Agile Software Requirements

Lean Requirements Practices for Teams, Programs, and the Enterprise

Dean Leffingwell

Foreword by Don Reinertsen

Agile Software Development Series

Alistair Cockburn and Jim Highsmith, Series Editors

Library of Congress Cataloging-in-Publication Data

Leffingwell, Dean.
Agile software requirements : lean requirements practices for teams, programs, and the enterprise / Dean Leffingwell.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-63584-6 (hardcover : alk. paper)
ISBN-10: 0-321-63584-1 (hardcover : alk. paper)
1. Agile software development. I. Title.
QA76.76.D47L4386 2011
005.1—dc22

2010041221

ISBN-13: 978-0-321-63584-6
ISBN-10: 0-321-63584-1
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, December 2010

This page intentionally left blank

Contents

Foreword

Why do product development projects miss their economic objectives? Studies show that 80 to 85 percent of project failures are due to incorrect requirements. Experienced developers know that managing requirements is a greater challenge than technical execution. And, although we have known this for decades, we really haven't gotten much better at it. Why? At first, we were functionally organized, so we simply displaced the problem outside the boundary of engineering—we blamed marketing and product management. Later, as we adopted cross-functional teams, we told these teams to listen to the voice of the customer and assumed that this would solve the problem.

It didn't. We never challenged the idea that it was feasible to develop valid requirements up front—we just told people to try harder. We just told them to pay more attention to what the customer was asking for. We ignored the fact that many customers don't know what they want. We ignored that fact that even when they know what they want, they can't describe it. We ignored the fact that even when they can describe it, they often describe a proposed solution rather than the real need. For example, customers told us that they wanted suitcases that were easy to carry, and asked us to make them lightweight. We did this, but they rejected our elegant designs and bought the heavier designs of our competitors—the ones with wheels on them!

The sad truth is that there is no one "voice of the customer." It is a cacophony of voices asking for different things. Even at a single customer, we need to balance the needs of technical decision makers, end users, system operators, and financial decision makers. All of these actors weigh different attributes differently, and they change their weighting as they acquire more experience using the product. We also need to understand the needs of distributors, regulators, manufacturing, and field service. If we focus only on the user, we could miss what Dean calls the "nonfunctional requirements."

And this problem is dynamic, not static. In the course of our development effort, the context constantly changes—competitors introduce new products and customer needs evolve. If it is not feasible to develop valid requirements before we begin design, what is our alternative? In my opinion, we should start with the belief that even the best requirements will contain major errors, and that these errors grow exponentially with time. This shifts our focus. Instead of believing that we are hearing a high-fidelity signal coming from the customer, we need to recognize that it is a noisy, low-fidelity signal—a signal that must be continually checked for errors. Rather than using heavy front-end investment to create perfect requirements, we invest in creating processes and infrastructure that can rapidly detect and correct poor fits between our solution and the customer's evolving needs.

What better test for this alternative approach than the development of large systems? Many of the methods that work superbly on small projects break down on large ones. For example, in small systems, costs and benefits are typically local. System performance does not suffer when a team makes locally optimal decisions. This is not true for large systems where we must deal with economic effects that are dispersed physically, temporally, and organizationally.

We need better approaches to understanding and managing software requirements, and Dean provides them in this book. He draws ideas from three very useful intellectual pools: classical management practices, agile methods, and lean product development. By combining the strengths of these three approaches, he has produced something that works better than any one approach in isolation.

First, although it might be unfashionable to say this, classic management practices still offer us some very useful methods. Not all of our predecessors were stupid dolts, incapable of recognizing a working solution. For decades I have seen relatively simple concepts like technology and product roadmaps producing great results. They ensure that work on technology begins early enough to keep it off the critical path. They create strong logical links between technology efforts and the programs that they serve. We don't need to blindly accept all traditional practices, but we'd be foolish to discard everything our predecessors already learned. Dean shows you how to apply some of these great ideas at the program and portfolio level.

Second, the agile community has developed a very powerful set of ideas that has already produced impressive results. These methods have grown rapidly for a very good reason—they work. Agile decomposes the large batches of the waterfall model into a series of time-boxed iterations. These smaller batches dramatically accelerate feedback, producing enormous benefits.

Since much of agile's success has occurred in smaller projects, it is natural to ask whether it is equally useful in large systems. While I deeply respect the value of agile methods, I think Dean is correct in recognizing that these methods must be extended to meet the needs of large system development. It is quite risky to assume that large system architectures will naturally emerge and that any shortcomings can be refactored away. For example, a naval warship is designed for a 30-year operating life. Good naval architects anticipate evolving threats, emerging technologies, and changing missions. We do not create such systems by letting architecture "emerge." Once we recognize the unique challenge of managing at the system level, we can start investing in the organizational infrastructure needed to meet this challenge. Dean shows you how to do this with agile method extensions such as architectural runways.

