LEAN-AGILE SOFTWARE DEVELOPMENT

Achieving Enterprise Agility

ALAN SHALLOWAY
GUY BEAVER
JAMES R. TROTT

Net Objectives
Lean-Agile Series

Illustrations by Andrea Chartier Bain

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/netobjectives

Library of Congress Cataloging-in-Publication Data

Shalloway, Alan.
Lean-agile software development : achieving enterprise agility / Alan Shalloway, Guy Beaver, James R. Trott.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-53289-9 (pbk. : alk. paper) 1. Agile software development. I. Beaver, Guy. II. Trott, James. III. Title.
QA76.76.D47S47 2009
005.1—dc22

2009032621

ISBN-13: 978-0-321-53289-3
ISBN-10: 0-321-53289-9

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, October 2009

Series Foreword
The Net Objectives Product Development Series

Alan Shalloway, CEO, Net Objectives

If you are like me, you will just skim this foreword for the series and move on, figuring there is nothing of substance here. That would be a mistake. Unless you have read this foreword in another book in the series, please take a moment with me at the outset of this book (if you've already read a foreword from another book, please skip a couple of pages to This Book's Role in the Series).

I want you to consider with me a tale that most people know but don't often think about. That tale illustrates what is ailing this industry. And it sets the context for why we wrote the Net Objectives Product Development Series and this particular book.

I have been doing software development since 1970. To me, it is just as fresh today as it was four decades ago. It is a never-ending source of fascination to me to contemplate how to do something better, and it is a never-ending source of humility to confront how limited my abilities truly are. I love it.

Throughout my career, I have also been interested in other industries, especially engineering and construction. Now, engineering and construction have suffered some spectacular failures: the Leaning Tower of Pisa, the Tacoma Narrows Bridge, the Hubble telescope. In its infancy, engineers knew little about the forces at work around them. Mostly, engineers tried to improve practices and to learn what they could from failures. It took a long time—centuries—before they acquired a solid understanding about how to do things.

No one would build a bridge today without taking into account long-established bridge-building practices (factoring in stress, compression, and the like) but software developers get away with writing code based on "what they like" every day, with little or no complaint from their peers. Why do we work this way?

But this is only part of the story. Ironically, much of the rest is related to why we call this the "Net Objectives Product Development Series." The Net Objectives part is pretty obvious. All of the books in this series were written either by Net Objectives staff or by those whose views are consistent with ours. Why Product Development? Because when building software, it is always important to remember that software development is really product development.

By itself, software has little inherent value. Its value comes when it enables delivery of products and services. Therefore, it is more useful to think of software development as part of product development—the set of activities we use to discover and create products that meet the needs of customers while advancing the strategic goals of the company.

Mary and Tom Poppendieck, in their excellent book, *Implementing Lean Software Development: From Concept to Cash* (2006), note:

> It is the product, the activity, the process in which software is embedded that is the real product under development. The software development is just a subset of the overall product development process. So in a very real sense, we can call software development a subset of product development. And thus, if we want to understand lean software development, we would do well to discover what constitutes excellent product development.

In other words, software in itself isn't important. It is the value that it contributes—to the business, to the consumer, to the user—that is important. When developing software, we must always remember to look to what value is being added by our work. At some level, we all know this. But so often organizational "silos" work against us, keeping us from working together, from focusing on efforts that create value.

The best—and perhaps only—way to achieve effective product development across an organization is a well-thought-out combination of Lean principles to guide the enterprise, Agile practices to manage teams, and technical skills (test-driven development, design patterns). That is the motivation for the Net Objectives Product Development Series.

Too long, this industry has suffered from a seemingly endless swing of the pendulum from no process to too much process and then back to no process: from heavyweight methods focused on enterprise control to disciplined teams focused on the project at hand. The time has come for management and individuals to work together to maximize the produc-

tion of business value across the enterprise. We believe Lean principles can guide us in this.

Lean principles tell us to look at the systems in which we work and then relentlessly improve them in order to increase our speed and quality (which will drive down our cost). This requires

1. Business to select the areas of software development that will return the greatest value
2. Teams to own their systems and continuously improve them
3. Management to train and support their teams to do this
4. An appreciation for what constitutes quality work

It may seem that we are very far from achieving this in the software-development industry, but the potential is definitely there. Lean principles help with the first three and the understanding of technical programming and design has matured far enough to help us with the fourth.

As we improve our existing analysis and coding approaches with the discipline, mindset, skills, and focus on value that Lean, Agile, patterns, and test-driven development teach us, we will help elevate software development from being merely a craft into a true profession. We have the knowledge required to do this; what we need is a new attitude.

The Net Objectives Product Development Series aims to develop this attitude. Our goal is to help unite management and individuals in work efforts that "optimize the whole":

- **The whole organization** Integrating enterprise, team, and individuals to work best together.
- **The whole product** Not just its development, but also its maintenance and integration.
- **The whole of time** Not just now, but in the future. We want sustainable ROI from our effort.

This Book's Role in the Series

While Scott Bain's *Emergent Design: The Evolutionary Nature of the Software Profession* dealt with how to raise the bar in technical practices, this book is about how to raise the bar in product and project management. Both

books, as I suspect all books in the series will be, are based on the belief that there are laws (rules) that must be followed to be effective and efficient.

As Agile has matured, we're finding it useful to go beyond the mere mandate of building in stages and having teams solve their own problems. While both are sage advice, more is needed as our products become more complex. Management needs to become more intimately involved in solving the problems teams face. And although the development teams are the ones who actually deliver the value, they are not empowered to solve the organizational and cultural problems that get in their way.

We believe that Lean thinking provides a new way for management and teams to work together. We further believe that the next generation of Agile methods will be those that promote this cooperative effort instead of being neutral at best and negative at worst. This book is therefore about raising software development closer to a professional level throughout the organization.

The End of an Era, the Beginning of a New Era

I believe the software industry is at a crisis point. The industry is continually expanding and becoming a more important part of our everyday lives. But software development groups are facing dire problems. Decaying code is becoming more problematic. An overloaded workforce seems to have no end in sight. Although Agile methods have brought great improvements to many teams, more is needed. By creating a true software profession, combined with the guidance of Lean principles and incorporating Agile practices, we believe we can help uncover the answers.

I hope you find this book series to be a worthy guide.

Preface

This book was born from need and from knowledge. The need is to expand the knowledge base of software development in both the management and process worlds so as to create a new base. Integrating Agile has transformed the software-development process in less than a decade. Although its mandate applies to all of software development, its focus typically has been on the teams directly involved in the development of software and on the projects they work on. As Agile has begun to transcend the early-adopter phase and move on to the early-majority phase, there are new challenges to address as Agile is applied to quite different situations.

- Larger organizations are attempting to adopt Agile for the first time.
- Organizations that are already using Agile are expanding the scale of their adoption.
- Organizations that are somewhat dysfunctional are starting to adopt Agile.

Extending Agile to these new situations creates the need for a better understanding of what Agile is and a broader set of tools to apply Agile. These two issues are surprisingly tightly related. Many Agile early adopters have learned from any number of excellent books that present a set of practices, mostly oriented around the team. Unfortunately, few of these books explain why Agile works. Rather, they are filled with excellent practices that embody Agile's fundamental belief systems while providing a set of practices that work at the team level in many situations.

The wider adoption of Agility demands more. There is now a need for a greater scope of knowledge as well as an explanation of why the practices work. While almost all Agile methods sprang up independently of Lean thinking, Lean thinking provides insight into why Agile works. This is why most of Agile's methods are compatible with Lean. True knowledge is realized when one can apply principles and practices together to form solid understanding. We use the term "Lean-Agile" for the approach described in this book because it represents our contention that for Agile to work most effectively, it must be applied within the context of Lean.

