/category/big-picture/. However, in blog series form, it's hard to get the gestalt of this somewhat complex construct, as the posts are fairly short and quite numerous. Frankly, it takes some hard work on the part of the reader to piece together "the Big Picture of the Big Picture. So in this whitepaper, I'll explain the Big Picture at the summary level.

First, the highlights, starting at the bottom of the graphic and working towards the top:

> At the *Project* level, agile teams of 7+/- 2 team members define, build and test *user stories* in a series of iterations and release increments. In the smallest enterprise, there may be only a few such teams. In larger enterprises "*Pods*" of such teams work together to build value streams, be they a feature, component, product in a product suite, application, subsystem or whatever.
>
> At the *Program* level, development of larger scale systems functionality is accomplished via multiple teams in a synchronized "Agile Release Train" model – a standard cadence of time-boxed iterations and releases that is date fixed and quality fixed - but scope variable.
>
> There is typically separation of product definition (requirements) responsibilities at the *Project, Program* and *Portfolio* levels. For our purposes, we'll assign roles/titles of

Product Owner for the *Project* level, *Product Manager* for the *Program* level, and *Portfolio Manager* for the *Portfolio* level. The titles however are not so important as these vary from company to company, but the roles are critical to understanding who has the responsibility to manage requirements as they move through the value stream.

Different requirements artifacts – *User Stories, Features*, and *Epics* - are used to describe the system at these different levels. These labels will help control the level of abstraction used by the people in the roles, limiting early, overly constraining and overly detailed specification, thereby reducing work in process and freeing the teams to implement the intent in the manner best suited to their local knowledge and context.

Releases and *Release Increments* (Potentially Shippable Increments) are frequent, typically at fixed-schedule, 60-120 day maximum boundaries. These increments can be released to the customer or not, depending on the customers capacity to absorb new product as well as external events which can drive timing such as trade shows, annual market news cycles and the like.

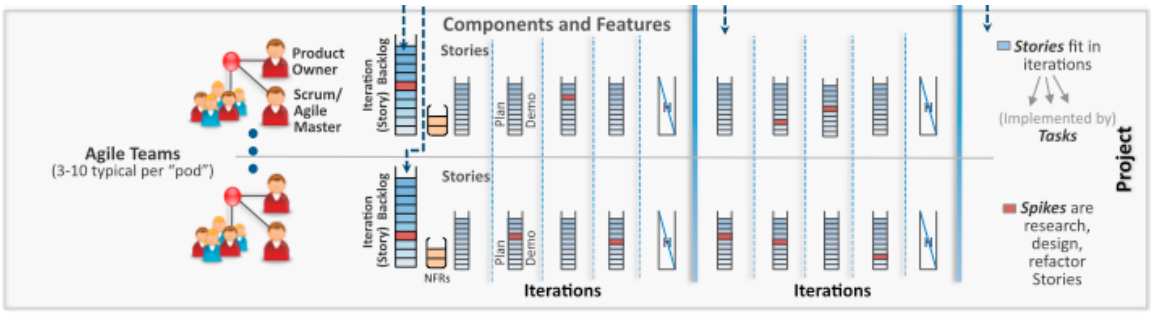
Since releasing products to the end user is the goal of the enterprise and can be quite a complex process, there are typically other teams involved at the *Program* level. The Big Picture identifies two typical, additional teams, the *System Team* and *Release Management Team*. These teams are typically responsible for system level testing and release readiness assessment, and release content management and deployment, respectively.

Architecture emerges in agile and it is the responsibility of the self-organizing teams to evolve an architecture that can reliably host the existing and new requirements. At enterprise scale, however, some amount of *Architectural Runway* will be necessarily to minimize large scale refactoring and to help disparate teams build features and components that integrate easily and behave in a like manner for the user. We'll use the term *Architectural Epics* to describe the key initiatives we'll use to develop architectural runway.

At the *Portfolio Management* level, we'll talk about a mix of *Investment Themes* that are used to drive the investment priorities for the enterprise. We'll use that construct to assure that the work being performed is the work necessary for the enterprise to deliver on its chose business strategy. Investment Themes drive the portfolio vision, which will be expressed in as a series of larger, *Epic*-scale initiatives that will be factored into upcoming releases over time.