Dean also draws upon the ideas of what I call "second-generation lean product development." Many of the initial attempts to use lean in product development focused on ideas such as standardization of work and variability reduction. They lacked agile's intrinsic appreciation that developing great new solutions requires learning to thrive in the presence of uncertainty. These lean product development methods have now evolved, and the results are impressive. For example, today's "kanban" approaches are limiting WIP, accelerating feedback, and making flow visible to all participants. You can see the influence of these ideas on Dean's approaches at the program and portfolio levels. Dean has also recognized the importance of the new emphasis on economics. This emphasis helps us make better decisions and it enables us to explain our choices to management in terms they readily understand.

As you read this book, I suggest paying attention to several things. First, try to understand the reasons *why* certain of these approaches work, not just *what* they are. If you understand why things work, then you can more easily adapt them to your own unique context. Second, treat these ideas as a portfolio of useful patterns rather than a rigid set of practices that must be adopted as a group. This will reduce the batch size of your adoption process, produce less resistance, and provide faster results. Finally, as you use these ideas, strive for balance. You will have a natural tendency to prefer certain ideas—they address issues you feel are important, and they feel comfortable. You may have given other areas little attention for a long time. Often the areas that have received little attention hold great untapped opportunity.

—Don Reinertsen
Author of The Principles of Product Development Flow:
Second Generation Lean Product Development

This page intentionally left blank

Preface

Introduction to the Book

In the past decade, the movement to lighter-weight and increasingly agile methods has been the most significant change to affect the software enterprise since the advent of the waterfall model in the 1970s. Originated by a variety of thought and practice leaders and proven in real-world, successful experiments, the methods have proven themselves to deliver outstanding benefits on the "big four" measures: productivity, quality, morale, and time to market.

In the past five years, the methods spread virally. Within the larger enterprise, the initiatives usually started out with individual teams adopting some or all of the practices espoused by the various methods, primarily XP, Scrum, Lean, Kanban (later), and various combinations and variants.

However, as the methods spread to the enterprise level, a number of extensions to the basic agile methods were necessary to address the larger process, organizational, application scope, and governance challenges of the larger enterprise.

Not the least of these is the challenge of agile requirements, which is the necessity to scale the basic, lightweight practices of team agile—product backlogs, user stories, and the like—to the needs of the enterprise's *Program* and *Portfolio* levels. For example, agile development practices introduced, adopted, and extended the XP-originated "user story" as the primary currency for expressing application requirements. The just-in-time application of the user story provided a much leaner approach and helped eliminate many waterfall-like practices, such as imposing overly detailed and constraining requirements specifications on development teams.

However, as powerful as this innovative concept is, the user story by itself does not provide an adequate, nor sufficiently lean, construct for reasoning about investment,

system-level requirements, and acceptance testing across the larger software enterprise's project Team, Program, and Portfolio organizational levels. That is the purpose of this book.

This book describes an agile requirements artifact model, corresponding practices, suggested roles, and an organizational model that provides a quintessentially lean and agile requirements subset for the agile project teams that write and test the code. Yet this model also scales to the full needs of the largest software enterprise.

Why Write This Book?

In 2000, after about 25 years of managing software development as an entrepreneur and executive, along with my coauthor Don Widrig, I published my first book: *Managing Software Requirements: A Unified Approach*. In 2003, we updated the book with a second edition: *Managing Software Requirements: A Use Case Approach*. These are considered to be definitive texts on managing application requirements—a lot of copies were sold, and the books have been translated into five languages. More importantly, many individuals, teams, and companies told me that these works helped them achieve better software outcomes. That was always the goal.

In the following years, I turned my attention to agile development methods. I continue to be more and more impressed with the power of these innovative methods, the quality and productivity results they delivered, and the way in which they reenergized and empowered software teams. Though the methods were developed and proven in small team environments, the challenges of building software at scale is a more fascinating puzzle—part science, part art, part engineering, part organizational psychology. As a result, I became engaged in helping a number of larger enterprises in adopting and adapting these methods in projects affecting hundreds—and then thousands—of software practitioners. Fortunately, with some extensions, the methods did scale to the challenge. Based on these experiences, in 2007 I published *Scaling Software Agility: Best Practices for Large Enterprises*, a book designed to help larger enterprises achieve the benefits of agile development.

Scaling Software Agility took a broad view of software methods and didn't focus much on software requirements. Even though the management of requirements continued to be a struggle for many agile teams, there were bigger organizational and cultural challenges, as well as a number of emerging agile technical practices, that needed to be addressed.

In the past couple of years, the movement to lean thinking in software development captured my interest, in part because I have some background in lean manufacturing

from earlier days. Generally, lean provides a comprehensive, deeply principled, rigorous, and mathematical framework for reasoning about product development economics and the increasingly important subset, software development.

