The Addison-Wesley Signature Series

A Mike Cohn Signature Book

The Scrum Field Guide

Agile Advice for Your First Year and Beyond

Second Edition

Mitch Lacey

Forewords by Jeff Sutherland and Kenneth S. Rubin

THE SCRUM FIELD GUIDE

SECOND EDITION

THE SCRUM FIELD GUIDE

SECOND EDITION

Agile Advice for Your First Year and Beyond

Mitch Lacey

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2015955194

ISBN-13: 978-0-13-385362-9
ISBN-10: 0-13-385362-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

This book is dedicated to two teams; The first team is my family. My wife, Bernice, and my kids, Ashley, Carter, and Emma—without their support and constantly asking "are you done yet?" this book would not be here. They kept me focused and supported me throughout.

The second team is the group of guys from the Falcon project while at Microsoft. John Boal, Donavan Hoepcke, Bart Hsu, Mike Puleio, Mon Leelaphisut, and Michael Corrigan (our boss), thank you for having the courage to leap with me. You guys made this book a reality.

This page intentionally left blank

Contents

Foreword
by Jeff Sutherland

Mitch and I have worked together for many years training developers in Scrum. Studying this book can help users overcome the biggest challenges that have occurred in recent years as agile practices (75 percent of which are Scrum) have become the primary mode of software development worldwide.

Ten years after the Agile Manifesto was published, some of the original signatories and a larger group of agile thought leaders met at Snowbird, Utah, this time to do a retrospective on ten years of agile software development. They celebrated the success of the agile approach to product development and reviewed the key impediments to building on that success. And they came to unanimous agreement on four key success factors for the next ten years.

1. Demand technical excellence.
2. Promote individual change and lead organizational change.
3. Organize knowledge and improve education.
4. Maximize value creation across the entire process.

Let's see how Mitch's book can help you become an agile leader.

Demand Technical Excellence

The key factor driving the explosion of the Internet and the applications on smartphones has been deploying applications in short increments and getting rapid feedback from end users. This process is formalized in agility by developing products in short sprints, always a month or less and most often two weeks in length. We framed this issue in the Agile Manifesto by saying that "we value working software over comprehensive documentation."

The Ten Year Agile Retrospective of the Manifesto concluded that the majority of agile teams are still having difficulty developing products in short sprints (usually because the management, the business, the customers, and the development teams do not demand technical excellence).

Engineering practices are fundamental to software development, and 17 percent of Scrum teams implement Scrum with XP engineering practices. The first Scrum team did so in 1993 before XP was even born. It is only common sense to professional engineers.

Mitch says in the first chapter that he considers certain XP practices to be mandatory—sustainable pace, collective code ownership, pair programming, test-driven development, continuous integration, coding standards, and refactoring. These are fundamental to technical excellence, and the 61 percent of agile teams using Scrum without implementing these practices should study Mitch's book carefully and follow his guidance. Neglecting to use these mandatory XP practices is the reason they do not have shippable code at the end of their sprints!

Mitch's book contains much more guidance on technical excellence, and agile leaders, whether they are in management or engineering, need to demand the technical excellence that Mitch articulates so well.

Promote Individual Change and Lead Organizational Change

Agile adoption requires rapid response to changing requirements along with technical excellence. This was the fourth principle of the Agile Manifesto—"respond to change over following a plan." However, individuals adapting to change is not enough. Organizations must be structured for agile response to change. If not, they prevent the formation of, or destroy, high-performing teams because of failure to remove impediments that block progress.

Mitch steps through the Harvard Business School key success factors for change. A sense of urgency is needed. Change is impossible without it. Agile leaders need to live it. A guiding coalition for institutional transformation is essential. Agile leaders need to make sure management is educated, trained, on board, and participating in the Scrum implementation.

Creating a vision and empowering others is fundamental. Arbitrary decisions and command and control mandates will kill agile performance. Agile leaders need to avoid these disasters by planning for short-term wins, consolidating improvements, removing impediments, and institutionalizing new approaches. Agile leaders need to be part of management or must train management as well as engineering, and Mitch's book can help you see what you need to do and how to do it.

Organize Knowledge and Improve Education

A large body of knowledge on teams and productivity is relatively unknown to most managers and many developers. Mitch talks about these issues throughout the book.

Software Development Is Inherently Unpredictable

Few people are aware of Ziv's law: Software development is unpredictable. The large failure rate on projects worldwide is largely due to lack of understanding of this problem and the proper approach to deal with it. Mitch describes the need to expect and adapt to constant change. The strategies in this book help you avoid many pitfalls and remove many blocks to your Scrum implementation.

Users Do Not Know What They Want until They See Working Software

Traditional project management erroneously assumes that users know what they want before software is built. This problem was formalized as Humphrey's law, yet this law is systematically ignored in university and industry training of managers and project leaders. This book can help you work with this issue and avoid being blindsided.

The Structure of the Organization Will Be Embedded in the Code

A third example of a major problem that is not generally understood is Conway's law: The structure of the organization will be reflected in the code. A traditional hierarchical organizational structure negatively impacts object-oriented design, resulting in brittle code, bad architecture, poor maintainability and adaptability, along with excessive costs and high failure rates. Mitch spends a lot of time explaining how to get the Scrum organization right. Listen carefully.

Maximize Value Creation Across the Entire Process

Agile practices can easily double or triple the productivity of a software development team if the product backlog is ready and software is done at the end of a sprint. This heightened productivity creates problems in the rest of the organization. Their lack of agility will become obvious and cause pain.

