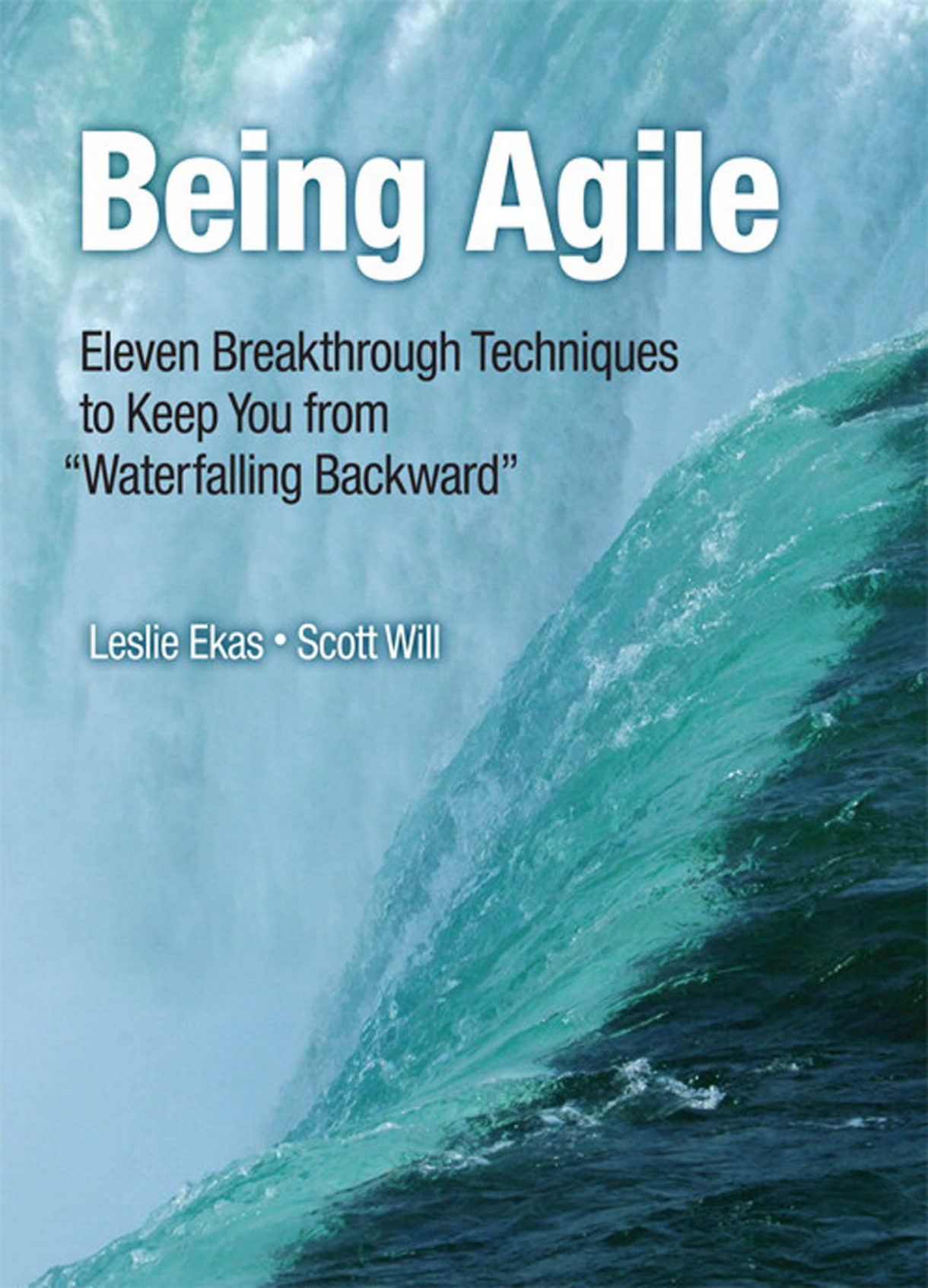
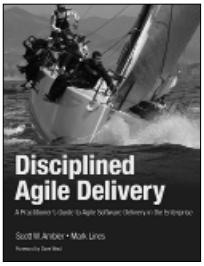

Being Agile
Eleven Breakthrough Techniques
to Keep You from
"Waterfalling Backward"
Leslie Ekas • Scott Will

Related Books of Interest



Disciplined Agile Delivery

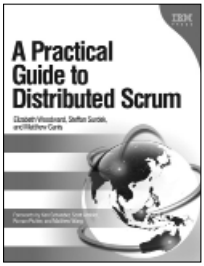
A Practitioner's Guide to Agile Software Delivery in the Enterprise

By Scott W. Ambler and Mark Lines

ISBN-13: 978-0-13-281013-5

It is widely recognized that moving from traditional to agile approaches to build software solutions is a critical source of competitive advantage. Mainstream agile approaches that are indeed suitable for small projects require significant tailoring for larger, complex enterprise projects. In *Disciplined Agile Delivery*, Scott W. Ambler and Mark Lines introduce IBM®'s breakthrough Disciplined Agile Delivery (DAD) process framework, which describes how to do this tailoring. DAD applies a more disciplined approach to agile development by acknowledging and dealing with the realities and complexities of a portfolio of interdependent program initiatives.

Ambler and Lines show how to extend Scrum with supplementary agile and lean strategies from Agile Modeling (AM), Extreme Programming (XP), Kanban, Unified Process (UP), and other proven methods to provide a hybrid approach that is adaptable to your organization's unique needs.



A Practical Guide to Distributed Scrum

By Elizabeth Woodward, Steffan Surdek, and Matthew Ganis

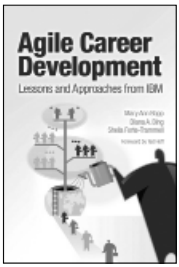
ISBN-13: 978-0-13-704113-8

This is the first comprehensive, practical guide for Scrum practitioners working in large-scale distributed environments. Written by three of IBM's leading Scrum practitioners—in close collaboration with the IBM QSE Scrum Community of more than 1,000 members worldwide—this book offers specific, actionable guidance for everyone who wants to succeed with Scrum in the enterprise.

Readers will follow a journey through the lifecycle of a distributed Scrum project, from envisioning products and setting up teams to preparing for Sprint planning and running retrospectives. Using real-world examples, the book demonstrates how to apply key Scrum practices, such as look-ahead planning in geographically distributed environments. Readers will also gain valuable new insights into the agile management of complex problem and technical domains.

Sign up for the monthly IBM Press newsletter at
ibmpressbooks.com/newsletters

Related Books of Interest



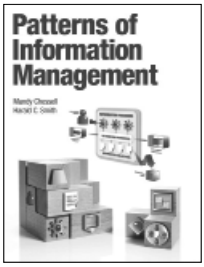
Agile Career Development
Lessons and Approaches from IBM

By Mary Ann Bopp, Diana A. Bing, Sheila Forte-Trammell

ISBN-13: 978-0-13-715364-0

Supercharge Performance by Linking Employee-Driven Career Development with Business Goals

How do you make career development work for both the employee and the business? IBM® has done it by tightly linking employee-driven career development programs with corporate goals. In Agile Career Development, three of IBM's leading HR innovators show how IBM has accomplished this by illustrating various lessons and approaches that can be applied to other organizations as well. This book is for every HR professional, learning or training manager, executive, strategist, and any other business leader who wants to create a high-performing organization.



Patterns of Information Management

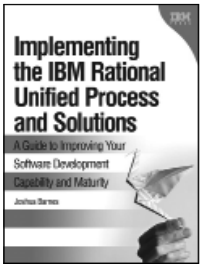
By Mandy Chessell and Harald Smith

ISBN-13: 978-0-13-315550-1

Use Best Practice Patterns to Understand and Architect Manageable, Efficient Information Supply Chains That Help You Leverage All Your Data and Knowledge

In the era of "Big Data," information pervades every aspect of the organization. Therefore, architecting and managing it is a multi-disciplinary task. Now, two pioneering IBM® architects present proven architecture patterns that fully reflect this reality. Using their pattern language, you can accurately characterize the information issues associated with your own systems, and design solutions that succeed over both the short- and long-term.