So, my thinking, along with that of many others, evolved further. Many of us started to see agile development, especially agile at scale, as a "software instance of lean." In addition, lean scales beyond the software development labs and provides tools to address changes in other departments such as deployment, IT, distribution, and program and portfolio management. Simply put, lean provides a broader framework for organizational change, and it helps us address these larger challenges. I'm a big fan of lean thinking.

At its core, lean focuses on the value stream and provides philosophies, principles, and tools to continually decrease time to market, enhance value delivery, and eliminate waste and delays. As enterprises head down the lean path, it is again beneficial to focus on optimizing the understanding and implementation of software requirements, because they are the unique carriers—or at least the best proxy—for that value stream.

Lean thinking brings us full circle. Once again, it is useful to focus on requirements management practices in our agile—and increasingly lean—software development paradigm. That's why I wrote this book.

My hope is that the book will help the individual software practitioner, project team, program, and enterprise adopt and adapt agile and lean practices, deliver better solutions to their users and stakeholders, and thereby achieve the personal and business benefits that success engenders. After all, you can never be to too rich or too lean.

How to Read This Book

With this book, I'm hoping to tell a somewhat complex story—how to address the challenge of managing software requirements in an agile enterprise that may employ just a few developers building a single product to those employing hundreds or even thousands of software practitioners building systems of previously unseen complexity—in a practical, straightforward, and understandable manner.

To do so, the book is written in four parts, the last three of which are dedicated to describing specific agile requirement practices at increasing levels of sophistication and scale.

Part I, Overview: The Big Picture of Agile Requirements in the Enterprise

In Part I, we describe an overall process model intended to communicate the "Big Picture" of how to apply agile requirements practices at the project Team, Program, and Portfolio levels.

We provide a brief history of software methods, describing the evolution from waterfall through iterative and incremental development, to agile and lean. We describe the big picture of agile requirements—an organization, requirements, and process model that works for the team and yet scales to the full needs of the enterprise.

We then provide an overview of the model and illustrate how it can be applied in agile requirements for the team, agile requirements for the program, and agile requirements for the portfolio.

If you need an introduction and orientation to the concepts, terms, and general practices of managing agile requirements, this part is intended to stand alone.

Part II, Agile Requirements for the Team

In Part II, we describe a simple yet comprehensive model for managing requirements for agile project teams. This portion of the model is designed to be as lightweight as possible, quintessentially agile, and to not encumber the agile teams with any unnecessary complexity and overhead. We introduce the agile team, user stories, stakeholders, users and user personas, iterating, agile estimating and velocity, acceptance testing, the role of the product owner, and, finally, methods for discovering requirements.

If your teams are using agile, this comprehensive, explanatory guide to applying agile requirements is intended for you.

Part III, Agile Requirements for the Program

Part III is intended for those involved in building more complex systems that often require the cooperation of a number of agile teams. We expand the picture and introduce additional requirements artifacts, roles, organizational constructs, and effective practices designed for this purpose. We describe Vision, product and system features, the product Roadmap, the role of the product manager, the Agile Release Train, release planning, nonfunctional requirements, techniques for requirements analysis, and use cases.

If you are a developer, tester, manager, team lead, QA, architect, project or program manager, or development director/executive involved in building systems of this scope, this part is intended for you.

Part IV, Agile Requirements for the Portfolio

In Part IV, we describe the final, *Portfolio* level, of requirements practices. This level is intended to guide enterprises building ever-larger systems of systems, application suites, and product portfolios. These often require the coordination and cooperation of large numbers (20 or 50 or 100 or more) of agile project teams. We introduce additional requirements artifacts, roles, organizational constructs, and practices designed for this purpose. We describe the role that larger-scale, intentional, system-level architectures play in agile development. We introduce a kanban system for reasoning about how to evolve and, when necessary, rearchitect, such systems in an agile manner. We also describe some of the legacy thinking in portfolio and project management and give some suggestions as to what to do about it. We conclude with a chapter describing investment themes, epics, and, finally, one of the ultimate objectives—agile portfolio planning.

If you are a program manager, development director, system architect, executive, or portfolio manager or planner who is involved in managing investments for a portfolio of products, systems, software services, or IT applications, this part is intended for you.

This page intentionally left blank

Chapter 2

The Big Picture of Agile Requirements

This would all be a lot easier to understand if you could just draw me a picture.

—Anonymous senior executive

Effectively implementing a new set of lean and agile requirements principles and practices in a project team, program, or enterprise is no small feat. Even the language is different and seemingly odd (user stories, sprints, velocity, story points, epics, backlog?). In addition, further "leaning" the organization often requires eliminating or reducing requirements specifications, design specifications, stage-gated governance models (with incumbent requirements reviews), sign-offs (with incumbent delays…), implementing work-in-process limits (which may seem counterproductive to those who measure "utilization"), and so on. So, there will likely be many challenges.