Lack of Agility in Operations and Infrastructure

As soon as talent and resources are applied to improve product backlog, the flow of software to production will at least double—and in some cases be five to ten times higher. This exposes the fact that development operations and infrastructure are crippling production and must be fixed.

Lack of Agility in Management, Sales, Marketing, and Product Management

At the front end of the process, business goals, strategies, and objectives are often not clear. This lack of clarity results in a flat or decaying revenue stream even when production of software doubles.

For this reason, everyone in an organization needs to be educated about and trained on how to optimize performance across the whole value stream. Agile individuals need to lead this educational process by improving their ability to organize knowledge and train the whole organization.

The Bottom Line

Many Scrum implementations make only minor improvements and find it difficult to remove impediments that embroil them in constant struggle. Work can be better than this. All teams can be good, and many can be great! Work can be fun, business can be profitable, and customers can be really happy!

If you are starting out, Mitch's book can help you. If you are struggling along the way, this book can help you even more. And if you are already great, Mitch can help you be greater. Improvement never ends, and Mitch's insight is truly helpful.

—Jeff Sutherland
Scrum Inc.

FOREWORD
by Kenneth S. Rubin

In 1988, I was the first employee hired by ParcPlace Systems after the Smalltalk research team was spun out of Xerox PARC (Palo Alto Research Center). Our mission was to commercialize the use of object-oriented technology. During those early days of the object-technology movement, we often discussed writing a sort of recipe book to help companies get started with object technology. The idea was to collect the most important situations/issues that we saw companies encountering and present them as a set of patterns or recipes in the format, "If you find yourself in this situation, try doing the following. . . ." We never did write that particular book.

Fast-forward over 20 years to the era of agile development, and Mitch Lacey has written his own recipe book: a field guide on the topic of Scrum. In it, he shares his wealth of Scrum experience with companies that are gearing up to use Scrum or those that are still in the nascent stages of applying it.

I first met Mitch in 2007, shortly after I became the very first managing director of the worldwide Scrum Alliance. At that time, Mitch was already a Certified Scrum Trainer (CST) and had been applying Scrum for some number of years both inside Microsoft, where he was first exposed to Scrum, and later with both large and small companies as a Scrum trainer and coach.

I could tell from our first meeting that Mitch was passionate about helping people be successful with Scrum. At that first encounter, he took out his laptop and started walking me through data he had been collecting. His goal was to reinforce anecdotal success stories with real data drawn from his experiences. In hindsight, this exchange was foreshadowing for what was to become *The Scrum Field Guide*.

When you read this book, you will experience what I did during that 2007 conversation and in numerous discussions and debates I have had with Mitch ever since—that he has a keen ability to collect and analyze real-world experiences and synthesize them into actionable advice. You will benefit from this advice in each and every chapter! Each chapter begins with a story that captures the culmination of Mitch's experiences on a specific topic. I find this technique to be very effective, since I like a good story and Mitch is quite an effective storyteller. But don't just stop after reading a chapter's intro story! Following each story is an expansion and interpretation of the concepts presented in the story that amplify the advice being offered.

One thing you will notice while reading this book is that Mitch isn't trying to teach you theory. He's trying to save your bacon. Even the major section names convey this intent: "Getting Prepared," "Field Basics," "First Aid," "Advanced Survival Techniques," and "Wilderness Essentials." A true field guide indeed!

You also have the benefit of reading the second edition of this book. Mitch has not been sitting still since the first edition was published in 2012. This new edition includes updated versions of many of the original chapters as well as five completely new chapters in the new section "Wilderness Essentials." These chapters address topics as diverse as "getting to done," the relationship of story points to hours, techniques for interviewing and hiring, how to align incentives with outcomes, and the all-important topic of managing risk during agile development. If you are already familiar with the first edition, I am sure you will benefit from the changes and additions Mitch has made to this latest edition!

It is an honor to have Mitch and *The Scrum Field Guide, Second Edition,* in the same book series as my *Essential Scrum* book: The Mike Cohn Signature Series. I have high regard for the authors and books in this series. They all pass my litmus test: "Would I recommend a book in the series to my clients without reservations?" I am happy to say that I can definitely recommend Mitch's book!

And, for those readers who are familiar with my book, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, I am confident that you will find my book and this book to be good complements to one another, even (and especially) where the two differ on particular issues or approaches. And, apparently, many others agree! A quick look on Amazon.com at the first edition of *The Scrum Field Guide* shows that the book most frequently purchased along with it is my *Essential Scrum* book. So, like fine wine and cheese, I hope you enjoy the pairing!

—Kenneth S. Rubin
Managing Principal, Innolution, LLC,
and author of *Essential Scrum:*
A Practical Guide to the Most Popular Agile Process

Preface

Welcome to the Second Edition of *The Scrum Field Guide*

When I mentioned that I was thinking of updating the original *Field Guide*, my wife questioned my sanity. After all, she reminded me, the first book had nearly done me in. Yet as I reflected on my first effort, I felt that not only did I have more to say, but I also wanted to adjust some of the content I'd already published. Put simply, I wanted to refactor, add some new features, and release a 2.0 version. So here we are.

You should still read this book in the same way you did the first edition: Pick a chapter that addresses a problem you see in your company—and read it. Then go apply my advice and see what happens.

Agile is a journey. I've learned a great deal since the first edition was published in 2012. If you have read this book before, you will immediately see that I have added new ideas and concepts to the original chapters. Many chapters have been rewritten by more than 80 percent; others by only 10 percent. You will see a new section: Part V, "Wilderness Essentials," containing additional tips for the field, inspired by my firsthand experience working with organizations around the globe. These new chapters include managing risks, interviewing, the fallacy of getting it right the first time, and more.