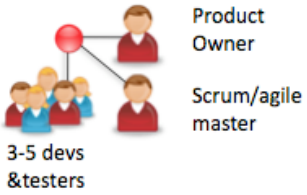
In the rest of this whitepaper, we'll "walk" through the various elements of the Big Picture to describe how it works. We'll highlight the requirements value delivery stream, as well as the individual roles, teams, and processes that are necessary to deliver a continuous flow of value to the users. For additional depth of explanation on any of the topics we'll cover below, refer directly to the blog series.

Big Picture – Project Level



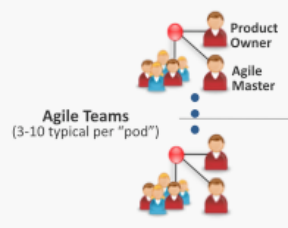
The Agile Team

The "fighting force" for software development consists of some number of *agile teams* that collaborate on building the larger system. It's appropriate to start with the team, because in agile, the *team is the thing*, as they write and test all the code that delivers value to the end user. Since it's an agile team, it has a maximum of 7-9 members, and includes all the roles necessary to define/build/test[2] the software for their *component* or *feature*. The roles include a Scrum/Agile Master, Product Owner and a small team of dedicated developers, testers and (hopefully) test automation experts, and maybe a tech lead. In its daily work, the team is supported by architects, tech writers, SCM/build/infrastructure support personnel, internal IT and whoever else it takes such that the team is fully capable of *defining, developing, testing and delivering working and tested* software into the system baseline.



These teams are typically organized around a software **component** or a **feature**. Most enterprises will have a mix of both, some **component teams** focused on shared infrastructure, subsystems, and persistent, service-oriented architectural components, and some **feature teams** focused on current larger scale, value-delivery initiatives. Agile teams are self-organizing and reorganize themselves continuously based on the work in the release backlog. Over time, the makeup of the teams themselves is more dynamic than static; static enough to norm and storm[3] for reasonable periods of time, dynamic enough to flex to the organizations changing priorities.

Pods of Agile Teams



In addition, within the larger enterprise, there are typically some number (3-10) or so such teams who cooperate to build a larger feature, system or subsystem (the *Program* domain in the Big Picture). While this isn't a hard or fast rule, experience has shown that even for VERY large systems, the logical partitions defined by system or product family architecture tend to cause "pods" of developers to be organized around the various implementation domains. This implies that perhaps 50-100 people must intensely collaborate on building their "next bigger thing" in the hierarchy. This is also about the maximum team size for collaborative, face-to-face, release planning.

[2] See Chapter 6 of Scaling Software Agility: Best Practices for large Enterprises [Leffingwell 2007]

[3] See the Forming – Storming – Norming – Performing model of group development proposed by Bruce Tuckman at http://en.wikipedia.org/wiki/Forming-storming-norming-performing

Of course, even that's an oversimplification for a really large system, as there are likely to be a number of such *Programs*, each contributing to the *Portfolio* (product portfolio, application suite, systems of system).

Roles in the Agile Team

Product Owner

Scrum is the dominant agile method used in the market today and the product owner role is uniquely, if somewhat arbitrarily, defined therein. In Scrum, the product owner is responsible for determining and prioritizing user requirements, and maintaining the product backlog. Moreover, even if a team is not using Scrum, it has been my experience that implementing the product owner role - as largely defined by Scrum - can deliver a real breakthrough in simplifying the team's work and organizing the entire team around a single, prioritized backlog.

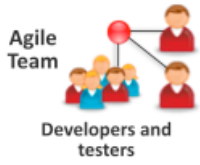
But the product owner's responsibilities don't end there. In support of Agile Manifesto principle #4[4] - *Business people and developers must work together daily throughout the project,* the product owner is ideally collocated with the team and participates *daily* with the team and its activities.

Scrum/Agile Master

For teams implementing Scrum, the ScrumMaster is an important role. The ScrumMaster is team-based management/leadership proxy whose role is to assist the team in its transition to the new method and continuously facilitate a team dynamic intended to maximize performance of the team. Thought the teams are self-organizing and self-managing, they are not leaderless and in a Scrum implementation, this leadership role falls to the ScrumMaster.

In teams that do not adopt Scrum, a comparable leadership role typically falls to a team lead, an internal or external coach, or the team's line manager. As their skills develop, many of these *Agile Masters* become future leaders of the enterprise by demonstrating their ability to help teams deliver products to the market and by driving continuously improving agile practices.