Even for the fully committed, it can take six months to a year to introduce and implement the basic practices and even more time to achieve the multiples of productivity and quality results that pay the ultimate dividends in customer satisfaction, revenue, or market share. To achieve these benefits, we must change many things, including virtually all of our former requirements management practices. However, many of the existing required artifacts, milestones, and so on, serve as safeguards to "help" avoid the types of project problems that software has often experienced. So, we have a dilemma—how do we practice this new high-wire act without a safety net, when the safety net itself is a big part of the problem?

Fortunately, we are now at the point in time where a number of organizations have made the transition before us and some common patterns for lean and agile software process success have started to emerge. In our discussions with teams, managers, and executives during this transition, we often struggled to find a language for discussion, a set of abstractions, and an appropriate graphic that we could use to quickly describe "what your enterprise would look like and how it would work after such an agile transformation."

To do so, we need to be able to describe the new software development and delivery process mechanisms, 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 model, because those artifacts are the proxy for the value stream.

Eventually, and with help from others, we arrived at something that worked reasonably well for its purpose.[1] We call it the *Agile Enterprise Big Picture*, and it appears in Figure 2–1.

THE BIG PICTURE EXPLAINED

In this chapter, we'll explain the Big Picture in a summary format intended to provide the reader with a quick gestalt of this new, agile, leaner, and yet fully scalable software requirements model.

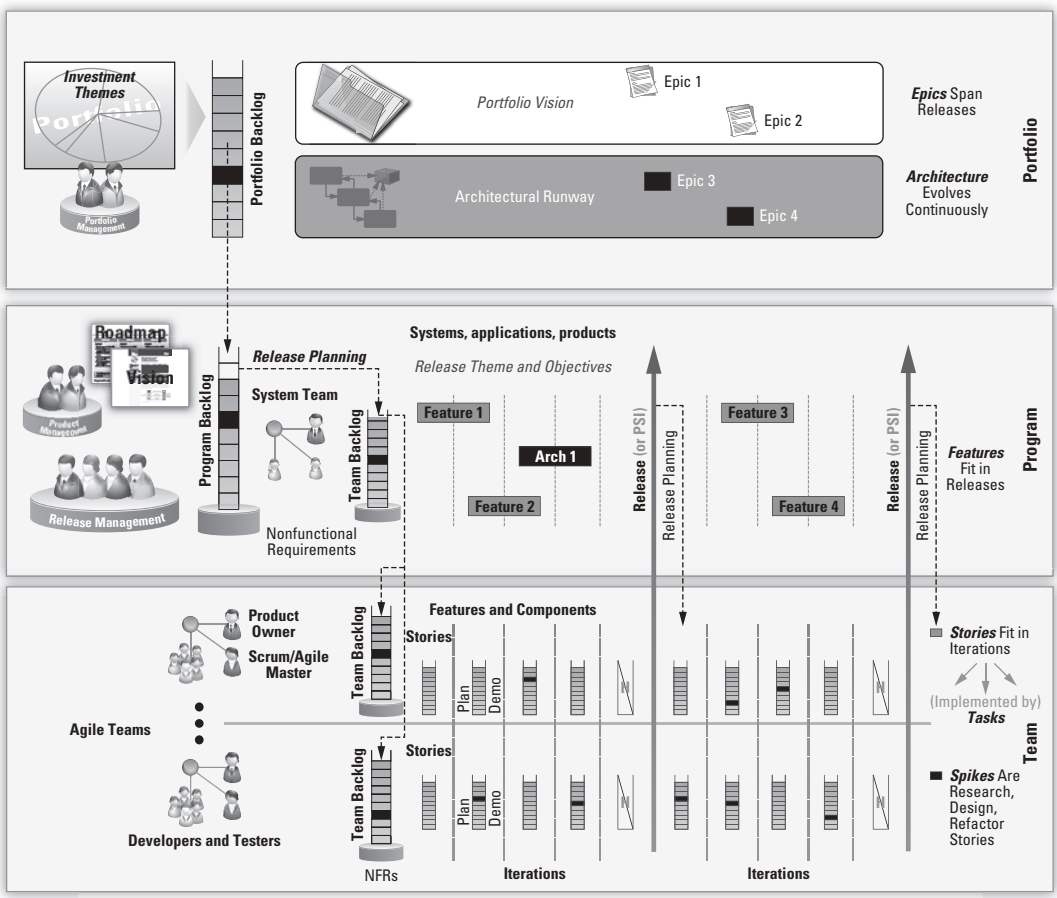


Figure 2–1 The Agile Enterprise Big Picture

1. Special thanks to Matthew Balchin and others at Symbian Software, Ltd., and Juha-Markus Aalto of Nokia Corporation.

In the remaining chapters of Part I of this book, we'll describe the basic big-picture requirements management practices for the individual *Team*, *Program*, and *Portfolio* levels. In Parts II, III, and IV, we'll further elaborate on the requirements management artifacts, roles, and activities at a level of detail suitable for implementation and action.