Related Books of Interest



Implementing the IBM® Rational Unified Process® and Solutions

By Joshua Barnes

ISBN-13: 978-0-321-36945-1

This book delivers all the knowledge and insight you need to succeed with the IBM Rational Unified Process and Solutions. Joshua Barnes presents a start-to-finish, best-practice roadmap to the complete implementation cycle of IBM RUP—from projecting ROI and making the business case through piloting, implementation, mentoring, and beyond. Drawing on his extensive experience leading large-scale IBM RUP implementations and working with some of the industry's most recognized thought leaders in the Software Engineering Process world, Barnes brings together comprehensive "lessons learned" from both successful and failed projects. You'll learn from real-world case studies, including actual project artifacts.

Work Item Management with IBM Rational ClearQuest and Jazz

A Customization Guide

Shmuel Bashan, David Bellagio

ISBN-13: 978-0-13-700179-8

Software Test Engineering with IBM Rational Functional Tester

The Definitive Resource

Davis, Chirillo, Gouveia, Saracevic, Bocarsley, Quesada, Thomas, van Lint

ISBN-13: 978-0-13-700066-1

Enterprise Master Data Management

An SOA Approach to Managing Core Information

Dreibelbis, Hechler, Milman, Oberhofer, van Run, Wolfson

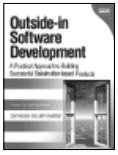
ISBN-13: 978-0-13-236625-0

An Introduction to IMS

Your Complete Guide to IBM Information Management Systems, 2nd Edition

Barbara Klein, et al.

ISBN-13: 978-0-13-288687-1



Outside-in Software Development

A Practical Approach to Building Successful Stakeholder-based Products

Carl Kessler, John Sweitzer

ISBN-13: 978-0-13-157551-6

Sign up for the monthly IBM Press newsletter at
ibmpressbooks.com/newsletters

This page intentionally left blank

Being Agile

This page intentionally left blank

IBM PRESS

Being Agile

Eleven Breakthrough Techniques to Keep You from "Waterfalling Backward"

Leslie Ekas

Scott Will

IBM Press
Pearson plc
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ibmpressbooks.com

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

© Copyright 2014 by International Business Machines Corporation. All rights reserved.

Note to U.S. Government Users: Documentation related to restricted right. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer

Cover design: IBM Corporation

Executive Editor: Bernard Goodwin

Marketing Manager: Stephane Nakib

Publicist: Heather Fox

Managing Editor: Kristy Hart

Designer: Alan Clements

Project Editor: Elaine Wiley

Copy Editor: Apostrophe Editing Services

Indexer: Erika Millen

Senior Compositor: Gloria Schurick

Proofreader: Jess DeGabriele

Manufacturing Buyer: Dan Uhrig

Published by Pearson plc

Publishing as IBM Press

IBM Press 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
1-800-382-3419
corpsales@pearsontechgroup.com.

For sales outside the United States, please contact

International Sales
international@pearsoned.com.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both: IBM, the IBM Press logo, FileNet, Rational, ClearCase, ClearQuest, Rational Team Concert, AIX, and POWER6. A current list of IBM trademarks is available on the web at "copyright and trademark information" as www.ibm.com/legal/copytrade.shtml. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.

Library of Congress Control Number: 2013946079

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-337562-6
ISBN-10: 0-13-337562-5

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing: October 2013

This page intentionally left blank

Contents

Preface

By Leslie Ekas

Your team embarked on adopting agile a while ago, but the results you expected to see by now just haven't materialized. To be honest, you're kind of surprised, especially given all the hype about agile, the stories of wild success you've seen online, and the tremendous enthusiasm expressed by the team early on. There might even be some whispered talk in the hallway about "chucking this agile thing" and going back to waterfall. If you find yourself in a situation similar to this, then this book is for you.

The target audience is software engineers and leaders who understand how to apply agile to software development but may find their teams falling back into old habits when the going gets tough or because an old waterfall approach seems like the right thing to do. It is also for those teams that have adopted agile but do not feel like there has been a significant improvement. Our goal in writing this book is to give you the means to *be agile* as well as to help keep you from "waterfalling backward."

Transforming to agile from waterfall is no small undertaking. Resorting to old habits when trouble hits is what our instincts tell us to do. When I started to learn agile, the first organization I led experienced challenges adopting agile but, fortunately, I got some unwavering encouragement and then some timely help. Let me explain: I first learned about agile in early 2007 in a class led by Tom and Mary Poppendieck. Following the class one of my teams jumped into agile and after approximately 6 months concluded that it was enabling us to deliver better value to our customers. However, the change was not compelling, we were not getting all the practices to succeed, and we were wondering when "the big moment" would happen when significant benefits became obvious. My team leaders and I had the opportunity to get additional coaching from Tom and Mary after the class to help our team address our specific problems. We described the challenges we faced with our agile adoption: not getting to "Done!" each iteration, not working well

together as a team, failing to break our pattern of building up project debt, and so forth. We were discouraged and hoped that they would give us the key to resolving our problems. Tom and Mary both smiled and said, "You'll do fine." I wanted to scream! "What do you mean by 'fine'?!?!? We can't get it right! And you can't believe how many problems we have to fix!"

It took me a while to understand their reaction. They did give us the key to resolving our problems, or at least they told us that we already had the key. They knew we were trying new ideas and wanted to have more success than we had experienced. They knew that with more success, we would try harder and get better—and they were right. Teams that are actually trying to get better are well on their way to becoming agile.

In our quest to get better, my team did finally experience a pivotal moment that not only fixed a significant waterfall problem that we had but also changed the way that we thought about working together. We wrote this book for anyone who wants to break through old ways of thinking—but may need a few tricks to get there.

Now let me start from the beginning and describe the events that led Scott and me to write this book. I worked for years as a software developer and then as manager in typical waterfall-style projects. Our teams had good engineering discipline and shipped good products; however, I did not think that we were delivering high enough value or quality to our customers. I was frustrated because I did not know how to fix the problems. We looked at several popular approaches but none of them got my attention until I learned about agile. When I started practicing agile in 2007, I was excited because this approach made sense, and the literature indicated that adopting agile produced some believable and compelling results.

My team started its agile journey implementing daily standup meetings, using 1-month iterations, completing iteration planning, demonstrations, and reflections, and even trying to build a product backlog. I noticed several positive changes taking place within the team—but did I think the move to agile was compelling? To be honest, no. Our results were not earth shaking. We were productive before moving to agile and that had not changed significantly.

At first, my team was organized like many traditional teams are—by discipline. We had very skilled teams working on the product; however, they were used to working in their silos—and that is where they stayed. They met as a group but worked in their individual disciplines. This same problem kept surfacing during our reflections at the end of each iteration—we were not working well together. We tried to fix this problem in a variety of ways but nothing worked, and most of us sensed that not solving this problem would lead to defeat with agile. An outside consultant suggested that we try *bullpens* (more on this in Chapter 1, "Whole Teams")—and it worked. I mean, it really worked, immediately. Our team started to become a whole team, and it made a difference in our results. The change was so compelling that we never looked back. This was our breakthrough. Given how eye opening my team's first breakthrough was, it prompted us to continually look for other breakthroughs. This book is the result of our efforts and experiences in discovering other breakthroughs.

The goal for this book is to help teams that have adopted agile but are struggling to make it stick, or struggling to get compelling value from it, or both. Old habits die hard, and in those instances when agile does not stick, it is often because teams have not experienced an "Ah-ha!" moment that changes the way they think. When no compelling improvements are noted, teams tend to "waterfall backward" one compromise (or excuse) at a time. Symptoms of waterfalling backward include moving to longer iterations, resetting iteration end dates, not breaking down user stories to fit into an iteration, measuring project progress based on individual disciplines instead of focusing on "working software," and an inability to share the work due to limited domain expertise.

In this book we offer several breakthrough techniques that enable teams to experience enough of an "Ah-ha!" moment that it breaks typical, reflexive waterfall thinking, thus allowing agile thinking which, in turn, helps transform software engineering teams from simply "doing agile" to actually "being agile"—with the resulting increase in realized benefits.

So back to my opening comments: If you're looking for ways to get better, you've passed the first hurdle to actually getting better. In addition, if you know that your team needs to get better but have difficulty convincing others that more benefits can be gained from moving further down the agile path, this book is also for you. Our hope is that you discover valuable techniques and gain new insights that help you continually improve. We also hope that your successes excite other teams in your organization (especially those on the verge of giving up) to press on. We like to see teams not only adopt agile, but also make it stick. Agile is fun and it helps teams produce higher value and higher quality software.

Join the Conversation

We encourage you to join the agile conversation on our blog: "Being Agile." You can find it at www.ibm.com/developerworks/community/blogs/beingagile/?lang=en.