This book fills the need both to understand why Agility works as well as to expand its base of principles and practices in order to apply it to the enterprise. It builds on the work of others, most particularly, those of David Anderson, Kent Beck, Jane Cleland-Huang, Alistair Cockburn, Jim Coplien, Ward Cunningham, W. Edwards Deming, Mark Denne, Ron Jeffries, Daniel Jones, Michael Kennedy, Corey Ladas, David Mann, Bob Martin, Rick Mugridge, Taichi Ohno, Mary Poppendieck, Tom Poppendieck, Don Reinertsen, Peter Scholtes, Ken Schwaber, Jeff Sutherland, James Womack, Alan Ward, and so many others. This blend of Lean, Agile, XP, Scrum, and other disciplines creates the synergistic blend essential to providing answers, both deep and broad, that the enterprise requires.

I want to give particular thanks to a few people who have helped us personally in our endeavors.

- Mary and Tom Poppendieck for helping me get my start in Lean training. Both have been invaluable to my personal development with their combination of suggestions for improvement tailored by respect and compassion.
- Don Reinertsen for his kindness and encouragement, not to mention the amazing amount of knowledge that his books have conveyed to the community.
- David Anderson for his outspokenness and out-of-the box thinking. He's been an inspiration to go further in my thinking than I have typically dared.
- Ward Cunningham. I know few people smarter than Ward, balanced with such an unassuming nature. His wisdom and manner have been invaluable.
- Our own Alan Chedalawada, who may not have contributed to the writing in this book, but whose ideas formed the basis for much of

what we are presenting here that is new. Many of these ideas he first manifested in the real world.

- Our own Amir Kolsky and Ken Pugh for insights into the role of acceptance test-driven development.

While it may seem odd for one author to acknowledge another, I must acknowledge Jim Trott—both a close associate and one of my dearest friends. Without his encouragement, hard work, and efforts on keeping me focused, this book may not have happened.

Alan Shalloway
CEO, Net Objectives
Achieving Enterprise and Team Agility

Introduction

"We can't solve problems by using the same kind of thinking we used when we created them." —Albert Einstein

One of the goals of this book is to give you a better perspective on Lean and Agile and how to use them in software development. This requires an understanding of the roots of Agility, the software development "pendulum," and the importance of paradigms and practices and of being pragmatic. Lean offers a way forward.

This book takes the reader beyond Agile's standard practices by teaching how to incorporate Lean thinking into software development. Although Agile, as it is usually practiced, is effective at the team level, it gives little guidance on how it fits at the enterprise level. This is somewhat for historical reasons, as you will see. Lean-Agile is an approach to Agile software development using Lean principles and practices for guidance.

You can think of Agile in one of two ways: as a set of values and beliefs that leave it to the practitioners to decide how to apply them or as a set of practices that are suggested to manifest good results. Practitioners typically use a combination of both, believing the mandate of the Agile Manifesto and then using either Scrum[1] or eXtreme Programming[2] (or some of each) as the basis for their methods. The challenge with this approach is two-fold—one resulting from the roots of Agility and the other from the lack of a theoretical foundation for the Agile practices themselves—as we will discuss later.

1. Scrum is a popular Agile process created by Jeff Sutherland and Ken Schwaber. It is commonly used at the team level and is characterized by self-organizing, cross-functional teams doing iterative development in what are called sprints.
2. eXtreme Programming is an iterative development process for teams centered around several engineering practices. The most common of these is test-driven development, paired programming, and continuous integration.

How This Book Will Help You

This book aims to change how you look at software development. Doing so will enable you to solve seemingly intransigent problems with much less effort than you might have thought possible. One of our guiding principles is that we need to drive from business value: Deliver the value (software) that will provide the greatest return to the business by providing the greatest value to the business's customers. For an IT development group, this could mean either internal or external customers.

Together, we will explore what software development actually is and how it must be managed. We will investigate ways to help our customers through the process of selecting what work to accomplish through development, deployment, and, ultimately, ongoing support and enhancement.

We will drive from principles throughout the book and provide a good many that you can apply in Lean Software Development. This book will not give you all the answers; instead, it will help direct your thinking so you can create answers that will work for you in your company, in your situation, for your customers, and with your products.

The Roots of Agility

The development of the Agile Manifesto (Beck et al. 2001) was a breakthrough event for the software industry. The manifesto, shown in Figure I.1, and its Twelve Principles, shown in Figure I.2, describe the essential ideology that underpins Agile software development.

The Software Development Pendulum

The Manifesto is a strong statement. It is consistent with the intentions of most people in the software development industry. But it says that we must develop in a way that is different from the ways we have often tried in the past. It stands in opposition to the myth that the way to create quality, sustainable software is to conceive large plans and then use command-and-control management[3] to realize them. When it was written, the Agile Manifesto presented a great opportunity for exploring new, better ways of

3. Apologies to military experts who properly use this term to mean vision at the top with implementation at the bottom. We're using this term in the way most people interpret it—top-level people telling lower-level people how to get their job done.

Manifesto for Agile Software Development[4]

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Figure I.1 Manifesto for Agile Software Development

Principles behind the Agile Manifesto

We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances Agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Figure I.2 Twelve Principles behind the Agile Manifesto

4. Copyright © 2001 Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas; this declaration may be freely copied in any form, but only in its entirety through this notice.

developing software. Unfortunately, it also left a huge hole. It did not attempt to describe how to achieve the promise.

This lack of instructions is not a shortcoming of the Agile Manifesto. The Manifesto's purpose was to create a vision for a better way to develop software. It is instructive to look at the Manifesto in its historical context. During the decades preceding the Manifesto, the principles of and approaches to software management swung like a pendulum, between free-form and command-and-control, from little process to too much process. Each was responding to the challenges of the other.

In the 1960s, several large system failures demonstrated the need for both better engineering methods and better processes. Certainly, software development during this time was not an ad-hoc affair, but the industry was new and there was little experience with large-scale systems. In the 1970s, the idea of software as "engineering" surfaced. We began to use structured analysis and design, top-down programming, and structured programming (`goto` statements were considered bad form). Notably, the Waterfall model emerged. The industry was growing up, and standard practices for design, programming, and management arose. By the 1980s, PCs and fourth-generation languages enabled small projects to flourish. Small teams produced much more software than large teams did. Prototyping was popular. Speed was king. If you were the first, you were the best.

But quality often suffered. Speed to entry was so important that a product's sustainability was often ignored. This led to different kinds of failures. Since it was easy for anyone to enter the market, the competitive edge of getting in first was lost if the product lacked quality. Failures in this era triggered an upsurge in rigorous process. The sense was that if we can't do it ad hoc, then we'd better control it.

Tick tock. Tick tock. The pendulum continued to swing. Maybe even faster.

The 1990s brought us the Capability Maturity Model (CMM). Y2K dominated the last few years of the decade, emphasizing the need for planning ahead. But the '90s also brought us the Internet, which again enabled small teams to have great impact. The dot-com boom brought rapid software development. Again, a proliferation of small teams found initial success but subsequently had difficulty maintaining the software that they had developed.

Now, the twenty-first century has given rise to Agility—small teams working with customers to develop software quickly. There have been many successes and there have been many failures.

And the pendulum continues to swing. What can we do to stop it? Or can we at least find a balance?

The Agile Manifesto was an attempt to find such balance. Let's respect our teams. Let's respect our customers. Let's work with the business. Process can be good, but process that doesn't help a team get its job done is not.

Unfortunately, the world is messy and the promise of the Manifesto has not been entirely realized. The Manifesto itself showed the potential, but it did not provide a means to stop the pendulum. In fact, it has been used to justify letting teams rule. We have mostly lost the enterprise view, because that view seems to lead right back down the path of command-and-control management. If the choice is between that and using teams with Agility, then abandoning command-and-control seems reasonable.