Big-Picture Highlights

Because this picture serves as both the organizational and process model for our agile requirements practices, we'll have time throughout this book to explore its many nuances. However, from an overview perspective, the following highlights emerge.

The Team Level

At the *Team level*, agile teams of 7±2 team members define, build, and test *user stories* in a series of *iterations* and *releases*. In the smallest enterprise, there may be only a few such teams. In larger enterprises, groups, or *pods*, of agile teams work together to support building up larger functionality into complete products, features, architectural components, subsystems, and so on. The responsibility for managing the *backlog* of user stories and other things the team needs to do belongs to the team's *product owner.*

The Program Level

At the *Program level*, the development of larger-scale systems functionality is accomplished via multiple teams in a synchronized *Agile Release Train* (ART). The ART is a standard cadence of timeboxed iterations and milestones that are date- and quality-fixed, but scope is variable (no iron triangle). The ART produces *releases* or *potentially shippable increments* (PSIs) at frequent, typically fixed, 60- to 120-day time boundaries. These evaluable increments can be released to the customer, or not, depending on the customer's capacity to absorb new product as well as external events that can drive timing.

We'll use the generic *product manager* label as the title for those who are responsible for defining the features of the system at this level, though we'll also see that many other titles can be applied to this role.

The Portfolio Level

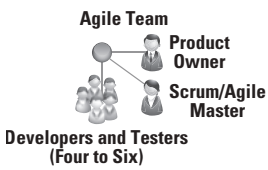
At the *Portfolio level*, we'll talk about a mix of *investment themes* that are used to drive the investment priorities for the enterprise. We'll use that construct to assure that the work being performed is the work necessary for the enterprise to deliver on its chosen business strategy. Investment themes drive the portfolio vision, which will be expressed in as a series of larger, *epic*-scale initiatives, which will be allocated to various release trains over time.

In the rest of this chapter, we'll walk through the various elements of the Big Picture to describe how it works. While we'll highlight the requirements value delivery stream, we'll also expose the rest of the picture including the roles, teams, and processes that are necessary to deliver value. In this way, we'll provide a systemic view of our *lean and agile requirements process that works for teams and yet scales to the full needs of the enterprise*.

BIG PICTURE: TEAM LEVEL

Figure 2–2 summarizes the Team level of the Big Picture.

The Agile Team



The "front line" of software development consists of some number of *agile teams* that implement and test code and collaborate on building the larger system. It's appropriate to start with the team, because in agile, the *team is the thing*, because they write and test all the code that delivers value to the end user. Since it's an agile team, each has a maximum of seven to nine members and includes all the roles necessary to define/build/test[2] the software for their *feature or component*. The roles include a Scrum/Agile Master, product owner, and a small team of dedicated developers, testers and (ideally) test automation experts, and maybe a tech lead.

In its daily work, the team is supported by architects, external QA resources, documentation specialists, database specialists, source code management (SCM)/build/infrastructure support personnel, internal IT, and whoever else it takes such that the core team is fully capable of *defining, developing, testing, and delivering working and tested* software into the system baseline.

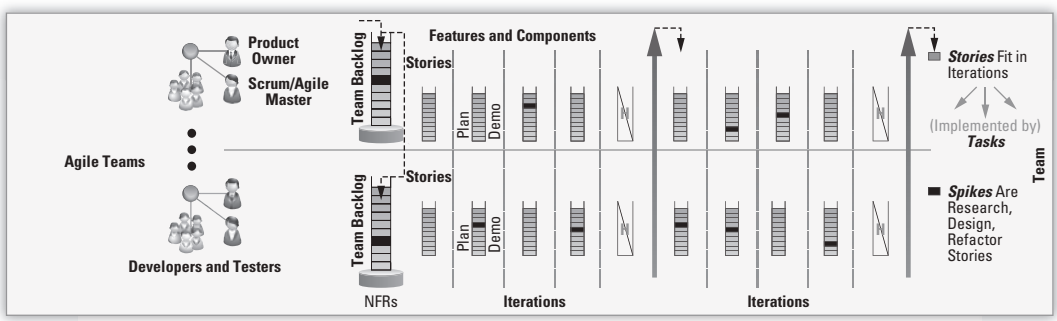


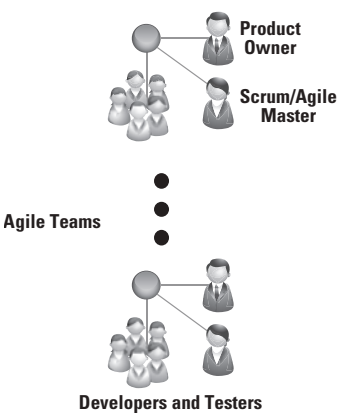
Figure 2–2 Team level of the Big Picture

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

Since testing software is integral to value delivery (teams get *no* credit for untested code), testers are integral to the team. Often the testers are logically part of the QA organization but are physically assigned and dedicated to an agile team. In this matrix fashion, their primary allegiance is to the team, but as members of the QA organization, they can leverage other QA teammates and managers for skills development, automation expertise, and any specialty testing capabilities that may be necessary at the system level. In any case, it must be clear that the agile team itself is responsible for the quality of their work product and that responsibility cannot be delegated (or abrogated!) to any other organization, in or out of house.