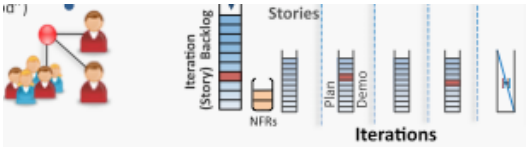
Developers and Testers



The rest of the core team includes the developers and testers who write and test the code. Since this is an agile team, the team size is limited to about 3-4 developers plus typically, 1-2 testers, who are collocated and work together to *define*, *build*, *test* and *deliver* stories into the code baseline. Of course, team members may also include full or part time architects, tech leads, user experience and documentation experts, CM/build masters or whatever – covering such other functions as are necessary to assure the delivery of a quality solution.

[4] www.agilemanifesto.org

Iterations

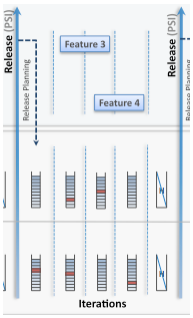


In agile development, new functionality is typically built in short time-boxed, events called *iterations* (*Sprints* in Scrum). In larger enterprises, agile teams are usually organized around a standard iteration length, and typically share start and stop boundaries so that the code asset maturity is comparable at each iteration-boundary system integration point.

Each iteration represents a valuable increment of new functionality, accomplished via a constantly repeating standard pattern: *plan the iteration*, *build and test stories*, *demonstrate the new functionality to stakeholders*, *repeat.* The iteration is the truly the "heartbeat of agility" for the team and teams are almost entirely focused on developing new functionality in these short time boxes.

In the Big Picture, the iteration lengths for all teams are the same as that is the simplest organizational model and one that supports continuous integration by driving comparable asset maturity at the iteration boundaries. While there is no mandated length, most have converged on a recommended length of *two weeks*.

Number of Iterations per "Release"



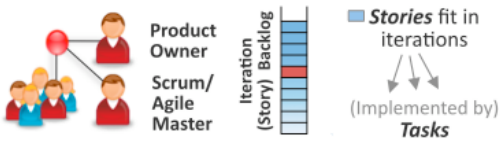
A series of iterations is used to aggregate larger, system wide, functionality for release (or potential release) to the external users. In the Big Picture, we've illustrated four *development* iterations (indicated by a full iteration backlog) followed by one *hardening* (or stabilization) iteration (indicated by an empty backlog) prior to each release increment.

This pattern is arbitrary and there is no fixed rule for how many times a team iterates prior to a release increment. Many teams apply this model with 4-5 development iterations and one hardening iteration per release, creating a cadence of a fully shippable increment about every 90 days. This is a fairly natural production rhythm that corresponds to a reasonable external release frequency for customers and also provides a nice quarterly planning cadence for the enterprise itself.

In any case, the length and number of iterations per release increment, and the decision as to when to actually release an increment, is left to the judgment of each enterprise.

User Stories and the Iteration Backlog

User Stories



User Stories (stories for short) are the requirements workhorse of agile development. They are the general-purpose, agile substitute for what traditionally has been referred to as "software requirements".

Originally developed within the constructs of XP,

user stories are now endemic to agile development in general and are typically taught in Scrum, XP and most other agile implementations. In agile, *user stories are the primary currency that carries the customer's requirements through the value stream into code and implementation.*

As opposed to requirements (which are something the system *must* do) user stories are *brief statements of intent* that describes something the system needs to do for the user. As commonly taught, the user story often takes a standard, user-voice form of:

> As a <user role> I can <activity> so that <business value>

With this form, the team learns to focus on the business benefit that the new functionality provides to a user role. This construct is integral to agile's intense focus on value delivery.

Iteration Backlog

The team's iteration backlog consists of all the user stories the team has identified for implementation. Each team has its own backlog, which is maintained and prioritized by the team's product owner. While there may be other things in the teams backlog as well - defects, refactors, infrastructure work, etc. - the user stories carry the value stream and are the primary focus of the team.