How This Book Came to Be

When my daughter Emma was born, I felt out of my depth. We seemed to be at the doctor's office much more than we had been with our other children. I kept asking my wife, "Is this normal?" One night, I found my wife's copy of *What to Expect the First Year* on my pillow with a note from her: "Read this. You'll feel better."

And I did. Knowing that everything we were experiencing was normal for my child, even if it wasn't typical for me or observed before, made me feel more confident and secure. This was around the same time I was starting to experiment with Scrum and agile. As I encountered obstacles and ran into unfamiliar situations, I began to realize that what I really needed was a *What to Expect* book for the first year (and even beyond) of Scrum and XP.

The problem is, unlike a *What to Expect* book, I can't tell you exactly what your team should be doing or worrying about during months 1 to 3 or 9 to 12. Teams, unlike most children, don't develop at a predictable rate. Instead, they often tumble, stumble, and bumble their way through their first year, taking two steps forward and one step back as they learn to function as a team, adopt agile engineering practices, build trust with their customers, and work in an incremental and iterative fashion.

With this in mind, I chose to structure this book with more of an "I've got a pain here, what should I do" approach. I've collected stories about teams I've been a part of or witnessed in their first year of agile life. As I continued down my agile path, I noticed that the stories and the patterns in companies were usually similar. I would implement an idea in one company and tweak it for the next. In repeating this process, I ended up with a collection of real-world solutions that I now carry in my virtual tool belt. In this book, I share with you some of the most common pains and solutions. When your team is hurting or in trouble, you can turn to the chapter that most closely matches your symptoms and find, if not a cure, at least a way to relieve the pain.

The Scrum Field Guide, Second Edition, is meant to help you fine-tune your own implementation, navigate some of the unfamiliar terrain, and more easily scale the hurdles we all encounter along the way.

Who Should Read This Book

If you are thinking about getting started with Scrum or agile, are at the beginning of your journey, or have been at it a year or so but feel like you've gotten lost along the way, this book is for you. I'm officially targeting companies that are within six months of starting a project to those that are a year into their implementation—an 18-month window.

This is a book for people who are pragmatic. If you are looking for theory and esoteric discussions, grab another of the many excellent books on Scrum and agile. If, on the other hand, you want practical advice and real data based on my experience running projects both at Microsoft and while coaching teams and consulting at large Fortune 100 companies, this book fits the bill.

How to Read This Book

The book is designed for you to be able to read any chapter, in any order, at any time. Each chapter starts out with a story pulled from a team, company, or project that I worked on or coached. As you might expect, I've changed the names to protect the innocent (and even the guilty). Once you read the story, which will likely sound familiar in some fashion, I walk you through the model. The model is what I use in the field to help address the issues evident in the story. Some of the models might feel uncomfortable, or you might believe they won't work for your company. I urge you to fight the instinct to ignore the advice or to modify the model. Try it at least three times and see what happens. You might be surprised. At the end of each chapter, I summarize the keys to success—those factors that can either make or break your implementation.

This book is organized in five parts.

- Part I, "Getting Prepared," gives you advice on getting started with Scrum, helping you set up for success. If you are just thinking about Scrum or are just beginning to use it, start there.
- Part II, "Field Basics," discusses items that, once you get started down the agile path, help you over some of the initial stumbling blocks that teams and organizations encounter. If you've gotten your feet wet with Scrum but are running into issues, you might want to start here.
- Part III, "First Aid," is where I deal with some of the larger, deeper issues that companies face, such as adding people to projects or fixing dysfunctional daily standup meetings. These are situations you'll likely find yourself in at one point or another during your first year. These chapters help you triage and treat the situation, allowing your team to return to a healthy state.
- Part IV, "Advanced Survival Techniques," contains a series of items that teams seem to struggle with regardless of where they are in their adoption—things such as costing projects, writing contacts, and addressing documentation in agile and Scrum projects.
- Part V, "Wilderness Essentials," contains chapters that focus on overlooked, yet just as costly, problems that most organizations face when they are in the middle of their agile adoption, such as managing risks, interviewing, getting it right the first time, and more.

If you are starting from scratch and have no idea what Scrum is, I've included a short description in the appendix to help familiarize you with the terms. You might also want to do some more reading on Scrum before diving into this book.

Why You Should Read This Book

Regardless of where we are on our agile journey, we all need a friendly reminder that what we are experiencing is normal, some suggestions on how to deal with issues, and a few keys for success. This book gives you all that in a format that allows you to read only the chapter you need, an entire section, or the whole thing. Its real-life situations will resonate with you, and its solutions can be applied by any team. Turn the page and read the stories. This field guide will become a trusted companion as you experience the highs and lows of Scrum and Extreme Programming.

Supplemental Material for This Book

Throughout this book, you may find yourself thinking, "I wish I had a tool or downloadable template to help me implement that concept." In many cases, you do. If you go to http://www.mitchlacey.com/supplements/, you will find a list of various files, images, spreadsheets, and tools that I use in my everyday Scrum projects. While some of the information is refined, most of the stuff is pretty raw. Why? For my projects, I don't need it to be pretty; I need it to be functional and to work. What you get from my website supplements will be raw, true, and from the trenches.

Acknowledgments

When I first had the idea for this book, it was raw. Little did I know that I was attempting to boil the ocean. My wife, Bernice, kept me grounded, as did my kids. Without their strength, this book would not be here today.