Teams are typically organized to deliver software *features or components*. Most enterprises will have a mix of both types—some *component teams* focused on shared infrastructure, subsystems, and persistent, service-oriented architectural components and some *feature teams* focused on vertical, user-facing, value-delivery initiatives. Agile teams are self-organizing and reorganize when necessary based on the work in the program backlog. Over time, the makeup of the teams themselves is more dynamic than static—static enough to "norm, storm, and perform"[3] for reasonable periods of time and dynamic enough to flex to the organization's changing priorities.

Pods of Agile Teams



In addition, within the larger enterprise, there are typically some number (three to ten) or so of such teams that cooperate to build a larger feature, system, or subsystem (the *program* domain in the Big Picture). Although 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 to 100 people must intensely collaborate on building their "next bigger thing" in the hierarchy, which we'll call a *program*. As we'll discover later, this is also about the maximum size for face-to-face, collaborative *release planning*.

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

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*.

Roles in the Agile Team

Product Owner

As we have described, Scrum is the dominant agile method in use, and the product owner role is uniquely, if 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 our 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.

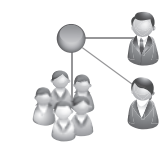
But the product owner's responsibilities don't end there. In support of Agile Manifesto principle #4—*Business people and developers must work together daily throughout the project*—the product owner is ideally co-located with the team and participates *daily* with the team and its activities.

Scrum/Agile Master

For teams implementing Scrum, the Scrum Master is an important (though sometimes transitional[4]) role. The Scrum Master is the 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.

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 by illustrating their ability to deliver user value 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 typically limited to about three to four developers plus one to two testers, who are (ideally) co-located and work together to *define, build, test, and deliver* stories into the code baseline.

Iterations

In agile development, new functionality is built in short timeboxed events called *iterations* (*sprints* in Scrum). In larger enterprises, agile teams typically adopt a

4. As the teams master the agile process, the role becomes less critical. Some very agile teams, even those who have adopted Scrum, no longer have a Scrum Master per se. Everybody knows the rules, and they are self-enforced.

standard iteration length and share start and stop boundaries so that code 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*, *inspect and adapt*, *repeat.*

The iteration is the "heartbeat of agility" for the team, and teams are almost entirely focused on developing new functionality in these short timeboxes. In the Big Picture, the iteration lengths for all teams are the same since that is the simplest organizational and management model. Although there is no mandated length, most have converged on a recommended length of *two weeks.*

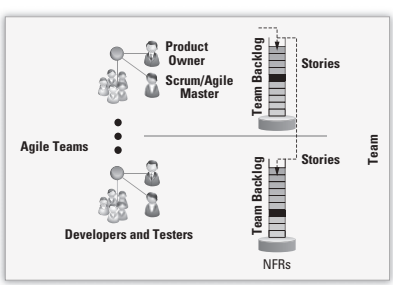
Number of Iterations per "Release"

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 *potentially shippable increment* (PSI). Many teams apply this model with four to five development iterations and one hardening iteration per release, creating a cadence of a potentially shippable increment about every 90 days. This is a fairly natural production rhythm that corresponds to a reasonable external release frequency for customers, and it 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, are left to the judgment of the enterprise.

User Stories and the Team Backlog



User stories (*stories* for short) are the general-purpose agile substitute for what traditionally has been referred to as *software requirements* (the stuff in the middle of the iron triangle of Chapter 1).

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 objects that carry the customer's requirements through the value stream—from needs analysis though code and implementation.*

As opposed to requirements (which by common definition represent something the system *must* do to fulfill a business need or contractual obligation), user stories are *brief statements of intent* that describe something the system *needs* to do for some *user*. As commonly taught, the user story often takes a standard user-voice form of the following:

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

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

Team Backlog

The team's backlog (typically called a *project* or *product* 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. Although there may be other things in the team's backlog as well—defects, refactors, infrastructure work, and so on—the yet-to-be-implemented user stories 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.

Therefore, we will spend much of the rest of this book further describing processes and practices around user stories.

Tasks

For more detailed tracking of the activities involved in delivering stories, teams typically 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, because this keeps the team focused on business value, rather than individual tasks. Tasks provide a micro–work breakdown structure that teams can use (or not) to facilitate coordinating, estimating, tracking status, and assigning individual responsibilities to help assure completion of the stories—and thereby—the iteration.