It is not either-or. There is a way to balance command-and-control with the need for effective teams. Lean provides the way. To see why, we must first examine the beliefs, principles, and paradigms on which we build our thinking.

Principles and Paradigms

A principle is a comprehensive and fundamental law, doctrine, or assumption. Principles may exist at the level of the individual, may be held by a community, or may even apply universally. For example, individual principles may relate to one's integrity or one's way of living. A communal set of principles might include moral or religious beliefs or a set of beliefs that the community accepts as the true way to be living. Universal principles are those that apply everywhere—beyond the effect of the beliefs of any set of individuals. Perhaps we should actually call these laws of the universe. Principles are often stated in the form of guidance since there is often a corresponding principle (law) that should be followed. For example, one of the best known Lean principles is "eliminate waste." That's not a law as much as it is something you should do, as a rule.

A paradigm is a combination of assumptions, values, beliefs, and practices that define how to view reality, how to look at a situation. It is a worldview that characterizes what is true. Paradigms tend to last a long time (consider how long people believed the earth was the center of the universe). Paradigms are shared by a particular community or group of people. In the software world, Waterfall, Scrum, and Lean-Agile each have their own paradigms, or way of looking at how to best build software.

Since a paradigm defines what is real and true for someone, changing one's paradigm is quite difficult. It requires the individuals and their community to grapple with the underlying assumptions, values, and beliefs and assess whether the paradigm actually squares with what is indeed "real" or whether some shift is required.

The paradigms we hold constrain what we consider possible and shape what we do. Unexamined paradigms can therefore be very limiting.

A Pragmatic Approach

Software development professionals are pragmatists (pragmatism is part of our worldview). We favor what works over what is represented as theoretically "correct." It is not that theory is bad, but theory must be grounded in real work if we are going to embrace it.

With that in mind, we would like to suggest taking a pragmatic approach to evaluating the essential paradigms we, as software developers, hold. That approach is to use the scientific method in whatever we do: Propose a hypothesis and then run an experiment to validate or invalidate it. If the experiment supports the hypothesis, then we have some evidence that the hypothesis is correct. If it doesn't, then the hypothesis is incorrect and must be modified.

We suggest that in the software development world, our processes must be consistent with our hypotheses about the best way to practice software development. If we get good results, we have evidence that our process (that is, our hypothesis) is good. If we get poor results, our process needs updating.

Critique the Process, Work Together

Let's be clear: This is all about critiquing the process, not the people involved. How many teams have run into problems because they are following a poor process and yet management, being overly committed to the process, blames the people? Assuming the process is right, they believe "if only the people had done it right, it wouldn't have been such a disaster."

Or how many projects have failed because teams decide to follow their own approaches regardless of the larger needs of the business? They assume management is just getting in the way—bureaucrats who must be worked around.

It seems that the tendency is for management to over-focus on process while teams underestimate its value. One side sees management as crucial to making the process work; the other wants to be protected from management's command-and-control mentality so that they can just get their work done. And so they go back and forth, not working in concert.

What we need is a new attitude about process and how to manage process. Processes must be designed to assist the team in achieving management's goals. Processes help the team get its job done: They represent accountability among team members about how they will work. The team is the steward of its processes—creating, sustaining, and improving them so that the team can improve constantly. Processes are dynamic: They are the team's baseline for change.

Lean Provides the Way Forward

Is this possible? Yes! Lean provides the principles we need to do this. And we will not follow these principles blindly. Blind faith doesn't work. Instead, we will use Lean as a guide and use our own experience to refine our own process.

If you have been building software for a few years, we invite you to use the hypothesis-and-test approach yourself: Run "backward-looking experiments,"[5] that look back over your own past experiences to validate or invalidate the process we are developing. This is a lot more pragmatic and a lot less painful than trying new processes on future projects. You will be able to verify relatively quickly whether the process works.

As we do this, we will be building a pragmatic "theory" about why and how software development works. We recognize the truth in Jan L.A. van de Snepscheut's or Yogi Berra's comment "in theory, theory and practice are the same, but in practice, they are different." We also believe Kurt Lewin's notion that sometimes "there is nothing more practical than a good theory." In other words, do not follow theory when it does not match practice. But when you are not sure what to do, an understanding of why your practices work may give you guidance in unfamiliar situations.

---

5. A backward-looking experiment is a term Alan Shalloway coined to mean looking into your past to validate or invalidate an hypothesis made in the present. For example, if someone says "coding conventions help" he is actually postulating that coding conventions result in better code. You can actually look into your past to see when that was true (adding evidence to the hypothesis) or when it was false (disproving the hypothesis). If you disprove the hypothesis, you can modify it with a condition to see if there is a set of circumstances that would make it true. This enables us to learn about and test our understanding by taking advantage of our past experience.

This pragmatic approach embraces the principles (or laws) that we have discovered work in all situations. For example, the principle that overloading an individual with work, that is, giving her many tasks to do at the same time, degrades her performance.

Principles lead to many practices. However, practices must change depending upon the context, or situation, in which they are used. Relying totally on principles may not work unless the principles are proven. Relying totally on practices will work only if you are in situations you've been in before. Effectiveness requires a proper blend of proven principles with practices appropriate for the situation in which they are being used.

Evaluating Paradigms

As we begin this approach, let's look at some of the core beliefs upon which the Waterfall model and the Agile framework are based. These are described in Figures I.3 and I.4. Are these universal principles? Or are they unexamined paradigms—rules that just must be followed?

We believe that the core beliefs of Agile are more helpful than the core beliefs of Waterfall. Agile's beliefs are helpful, but they are not enough. To follow them effectively, more is required. That is where Lean comes in.

The Core Beliefs of Waterfall

- You can know everything required to build a software product properly at the start of the project.
- Customers can accurately tell you what they want at the start of the project.
- You don't need to get feedback from the customer until the end of the project.
- Managers, developers, and customers can gauge the status of a project by looking at completed milestones as reflected in documentation. That is, given proper documentation, it is not necessary to deliver complete, tested software until the very end of the project.
- You can effectively have separate groups do analysis, design, code, and test. That is, there is little loss of information in the handoff between these groups.
- Handoffs between people in different roles can be done efficiently by writing down what was done in each step.
- You can test at the end of a project and achieve the required quality.
- Management can demand that certain work be done at a certain time and should expect it to happen.
- Giving people many projects to work on simultaneously is a good approach to achieving 100% productivity because then everyone is always busy.

Figure I.3 Core beliefs of Waterfall

The Core Beliefs of Agile

- You cannot know everything required to build a software product at the start of the project.
- Customers cannot accurately tell you what they want at the start of the project; instead, they will gain clarity as the project proceeds.
- You want feedback from the customer as often as possible and you want to give developers feedback on how they are doing as soon as possible.
- Working code is the most accurate way of seeing the progress of the development effort.
- Groups working together minimizes delays as well as the loss of information between people.
- Moving test to the front of the development cycle improves the conversation between developers and customers and testers and, thus, improves the quality of the code.
- While management can set expectations for what work is done, management must not demand how that work is done.
- Working on one project at a time improves the productivity of a team.

Figure I.4 Core beliefs of Agile

We Do Not Know It All

Although software development is not exactly like other types of product development, we can still learn a lot from how other industries approach product development. In particular, Lean gives us a lot of information, based on decades of experience, that can be particularly useful to Agile teams. In fact, Scrum, one of the more popular Agile methods, is based on Lean principles. Unfortunately, an understanding of Lean is not widespread in the software community. It is unfortunate because teams lose out on the potential guidance that Lean offers. Moreover, without a grounding in Lean, software developers often lack the basis for explaining to management why certain practices would be useful. Lean provides a new set of beliefs, shown in Figure I.5. The question still remains: Even if these beliefs are true, how do we manifest good practices that are consistent with them?