David Anderson, Ward Cunningham, and Jim Newkirk were all instrumental in helping me and my first team get off the ground at Microsoft. Each of them worked there at the time and coached us through some rough periods. I still look back at my notes from an early session with Ward, with a question highlighted that read, "Can't we just skip TDD?" Each of these three people helped turn our team of misfits into something that was really special. David, Ward, and Jim—thank you.

I thank Mike Cohn and his authors for accepting this title into the Mike Cohn Signature Series. It is quite an honor to be among some of the best agile authors on the planet.

I could not have done this without the help of my wife, Bernice Lacey, and the best editor on the planet, Rebecca Traeger. Both put in countless hours editing, keeping me on track, keeping me focused, and helping me turn my raw thoughts and words into cohesive chapters.

I would also like to once again thank the following friends, each of whom helped craft this book into what it is today. Everyone listed here has given me invaluable feedback and contributed many hours either listening to me formulate thoughts or reading early drafts. I cannot thank each of you enough, including Tiago Andrade e Silva, Adam Barr, artist Tor Imsland, Brent Barton, Martin Beechen, Arlo Belshee, Jelle Bens, John Boal, Jedidja Bourgeois, Stephen Brudz, Brian Button, Sharon Button, Mike Cohn, Jim Constant, Michael Corrigan, Scott Densmore, Esther Derby, Stein Dolan, Marc Fisher, Paul Hammond, Bill Hanlon, Christina Hartikainen, Christian Hassa, Martina Hiemstra, Jim Highsmith, Liz Hill, Donavan Hoepcke, Bart Hsu, Wilhelm Hummer, Ron Jeffries, Lynn Keele, Clinton Keith, James Kovaks, Ben Linders, Rocky Mazzeo, Steve McConnell, Jeff McKenna, Brian Melton, Ade Miller, Raul Miller, Jim Morris, Jim Newkirk, Jacob Ozolins, Michael Paterson, Bart Pietrzak, Dave Prior, Peter Provost, Michael Puleio, Scott Robinson, René Rosendahl, Ken Schwaber, Tammy Shepherd, Lisa Shoop, Michele Sliger, Ted St. Clair, Jeff Sutherland, Gaylyn Thompson, Isaac Varon, Bas Vodde, and Brad Wilson.

I'd also like to thank the team at Addison-Wesley, including Elizabeth Ryan, Chris Zahn and Chris Guzikowski. I appreciate all that your team did. Also, Carol Lallier and Kim Arney—copyeditor and production project coordinator—thank you. You caught so many things that I overlooked and made this very easy.

Books don't just pop out of your head and onto paper. They, like most projects I've ever encountered, are truly a team effort. The people I have mentioned (and likely a few that I forgot) have listened to me, told me where I was going astray, given me ideas to experiment with on my teams and with clients, and been there for me when I needed reviews. I imagine they are as glad as I am that the second edition of this book is finally in print. I hope that after you read this, you too will join me in thanking them for helping to make this guide a reality.

About the Author

Mitch Lacey, founder of Mitch Lacey & Associates, Inc., helps companies reach full potential by building performing organizations through the adoption of agile practices, including Scrum and XP.

Mitch is a self-described tech nerd who started his technology career in 1991 at Accolade Software, a computer gaming company. After working as a software test engineer, a test manager, a developer, and at a variety of other jobs in between, he settled on his true calling—project and program management.

Mitch was a formally trained program manager before adding agile to his project tool belt in 2005. He first practiced agile at Microsoft Corporation, where he successfully released core enterprise services for MSN and assisted in driving agile adoption across multiple divisions.

Mitch's first agile team was coached by Ward Cunningham, Jim Newkirk, and David Anderson. He has worked extensively in all Scrum roles on a variety of projects. Today, with decades of experience under his belt, Mitch continues to develop his craft by experimenting and practicing with both leadership and project teams at many different organizations.

Mitch's rich, practical experience and his pragmatic approach are trusted by many companies including Adobe Systems, Aera Energy, Bio-Rad, EchoStar, Microsoft, Oracle, Qualcomm, Salem Hospital, SAP, Sony, and more. He is a Certified Scrum Trainer (CST), a PMI Project Management Professional (PMP), and an Agile Certified Practitioner (ACP).

Mitch has presented at a variety of conferences worldwide, chaired Agile2012 and Agile2014, and served on the board of directors for the Scrum Alliance and the Agile Alliance.

For more information, visit www.mitchlacey.com, where you will find Mitch's blog as well as a variety of articles, tools, and videos that will help you with your Scrum and agile adoption. He can also be found on Twitter at @mglacey and by email at mitch@mitchlacey.com.

This page intentionally left blank

This page intentionally left blank

Chapter 27

Documentation in Scrum Projects

We've all heard the common myth, *Agile means no documentation.* While other agile fallacies exist, this is a big one, and it could not be farther from the truth. Good agile teams are disciplined about their documentation but are also deliberate about how much they do and when. In this chapter's story, we find a duo struggling to explain that while they won't be fully documenting everything up front, they will actually be more fully documenting the entire project from beginning to end.

The Story

"Hey, you two," said Ashley, stopping Carter and Noel in the hallway as they passed by her office. "I've been sensing some resistance from you over the initial project documentation. I need it by next Friday for project sign off, okay?" Ashley looked back at her computer and began typing again, clearly expecting a quick answer.

Carter and Noel looked at each other, then back at Ashley, their manager, before replying. They had known this conversation was coming but didn't realize they'd be accosted in the hallway by an obviously harried Ashley when it did.