BIG PICTURE: PROGRAM LEVEL

Figure 2–3 summarizes the Program level of the Big Picture.

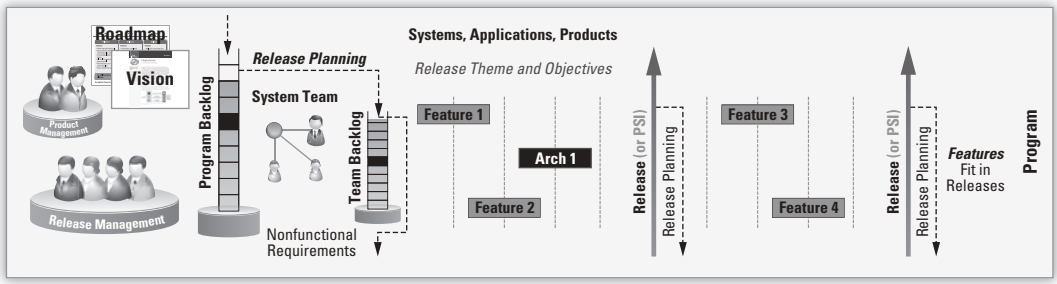
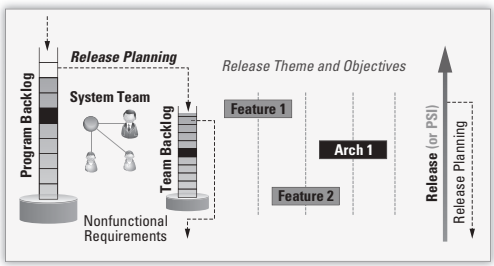


Figure 2–3 The Program level of the Big Picture

Here, we find additional organizational constructs, roles, processes, and requirements artifacts suited for building larger-scale systems, applications, products, and suites of products.

Releases and Potentially Shippable Increments



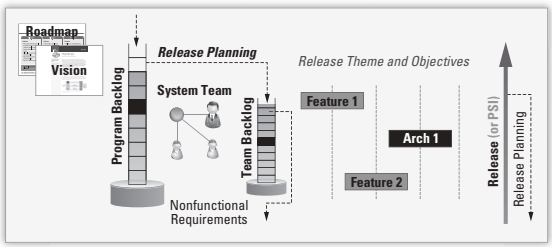
Although the goal of every iteration is to produce a shippable increment of software, teams (especially larger-scale enterprise teams) find that it may simply not be practical or appropriate to ship an increment at each iteration boundary. For example, during the course of a series of iterations, the team may accumulate some *technical debt* that needs to be addressed before shipment. Technical debt may include things such as defects to be resolved, minor code refactoring, deferred system-wide testing for performance, reliability, or standards compliance, or finalization of user documentation. *Hardening iterations* (indicated by an iteration with an empty backlog) are included in the Big Picture to provide the time necessary for these additional activities.

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

- Potential interference with a customer's licensing and service agreements
- Potential for customer overhead and business disruption for installation, user training, and so on
- Potential for disrupting customer's existing operations with minor regressions or defects

For these and other reasons, most programs aggregate a series of iterations into a potentially shippable increment, which can be released, or not, based on the then-current business context.

Vision, Features, and the Program 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 the following.

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

The Primary Content of the Vision Is a Set of Features

A Vision may be maintained in a document, in a backlog repository, or even in a simple briefing or presentation form. But no matter the form, the prime content of the Vision document is a prioritized set of *features* intended to deliver *benefits* to the users.

Nonfunctional Requirements

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

Undelivered Features Fill the Program Backlog

In a manner similar to the team's backlog, which contains primarily *stories*, the program (or *release*) backlog contains the set of desired and prioritized *features* that have not yet been implemented. The program 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.

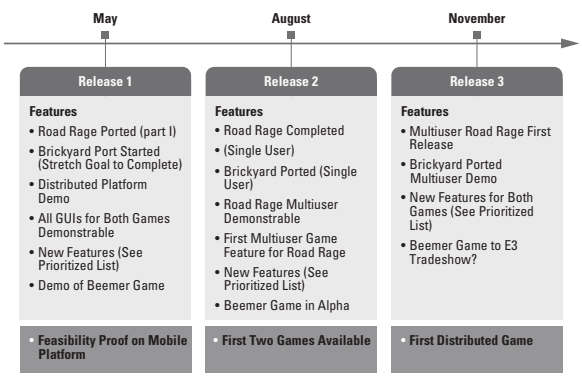
Release Planning

In accordance with emerging agile enterprise practices, each release increment timebox 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 a 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 PSI timebox. They also negotiate scope trade-offs 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. In addition to the plan itself, another primary result of this process is a commitment to a set of release objectives, along with a prioritized feature set.