Of course, merely believing something doesn't make it so. It is worth looking at the beliefs presented here and then deciding which ones represent actual principles. We suggest using backward-looking experiments for this.

The Core Beliefs of Lean

- Most errors are due to the system within which people work rather than to the individuals themselves.
- People doing the work are the best ones to understand how to improve the system.
- Ad hoc is not an acceptable process.
- Looking at when things are done in a process is a more useful guide than trying to make sure every step of the way is as efficient as possible.
- Our measure for success must be related to the amount of time between when ideas come in and when they are manifested as value to our customers.
- Management must work with the team to improve the way it works to improve its own efficiency.
- Teams are most efficient when the amount of work is limited to their capacity.
- Team eficiency improves by minimizing the amount of work in progress at any one time.
- When evaluating actions, we must optimize the whole, not merely improve individual steps in the process.
- There are principles in software development that must be followed in order to reduce waste.

Figure I.5 Core beliefs of Lean

Lean Provides More than Beliefs

Fortunately, Lean provides more than a paradigm and a belief system. It provides a set of principles in its own right as well as many practices based on them. While these practices cannot usually be taken straight from Lean (since practices must change depending upon the context in which they are used), Lean principles and practices can be readily adapted to software development. By learning these principles and practices, one can manifest the intention of the Agile Manifesto—developing software effectively. And we can do it at both levels—enterprise and team.

We will see that Lean provides a paradigm of management in which managers are not encouraged to command and control teams and developers are not required to insist they are craftsmen who cannot and should not be managed. Rather, Lean provides a paradigm under which managers and developers can work together toward a common goal—providing the best return on software development efforts. Lean provides such a paradigm through its focus on the process by which the team works—but a process that must be the best one for the team to get its job done.

Process is no longer something imposed on the team, but rather something owned by the team to make its work more productive as well as more enjoyable.

Lean combines this management paradigm with concepts, tools, and practices that give both sides a way to work together and improve visibility to management, direction from management, and team productivity.

Going beyond Lean

Of course, Lean is not all there is from which to pull. But our experience is that it is consistent with other useful paradigms, beliefs, and principles that come from other disciplines. For example, we learn from the building-architecture discipline and the software-design-patterns community that we should develop products by starting with the big picture. That is, don't try to create a product by building it from small pieces. Keep the big picture in mind. This, unfortunately, is a lesson many Agile practitioners and consultants have long ignored (probably due to Agile's heritage, which sprang up on smaller projects).

In this book, we incorporate what may be non-Lean practices but they are otherwise consistent with a central principle of Lean: "Optimize the whole." In particular, we'll see this in these areas:

- An enterprise focus instead of a team focus both for product-portfolio management and for team coordination, thereby providing a working alternative to Scrum-of-Scrums.
- A product focus instead of a project focus (where projects are enhancements to products).
- Managing requirement elicitation from the big picture instead of starting with stories and combining them into epics and themes.
- Driving release planning from business value instead of trying to manage the effective release of a collection of stories.

Summary

This introduction explored the roots of Agility, starting with the Agile Manifesto, its principles, and its historical context in the swing between management command-and-control and development teams wanting to apply their local knowledge to get work done. What is needed is a proper

understanding of process as both/and: both as a tool for management and a responsibility of the team to steward what it knows.

Getting to this better understanding involves examining core paradigms, principles, and practices that everyone in software development holds. Lean-Agile offers a thinking practice to help form a better way of understanding. It is based on the solid foundation of Lean thinking and is entirely consistent with Agile practices.

Try This

These exercises are best done as a conversation with someone in your organization. After each exercise, ask each other if there are any actions either of you can take to improve your situation.

- Look at the beliefs of Waterfall listed in Figure I.3. Which of these are true?
- Look at the beliefs of Agile listed in Figure I.4. Which of these are true?
- Look at the beliefs of Lean thinking listed in Figure I.5. Which of these are true?

CHAPTER 7

Lean-Agile Release Planning

"If anything is certain, it is that change is certain. The world we are planning for today will not exist in this form tomorrow." —Philip Crosby

"In preparing for battle I have always found that plans are useless, but planning is indispensable." —Dwight D. Eisenhower

IN THIS CHAPTER

A major reason enterprises transition to Lean-Agile software development is the need to plan releases predictably and accurately. Release planning is the process of transforming a product vision into a product backlog. The release plan is the visible and estimated product backlog itself, overlaid with the measured velocity of the delivery organization. It provides visual controls and a road map with predictable release points.

Lean-Agile says the release plan must be driven by the needs of the business. We prioritize to maximize value to the business. We sometimes call this approach "business-driven software development."

To understand how to do this, we must understand some fundamental concepts about process. Therefore, the chapter begins with a conversation about the issues that underlie process—predictability, level of definition, and requirements for feedback.

Takeaways

Key insights to take away from this chapter include

- Release planning is a continual activity of the Lean-Agile enterprise. It is the transformation of the product vision or business case into a prioritized and estimated list of features.

- Once a feature list is created, its completion plan is determined by the velocity of the teams involved. Effective release planning requires a delivery organization that is skilled in predictable estimation, a skill readily gained by leveraging short-cycle iterations in order to get rapid feedback.
- Effective release planning emphasizes rapid return by focusing on discovering and manifesting minimum marketable features.

Issues that Affect Planning

One of the most frequent questions we get is, "How can you predict what is going to happen if you are working with an Agile process?" We believe that this question comes from a misunderstanding of some key issues that underlie process.[1]

Evaluating Processes

We think of processes as having the following:

- A degree of process definition; that is, to what extent the process has been defined
- A degree of predictability, or the randomness of its output
- A degree of feedback, or the amount of feedback that the process uses

Degree of Process Definition

Let's first clean up the terminology: We can view the output of a process as deterministic or nondeterministic (stochastic). In a deterministic process, the outputs are 100 percent determined by the inputs; in a stochastic one, the output is a random variable—it has different values that occur with different probabilities.

1. Special thanks to Don Reinertsen for an e-mail laying out many of these ideas. Used with permission; any inaccuracies should be considered ours.

Fully determined systems do not exist, except in academia and thought experiments. Virtually all real-world manufacturing and development systems have stochastic outputs. That is, they are partially determined.

It is useful to distinguish between a process that is fully determined versus one in which its *output* is fully determined. Although many people tend to assume that a defined process produces a deterministic output, this is not always true—a precisely defined process can still produce a random output. For example, the process for obtaining and summing the results of fair coin flips may be precisely defined; its output is a random variable.

Well-defined systems can produce outputs that range on a continuum from deterministic to purely stochastic. Just as we can structure a financial portfolio to change the variance in its future value—by ranging from all cash to all equity—we can make design choices that affect the amount of variance in a system's output.

Degree of Predictability

Thinking of system output as a random variable may be more useful than labeling it as either unpredictable or predictable. We could think of it as completely unpredictable, macroscopically predictable, or microscopically predictable. It is unclear if anything falls into the first category—even a random number generator will produce uniformly distributed random numbers. It is the zones of what we would call "macroscopic" and "microscopic" predictability that is most interesting.

We can make this distinction using the coin-tossing analogy. When we toss a fair coin 1,000 times, we cannot predict whether the outcome of the next coin toss will be a head or tail—we would call these individual outcomes "microscopically unpredictable." There may be other microscopic outcomes that are fully determined since we have a fully defined process. For example, we could define this process such that there is a zero percent chance that the coin will land on its edge and remain upright. (If the coin lands on its edge, then re-toss the coin.)