Acknowledgments

We really enjoyed writing this book because it helped us to distill our years of thinking about these topics. We have had many discussions through our work with various teams on what is required to help them succeed and our hope is that we can help even more teams with this book.

We received our initial education on agile software development from Tom and Mary Poppendieck. In addition to their education, the Poppendiecks gave IBM® tremendous support that we have leveraged in our coaching over the years. The Poppendiecks certainly gave Leslie and her team the motivation early on to stick with their efforts to make agile work and to get better in the process of doing so. We also want to thank Tom in particular for providing a review of our initial book proposal and his subsequent reviews of various chapters.

Leslie would also like to thank Stan Rifkin whose guidance early on helped her team achieve an "Ah-ha!" moment that really made its agile transformation become permanent.

Also, we want to thank Pramod Sadalage who provided reviews on every chapter. He helped us clarify and improve our explanations of several topics that, otherwise, might have been a little too IBM-centric.

We thank Scott's daughter Karoline Strickland for her review of our chapters in the final weeks. Her strength in writing combined with her lack of subject matter expertise made her an outstanding reviewer because she could quickly detect when our thoughts were not coming across clearly. Karoline now calls herself a "zero-gravity reviewer" (you'll just have to read the Deliver Value chapter to get the joke).

Mark Wainwright was a fellow coach in the IBM Software Group Agile Center of Competence prior to retiring from IBM. He provided very useful feedback early in the process of writing our proposal that helped us redesign our approach in conveying the material in the book. We had many great times working with Mark and his insights always kept us on track. We thank him for all his support, the things we learned from him, and for his friendship.

Dibbe Edwards, a Vice President in the IBM Rational Software organization, provided IBM executive sponsorship for this book. She has championed the agile mission in IBM for a long time and has supported our efforts to help teams realize the rewards that agile adoption has to offer. Dibbe willingly supported new approaches to solve some difficult problems and her leadership helped her teams achieve ever-higher levels of productivity when adopting agile.

As we've continued to work with teams across all of IBM, it's always gratifying to see others take up the flag and help drive agile adoption as well. Yvonne Matute is one such person and Matthew Stave is another. We would like to thank Yvonne for letting us use quotes from an email she sent to us describing her team's successes in adopting agile and to Matthew for letting us include his list of items regarding the adoption of short iterations that we've included in the Appendix of the book.

We would also like to thank Carl Kessler for his early and ongoing encouragement to all of IBM to move to agile. We both had the distinct pleasure of working for Carl at various points in our careers and, even though Carl retired from his role as a senior development executive in IBM several years ago, his name still regularly comes up in conversations revolving around helping teams adopt agile. Thank you, Carl!

It should come as no surprise that writing this book took time away from our respective families, so we would both like to publicly thank our families for putting up with late nights, missed dinners, and not doing all the other things we could have been doing with you. The support we received from you all was critical in seeing this dream come to fruition. Thank you so much!

We heartily thank all the teams that we have worked with because they persevered and got better, and they enabled us to learn and leverage their experiences and their unique perspectives for our coaching responsibilities. If it wasn't for your willingness to hear us out and to try new things, this book would never have been written. You know who you are! And now you know how thankful we are to you!

Finally, we would like to thank Steven Stansel from IBM Press, and Bernard Goodwin, Michelle Housley, and Elaine Wiley of Pearson Press for all their incredible help in making this book a reality.

We would also like to acknowledge that, as with any major undertaking, not everything is going to be perfect. Any errors and any lack of clarity you may find while reading the book remain the sole responsibility of the authors.

About the Authors

Leslie Ekas has worked in software development for over 20 years as a developer, manager, and agile coach. Her industry experience ranges from a startup, to a mid-sized company, and now IBM. She has led multiple products to market successfully over the years. She has managed teams of all sizes and many disciplines and across broad geographies. Leslie helped start the IBM Software Group Agile Center of Competence after her team's early success transforming to agile. After coaching for several years, she returned to development to lead the worldwide Rational ClearCase team. In her new job as the Smarter Infrastructure Portfolio Manager, she is helping the business team adopt an agile operational approach.

Scott Will has been with IBM for more than 22 years, the last six as an agile consultant. His experience ranges from providing consulting for small, co-located teams to teams with hundreds of engineers scattered across the world. Previously Scott was a successful programmer, tester, and customer support team lead, and he was in management for years. He is a contributing author to the book *Agility and Discipline Made Easy*, an IBM Master Inventor with numerous patents, a former Air Force combat pilot, and a graduate of Purdue University with degrees in Computer Science, Mathematics, and Numerical Analysis. He also completed his MBA while in the Air Force.

This page intentionally left blank

Introduction

By Leslie Ekas

Who This Book Is For

Transforming from a waterfall-based methodology to agile is no small undertaking. This book is for people who may find their team falling back into old habits when the going gets tough or just because an old waterfall approach seems like the right thing to do. It is also for those teams that have adopted agile but do not feel like there has been a significant improvement. The target audience includes both leaders and members of agile teams. The goal in writing this book is to give you the means to react to situations and challenges in an instinctively agile way and, thus, secure the real benefits that agile promises.

What Is Our Approach?

Adopting agile requires a change in thinking—it's not just adopting a set of practices. Too many teams have adopted a list of practices and called themselves "agile." This book can help teams get past a typical, rote approach to adopting agile and start gaining the real benefits that agile promises. Even if you are already experienced and successful with agile, you can gain additional insights that can help you and your teams be even more successful. (*Continuous improvement* is one of the fundamental concepts of agile.) Basically, we don't care what specific agile practices you've adopted—what we're concerned with in this book is whether you've gained the *benefits* from adopting those practices and, if not, we can provide some help in showing you how to do so.

Agile continues to grow in popularity because the benefits promised are substantial. As markets grow more competitive, and products become more sophisticated, software development teams need to become more efficient and effective while still ensuring high product quality and delivering real customer value. Agile gives teams a proven way to address these challenges. However, being an agile team is more than having daily standups, chunking up work into short, time-boxed iterations, and always having working software. Agile means thinking differently:

focusing on customer-value, continuous high-quality development, constant improvement, and more. If teams simply adopt some practices—without understanding the principles behind them—they may never get the full value of agile.

Through our years of working with teams, we've found that teams need ways to overcome their reflexive waterfall habits to really understand and benefit from agile. In our experience, if teams couldn't break an old habit, they would typically wind up with a modified waterfall approach instead of truly becoming agile.

This book reviews several of the foundational concepts in agile, covers the principles that undergird each of the concepts, and then discusses the corresponding practices that complement the principles. At the end of each chapter we offer a breakthrough technique that can provide a mechanism for teams to move toward the goal of "being agile" instead of just "doing agile."

And now for a warning: To make a breakthrough, we recommend techniques that are radical enough that you CANNOT fall back into a waterfall habit. You have to remove the safety net—no "cheating" allowed. One of the differentiators between this book and other agile books is that we cover more than just agile practices; we discuss the principles on which the practices are built and also offer breakthrough techniques that can help break old, bad habits. We broke through ourselves and have seen many other teams succeed with these methods.

This may sound funny, but it is our hope that when you read any of the breakthrough techniques in this book, your first reaction is to tell us that we're nuts! We want the ideas to feel edgy enough that they make you feel uncomfortable. Getting out of your comfort zone is how it works. Not all the ideas may strike you this way depending on how your team works currently—and that's OK.

An Example of a Breakthrough

The following is an example of the type of habit-breaking we're referring to, but it comes from outside the realm of software engineering. Scott is a competitive marksman, as is his wife and several family members. However, his oldest daughter always had problems with flinching when she would shoot—she was anticipating the recoil of the firearm and it would inevitably cause her shots to be off the mark. Scott had tried numerous ways to help his daughter overcome the flinching habit, but often with little improvement noted. One day he suggested that his daughter try shooting with both eyes open. (Note that most shooters will shoot with only their dominant eye open, focusing primarily on the firearm's front sight. When shooting with both eyes open, the focus must transition to the target instead of the front sight.) When his daughter tried shooting with both eyes open, suddenly the flinching stopped and she started hitting the target right where she was aiming. The radical change in the shooting fundamentals likely "overloaded" her reflexive, flinching response and helped her instead to become an excellent shooter, almost immediately. It truly was a *breakthrough*...

What Does This Book Cover?