Thereafter, the teams 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



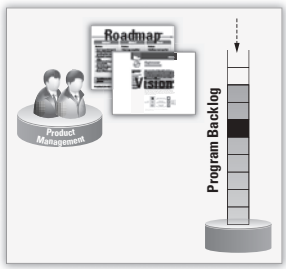
An Updated, Themed, and Prioritized "Plan of Intent"

The results of release planning are used to update the (product or solution) *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, a 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 enterprise's 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 generally be used to create any external commitments.

Product Management



In agile, there can be a challenge with the apparently overlapping responsibilities of the *product manager* and the product owner. For example, in Scrum, the product owner is responsible for the following:

> 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.[5]

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 market or program-based *product managers*, who carry out their traditional responsibilities of both defining the product *and* presenting the solution to the marketplace.

However, we also note that the title of the person who plays this role may vary by industry segment, as shown in Table 2–1.

Responsibilities of the Agile Product Manager in the Enterprise

No matter the title (we'll continue to use *product manager* generically), when an agile transition is afoot, the person playing that role must fulfill the following primary responsibilities:

- Own the Vision and program (release) backlog
- Manage release content
- Maintain the product Roadmap
- Build an effective product manager/product owner team

5. [Schwaber 2007]

Table 2–1 Product Manager Role May Have Different Titles

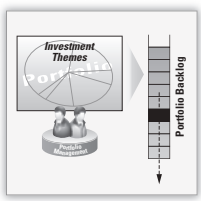
Industry Segment	Common Title for the Role
Information systems/information technology (IS/IT)	Business owner, business analyst, project or program manager
Embedded systems	Product, project, or program manager
Independent software vendor	Product manager

Big-Picture Elements: Portfolio Level

Figure 2–4 summarizes the Portfolio level of the Big Picture.

At the top of the Big Picture, we find the portfolio management function, which includes those individuals, teams, and organizations 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 together create the *portfolio vision*.

Investment Themes



A set of *investment themes* establishes the relative investment objectives for the enterprise or business unit. These themes drive the vision for all programs, and new epics are derived from these themes. The derivation of these decisions is the responsibility of the portfolio managers, either line-of-business owners, product councils, or others who have fiduciary responsibilities to their stakeholders.

The result of the decision process is a set of themes—*key product value propositions that provide marketplace differentiation and competitive advantage*. 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.

Epics and the Portfolio Backlog

Epics represent the highest-level expression of a customer need. Epics are development initiatives that are intended to deliver the value of an investment 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.

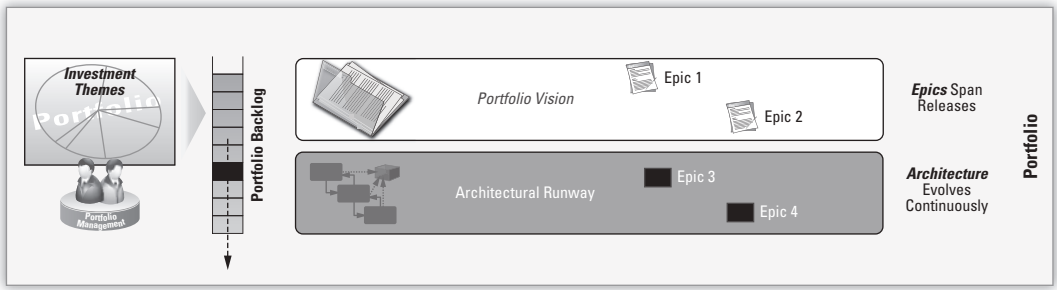


Figure 2–4 Portfolio level of the Big Picture

Epics may be expressed in bullet form, in user-voice story form, as a sentence or two, in video, in a 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

In Chapter 1, we described how design (architecture) and requirements are simply two sides of the same coin—the "what" and the "how." In this book, we'll have time to explore this topic in more detail, and we'll provide some discriminators that help us think about the differences in architecture and requirements, as well as the commonalities. However, even though this book focuses on requirements, we can't ignore architecture, because experience tells us that teams that build some amount of *architectural runway*, which is the ability to implement new features without excessive refactoring, will eventually emerge as the winners in the marketplace. So, any effective treatment of agile requirements must address the topic of architecture as well.

Therefore, system architecture is a first-class citizen of the Big Picture and is a routine portfolio investment consideration for the agile enterprise.

Summary

In this chapter, we introduced the Big Picture as the basic requirements artifact, process, and organizational model for managing software requirements in a lean and agile manner. For agile teams, the model uses the minimum number of artifacts, roles, and practices that are necessary for a team to be effective. However, the model expands as needed to the Program and Portfolio levels, in each case providing the leanest possible approach to managing software requirements, even as teams of teams build larger and larger systems of systems. In the next few chapters, we'll elaborate on each of these levels.