> *Identifying, maintaining, prioritizing, scheduling, elaborating, implementing, testing and accepting user stories is the primary requirements management process at work in the agile enterprise.*

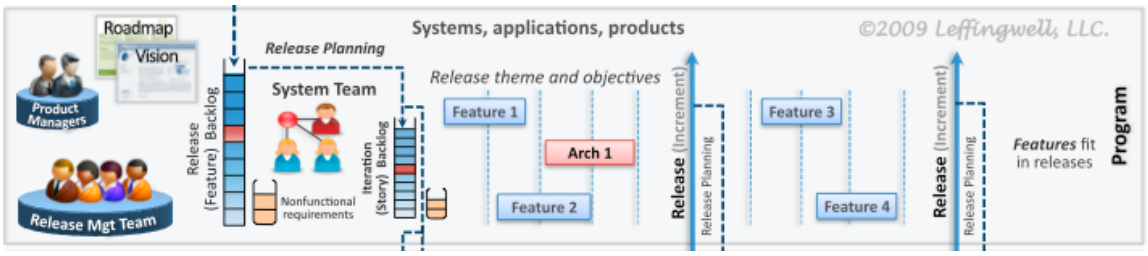
Tasks

For more detailed tracking of the activities involved in delivering stories, many teams decompose stories into *tasks* that must be accomplished by individual team members in order to complete the story. Indeed some agile training uses the task object as the basic estimating and tracking metaphor.

However, the iteration tracking focus should be at the story level, as it is better aligned to business value than are individual tasks. Tasks are simply a means to that end, a micro-work breakdown structure that facilitates estimating and the taking of individual responsibilities to help assure completion of the iteration, one story at a time.

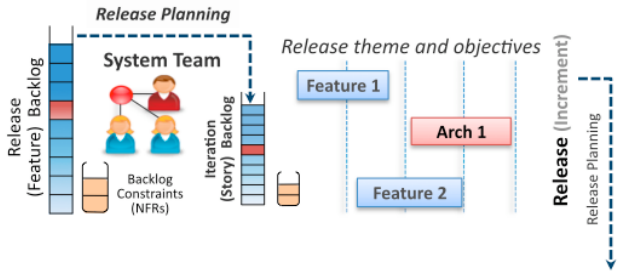
Many mature teams tend to use queue-based assignment – tasks aren't assigned until a given person has availability. In any case, the use of tasks is entirely optional and many mature teams tend to eliminate them over time.

Big Picture – Program Level



At the *Program* level of the Big Picture, we find additional organizational constructs, roles, processes and requirements artifacts suited for building larger-scale systems, applications, products and suites of products.

Releases and Release Increments



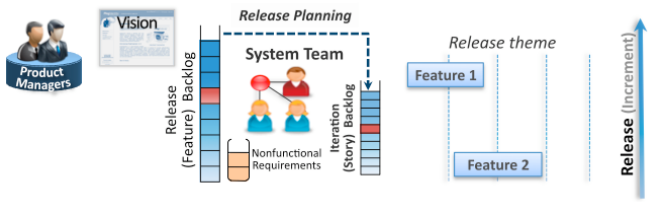
While the goal of each agile iteration is to produce a "potentially shippable increment" of software, teams - especially larger-scale enterprise teams - find that it may simply not be practical or even appropriate to ship an increment at each iteration boundary. For example, during the course of a series of iterations, the team may accumulate some *technical debt*, which needs to be addressed before shipment. Technical debt may include things such as defects to be resolved, minor code refactoring and deferred system-wide testing for performance, reliability or standards compliance. Resolving technical debt is the primary reason that a *hardening iteration* is included in the Big Picture model.

Moreover, there are legitimate business reasons why not even every release increment should be shipped to the customer. These include:

- Potential interference with a customer's licensing and service agreements
- Potential for customer overhead and business disruption for installation, user training etc.
- Potential for disrupting customer's existing operations with minor regressions or defects
- Ability for the customer to consume the releases

For these reasons and others, most teams aggregate a series of iterations into a *release increment*, which can be released or not at the discretion of the enterprise.

Vision, Features and the Release Backlog



Within the enterprise, the Product Management (or possibly Program Management or Business Analyst) function is primarily responsible for maintaining the Vision of the products, systems, or application in their domain of influence.

The vision answers the big questions for the system, application or product, including:

- What problem does this particular solution solve?
- What features and benefits does it provide?
- For whom does it provide it?
- What performance, reliability, etc. does it deliver?
- What platforms, standards, applications, etc. will it support?