"Listen, we can document everything up front like you ask," Noel began, as she and Carter moved to stand close to Ashley's doorway. "But we don't think it's the best approach. Things change, and we cannot promise you that things will go as planned. Further—" Ashley stopped typing and looked up, interrupting Noel mid-stream.

"Look, I don't want to argue about something as basic as documentation. I just need it on my desk by Friday."

Carter spoke up.

"Ashley," he began. "Can I have five minutes to communicate a different approach? I know you have a full plate, but I think it's important for you to try to understand this point before we table our discussion."

Ashley glanced at her watch and nodded. "Five minutes. Go."

"When I was in college, I worked for our university newspaper," Carter explained. "I was a sports photographer and always attended local football games with the sports writers. I was on the field, and they were in the stands.

"It probably won't surprise you to hear that not one of those sports writers came to the football game with the story already written. Now, they might have done some research on the players. They might have talked to the coaches about their game plans. They might even have asked me to be sure to get some shots of particular players. But they never wrote the article before the game even began.

"That's kind of what you are asking us to do with the software. You want the complete story of how this application will unfold, including the final game score, before we've even started playing," said Carter.

Ashley replied, "Well, that's how we get things done around here. Without the up-front documentation—design docs, use cases, master requirements docs, architectural designs, and release plans—I won't get project approval, and I can't be sure that you two and the rest of the team understand what we need to build."

"Right. I get that," agreed Carter. "It's not unreasonable for you to want *some* information before we get started. And you should expect to receive frequent updates from us on what's going on with the project. After all, the reporters I used to work with would take notes and write snippets of the article about the game as it was unfolding. Periodically, we would get together to discuss the angle they were working on and some of the shots I had captured so far.

"But to ask us to tell you what the software will look like, exactly how much it will cost, and precisely when we'll be done is like asking us to predict the final score of the football game. We can tell you how we *think* it's going to go, but when things are changing and unfolding, it's difficult to predict all the details."

Ashley nodded. "But things aren't always that volatile with our projects. We know basically what we want this to look like. It's only some of the features we aren't sure of."

"Right," chimed in Noel. "And on projects where we can nail down most of the variables and have a clear picture of the final product, we can give you more up-front documentation."

Carter nodded. "To go back to my sports writer analogy, there were times when one team was clearly dominating—the game was a blowout. In those cases, the reporters had their stories mostly written by halftime. They'd already come up with the headline, filled in a lot of the details, and were just waiting until the end of the game to add the final stats and score.

"Most times, though, the games were close and the outcome uncertain. In those cases, the reporters would keep filling in the skeleton of the story with the events as they happened in real time. They would come down to the field at halftime, and we would discuss the unfolding story. We'd strategize and say, 'If the game goes this way, we'll take this approach. But if it goes that way, we'll take this other approach.'

"Likewise, the level of detail in our documentation at any given point in the project should depend on how certain we are that things aren't going to change."

Ashley leaned back in her chair with her hand on her chin, deep in thought. Noel decided to go in for the kill.

"Ashley, remember the 1989 San Francisco earthquake, or the quake and tsunami in Japan? Or when either Reagan or Kennedy were shot?"

Ashley nodded.

"Well, you would notice a trend in all these events. In the initial accounts, the media headlines conveyed the tragic event, but with very few details. All they could tell us at first was generally what had happened (explosion/quake/tsunami/shots fired), when, and where. Why? Because the events were still unfolding, and that was all anyone knew. As the reporters on scene learned more, they added the new facts and changed the headlines.

"All the little updates, facts, and details were important to capture in real time, even if they later had to be changed to reflect new information. Without continuous briefings, many of the details surrounding the events would have been forgotten in the chaos of the devastation. The reporters didn't try to write more up front than the facts they knew. Instead, the reporters recorded what they did know as they went along. Later, after the details had solidified, they went back through the various articles and wrote a larger, encompassing synopsis that outlined the specific event from the initial reports to the current state," Noel explained.

"That's what we're suggesting we do: Make our documentation a story in progress. Are we making sense?" asked Carter.

Ashley sat forward.

"I think I get it now. What I originally heard you say was, 'I can't give you documentation.' But what you're actually saying is that you will document certain things up front, most things in real time, updating them as necessary to reflect reality, and some things after the fact. But what does that mean in terms of software exactly? I need certain documents to get the project approved, such as the master requirements documentation outlining all the use cases."

Noel answered, "We will provide that MRD, but it will be in the form of a product backlog, with stories at different levels of detail. Some of the stories will be headlines only. Others will be fully fleshed out, maybe to the point of use cases. For example, the team has a good idea of the architectural direction, which gives us insight into what stories to build for the first six or so sprints, but after that, things get fuzzy. We know that you want us to capture the details, and we have those details for the higher-priority stories, but we don't have them for the stories off in the distance. For those, we might only have headline-level information."

Ashley asked, "But if we don't capture details up front, won't they get lost? And don't you then have to spend a significant amount of time getting and documenting that information during the project, if they can be found at all? How can you ensure that what is being built is correct?"

"Remember, like any good journalist, we will document as we go, as soon as we can, without doing too much. That way when we get to the point where the UI stabilizes, let's say, we can create the more detailed documentation that the company needs and that we need to deliver in the form of user manuals and such. We won't lose our details because we'll be going over those details *every sprint* with the team, documenting them as part of our definition of done, and checking with the stakeholders to ensure we're on track, to ensure that what is being asked is what we are building and matches what the customers meant. Keeping documentation up to date a little at a time is just part of the cost of building working software. If things change along the way, we will update what we have written to reflect the change. It's a balance between stability and volatility. The more volatile something is, the more careful we need to be in what level we document. If it's stable, we can do something such as a large database diagram model in a tool. If it's volatile, we might just draw a picture on the white-board—again, both are documents, database models to be exact, but they are very different in terms of formality," finished Noel.