In this book, Scott and I have distilled our collective 12 years' experience in both leading software teams through adopting agile and subsequently coaching many, many additional teams. The book is divided into chapters that focus on 11 crucial topics for agile organizations. Chapter 1, "Whole Teams," explores the vital need for developers, testers, and product documentation writers to work together during each iteration to accomplish a small but valuable slice of functionality. Each small portion of functionality can—and should—be regularly demonstrated to customers, as described in Chapter 2, "Active Stakeholder Interaction," where we provide guidance on how to get the most from these regular interactions.

You may be challenged with getting to "Done!" every iteration. In Chapter 3, "Queuing Theory," you'll see how working on—and regularly completing—some small amount of functionality allows teams to be much more productive and efficient than has ever been possible with typical waterfall approaches. A common refrain in agile circles is, "Stop starting and start finishing!"

One of the primary obstacles to regularly finishing small amounts of work is the pervasiveness of multitasking, and in Chapter 4, "No Multitasking," we help you see why multitasking (sometimes referred to as task-switching) is inherently inefficient. Here we help you see the difference between *busyness* and *productivity*.

One of the principles of the Agile Manifesto is "Simplicity—the art of maximizing the amount of work not done—is essential."[1] To maximize the amount of work *not* done, there has to be an intense focus on eliminating waste. Waste can take many forms in software development. Chapter 5, "Eliminate Waste," provides numerous ways to help you both to see waste and to get rid of it.

The flip-side of eliminating waste is ensuring that what is created is valuable—especially with respect to your stakeholders. Chapter 2 covers how to engage with your stakeholders during a release. In Chapter 7, "Deliver Value," we discuss ways to ensure that what your team is creating actually provides value to your stakeholders.

And the best way to engage with your stakeholders regularly is to always have working software. Two-hundred lines of code that provide some small amount of functionality, that are "release-ready," and that can be demonstrated to customers to get their feedback is far more valuable than 2,000 lines of code that haven't been tested and can't be shown to customers. Teams should focus on always having working software, and Chapter 6, "Working Software," covers just how important of a practice this is for agile teams.

Always having working software allows teams to have much greater flexibility for actually releasing a product. Scott and I come from the enterprise application software world in which it is common for products to release once every 2 to 3 years. Given the rapid changes going on in the industry today, with the advent of cloud technologies and continuous delivery approaches, shorter release cycles are becoming the norm. But even apart from these, shorter release cycles

1. http://agilemanifesto.org/principles.html

are more efficient and provide more flexibility to teams than longer release cycles. Chapter 8, "Release Often," provides much food for thought on this topic.

What do you do when a critical problem surfaces? Do you just address the symptoms and move on? Chapter 9, "Stop the Line," encourages teams to immediately stop work when a critical problem surfaces and fix the problem at its root. Doing so means that the problem will never surface again.

As with stop the line behavior, many things covered in the book will likely seem counterintuitive, especially given the hold that waterfall thinking has had on software development for decades. Real transformation requires both a technical and executive leadership that genuinely understands agile principles and practices, is willing to learn more, encourages continuous improvement, and provides a penalty-free environment. Chapter 10, "Agile Leadership," shows that meaningful transformation cannot occur without leaders who engage with their teams when embarking on an agile transformation.

Perhaps the most important chapter in this book is Chapter 11, "Continuous Improvement." We frequently tell teams that there is no such thing as "100 percent agile" because "being agile" means always looking for ways to get better. The best teams we've worked with are those that continually look for ways to get better. Chapter 11 provides a lot of encouragement to adopt a continuous improvement mindset as part of your day-to-day thinking.

The chapters are organized so that they can act as a ready reference for each topic. In addition, the chapters can be read in any order; in general, one chapter does not build on any previous chapter. However, because many of the concepts and principles have a lot of affinity with each other, you can see that we've emphasized some items in more than one place in the book. Adopting one agile concept or practice tends to pull other concepts and practices along with it. For instance, the concepts of Stop the Line, Eliminate Waste, and Continuous Improvement overlap in both intent and practice. However, they are separated in the book in order to reinforce the value of each concept in isolation.

As a side-note, Scott and I firmly believe that teams should not try to "tip-toe" into adopting agile. We recommend jumping in with both feet. Yes, it will be messier and more chaotic than tip-toeing in, but the benefits will accrue faster because teams will learn faster. Pulling stuff in slowly delays many of the benefits and can often actually derail motivation for going further. ("We haven't seen much benefit yet, why should we take on something else? Why not just go back to waterfall...?") Jumping into agile allows you to change your thinking, and consequently your reflexive habits much more quickly.

An Overview Of The Content

Each chapter is divided into four main sections: Principles, Practices, Metrics, and Breakthrough. The Principle section provides the conceptual foundation upon which the topic is built and shows how the ensuing practices support the concept. These principles often provide the necessary insight to help teams understand the "why" of adopting any given practice. Each principle is

described in terms of why it is important for agile thinking. The practices described are those necessary for really making a transformation in an agile way. There is not a one-to-one mapping between the principles and the practices—there is actually a lot of overlap between all the principles and practices. In fact, in some cases it can be hard to classify an item as either a principle or a practice because it can easily fit into both categories. So if this seems a bit confounding, don't worry about it. The content of the concept matters more than the context.

If things are working well you generally know it, but it might be useful to have a way to measure progress. Metrics are often useful to entice another team to try something similar, or just to give you some context for your own results. The metrics section provides possible ways to evaluate if the principles are sticking. Be careful, though, to not let them be your only guide in understanding improvement.

The set of concepts, principles, and practices that we've selected are ones that we have found to be critical in helping our teams succeed. We are not trying to be exhaustive—there are much better references for that. Our intent is to build a better understanding of each concept so that you understand the thinking that is achieved by the breakthrough method. At the end of each chapter we introduce a breakthrough technique to help make lasting transformations. Each technique is intended to help teams break their traditional, reflexive (waterfall) thinking patterns. We did not invent all the breakthrough techniques—some have been adopted from our interaction with other agile practitioners (such as the bullpen technique mentioned in the Preface). The others are ones we've discovered through our coaching experiences. We know there are more techniques out there and hope that this book will encourage teams to share what's worked for them.

We've also included a number of stories from teams with which we have worked. Some of the stories are of successes. Others describe challenging situations we faced when first working with a team—situations that forced us to think of out-of-the-box ways to help them. Some are simply illustrative.

As mentioned earlier, the focus of much of our coaching experience has been in helping enterprise application software development teams. Enterprise application software is generally defined as software that enables organizations to run their businesses better. It is not intended just for individual usage. As such, it is expected to have a life cycle of up to 20 years or more in some cases, is deployed to thousands of users, can run on a variety of hardware and software platforms, can typically be enhanced via supported interfaces such as APIs, and generally is built with millions of lines of code. Examples include Facilities Management for global companies, Enterprise Content Management that protects intellectual property, and Business Process Management that guarantees audit-ready process compliance for regulated industries. Enterprise software obviously needs to operate continuously, needs to provide the ability to scale, must ensure that the business value offered is secure, and more. For these reasons, enterprise software development environments have tended to have heavier-weight processes as well as long-entrenched waterfall practices. Teams tend to be large and are often spread out geographically and functionally. However, we are convinced that if these principles, practices, and breakthroughs work in enterprise application software development, they can easily be applied to smaller development projects.

What Do You Have To Do?

For this book to be useful, teams have to be willing to "jump off the cliff" (so to speak) and give these breakthrough techniques a try. Many of the techniques can be tackled in a couple of weeks, but adopting one or more means removing other distractions and staying focused so that the techniques get the attention they require. Everyone on the team needs to have "skin in the game" and make a conscious effort to succeed. Such a mindset can serve the team well on its agile transformation journey.

Making agile stick is hard work. You may have to update your technologies, get rid of your project debt, be rigorous in your planning and execution, learn how to work as a team, listen to your stakeholders when they tell you what you may not want to hear, and keep the focus on getting better. But your efforts will succeed. We can generally tell when teams are doing well with their agile transformation because they are *having fun*.

What Benefits Can You Get from Reading This Book?

In this book, you can

- Learn techniques that will break typical waterfall response patterns learned over many years and which can also help with adopting a new agile way of thinking.
- Learn why agile recommends various practices and why they are key to being successful.
- See how to gain the real benefits that agile promises.
- Benefit from the authors' combined 12 years' experience at IBM® in coaching IBM enterprise software teams in agile.
- Benefit from the experiences of other teams who have adopted agile.
- Use a short questionnaire at the end of the book to assess whether your team is just "doing agile" or if you are actually "being agile."
- Add a useful, continually referenced agile book to your library.