The Primary Content of the Vision is a Set of Features

A vision may be maintained in document, backlog repository, or even in a simple briefing form. But no matter the form, the prime content of the vision document is a prioritized set of features. Features are "Big Picture" system behaviors that can be described in a sentence or two and which are written so that customers can actually understand, debate and prioritize them. In [Leffingwell, 2003], we posited that by managing the level of abstraction, a system of arbitrary complexity (from the space shuttle to the spellchecker on this editor) can be described in a list of 25 or so features. That still works for agile teams describing a vision today.

Undelivered Features Fill the Release Backlog

In a manner similar to the iteration backlog, which contains *stories*, the release backlog (the *product backlog* in some agile trainings) contains the set of desired and prioritized *features* that have not yet been implemented. The release backlog may or may not also contain estimates for the features. However, any estimates at this scale are coarse grained and imprecise, which prevents any temptation to over-invest in inventory of too-early feature elaboration and estimation. If and when a feature reaches a priority such that it hits a release-planning boundary, it will be broken into user stories, which can then be estimated by the team.

Nonfunctional Requirements

In addition, the Vision must also contain the various nonfunctional requirements, such as reliability, accuracy, performance, quality, compatibility standards, etc. that are necessary for the system to meet its objectives.

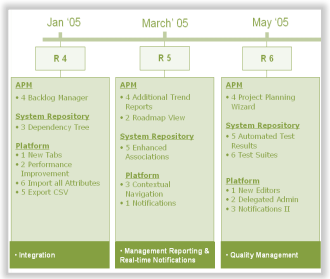
Release Planning

In accordance with emerging agile enterprise best practices, each release increment time-box has a kickoff release planning session that the enterprise uses to set the company context and to align the teams to common business objectives for the release. The input to the release planning session is the current vision, along with set of objectives and a desired, prioritized feature set for the upcoming release.

By breaking the features into stories, and applying the agreed to iteration cadence and knowledge of their velocity, the teams plan the release, typically in a group setting. During this process, the teams work out their interdependencies and design the release by laying stories into the iterations available within the release timebox. They also negotiate scope tradeoffs with product management, using the physics of their known velocity and estimates for the new stories to determine what can and can't be done. The result of this process is a commitment to a revised set of release objectives, along with a prioritized feature set.

Thereafter, they endeavor to meet their commitment by satisfying the primary objectives of the release, even if it turns out that not every feature makes the deadline.

The Roadmap

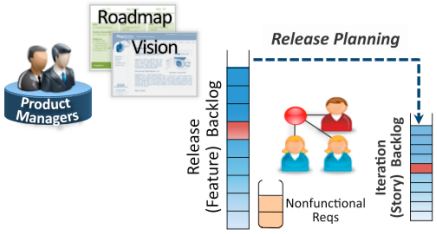


The output of this process is a *Roadmap*, which provides a sense of how the enterprise hopes to deliver increasing value over time.

The roadmap consists of a series of planned release dates, each of which has a theme, set of objectives and a prioritized feature set. The "next" release on the roadmap is *committed to the enterprise*, based on the work done in the most recent release planning session. Releases beyond the next one are not committed, and their scope is fuzzy, at best.

The roadmap then, represents the enterprises current "plan of intent" for the next, and future releases. However, it is subject to change - as development facts, business priorities and customers need change – and therefore release plans beyond the next release should not be used to create any external commitments.

Product (Program) Management

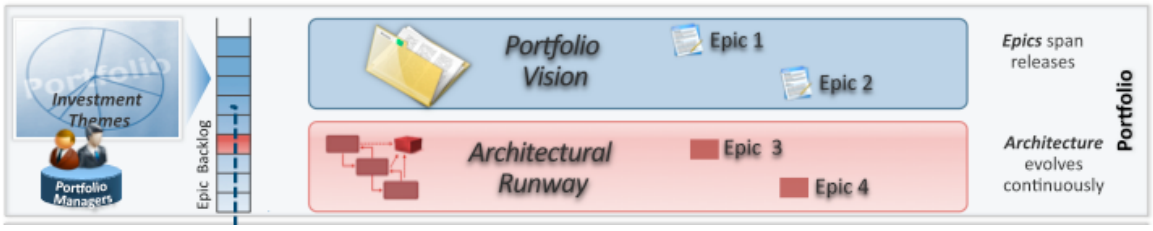


Schwaber [Schwaber 2007] describes the Scrum perspective on the role of the product owner as follows:

"is responsible for representing the interests of everyone with a stake in the resulting project ...achieves initial and ongoing funding by creating the initial requirements, return on investment objectives and release plans."