"So, are we on the same page?" asked Carter.

"Yes," said Ashley. "I get it now. I think this is a good approach and something that I will advocate, provided you give me regular feedback so I can update executive management. But I still need the big headlines by Friday. Agreed?"

"Agreed," said Carter and Noel together.

And that was that.

The Model

Many people can quote the part of the Agile Manifesto that states, "Working software over comprehensive documentation," but they fail to mention the very important explanatory component that follows: "While there is value in the items on the right, we value the items on the left more" [BECK]. Scrum teams still value documentation; they just change the timing of that documentation to be more consistent with their level of knowledge.

For example, imagine you are in a university world history class. You get to the point in the class when it's time to discuss Western European history. Your professor says, "I want each of you to buy my new book *Western European History: The 30th Century.* Come prepared for an exam on the first five chapters in two weeks."

You would probably look around the room, wondering if what you just heard was correct and ask a fellow student, "Did he just say *thirtieth* century history?"

Common sense tells you that without time machines, it is impossible to read a factual account of future events—they haven't happened yet! Sure, there are predictors and indicators that suggest what *might* happen, but nothing is certain. This then begs the question: If this approach is wrong for a university class that *may* reach a few thousand, why is the exact same approach accepted when developing software that has the potential to reach millions?

Before we've begun any work on a project, we are often asked for exact details as to what will be delivered, by when, and at what cost. To determine these things, teams often write volumes of documents detailing how the system will work, the interfaces, the database table structures—everything. They are, in essence, writing a history of things that have yet to occur. And it's just as ludicrous for a software team to do it as it would be for your history professor.

That doesn't mean we should abandon documents, and it doesn't mean that we should leave everything until the end, either. A certain amount of documentation is essential at each stage of a project. Up front, we use specifications or user stories to capture ideas and concepts on paper so that we can communicate project goals and strategies. When we sign off on these plans, we agree that what we have documented is the right thing to do.

The question, then, is not whether we should document, but *what* we should document and *when*. The answer has everything to do with necessity, volatility, and cost.

Why Do We Document?

Every project needs a certain amount of documentation. In a 1998 article on Salon.com titled "The Dumbing-Down of Programming," author Ellen Ullman notes how large computer systems "represented the summed-up knowledge of human beings" [ULLMAN]. When it comes to system documentation, we need to realize that we're not building or writing for us; we are writing for the future. I think Ullman summarizes it best with this snippet from the same article:

> *Over time, the only representation of the original knowledge becomes the code itself, which by now is something we can run but not exactly understand. It has become a process, something we can operate but no longer rethink deeply. Even if you have the source code in front of you, there are limits to what a human reader can absorb from thousands of lines of text designed primarily to function, not to convey meaning. When knowledge passes into code, it changes state; like water turned to ice, it becomes a new thing, with new properties. We use it; but in a human sense we no longer know it.*

Why is Ullman's concept of code as a process that changes state important? Because we need to realize that, in a human sense, we use the system and we know the system. That is why we document.

So, what is essential to document, and what is needless work? Much of that depends on the type of system you are building and the way in which you work. Teams that are colocated need to document less than do teams distributed across continents and time zones. Teams that are building banking systems need to satisfy more regulatory requirements than do teams building marketing websites. The key is to document as much as you need and nothing more.

What Do We Document?

The list of essential documents is different for every project. Going through my list of past projects, some frequent documentation items include the following:

- End-user manual
- Operations user guide

- Troubleshooting guide
- Release and update manual
- Rollback/failover manual
- User stories and details
- Unit tests
- Network architecture diagram
- Database architecture diagram
- System architecture diagram
- Acceptance test cases
- Development API manual
- Threat models
- UML (Unified Modeling Language) diagrams
- Sequence diagrams

We didn't write all these documents before the project began. And we didn't wait until the final sprint to start them either. We did them as the information became available. Many of the user stories, for instance, were written up front. But some of them were changed, and others were added as the project progressed and requirements became clearer. Our unit tests were written as we coded. And at the end of every sprint, we updated the end-user manual to reflect new functionality. We included in our definition of done what we would document and when we would write it (see Chapter 7, "How Do You Know You're Done?").

When and How Do We Document?

So, if we don't do it all up front and we don't save it all for the end, how does documentation happen in an agile project? Documentation, any documentation, costs money. The more time it takes to write and update, the more it costs. What agile projects strive to do is minimize write time, maintenance time, rework costs, and corrections.

Let's look at a few approaches we can take when documenting our projects:

- Document heavily in the beginning.
- Document heavily in the end.
- Document as we go along.

Document Heavily in the Beginning

Traditional projects rely on early documentation. As you can see from the diagram in Figure 27-1, a typical waterfall team must capture requirements, build a project plan, document the system architecture, write test plans, and do other such documentation at the beginning of the project. If we were to overlay a line that represented working software, it would not begin to move up until the gray line started to flatten.