Even when the outcome of an individual trial is "microscopically unpredictable," it is still a random variable. As such, it may have "macroscopic" or bulk properties that are highly predictable. For example, we can forecast the mean number of heads and its variance with great precision. Thus, just because the output of a process is stochastic, and described by a random variable, does not mean that it is "unpredictable." This is important because the derived random variables describing the "bulk properties" of a system are typically the most practical way to control a

stochastic process. That is, even though a process may be unpredictable on its own, it can still be controlled with feedback.

Degree of Feedback

The degree of feedback needed is another variable we should add to our duo of degree of predictability and degree of process-definition. In the software-development world, feedback is probably essential; in other areas it may not be. But for us, feedback is likely the most cost-effective way to achieve our goal—but deciding how and when to use it is really an economic issue.

It is important not to confuse process definition with the level of determinism or the amount of feedback required to keep things on track. The key to this section is to understand that although we may not be able to predict microscopically the result of each story, we should be able to predict macroscopically the timing and the cost of the business capabilities encompassed in our features.

Transparent and Continuous Planning

Lean-Agile release planning is a continuous activity that the entire organization can observe. This makes it possible for anyone to contribute to discussions about the value of items in the plan and the effort required to produce them. Release plans enable delivery in small, end-to-end slices. This enables validation in a regular, predictable rhythm that is defined by the iteration length. As we described in chapter 4, Lean Portfolio Management, we want the product portfolio to serve as the transparent focal point for the business to sequence releases of minimal marketable features.

In all but the simplest cases, a feature requires several iterations before it is ready to be released to the customer. Reasons for this include

- The feature is too big to finish in one iteration.
- Multiple features may need to be released together in one package.
- The customer can only "consume," or put to use, features at a certain pace or at a certain time of year.
- Marketing, training, support, and packaging for an otherwise completed feature will not be ready after a single iteration.

Release planning must account for all of these when developing the release schedule.

We think of release planning as continuously decomposing a product vision while focusing on those features of greater priority (value) to the business. This decomposition uses just-in-time methods to prevent wasted effort on lower-priority or unneeded features. That is, we expand on features just as much as we need to according to our expectations of when we will build them (this order is determined by the value they provide to the customer). This plan enables the team to look ahead responsibly so that large-effort activities can be broken down in small enough segments (right-sized work) and balanced against higher priority items that come up. A good release plan provides a clear visual control and obviates the need to look too far ahead and work too far in advance on future, larger features. The continuous activity model is shown in Figure 7.1.

Release planning starts with a vision provided by the product champion, who can make decisions regarding value priority for both the customer and the business. We typically look to the organization that creates project charters to find ideal candidates for this role. The vision should be reviewed and understood by the delivery team and should be revisited as market conditions change priorities. The vision should be visible (for example, with posters on walls) and re-reviewed as part of every iteration's planning session.

Target dates are determined by looking at the estimates in relation to the team's velocity. For example, if a team can deliver 40 story points in

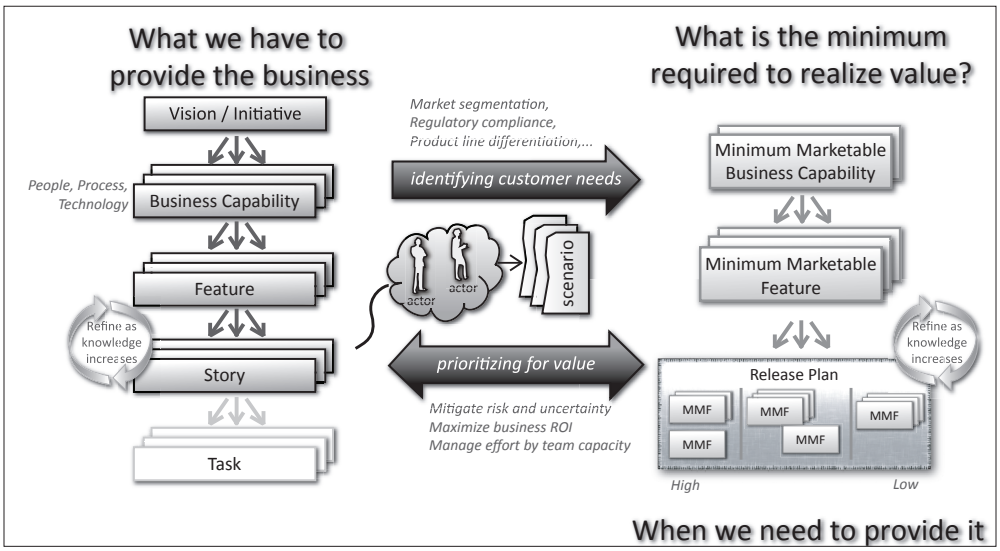

Figure 7.1 The continuous activities involved in release planning

a two-week iteration and we have 200 story points to achieve, we can fairly estimate that it will take five two-week iterations to complete the work at hand. Short cycle times (one to four weeks) enable quick feedback on both the rate of completion and how well we are meeting our customers' needs. During each iteration, teams must focus on coding only the most important feature at any one time. This provides a clear picture of business value (features) juxtaposed against system constraints (technical stories) and enables high-value decisions regarding minimum releasable features.

A project charter should make a business case for new capabilities or capability enhancements. We look to these capabilities to find business features, or "features." It is important to realize that features derive from the vision and capabilities; they do not appear by aggregating lower-level requirements into larger chunks, which is sometimes suggested in the literature as the creation of "epics." Trading business value against effort in search of minimum marketable features leads to decomposing capabilities to form features and stories.

To perform Lean-Agile release planning effectively, the development organization must visually establish (and continuously improve) its ability to determine velocity (story points per iteration), as described in chapter 4, Lean Portfolio Management. The visible velocity is a powerful measure of enterprise capacity (see Figure 4.13 on page 69). This approach requires that the delivery organization be skilled in the art of three-level story point estimation (feature, story, task). Here is another opportunity to emphasize the importance of short cycle time (two-week iterations): The organization is able to recalibrate the quantity associated with story points, as well as get feedback and institutional learning regarding how complex the capabilities, stories, and tasks are.

These multiple levels of continuous decomposition enable an organization to provide estimates required for creating a visible release plan predictably and fearlessly. This is especially worth noting when estimates are required at the feature level, when the least amount of information is known. Experienced Agile teams are confident in providing estimates because the precision required for large features is low, and they know that they are required to commit only when features have been broken down at least two more levels (stories and tasks), and then only commit to two-week iterations with known tasks (which should be about four hours in size). In a transition to Lean-Agile, allow three to four iterations for this skill to mature well enough to produce reliable release plans. Table 7.1 shows the various levels of requirements, their sources, and estimation units.

Table 7.1 Various Levels of Top-Down Requirements Utilized in the Lean-Agile Approach

Requirement Level	Description	Source	Units
Feature	Business solution, capability or enhancement that ultimately provides value to the business and/or its customers	Business/customer value, charter document, business case	Story Points
User Story	Describes interaction of users with the system	Feature	Story Points
Story	Any requirement that is *not* a user story (e.g., technical enabling, analysis, reminder to have conversation)	Development team, analysis work, large story decomposition	Story Points
Task	Fundamental unit of work that must be completed to make progress on a story	Development team (during iteration planning)	Hours

The rate at which teams complete features can be measured in average story points completed per iteration. This provides a velocity of production. After a few iterations this should converge to a somewhat steady rate. If it doesn't, the teams need to investigate why it hasn't yet happened. Once a reasonable velocity is established, it can be used to estimate delivery dates of the releases. Prior to this, release planning will need to rely on comparing current work to the amount of time it took to perform similar work in the past.