In some smaller organizational contexts, that definition works adequately and one or two product owners are all that are needed to define and prioritize software requirements. However, in the larger software enterprise, the set of responsibilities imbued in the Scrum product owner is more typically a much broader set of responsibilities shared between team and technology-based, product owners and a number of market or program-based, *product or program managers*, who carry their traditional responsibilities of both defining the product *and* positioning the solution to the marketplace.

Big Picture – Portfolio Level



At the top of the Big Picture, we find the *portfolio management* function, those teams and functions dedicated to managing the investments of the enterprise in accordance with the enterprise business strategy. We also find two new artifact types, *Investment Themes* and *Epics*,

which we'll use to describe the set of initiatives that drive the enterprises investment in systems, products and applications.

Investment Themes

The set of investment themes for an enterprise, or a business unit within the larger enterprise, establishes the relative investment objectives for the entity as the pie chart below illustrates:

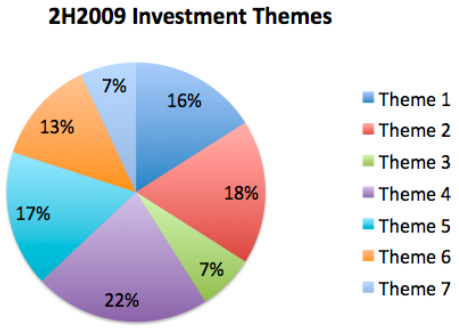


Figure 3 - Investment Themes

These themes drive the vision for all product teams and new *epics* are derived from this decision. The derivation of these decisions is the responsibility of those who have fiduciary responsibilities to their stakeholders. In most enterprises, this happens at the business unit level based on annual or twice annual budgeting process.

Within the business unit, the decisions are based on some combination of:

- Investment in *existing* product offerings – enhancements, support and maintenance
- Investment in *new* products and services - products that will enhance revenue and/or gain new market share in the current or near term budget period
- Investment in *futures* - product and service offerings that *require* investment today, but will not contribute to revenue until outlying years.
- *Sunset* strategies for existing product offerings – proactively determining the end life of products

The result of the decision process is a set of themes - *key product value propositions that provide marketplace differentiation and competitive advantage*. Investment themes have a much longer life span than epics, and a set of themes may be largely unchanged for up to a year or more.

Why Investment Mix Rather than Backlog Priority?

Investment themes are not contained or represented in a backlog, as such. Backlog items (stories for the iteration, features for the release and epics for the portfolio) are intended to be implemented in priority order. Investment themes are intended to be addressed on "a percentage of time to be made available basis." For example, the lowest priority feature on a release backlog may not be addressed at all in the course of a release and yet the release could well be a success (meet its stated objectives). However, if the lowest priority (smallest investment mix) investment theme is not addressed over time, the enterprise may ultimately fail in its mission, as it is not making its actual investments based on the longer-term priorities it has decided.

Epics and the Portfolio Backlog

Epics represent the highest-level expression of a customer need as this hierarchical graphic shows.

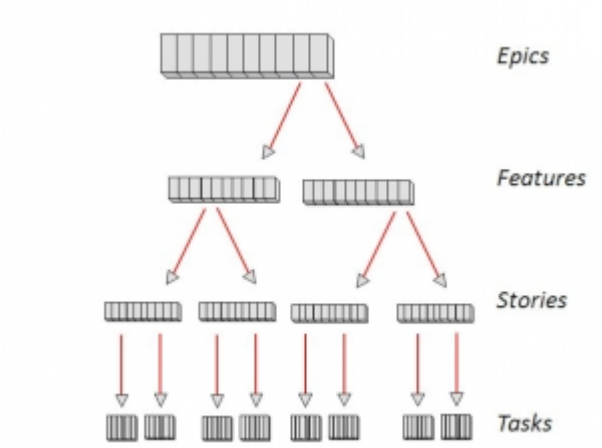


Figure 4-Epics are the highest level requirements artifact

Derived from investment themes, epics are development initiatives that are intended to deliver the value of the theme and are identified, prioritized, estimated and maintained in the *Portfolio Backlog***.** Prior to release planning, epics are decomposed into specific features, which in turn, are converted into more detailed stories for implementation.