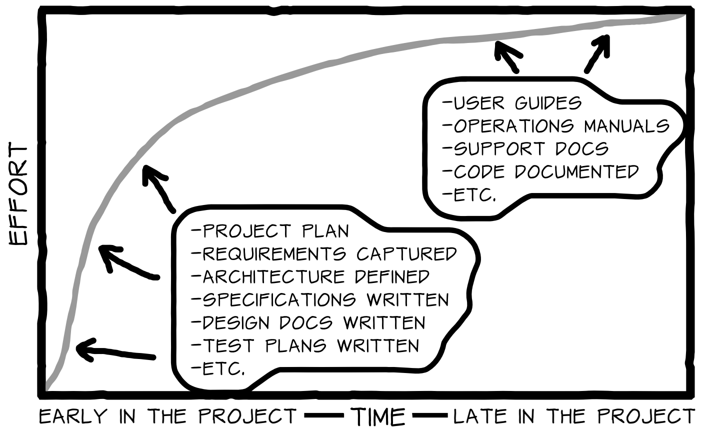

FIGURE 27-1 Traditional project with up-front documentation

The benefit of this approach is that people feel more secure about the system being built. The major drawback is that this sense of security is misleading. In point of fact, though a great deal of time, effort, and money has gone into writing the documents, no working software has yet been created. The chances of getting everything right up front are marginal on stable projects and next to zero on volatile projects. That means factoring in costly rework and extra time. Chances are good that these high-priced, feel-good documents will turn into dusty artifacts on the project bookcase.

Document Heavily at the End

When we document heavily at the end, we document as little as possible as the software is developed and save all the material needed to release, sustain, and maintain the system over time until the end of the project. Figure 27-2 illustrates this approach.

The benefits of this approach are that working software is created quickly and that what is eventually written *should* reflect what the system does.

There are many problems with this approach. People often forget what was done and when, and what decisions were made and why. Team members on the project at the end are not necessarily the people on the project in the beginning; departing team members take much of their knowledge with them when they go. After the code for a project is complete, there is almost always another high-priority project that needs attention. What usually happens is that most of the team members move on to the new project, leaving the remaining team members to create the documentation for the system by themselves. Countless hours are spent hunting for data and trying to track down and access old team members, who are busy with new work and no longer have time for something "as insignificant as documentation."

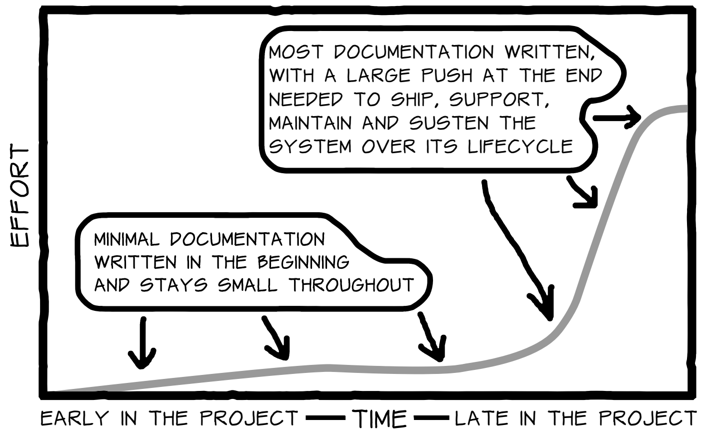

FIGURE 27-2 Documenting heavily at the end of the project

Though saving documentation until the end is cheaper in the beginning because more time is spent on actual software development, it is usually expensive in the end because it can hold up a release or cause support and maintenance issues, as it will likely contain gaps and faulty information.

Document as We Go Along

Agile projects do things differently. We acknowledge that while we can't know everything up front, we do want to know some things. We also maintain that documentation should be part of each story's definition of done, so that it is created, maintained, and updated in real time, as part of the cost of creating working software. Figure 27-3 illustrates the document-as-we-go approach.

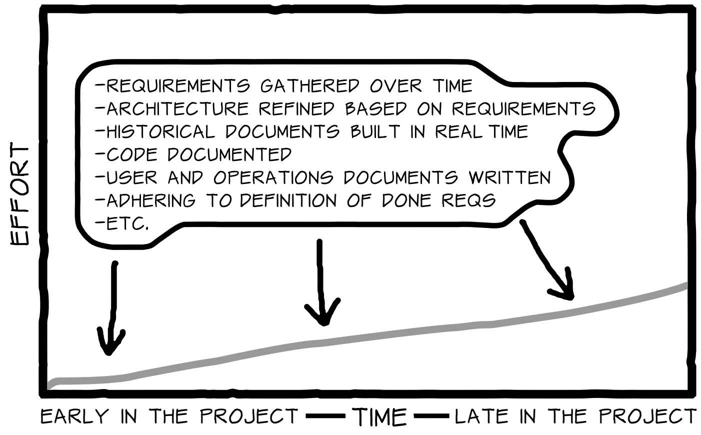

FIGURE 27-3 Documenting as you go

The product owner works with the stakeholders and customers to build the requirements while the team works with the product owner to achieve emergent design and architecture. The team keeps the code clean, creates automated tests, and uses code comments and other tools to slowly build other required documentation for the system, such as the user manuals, operations guide, and more.

The one drawback is that it does take a little longer to code when you document as you go than it would to fly through the code without having to write a comment or update an architectural diagram. This drawback is more than offset, though, by the benefits. There is less waste, less risk of eleventh-hour holdups, and more emphasis on working software. Much of the documentation is updated automatically as changes are made to the code, reducing maintenance and rework costs. Just as news reports capture the details of a story for posterity, real-time documentation of decisions and behavior minimizes gaps in knowledge and creates a living history of the software for future teams and projects.

Documenting in an Agile Project

So, we agree that in most cases, agile teams will want to document as they go. What exactly does that look like on a typical software project? To illustrate, let's use a document that is familiar to almost everyone: the user manual. A waterfall approach would be to write the entire manual at the end. We've discussed why this method is a workable but risky solution. The more agile way to approach a user manual is to include "update the user manual" as one of the acceptance criteria for a story that has to do with user-facing functionality. By doing that, the manual is updated each time working software is produced.