Who Are We?

As with adopting agile, learning how to coach effectively requires years of experience and continuous improvement. Both Scott and I became agile coaches and have spent many hours discovering how to help teams get better. Just like agile adoption, coaching is a journey. Scott's years as an instructor pilot in the Air Force trained him to communicate techniques effectively, often requiring him to find different ways to communicate the same principle to people who learned and understood things differently. One of Scott's favorite sayings is, "Communicate, communicate, communicate!" This book has the same intention: to give you different ways to understand critical agile concepts so that they can be communicated to a wide spectrum of teams.

After leaving the Air Force in 1991, Scott joined IBM and has been a developer, Chief Programmer, Customer Support team lead, Development Operations manager, System Test manager, Quality Engineering manager, and had responsibility for the Quality Management System for all IBM Software Group—all prior to becoming an agile coach.

In addition to coaching agile teams, Leslie has been the senior development manager for several organizations in IBM making the transition to agile. Her efforts started with her heritage teams in FileNet® working on Enterprise Content Management and continued to Rational® working on the ClearCase® and ClearQuest® products, and then onto the Cloud and Smarter Infrastructure organization focusing on the Smarter Infrastructure products. She worked for several companies prior to joining IBM, including Boeing, Tandem, the Saros Corporation, and FileNet (acquired by IBM in 2006). Teams are often tasked with "adopting agile" without much management participation and, given her management background, her perspective is that successful adoption starts with leadership—and Scott is in complete agreement. We drive this point throughout the book.

As agile coaches, we enjoy the "I get it!" moments that people experience after they make a breakthrough in their thinking. This is the goal we have for you. And if you don't "get it" from reading this book, hopefully you'll "get it" from trying the breakthrough techniques. After you do get it, it feels good—and you will want to find yet more new ways to drive improvement. And that, in turn, is fun!

Join the Conversation

We encourage you to join the agile conversation on our blog: "Being Agile." You can find it at: https://www.ibm.com/developerworks/community/blogs/beingagile/?lang=en

This page intentionally left blank

CHAPTER 1

Whole Teams

Being agile requires whole teams because the synergy derived from cross-disciplined and cross-component teams working together enables teams to be more productive than working in isolation.

By Leslie Ekas

Do any of these phrases sound familiar?

- "When are you going to get your problem solved?"
- "Oops, I forgot to copy John on the email that discussed our proposed solution."
- "The test team can't start working on this release yet, so we will have to start writing code without them."

These are phrases typical of teams that work in silos. If you regularly hear such comments from your team, you are likely not experiencing the benefits of whole teams. Whole teams are composed of people who work together to deliver a product.

The first time I saw the power of whole teams went something like this: While we were building a product, our entire team met together to discuss various challenges. One developer was working on a search interface, and innocently brought up a usability problem he was tackling. This search interface showed various related fields that the user could select. (Just to describe the scenario, think of one field being the state selection and another field being the city selection. The city selection contents would change based on which state the user selected.) The developer casually mentioned that if a user selected the state and city but then changed the state, he would receive an error dialog alerting him that the city was no longer valid. One of the testers instantly became indignant and complained that she hit this issue all the time and getting an error message would drive her crazy! If she had to close an error message every time she switched the state selection, she would despise the interface and the application. The meeting came to a halt in a stunned silence.

Luckily the whole team was in attendance and that included the user experience (UX) expert. He agreed that the error dialog was a poor choice and recommended using a visual warning that the user could open or ignore. This may not seem like a big deal, but the team had already

used error dialogs in the code, so this was a change in the design pattern. But everyone liked it, and after they discussed it, it seemed like it should have been the obvious solution all along. In fact the idea was so well received that over time they updated the application to use it wherever possible.

What occurred to me while I listened was that, had this problem been raised after the entire interface was written and was in test, it would have been considered an enhancement request. As such, it would not likely have seen the light of day because it would have been too late to do anything but fix broken functionality—forget about making the application a little easier to use. The more code that is written before a good suggestion comes in, the harder it is to make a change because it affects more code and subsequently requires more testing. So the team experienced "just in time feedback," which allowed them to make the product better.

But what was so compelling is that the entire team was there. Without the whole team present and working together, this problem would have taken weeks to solve. Using email, we would likely have missed a few critical team members needed to "seal the deal." As it was, in a few hours a problem was solved, the customer got more value from the solution, and the team did not have to revisit the issue again. During the meeting, feedback was given fearlessly and feedback was taken constructively. For me, this was an early, albeit unintentional, demonstration of the power of whole teams.

Principles

What Is a Whole Team?

In the context of agile development, whole teams are teams that are both cross-component and cross-discipline teams that work together throughout a product life cycle. The whole team is responsible for the success of its work. By cross-discipline I mean a team that includes developers, testers, and user documentation professionals (writers). By cross-component I refer to teams that are responsible—not for just a single component out of a larger project—but which have the necessary expertise to work on all the affected components. The whole team concept goes beyond just team composition though: The whole team concept is a way of thinking and acting that must become the norm. In fact, until a team starts to be a true whole team, the team may experience limited success with agile and may feel continually stuck in a beginner's rut.

Being agile requires whole teams because the synergy derived from cross-disciplined and cross-component teams working together is more productive than when each discipline works in isolation and/or when components are developed in isolation. Whole teams succeed because they capitalize on the combined skills of each team member working together to accelerate their deliverables. Working cooperatively, they can leverage each other's insights, instincts, and responses to ongoing work throughout a project.

This chapter describes why whole teams experience better communication, productivity, and collective knowledge sharing than traditional silo'd teams. It makes a case for keeping teams intact as well as protecting them from interruption during a product release.[1]

1. Note that in some agile circles, the term "integrated teams" is used instead of "whole teams."

Why Are Whole Teams Hard to Create?

Traditional software development organizational structures have advocated for teams that specialize in technology and are grouped by a common discipline, for example, development, test, user documentation, and so on. The reasoning goes that teams composed of people with similar skills can help each other within their own domains. Furthermore, it is believed that teams that share a common discipline can be "time-sliced" across various projects as needed instead of focusing on one project at a time. This is the epitome of the "job-shop" mentality in which engineers just do their specific job and lose sight of the bigger picture. Unfortunately, optimizing the efficiency of a particular discipline almost always sub-optimizes the organization—a point that is often not well understood. Lean thinking in particular focuses on process throughput optimization to improve efficiency, versus individual throughput (described more in Chapter 3, "Queuing Theory").

The whole team approach advocates the idea of team members being "generalizing specialists" who have deep skills in specific disciplines, domains, and technologies but who can also work outside their area of expertise to help achieve the team's iteration goals. At first, teams shy away from this aspect of the whole team concept because they interpret it to mean that everyone on the team must do everything. In small, high-performing agile teams, this may be the case, but it gets more difficult as projects grow in size and encompass many technology domains. However, teams do not have to achieve the ideal level to become a whole team—but they should at least move in the direction of becoming "generalizing specialists."

Cross-Component Teams

Software composed of multiple architectural components is often built by separate teams that develop their respective components independently. There may be a database team, an application server team, a user interface team, and so forth. After the components are developed, the parts are put together to create the product as a whole and tested during an integration phase. Teams working on a single component find this a convenient way to work because they can easily capitalize on their common knowledge. However, the overall focus of the work becomes rather narrow, and it can be tempting to hand off responsibility for some component to a remote team, further isolating the various parts of an organization. Individual component teams are likely to work together only at the beginning of a project, during initial design, and again at the end of the development cycle when trying to integrate the components (perhaps for the first time).

Building components in isolation rarely produces the best product because the bigger picture is often lost. And although it may seem counterintuitive, building independent components in isolation is less productive than building cross-component functionality from the outset. One of the basic aspects of whole teams is that they include team members from across technology disciplines that are required to develop the software. These integrated teams work together from the beginning of the project and continue together throughout the life cycle of the project. One of the core enablers of teams working together this way is continuous integration, which enables products to be automatically built, tested, and deployed from the beginning of the project. This

practice avoids the typical expensive integration problems that arise when large amounts of code get integrated for the first time.