Epics may be expressed in bullet form, in user voice story form, as a sentence or two, in video, prototype, or indeed in *any form* of expression suitable to express the intent of the product initiative. With epics, clearly, the objective is *strategic intent*, *not specificity*. In other words, the epic need only be described in detail sufficient to *initiate a further discussion* about what types of features an epic implies.

Architectural Runway

Design (architecture) and requirements are simply two sides of the same coin, the "what" and the "how" of the systems behavior. Experience tells us that teams that build some amount of *architectural runway* will eventually emerge as the winners in the marketplace.

I've defined architectural runway [Leffingwell 2007] as:

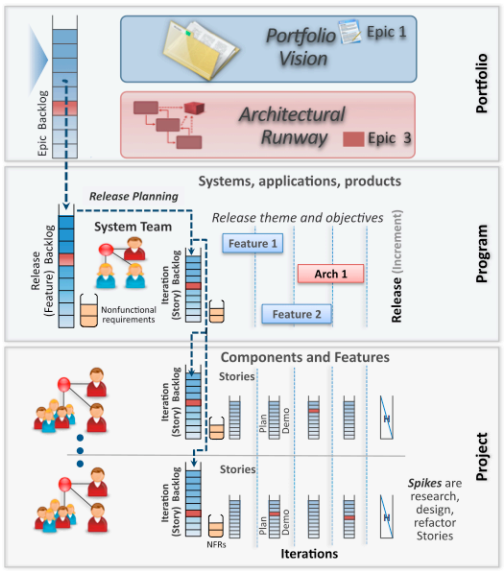
> *A system with architectural runway has existing or planned infrastructure sufficient to allow incorporation of current and near term requirements without excessive refactoring.*

Continuous build out and maintenance of new architectural runway is the responsibility of all mature agile teams. Failing to do so will call cause one of two things to happen, either of which is very bad:

1. Release dates will be missed as large-scale, just-in-time, infrastructure refactoring adds unacceptable risk to scheduling

2. Failure to extend the architecture systemically means that the teams will eventually run out of runway. New features cannot be added without major refactoring. Velocity slows. The system eventually becomes so brittle and unstable that it has to be entirely rewritten.

In the Big Picture, architecture appears as a "red thread" through various levels of the hierarchy.



Portfolio – At the Portfolio level, architectural runway is represented by ongoing infrastructure initiatives that have the same level of importance as the larger scale requirements epics that drive the company's vision forward. Some will require meaningful levels of investment, consume substantial resources and in the near term, reduce the velocity of new feature implementations. As failing to implement them will quickly compromise the company's position in the market, *architectural epics* must be visible, estimated and planned just like any other epic.

Program – Once identified, architects, system teams, project teams and other stakeholders translate the architectural epics into architectural features that are relevant to each release. They are prioritized, estimated and resourced like any other feature. And, like features, each architectural initiative must also be conceptually complete (no overhang, no implied but missing functionality) at each release boundary, so as to not compromise the release increment.

Project – At the Project level, refactors, design spikes, evaluations etc. that are necessary to extend the runway are simply a type of story that are prioritized like any other story. Like user value stories, architectural work is visible, accountable and demonstrable at every iteration boundary. This is accomplished by agreement and collaboration with the system architects, product owners, and agile tech leads that determine what spikes need to happen, and when.

In this manner, architecture is a first class citizen of the Big Picture and is a routine portfolio investment consideration for the agile enterprise.

Summary

This whitepaper describes an emerging enterprise pattern for the successful implementation of software agility at the project team, program and portfolio level. Future articles in this series will describe each of these levels – along with the roles, organizations and requirements artifacts – in additional detail. In the meantime, you can find additional depth of explanation of the Big Picture of Enterprise Agility at the blog: http://scalingsoftwareagility.wordpress.com/category/big-picture/.

References

Leffingwell, Dean. 2007. Scaling Software Agility: Best Practices for Large Enterprises. Boston, MA: Addison-Wesley.

Leffingwell, Dean, and Don Widrig. 2003. Managing Software Requirements, Second Edition: A Use Case Approach. Boston, MA: Addison-Wesley.

Schwaber, Ken. 2007. The Enterprise and Scrum. Redmond, WA: Microsoft Press.