Let's say, for example, that I'm writing the user manual for an update to Adobe Lightroom (my current favorite piece of software). I'm in sprint planning, and the product owner explains that the story with the highest priority is "As an Adobe Lightroom user, I can export a series of photographs to Adobe Photoshop so I can stitch them together to make a panorama." As we're talking through that story, I recommend that we add "update user manual to reflect new functionality" as one of the acceptance criteria for that story.

As I write the code or as I finish the feature, I would also edit a document that provides the user instructions on how to use the feature. Depending on how stable the feature is, I might even include screenshots that walk the user through the instructions for both Lightroom and Photoshop. If the feature is less stable, meaning the core components are built but the user interface team is still hashing out the user interface through focus groups, I would document the behavior but probably only include placeholders for the screenshots. The key here is that the story would not be done until the user manual is updated.

Updating the user manual would be appropriate to do at the story level, as I described, but could also be accomplished at the sprint level. For instance, if we have several stories that revolve around user-facing functionality, we might add a story during sprint planning that says, “As a user, I want to be able to learn about all the new functionality added during this sprint in my user manual.”

What I am doing is balancing stability versus volatility of the feature to determine how deep I go and when. It would not, for example, be prudent to make updating the user manual part of the definition of done for a task. Too much might change before the story is complete. Nor would it be acceptable to wait to update the user manual until right before a release. That’s far too late to start capturing the details of the new behaviors.

When determining when to document your own systems, you also should balance cost, volatility, and risk. (For more on determining your definition of done, refer to Chapter 7.)

Starting Projects without Extensive Documentation

One challenge you will face is to help stakeholders and customers understand why you are not documenting everything up front. Tell them a story similar to the one Carter told at the beginning of this chapter (or share that story with them). Remind them that while documenting heavily up front drives down the perceived risk, you never know what you don’t know until a working solution is in place.

Eschewing extensive documentation up front does not mean you are off the hook for a project signoff piece. But it does mean that the piece will look different to your stakeholders than it has on other projects. Rather than give them the specific artifacts they request, answer the questions they are asking in regard to schedules and requirements in the most lightweight way possible for your project and situation. A PMO might, for instance, ask for a Microsoft Project plan, but what the PMO really wants to know is what will be done by about when. By the same token, a stakeholder might ask you for a detailed specification, when what she really wants to know is, “Are you and I on the same page with regard to what I’m asking you to do?”

Signoff and approval will occur early and often. The product owner will hold many story workshops to build the product backlog, work with the team to build the release plan, and then communicate that information to all interested parties, soliciting enough feedback to ensure that the team will deliver what the stakeholders had in mind (which is rarely exactly what they asked for). The documents the product owner uses for these tasks are only a mode of transportation for ideas and concepts, and a document is not the only way to transfer those ideas. Up-front documentation can just as easily take the form of pictures of whiteboard drawings, sketches, mockups, and the like—it does not need to be a large formal document.

The beginning of the project is when you know the least about what you are building and when you have the most volatility. What your stakeholders need is the peace of mind that comes from knowing you understand what they need and can give them some idea of how long it will take to deliver. Expend the least amount of effort possible while still giving them accurate information and reassurance. At this point in the project, everything can and will change.

Keys to Success

The keys to success are simple:

- **Decide**—Determine what you need to document for your project and when it makes the most sense to produce that documentation. Some things, such as code comments, are easy to time. Other items, such as threat models, are more difficult. Work as a team with your product owner to determine the must-have documents at each stage of your project.
- **Commit**—Once you have a documentation plan, stick to it. Put it in your definition of done. Hold yourselves accountable. Documentation is never fun, even when it's broken into small chunks. Remind your team that a little bit of pain will eliminate a great deal of risk come release time.
- **Communicate**—If this is the first project to move forward without extensive up-front documentation, the stakeholders will be nervous. Help them out, especially at the beginning of the project, by sending frequent updates, pictures of whiteboards, and any other documents that are produced. Do as your math teacher always told you: show your work. Seeing working software and physical artifacts goes a long way toward calming the fears of even the most anxious executives.
- **Invest in automation**—Documentation is easier and ultimately cheaper if you invest a little time in automating either the system or the documentation itself. For example, if you can create an automated script to compile all the code comments and parse them into documentation, you've saved a manual step and instantly made your documentation more in sync with the actual code. It's also much easier to document acceptance test results and API documents automatically than it is to do it manually. On the flip side, you might find that automating the features themselves can save you a lot of documentation work. For example, a manual installation process might require a 40-page installation guide; an automated installation process, on the other hand, probably needs only a one-page guide and is better for the end user as well. Whenever possible, automate either your documentation or the features it supports. The results are well worth the investment.

Being agile does not equate to *no* documentation; it means doing timely, accurate, responsible documentation. Make sure that documentation is equally represented in your team's definition of done alongside things like code and automation. Remember that when change happens, it's not just the code that changes—the entire software package that you are delivering changes, documentation included. Lastly, remember that as much as you might wish otherwise, documentation is a part of every software project. When you do a little at a time and automate as much as possible, you'll find that while it's still an obligation, it's not nearly as much of a chore.

References

[BECK] Beck, Kent, et al. "Manifesto for Agile Software Development." Agile Manifesto website. http://agilemanifesto.org/ (accessed 16 January 2011).

[ULLMAN] Ullman, Ellen. Salon.com. http://www.salon.com/technology/-feature/1998/05/13/feature (accessed 18 November 2010).

Index