The emphasis on cross-component team composition also facilitates the practices of evolutionary architecture and emergent design.[2] Whole teams, using these practices start with a minimal architecture that is "good enough" to enable them to build a "thin slice" of cross-architecture/cross-component functionality in one of the earliest iterations. All the architectural details will not have been worked out yet, but the team can test the initial, basic architecture right away. This approach enables teams to validate their architecture early in the project and have greater confidence that it is a sound architecture—or adjust it as needed—which can be much easier now rather than later because the amount of work done up to this point is so small. Compare this with finding a major problem in the architecture late in a project when various components are integrated together for the first time.

Working together as a whole team iteratively on the architecture and design enables everyone on the team to understand the strengths and weakness of the software so that the team develops good instincts regarding any noted constraints. A whole team will also have a common history that enables the team to produce more consistent design patterns as more and more features are added. Furthermore, because this evolutionary approach enables real functionality to be manifested earlier, the team can demonstrate this functionality to its customers and get feedback earlier than was possible with waterfall's component-based development approach.

Cross-component, whole teams should be formed around epic stories or product features rather than technology. Agile teams, working together to build cross-component functionality from the beginning require that an end-to-end working environment be ready early in the project life cycle—that is, a continuous integration system that continually builds, validates, and deploys the product build. These are just a few of the items that need to be addressed for cross-component teams to function successfully in short iterations. Getting such an environment in place at the beginning of the project requires cross-component expertise, and this expertise can continue to be required as the project progresses and the environment requirements continue to grow along with the product.

Cross-Discipline Teams

Whole teams also include team members from each of the disciplines required to develop and successfully deliver the product. Teams should include developers, testers, and writers, and may also include a product manager, a build developer, a user interface designer, and so forth. It is important that teams have the collective skills to fill the required roles. Note, however, that a "person per role" is not required.

Teams organized by discipline often operate differently from cross-discipline teams. For instance, in waterfall, testing groups may wait until the design phase is complete before they develop a test plan, and testing itself doesn't typically begin until late in the cycle when a lot of

2. Note that we discuss evolutionary architecture and emergent design in Chapter 6, "Working Software," and Chapter 8, "Release Often" as well.

code has been written. Testers may even be prevented from joining the effort on the new release because they are finishing up leftover work from an earlier release. Getting to the point in which whole teams can begin work together on the first day of a project may take a significant amount of effort, and perhaps some behavioral changes as well. All available engineers should focus on completing any work that would prohibit them from starting together on the next release. This means getting the previous release out the door so that everyone can be available on Day 1 of the next project.

Why is it so important that a team actually start a release together? The goal for whole teams is to work together to produce a product, to learn together, and help each other get to "Done!" each iteration. With whole teams, no one gets credit for anything until the team has gotten to "Done!" Only then is any "credit" given for completing work, and it's *the team* that gets the credit. It's an all-for-one, one-for-all approach in which there's no "partial credit" and no "individual credit." For this to work, the team needs to start together so that coding, testing, user documentation, automation, defect fixing, code reviews, and so forth can all be accomplished as part of each iteration, beginning with the first one.

It is common to see the team members charged with system level testing joining an agile team well after the initial iterations have been completed. System level testing typically describes testing that covers scalability testing, failover testing, long-run testing, performance testing, and more. System testers tend to have a unique set of skills, so they tend to move from one project to another, always joining a project long after its start. When system testers actually do join a project, the rest of the team loses valuable time trying to get this set of contributors caught up and, more importantly, the project team misses the opportunity to have things such as scalability testing and performance testing executed early in the project when adjusting the code is much more manageable. If the system testers are not engaged early in the project, it is also harder for them to devise how best to test the software because much of the discussion and early decisions will not be known to them. Thus, system testers should be part of an agile whole team from the very beginning of a project.

One of the benefits of whole teams is that when the team discovers and solves problems together, every team member understands the context and history of the issues and resolutions. Together they gain a shared experience and build a knowledge foundation that will serve them throughout the entire project. As the team builds its working history, it improves its synergy, which allows the team to make decisions faster and even anticipate problems before they actually happen.

Cross-Geographical, Cross-Cultural, Large Teams

In enterprise application software projects, whole teams can consist of engineers from around the globe and from different cultures. Whole teams can work when team sizes are larger than the recommended 7 to 10 engineers. However, it should be no surprise that all the same principles still apply: These types of teams need to start a project together, plan their work together, and get to "Done!" each iteration together. The effort to create a whole team is even more critical in these situations but—we won't sugarcoat it—it is much harder.

The most difficult part of cross-geographical teams usually has to do with the need for daily communication. To establish trust and ensure rich communication channels, regular communication is an imperative. The communication needs to be open, the interaction needs to be constructive, and the focus needs to be on working together to succeed. There are a couple of ways to tackle the cross-geographical problem. The most successful way is for everyone to share a bit of the "pain." By this I mean that some team members may have to stay up late for calls with the other part of the team, or get up early, or perhaps even shift their schedules to align completely with the other part of the team. Teams can meet at different times throughout the week to "share the pain" of difficult meeting times. Some teams record their standups so that the team members separated by time-zones can understand what happened. There are many ways to make this communication successful, and teams should try a variety of mechanisms to determine what works best.

We know that this is not ideal but we encourage you to be creative, try different approaches, and keep experimenting until you find the right combination of changes that make it work for your team. However, please do not dismiss the need for regular communication (whether the team is local or geographically separated)—it's just *too important.*

There are a couple of positive benefits that cross-geographically organized teams can obtain: Teams that span time zones can coordinate work so that one part of the team can start where another part leaves off at the end of their day. In addition, teams that bring different cultural behaviors together can leverage the best from multiple working styles.

Stable, Dedicated, and Protected

Whole teams are more effective when they are committed to the project for the complete duration of the project. With this approach, teams should get new team members or lose team members only on an exceptional basis. Managers and other project leaders should do everything they can to protect teams from all interruptions so that the teams can get to "Done!" (which is very, very hard to do if management acts like a "fast-forward" button, immediately passing along every interruption to the team).

Although this probably sounds like common sense, it's not that common. Companies with multiple products typically juggle people between projects to align with changing schedules and revenue expectations. Engineers with particular knowledge or skills may get "time-shared" between various projects. People may get pulled from doing new development to manage a critical defect discovered at a customer site. And of course all teams are constantly asked to do more with less. Team churn ultimately slows down project progress. Furthermore, if team members slice their time between different projects, their lack of focus for any length of time can negatively affect their productivity and likely the quality of their work as well. The section "Maintain and Protect Dedicated Teams" discusses the evils of task-switching further, as does Chapter 4, "No Multitasking."

The Mythical Man Month by Fred Brooks[3]

Fred Brooks is a former IBMer who came out against the idea of throwing additional bodies at projects that were running behind. In his classic work titled, *The Mythical Man Month*, he discusses his Brook's Law, which reads, "Adding manpower to a late software project makes it later." My corollary to his point is, when throwing bodies at a project that is running behind, "What one engineer can do in one week, two engineers can do in two weeks." ☺

Expecting team members to work on multiple projects may feel like an efficient use of personnel, but it is likely not delivering the best product in the most efficient way. "Protect the team"—this is a cornerstone principle of whole teams, thus management must have as a primary goal protecting the team. Protecting the team means that the team should be shielded from distractions to meet its project goals.

Rotating people between agile teams is a common practice in some organizations. An individual may have a particular expertise that makes him valuable to multiple teams. There are three primary problems with moving an expert around, however. First, if an expert is moved out of an agile team for an iteration, the work the expert would normally have contributed will not be picked up by the rest of the team, thus putting the team behind. Second, if an expert spends limited time on a project, he will not have the shared history enabling him to make well-informed decisions regarding his own contributions. For example, he will have to learn what technology has been developed so that he can determine what to test. This will demand valuable time from the rest of the team. Finally, it will be difficult for others to learn from the expert due to the limited time the expert has available by virtue of being pulled in multiple directions at once.

One development project that I managed included new features that required aggressive performance tuning. Our company had a skilled team to measure product performance and make tuning suggestions, but that team had little time for my project until late in the schedule. We decided to bring the performance testing work inside the scope of our team and get some of our team members skilled with the performance testing tools. This way we could test and tune early in the project as the features were developed. Educating the developers, creating the environments to test, and managing the results took extra time initially for the project. However, by doing this, we had performance information in one of the earliest iterations, and the data showed that we had problems. The team stopped further feature work until it understood and fixed the performance problems. This behavior continued throughout the project, and by the middle of the project, the team felt confident that it had discovered and fixed the most critical performance issues. Furthermore it concluded that had the problems been found later in the cycle, the root causes and the solutions would have been harder to identify. The actual fixes also would have

3. Brooks, Frederick P., *The Mythical Man Month*. Boston: Addison-Wesley, 1995: page 232. Print. This is a book that should be on every software engineer's shelf.

been difficult and time-consuming to implement. This is also an example of "stop the line" thinking, which we address more in-depth in Chapter 9, "Stop the Line."