In practice, it is never possible to focus on only one feature at a time. Some features may require longer lead times due to dependencies and waiting to complete system-enabling work. WIP should be constrained by an overall focus on the delivery of features (as opposed to the completion of tasks). The constraint is naturally held to because the visual control would quickly expose a feature that is too large. The mature organization continuously challenges the need for large features to find the minimum scope required to deliver maximum return. Metaphorically, this means that sometimes the business value requires only a "bicycle," while the development organization is creating a "motorcycle." In organizations that

exhibit enterprise Agility, visible release plans serve as catalysts for communication, where business value and technical constraints are continuously decomposed and visible along with multiple options based on effort and value. The end result is an organization that incrementally demonstrates and evaluates the value of the release, one feature at a time. A business develops true Agility when it can make real-time verification that what it has built meets the minimum scope required for the feature to deliver its intended value. This is achieved by continuously fighting the waste that comes from building too much. The resulting increase in speed of delivery now enables the organization to meet the demands of rapidly changing markets, customer needs, and business opportunities.

Depending on the release structure of the organization, dedicated release iterations may be required to actually deploy the product to the enterprise production environment. It is an acceptable practice to have a so-called "release iteration" for this activity. It is important that this iteration is constrained to the minimum amount of time required by the release organization, and it should be used only to perform activities required for sign-off and compliance of the release acceptance organization (no new scope).

Releases and Elevations

In an ideal world we could release straight to the customers after every iteration. Unfortunately, for many reasons this is often impractical. For example, if you are on a team that builds embedded software, you may need to create an internal release for the integration team (a team that tests your software, and possibly others' as well) on a hardware platform. Or you may build code that another team will use, so you'll need to release it internally to the other team. There are also times you'll need to release code to selected customers to get feedback—possibly as an alpha test, but maybe just to play with.

We have coined the term "elevation" for all of these "releases" that are not quite real. We don't use "internal release," as elevations sometimes go to customers, but they are not the real releases.

Example: Release Planning Session

This section describes a typical release planning session. Such sessions often follow a sequence like this:

1. Identify features.
2. Prioritize features.

3. Split features using the minimum-marketable-feature perspective.
4. Estimate the value of the features.
5. Estimate the cost of the features.
6. Elaborate further by writing stories for features, repeating until you have reasonable clarity on what the features are and their high-level values.
7. Create a specific release plan by date or by scope.
8. Plan elevations.

How long does a release-planning session take? Small projects (three months or less) can often be done in a day. Larger projects will take a few days.

During the session, the team has to remember constantly that it is being driven by two forces:

- **Add value for the customer.** The focus is not on building software; it is to increase the value of the software product we create to those who will use it. The software is a means to an end, but it is not the value itself.
- **Get to market quickly**. Develop plans around minimum marketable features (MMF). View features from the MMF perspective: What is required to develop and release them?

Using Tools in Release Planning

We want tools to support the Lean-Agile process. The early stages of release planning, though, are best supported with lower-tech, higher-touch tools: everyone present in the room, using stickies or index cards on the wall.

This creates the best environment for the nonlinear, multi-dimensional thought processes release planning requires.

Once the release plan has been created, it is good to move the data into an Agile planning tool.

In the following sections, we examine each of the steps in a bit more detail.

1. Identify Features

Begin by writing features on stickies or index cards. Briefly describe each feature (usually just a few words), as shown in Figure 7.2. At this point, the team is just trying to establish a high-level scope of the system.

2. Prioritize Features, Left to Right

Once the features are identified, the team does an initial prioritization: Place the most important features on the left and the least important on the right, as shown in Figure 7.3. This only represents a first cut; the team is not committed to this order. It will certainly change as they work through the steps.

Even this initial prioritization should prompt some interesting conversations. The conversations should focus on sharing knowledge and helping everyone learn more about the features. Don't get hung up on whether the prioritizations are absolutely correct. Focus on learning as much as possible and consider all decisions tentative.

3. Split Features Using the MMF Perspective

Once the initial set of features is described, it is often easy enough to split up some into what could be called minimum marketable features and then further split into one or more enhancements to those MMFs.

For example, suppose Feature F in Figure 7.3 must be supported on five different platforms: Linux, Windows, Solaris, HP, and AIX. Talking with the customer, the team discovers that only Linux and Windows need to be supported at first. Feature F can be broken into two parts: the core MMF for Linux and Windows and an extension MMF for the others. Call these F1 and F2, respectively. Other features can likewise be decomposed, as shown in Figure 7.4.

4. Estimate the Value of Features

Since the product champion is driving from business value, the first thing to do is estimate the relative value of each feature. We can do this using the Team Estimation Game.[2] The value of each story is assigned business-value "points" (shown as "BV" in Figure 7.5). However, do not reorder

2. Appendix A, Team Estimation Game, contains a description of the Team Estimation game, which we prefer over "Planning Poker."

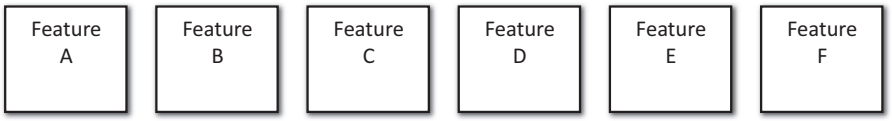

Figure 7.2 Initial features

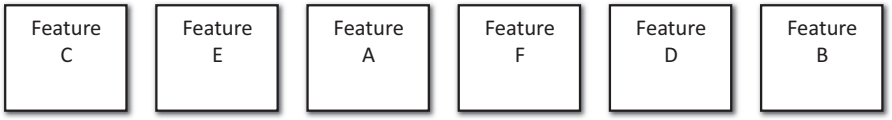

Figure 7.3 Initial features, prioritized, left to right

Figure 7.4 Splitting features up into an MMF and its extension

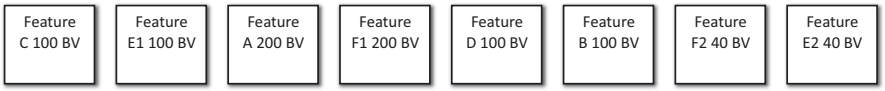

Figure 7.5 Assigning business value to the features

the features based just on these points. Features may have to be developed in a particular order or you may need to get a sense of the cost required for each business value.

You may find that you have difficulties quantifying features by points this way. In this case, just identify the general sequence in which the features need to be built. We have found that many organizations cannot initially set values to the core, required features. In some sense, this doesn't matter: They will all need to be built before release anyway. If that is the case, don't worry about it. You should find that, after the release of the core MMFs, you can set relative values for the remaining features.

Remember: Business or customer value is independent of cost. First, determine business or customer value and only then ask the team to estimate the cost. Then, you can calculate ROI.

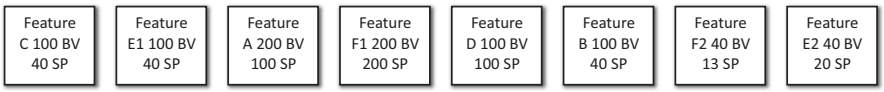

Figure 7.6 Assigning cost in story points to features

5. Estimate the Cost of Features

You can use the Team Estimation Game to estimate the cost of the features that are represented in "story points" (shown as "SP" in Figure 7.6).

Once you have the cost for each feature, the product team may decide to reprioritize them. In effect, you now have the capability to evaluate Return (business value) on Investment (cost), which enables new insight into selecting what brings the highest return to the business for the effort spent by the delivery team. A significant value of this technique is that it clearly de-couples business value prioritization from technical effort, which is an opportunity to drive from business value first. We find that most business organizations have lost the ability to prioritize based on business value alone because they are so used to batching up large requirement sets with faraway dates that they see no need to sequence features since "they are all important."

6. Elaborate Features

You might be surprised at how well this approach works at a high level. It works by comparing one item against another—something teams are reasonably good at. Going further requires more accuracy. This requires a more detailed understanding of the features.

Start by writing stories for each of the features, beginning with the higher-priority features, the ones you will be working on sooner. This is called "elaboration."

After elaborating a few features and increasing your understanding of what is required to build them, you may need to re-estimate both business value and cost. (Notice that this technique has a built-in feedback loop that continuously calibrates the accuracy of the feature estimates. The elaborated stories for each feature are individually estimated and then summed to compare with the feature.) Continue this process until you have a set of features comprised of the core and extension MMFs, along with a number of elaborated stories, and you are confident in the relative estimates of the features. This is shown in Figure 7.7.

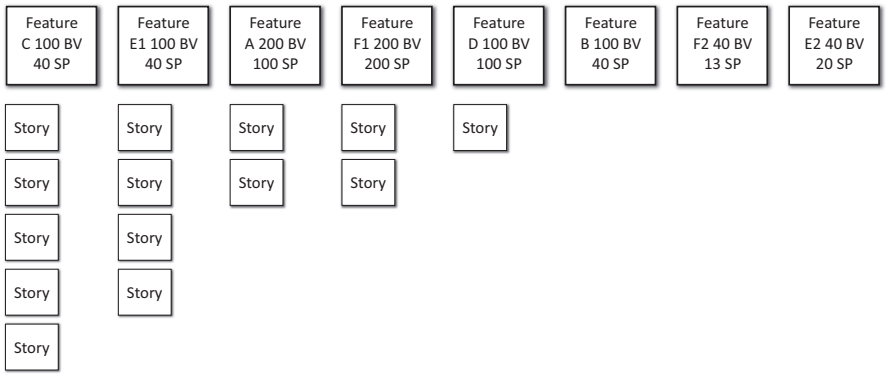

Figure 7.7 Result of feature and story elaboration

7. Create the Release Plan

Now the team is ready to plan releases. There are two approaches to this: planning by date and planning by scope. Which to use depends on your needs, which are often mandated by governmental regulations or market conditions.

Planning by Date

There are times when a project must come in by a certain date: Government regulations require certain features by a certain time, software is required for a conference, or our industry requires major releases at a certain time of year. If this is the case, then the release plan entails setting the date and ensuring the right amount of functionality can be achieved within the allotted time.

For example, suppose you have four months to finish product development and every feature except B, F2, and E2 is required by that date. The release plan is shown in Figure 7.8.

Add up the estimated number of story points for these features. That determines how many points must be completed in each iteration. In this example, there are 480 story points. There are 17 weeks available. Suppose Iteration 0 requires a week at the beginning and there are two weeks at the end for alpha testing. That means 480 points over 14 weeks for development, or 34 story points per two-week iteration.

Total Points/Number of Weeks Available for Development = Required Team Velocity

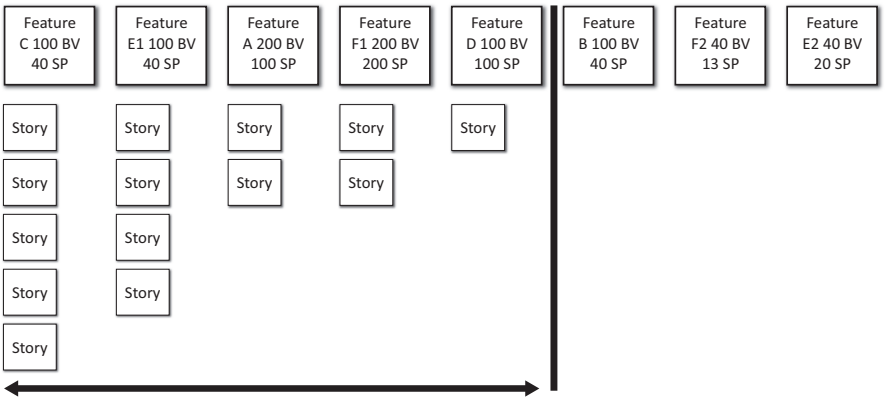

Figure 7.8 Planning by date

If the team can handle that level (velocity), that is great. If not, you have to focus on what is truly minimal for each of the identified features. What can be cut out? What must be left in? At the beginning, you cannot know for sure, which is why the focus must be on starting work on only the features, or aspects of features, that are truly essential. Iterative development will enable you to discover the core functionality need.

Agile Estimation Isn't Exact, but It Is Better

In our classes, we are often asked how we can get precise estimates with Agile methods. This question seems to imply that the asker is somehow getting these desired accurate estimates with his or her non-Agile method. We don't claim that using Agile methods will improve accuracy over non-Agile estimating at the very beginning. It will, however, create clarity at a faster pace. But when it comes to the claim that we must be accurate, we are reminded of the following joke: Two campers are awakened in the middle of the night by the sounds of a bear snuffling at their tent entrance. One calmly starts putting on his shoes. The other exclaims, "Are you crazy? You can't outrun a bear!" The other responds, "I don't have to outrun the bear, I only have to outrun you!"

This type of estimation does not necessarily give you better accuracy than traditional methods. But it does show you where you need to look to make your decisions. Very often it becomes clear that the true MMFs

can be built in time, whereas you are uncertain about features you would just *like* to have. Sometimes, it becomes clear you are in trouble. If you are somewhere in the middle, then at least you have an idea about which features you need to investigate further.

Planning by Scope

Another approach is to plan by scope. This works much like planning by date; however, this time you begin with those MMFs that are required. Calculate the number of story points in the MMFs, divide by the team's velocity (the ability to complete stories in an iteration) and the result is the time required to do the work.

Total Points/Team Velocity = Number of Weeks Required for Development

If the result is too long, reassess to see what features or elements can be dropped to make it shorter.

Proper Planning Avoids Risk

Both of these approaches help teams focus and avoid risk. They help teams:

- Work on the most important features
- Avoid starting less-important features until the more important ones are finished
- Minimize WIP

These are crucial. Consider a time when you were working on a project only to discover you were going to have to cut scope. The predicament is that at this point, you have:

- Already completed some less-important features—which you started because at the beginning of the project you were confident it was all going to happen; and
- Started some features you would like to cut but doing so now would cause you to lose work you've already done—you'd have wasted time and added complexity for no value (almost certainly the code that's in there for these features will stay in there).

Planning-by-date and planning-by-scope methods help ensure that the team works on the most important features known at the time and that other features are not started until the important ones are finished.

A Case Study

COMPANY PROFILE: Large software product company

CHALLENGES: Tightly coupled, complex product enhancements being built at the same time. Not clear of the exact scope of features.

INSIGHT: During a planning session where all related features were put on a wall and all interested parties were present, one of our consultants asked the question—"how many people here are 100% certain that all of these features will be built in the time frame we have?" To no one's surprise, no one raised their hand. Then the consultant asked—"which of these features must be done by the deadline or you don't have a product?" There was actually fairly consistent agreement on this question. These were the features selected for the first release.

Lean suggests doing the essential things first in the fastest time possible by building quality in. By de-scoping early, we focus on the Pareto Rule of 20% of the work providing 80% of the value. By time-boxing our development, we minimize the affect of Parkinson's Law that "work expands so as to fill the time allotted for its completion."

8. Plan the Elevations

There may be another degree of complexity to consider when there is more than one team involved in the software development or there is a subset of the software that can be tested but cannot yet be released.

The first case can be made more difficult if there is hardware on which to test as well. In these cases, an internal release is necessary to test the system—either its technical integrity through integration testing or its quality to customers through external system testing using alpha or beta testers. We call these pseudo/external releases "elevations." We are moving the software farther down the value stream, but not all the way to the customer. We will consider two different types of elevations.

Elevations for Integration Testing

Very often a team will build software must interact with software that other teams are building. You cannot be sure exactly how it will function until the other teams use it. Or teams are creating multiple products that

must be used on a hardware platform. Until the software is actually on the hardware, you cannot know for sure how it will function.