High-performing agile teams have dedicated members and are protected by their management from interruptions and from constant changes in team membership.

Practices

Start with Whole Teams

Agile teams are whole teams. Whole teams are created at the beginning of a project and stay together throughout the project. Whole teams consist of everyone needed to complete the planned product deliverables.

Team members each have unique skills and expertise that they use to contribute to the overall effort, but they are also expected to cross outside of their skill boundaries when required to help the rest of the team reach the iteration goals. Teams new to agile often respond to this notion with anxiety. "We cannot afford to have developers test." "I cannot learn how to develop." "I am not a good writer." Team productivity will obviously increase if the entire team could do any of the tasks; however, this is not the situation for many teams. Getting to the point in which team members can contribute outside of their particular skill area takes time for most teams to achieve, and some teams never achieve the ideal given the size of their projects. Nevertheless, becoming a whole team requires team members who are willing to jump in and help in any way they can and learn new skills as they go. This might mean helping to set up computers, looking over someone's shoulder to help find bugs, writing draft documentation, and so forth. But it does not necessarily require that everyone on the team learn the most complex aspects of each job, such as kernel-level debugging, the use of automated testing tools, or the usage of unique product documentation tools.

Teams willing to adopt a whole team approach can experience productivity growth. Furthermore, team members willing to jump into unfamiliar tasks can expand their domain knowledge and skill sets and also gain satisfaction and job growth as a consequence. And of course, they are making themselves more valuable to the team.

Maintain and Protect Dedicated Teams

Unless you've adopted the practice of self-organizing teams, it's typically management's responsibility to assign team members to a project and then protect the team. It is to management's benefit to protect the team so that the team can get its work done and ultimately deliver the product with customer value and high quality. Protecting the team means preventing any interruptions to the work the team has committed to accomplishing in its current iteration. Furthermore, management must keep each team member focused on and committed to a single project so that team members can maximize their output.

In the real world, teams have to respond to short-term crises and shifting priorities. The easiest way to manage such challenges and continue to protect teams is to keep iterations short. And by short iterations, we mean 2 weeks maximum. It is easier to protect a team during a short iteration than a longer iteration. In fact, this is one of the many reasons iterations should be short. If a critical interruption arises, and the team is halfway through a 1- or 2-week iteration, the manager can make a better case for the team being allowed to complete its work and stay productive, rather than stopping the team and getting nothing "Done!" for the iteration. It can use its next iteration to manage the crisis because the elapsed time is relatively short. Conversely, if the iteration is as long as a month, it is easy for management to believe new work can be absorbed by the team by simply having the team "work a little harder" (which, when translated into reality, implies overtime).

Highly productive agile teams may even adopt a "no iteration" methodology based on a "pull" or a "continuous flow" model. In these scenarios the team is maintaining working software even more aggressively. As teams get better at working together as a whole team, they should consider trying these lean models.

When teams get used to getting to "Done!" in short iterations, and they reliably repeat the pattern, management has a new defensive strategy: It has the data to show that protecting the team from interruption results in higher productivity. Any situations that arise that require immediate attention are added to the top of the backlog and addressed the next iteration. This should remove the typical distractions that are less urgent than maintaining progress on the project. Shorter release cycles also help protect the team. When teams deliver "just enough" capability in a short release cycle, they can better respond to new priorities that would negatively impact longer release plans.

Whole teams can get to "Done!" better by learning how to work together leveraging their unique skills. Each team is unique and develops its own team dynamics because it is made of unique people who have their own combinations of gifts and skills. Changing the team make-up regularly by moving people around disrupts this pattern and hurts a team's long-term effectiveness and predictability. Short release cycles also help enable dedicated teams because it is easier to keep a team together for shorter time periods.

The Conversation

The most critical aspect to succeeding as a whole team is effective communication. Communication is best achieved by interacting, or rather, by having conversations. Agile's primary mechanisms for this necessary communication include release planning, the daily standup, iteration planning, customer demonstrations, and reflections. There are additional ways that encourage increased communication, such as making it a team rule to help each other remove blocking issues, adopting pair programming, and team creation of user stories. Whatever the mechanism, the goal is *the conversation*—live interaction. Conversation enables interaction while ideas are fresh. It also enables collective thinking, that is, one person's input inspires another's feedback.

Furthermore, timely discussion with all the right people contributing (or maybe even just listening), results in faster progress. Working together enables a team to stay informed.

Teams often use documentation to communicate anything that needs to be referenced. For instance, engineers have a tradition of documenting designs, but often they do not review their designs with their team because it is written down—anyone can pick it up and read it. However, even the best writers often struggle to convey their point in written form.[4] So why do we expect that the written document will succeed in socializing the critical points that need to be conveyed? Many have heard the familiar quote from Blaise Pascal that goes like this, "I have made this letter longer than usual, only because I have not had time to make it shorter." Constructing a message that is easy to understand takes time. Documentation is useful and should not be abandoned, but teams need to use live discussions to make sure that they are communicating and that the information is understood.

Email threads often result in failed communication. After you send an email explaining a problem to a coworker, do you think your job is done? Have you ever received a long email thread with a message from the sender saying, "How do you want to handle this?" Do you have emails sitting in your inbox that have been there more than a week? Email is a great storage repository for unfinished business. Find a way to get critical issues out of email and get them handled.

Scott has a great saying, "If a picture is worth a thousand words, then a conversation is worth a thousand emails." Live discussions are necessary to get work completed in a timely fasion. Improving the communication at the daily standup is a great place to start. But it is hard to enable the teaming on work if you do not share enough descriptive information in the standup. Common—but frustrating—daily standup meeting comments we have heard are, "I am doing the same thing that I did yesterday," "I am still testing," and so on. This input is practically useless. The reason that we go through the typical standup protocol, "What I did since the last standup meeting," "What I am going to do next," and "What blocking issues do I have?" is to provide the opportunity to communicate about our work and help each other.

The same conclusions apply to teams that cross geographies and cultures. Everything that is important for local teams is also critical for cross-geographical teams, but communication alone may determine success or failure.

To succeed in agile, and even in your career, learn to communicate and learn to communicate well. Engage in live discussions, be willing to listen to feedback, be informative—but to the point—in your delivery, and most of all, keep trying to communicate better.

4. To drive this point home, Scott and I would much rather be having a live conversation with you about the topics covered in this book rather than simply having you read this book. We could cover the material in less time than it would take you to read the book and in a more comprehensive way. Additionally, you'd have the opportunity to ask questions of us and gain additional clarifications that you obviously aren't able to do by just reading the book.

Share the Same Truth

For any team to succeed, sharing the same priorities and the same information is critical because each team member must continually make decisions. The basis for decision making for the team has to be rooted in the same priorities using all the information available. A prioritized backlog of work provides a mechanism to ensure team synchronicity, and good tooling can provide the mechanism for sharing the same information.

One of the most significant tools for keeping teams coordinated is a team dashboard or an information radiator. Good dashboard products provide a flexible way to create widgets that can integrate live data from external tools (such as separate source code management systems, separate test case repositories, separate build environments, and so on). Providing a single view of commonly viewed data gives teams a real-time view of the same "truth," such as how many defects are active, where the latest build is deployed, what delta functionality is in the latest build, and how far along the team is on this iteration's user stories. Several years ago, a team that I was managing—a large team that spanned the globe—started to use such a tool to coordinate our work. We quit using most other mechanisms of reporting information because our online dashboard solution encompassed all aspects of the project and did so in real time. Because the data that was displayed was live data, it was never out of date.

This dashboard capability moved our large team a lot closer to succeeding as a whole team. The Agile Manifesto calls for interaction over tools and processes, but do not discount the power of excellent tooling to help enable necessary team interaction and communication.

No Partial Credit

Whole teams get credit for the work they complete each iteration. Individuals on the team do not get partial credit for the work they accomplish during the iteration. Team members may give each other "high fives" for finishing work as they go, but agile works when teams embody the whole team spirit and succeed or fail together.