One type of elevation planning is to look at the milestones the software must reach prior to true release. In a situation like this it could be

- Software passes a team's functional test.
- Software passes several teams' functional test.
- Software works on a specified hardware platform.
- Software has been alpha-tested by a set of customers.

This example would require three elevations prior to the final release:

1. Move the software to the other teams that will use it.
2. Load and test the software on the hardware platform using internal testers.
3. Enable external users to try out the software.

These elevations are shown graphically in Figure 7.9.

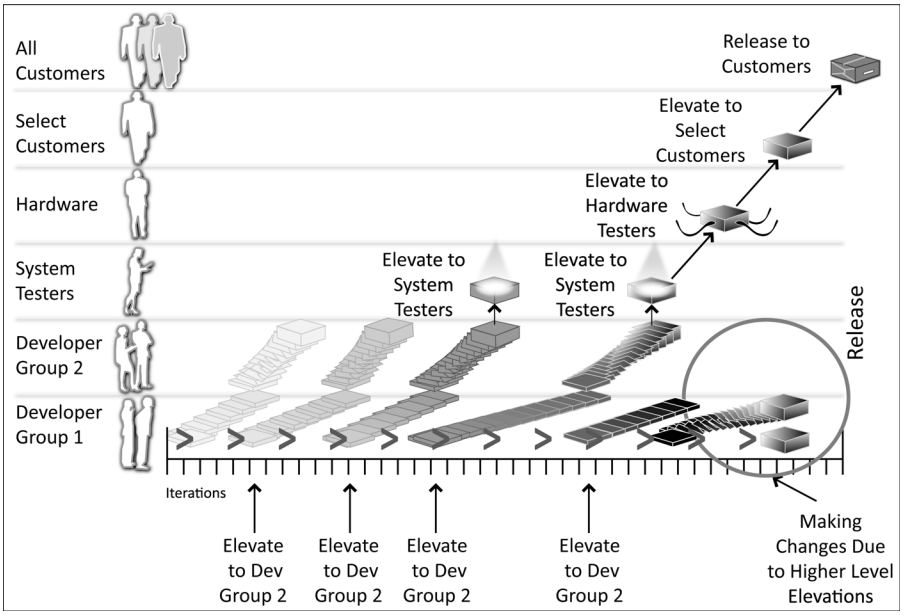

Figure 7.9 Elevations across teams and testing platforms

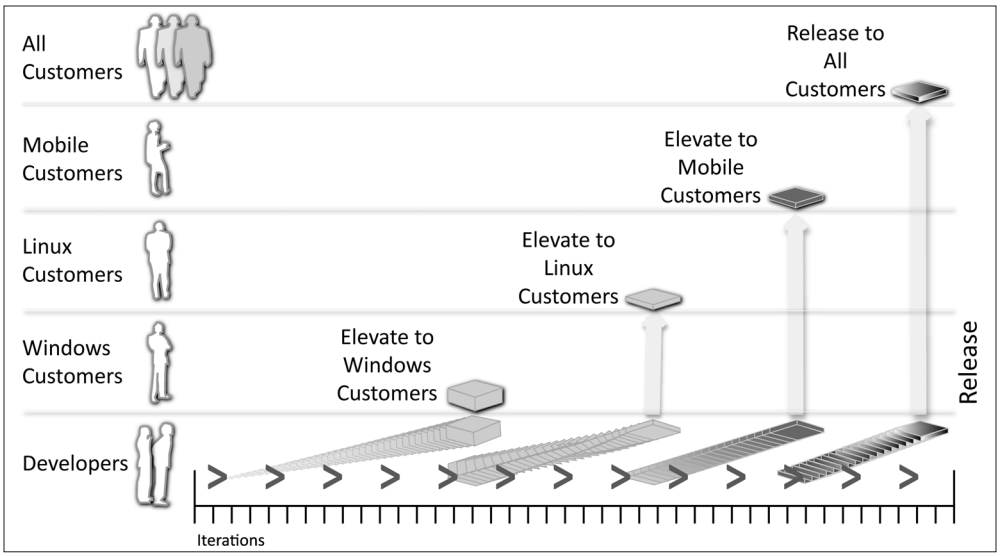

Figure 7.10 Elevations to different operating systems

Elevations to Different Platforms

A different elevation pattern exists when the software you are writing must work on different operating systems. For example, suppose you are writing software for Windows, Linux, and mobile platforms. Figure 7.10 illustrates that elevation plan.

Elevation Summary

There are no set rules for elevations. The ideal case is continuous integration across all development. But when different platforms, operating systems, hardware, customer bases, and so on are present, that is not always possible. Elevation planning, however, enables you to investigate the best way to get feedback about a larger, working part of the system. Acceptance Test-Driven Development with an emphasis on design patterns and refactoring enables the organization to benefit holistically from emergent design techniques. For example, skilled organizations that mock to test and refactor to design patterns can do more in-place and continuous integration than would be required to incorporate Lean-Agile in complex-release organizations that deliver across different platforms. Chapter 9, The Role of Quality Assurance in Lean-Agile Software Development, covers this in more detail.

A Few Notes

We end this chapter with a few more release-planning thoughts on estimation and risk—and Pareto versus Parkinson.

On Estimation and Risk

Many people think that there is risk attached to missing your estimate. At worst it might be embarrassing; however, the real risk is in missing your *delivery dates*. It is not important to be able to predict at the level of the story; what is important is predicting at the release level.

Risk also plays a role in prioritizing features. Usually, we prioritize by the business value each feature represents—possibly offset by the cost of creating it. However, sometimes prioritization is affected by the potential cost of delay. For example, let's say we have Feature A and Feature B. Feature A may be twice as important as Feature B, but we need Feature B for a conference coming up in three months. We may actually do Feature B first to ensure its completion before the conference if delaying Feature A is not too costly.

Pareto versus Parkinson

We have heard Lean software development likened to following Pareto's Law: 80 percent of the value comes from 20 percent of the work. In other words, find that 20 percent of features that will provide your customers with 80 percent of their value; then, find the next features that will provide the greatest value to your customers.

The problem with this is that if there is no time-boxing—no end-date—Parkinson's Law may apply: "Work expands so as to fill its time for completion." Parkinson's Law is particularly dangerous when multiple product managers are competing for a team's resources. Manager A is focusing the team on one thing and Manager B is concerned about when she will have the team's availability. You can counteract the effect of Parkinson's Law, by having the team follow Pareto's Law in the shortest amount of time they can. In other words, have the team always focus on building the smallest things as quickly as they can, end to end, while ensuring quality.

Add the most value possible in the least amount of time possible with the right level of quality.

Summary

An organization that maintains visible release plans that are driven by continuous validation of velocity have a powerfully competitive weapon—key tactical and strategic moves can be analyzed continuously for maximum value. Enterprise Agility is achieved when the delivery organization is actively engaged in the release planning activity, through estimation and the discovery of options based on effort.

Try This

These exercises are best done as a conversation with someone in your organization. After each exercise, ask each other if there are any actions either of you can take to improve your situation.

Consider a few typical past projects.

- Most successful Waterfall projects require de-scoping in order to reach target dates. If this was the case for any of your past projects, when did de-scoping occur?
- What would have happened if de-scoping would have occurred *before* the development team started implementation?
- How does release planning (with visible velocity) aid in the discovery of right-sized, high-value work?

Recommended Reading

The following works offer helpful insights into the topics of this chapter.

Denne and Cleland-Huang. 2003. *Software by Numbers: Low-Risk, High-Return Development*. Upper Saddle River, NJ: Prentice Hall.

Reinertsen. 1997. *Managing the Design Factory*. New York: Free Press.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V