Encourage your team to avoid this kind of thinking: "I have my code written and now it's up to the testers to finish testing while I move on to something else." Encourage them to think instead about how to help the testers finish their work so that the team can move on together. Staying tightly coordinated to finish work enables teams to be more productive and achieve working software every iteration. The notion of "no partial credit" drives this point home. Working software is the team's measure of progress. Having working software every iteration requires coordinated work from the whole team. They all get credit for achieving the goal.

This whole team "full credit and no partial credit" concept flies in the face of most companies' Human Resource (HR) practice of reviewing and rewarding individuals for their accomplishments. To enable whole teams to succeed in such an HR environment, managers have to strike a balance between encouraging individuals to grow their own skills and encouraging individuals to be successful team players who encourage and help each other. Many of the best sports teams have demonstrated repeatedly that teams that work well together make it to the championships. Teams dominated by one or two individuals ultimately lose because the burden is too great for just one player or two players. Whole teams leverage the valuable skills of all the players.

Team members need to set their goals around personal growth areas as well as team growth areas. These goals can naturally work together. To get to "Done!" as a team, each person should contribute to the effort using his strongest skills. But to make the team better, team members need to learn new skills so that they can help each other. When team members can help each other, they can better maintain working software, which is *the* measure of progress.

Offer Help

One critical aspect of whole team success is offering help. Each team member should offer to help other team members whenever needed. This may seem obvious, but it is contrary to mandates commonly practiced by development teams. Typically, individuals are instructed to get their own work completed, grow their own skills, and achieve individual feats to differentiate themselves from their peers. However, this focus on the individual has to be paired with the practice of individuals extending their time and skills to help others on the team. Teams that emerge from a traditional development culture may be unaware that their culture is not transforming successfully. They need to pay close attention to warning signs that they might be in trouble. For instance, *silence* should be painful if no one offers to help when a blocking issue is raised during a standup meeting. A whole team culture requires that team members get into the habit of offering help, even if they have to learn something new.

Every team has a go-to person. People usually figure out who it is and start to rely on her. The go-to person knows what the team is doing, how the code works, what the biggest issue is right now, and so forth. If that go-to person does not know the needed information, she will often start looking for it before you even finish asking. Agile's emphasis on whole teams working together should inspire a whole team of go-to people. Go-to people are not afraid to stretch beyond their job descriptions; in fact they enjoy learning and helping teams to work together well. You may be afraid that you do not have the personality for this, but those that try tend to experience additional job satisfaction, which improves their contributions significantly.

Metrics

To validate that you have a whole team, track the team membership each iteration, beginning with the first iteration, and review it regularly. Confirm that the whole team starts together and finishes together. Yes, this may seem like a relatively simple (or even simplistic) thing to track, but if the goal is to have stable, dedicated teams, then having an indicator immediately available to confirm that this happens can be a big incentive to actually having stable, dedicated teams—especially if you need to convince others in the organization that problems with team stability are real.

As another simple metric, make a rough estimate for the time required to coordinate information across the team at the beginning of a release. Find a mechanism to share the same truth with the entire team. Use a dashboard, a common whiteboard, a wiki, an information radiator, or whatever works. Use that mechanism as a way to share information during the daily standup.

Re-evaluate the time required to coordinate information across the team at the end of the several iterations. Compare the results. If significant progress was not made, or rather the time required for coordination did not decrease, try a new mechanism.

Breakthrough

Early in my experience with agile, a development team that I was managing was lucky enough to make a breakthrough with bullpens that transformed its behavior almost immediately. In fact, this breakthrough was so profound for the team that it changed my reflexive, waterfall-oriented thinking permanently. The team I was managing was having communication problems. A project consultant, Stan Rifkin, suggested we try bullpens to fix these challenges.

But let me back up and describe how we got to this recommendation. The first development team I managed that adopted agile got all the basics down quickly. Our daily standups became essential; we defined the work we planned to complete each iteration; and we demonstrated new functionality to our customers almost weekly. Despite our early success with many of the typical agile techniques, we were not working well as a whole team. Several developers would talk to each other in the hall, decide on a new development strategy, and then forget to tell the testers. An email discussing changes to how the product could be customized would go out to a part of the team, but the writers would be left off the email thread. Testers complained among themselves about the usability problems but did not bring them to the rest of the team. Tempers were mounting and disgruntled team members visited me saying, "He said this; she said that."

I took a piece of advice Mary Poppendieck gave me during a conversation about "stop the line" thinking and identified just one problem to fix during each iteration. If we failed to find a solution to the problem, then we worked on the same problem the next iteration, and the next, and so on, until it was fixed. The team agreed that it was not a whole team. We tried everything we could think of but nothing worked. Over time the team was feeling demoralized, members were banding together to place blame, and the whole agile effort was losing ground.

Luckily we got consulting help from Stan Rifkin, a consultant with 40 years' experience in data processing, management consulting, software engineering, and computer science. We reviewed our practices and issues with him. He suggested that we try bullpens. He explained that bullpens were multihour working meetings with the emphasis on *working*. Instead of meeting to discuss status, or to do planning, teams do real work together. Here are the rules for bullpens:

- Everyone on the agile team MUST attend and pay attention.
- Do real work together.

At first we struggled with what this actually meant. Do we write code together, do we test together, do we review documents together? In the end, it did not matter as long as we worked together and solved problems.

Interestingly enough, the majority of the team had a similar response to the suggestion, "No way!" (although a few people said, "Sounds great, let's try it"). After listening to all the

reasons we should not do it for more time than I can stand to remember, I pulled management rank and said, "We will try it beginning tomorrow." The team grumbled off and planned for their first 2-hour bullpen.

The team floundered a bit, wondering how to start, but a tester broke the silence by talking about a set of tests that were failing. That quickly led to discussions on how the tests were run, disagreements about the expected behavior, and candid feedback about how the functionality could be more usable. With that, the team was off and running.

The first bullpen worked well. Problems were solved in real time, and the entire team learned what was going on at the same time. The team proceeded to schedule multiple bullpens weekly. Because the entire team was not in the same room, let alone the same country, we had to do this over the phone and with an emeeting—sort of "virtually sitting next to each other." Adopting bullpens had the added benefit that when the team was not in a bullpen, it could get more focused time to work because most of the issues requiring cross-team communication had already been addressed.

Bullpens became so successful that the team started to use them for a variety of purposes. They had bullpens to jointly review code, discuss and solve difficult technology problems, code bash as a team, and more. We observed these positive outcomes: First, the team became a whole team. Accusations ceased and productivity increased. Second, the amount of email in my inbox dropped in size significantly. The team solved problems in real time with the whole team involved, so no additional communication was required.

Small agile teams that work in the same room usually develop a bullpen behavior simply based on their proximity. These may be called team rooms. Teams not residing in the same location can get similar value from bullpens. One advantage of bullpens over team rooms is that with bullpens, the rule is that everyone must attend, and it also leaves nonbullpen time to get focused work done that is less likely to be interrupted.

The story at the beginning of this chapter about the user interface design point that the tester disliked resulted from one of the first bullpens that I attended. The success of the meeting was stunning to everyone in the meeting. It forever changed our team dynamics and we were hooked.

When a team becomes a *whole team*, the team starts to think in a more agile way. When teams share the ownership of problems and their resolution, they can move faster together, leveraging everyone's collective strengths. Most whole teams have more fun working this way and certainly experience higher productivity.

Summary

Being agile requires whole teams because the synergy derived from cross-disciplined and cross-component teams working together enables teams to be more productive than working in isolation.

- Whole teams accelerate delivery and increase team capacity by working closely together to leverage each other's unique skills.

- Agile advocates the idea of "generalizing specialists" who can work outside their regular domains when required to achieve iteration goals.
- Whole teams work across the architecture stack to deliver end-to-end functionality from the beginning of the project and develop a common history so that consistent design patterns emerge.
- Whole teams start the project together; they get to "Done!" each iteration together; and they succeed or fail together. Whole teams may span the globe and cross cultures.
- Management must protect its teams from interruptions to enable them to deliver value to customers sooner.
- Regular, open, and constructive communication is critical to whole team success.
- Dashboards or information radiators enable teams to share the "same truth."
- Aspire to be a go-to team member.
- Bullpens are a mechanism that breaks teams out of silo'd behaviors and enables them to become whole teams. The basic rules of bullpens are 1. Everyone on the agile team MUST attend and pay attention and 2. Do real work together.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z