

Practices for Scaling Lean & Agile Development

*Large, Multisite, and Offshore
Product Development
with Large-Scale Scrum*

Craig Larman
Bas Vodde

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

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.

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

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

For sales outside of the U.S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Larman, Craig.

Practices for scaling lean & agile development : large, multisite, and offshore product development with large-scale Scrum / Craig Larman, Bas Vodde.

p. cm.

Includes bibliographical references and index.

ISBN 0-321-63640-6 (pbk. : alk. paper)

1. Agile software development. 2. Scrum (Computer software development)
I. Vodde, Bas. II. Title.

QA76.76.D47L3926 2010
005.1—dc22

2009045495

Copyright © 2010 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. 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. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-63640-9

ISBN-10: 0-321-63640-6

Text printed in the United States at Courier in Westford, Massachusetts.

First printing, January 2010

To our clients, and my friend and co-author Bas

To Lü Yi, Tero Peltola, and the little one

This page intentionally left blank

CONTENTS

- 1 **Introduction** 1
- 2 **Large-Scale Scrum** 9
- Action Tools**
- 3 **Test** 23
- 4 **Product Management** 99
- 5 **Planning** 155
- 6 **Coordination** 189
- 7 **Requirements & PBIs** 215
- 8 **Design & Architecture** 281
- 9 **Legacy Code** 333
- 10 **Continuous Integration** 351
- 11 **Inspect & Adapt** 373
- 12 **Multisite** 413
- 13 **Offshore** 445
- 14 **Contracts** 499
- Miscellany**
- 15 **Feature Team Primer** 549
- Recommended Readings** 559
- Bibliography** 565
- List of Experiments** 580
- Index** 589

This page intentionally left blank

PREFACE

Thank you for reading this book! We've tried to make it practical. Some related articles and pointers are at www.craiglarman.com and www.odd-e.com. Please contact us for questions.

Typographic Conventions

Basic point of emphasis or Book Title or minor new term. A **noticeable point of emphasis**. A **major new term** in a sentence. [Bob67] is a reference in the bibliography.

About the Authors

Craig Larman has served as chief scientist at Valtech, an outsourcing and consulting group with a division in Bangalore that applies Scrum, where he and colleagues created agile offshore development while living in India and also working in China. Craig was the creator and lead coach for the lean software development initiative at Xerox, in addition to consulting and coaching on large-scale agile and lean adoptions over several years at Nokia Networks, Schlumberger, Siemens, UBS, and other clients. Originally from Canada, he has lived off and on in India since 1978. Craig is the author of *Agile and Iterative Development: A Manager's Guide* and *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design & Iterative Development*.

After a failed career as a wandering street musician, he built systems in APL and 4GLs in the 1970s. Starting in the early 1980s he became interested in artificial intelligence (having little of his own). Craig has a B.S. and M.S. in computer science from beautiful Simon Fraser University in Vancouver, Canada.

Along with Bas Vodde, he is also co-author of the companion book *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*.

Bas Vodde works as a product-development consultant and large-scale Scrum coach for Odd-e, a small coaching company based in Singapore. Originally Bas is from Holland, and before settling in Singapore, he lived and worked in Helsinki (Finland) and Beijing and Hangzhou (China). Much of his recent work is in Asian coun-

tries—especially China, Japan, India, the Philippines, and Singapore—applying agile principles to offshore and multisite development. For several years he led the agile and Scrum enterprise-wide adoption initiative at Nokia Networks. He has been a member of the leadership team of a very large multisite product group adopting Scrum. Bas has worked as developer/architect in multimedia/real-time graphics product development and in embedded telecommunication systems. He is co-author of the CppUTest unit-test framework for C/C++ and still spends some time programming, and coaching agile-development practices such as refactoring and test-driven development.

Bas rushed through his B.S. in computer science so that he could write *real* software. He has been waiting for some university to give him an honorary Ph.D. but is afraid he will actually have to work for it. He is a passionate book collector—especially historical books related to product development and management.

Acknowledgments

Many thanks for the contributions and reviews from...

Peter Alfvín, Bruce Anderson, Brad Appleton, Tom Arbogast, Alan Atlas, James Bach, Sujatha Balakrishnan, Gabrielle Benefield, Bjarte Bogsnes, Mike Bria, Larry Cai, Olivier Cavrel, Pekka Clärk, Mike Cohn, Lissa Crispin, Ward Cunningham, Pete Deemer, Esther Derby, Jutta Eckstein, Janet Gregory, James Grenning, Elisabeth Hendrickson, Kenji Hiranabe, Greg Hutchings, Michael James, Clinton Keith, Joshua Kerievsky, Janne Kohvakka (and team), Venkatesh Krishnamurthy, Shiv Kumar MN, Kuroiwa-san, Diana Larsen, Timo Leppänen, Eric Lindley, Steven Mak, Shiva-kumar Manjunathaswamy, Brian Marick, Bob Martin, Gregory Melnik, Emerson Mills, John Nolan, Roman Pichler, Mary Poppendieck, Tom Poppendieck, Jukka Savela, Ken Schwaber, Annapoorani Shanmugam, James Shore, Maarten Smeets, Jeff Sutherland, Dave Thomas, Ville Valtonen, and Xu Yi.

Current and past Flexible company team members (and reviewers), including Kati Vilki, Petri Haapio, Lasse Koskela, Paul Nagy, Ran Nyman, Joonas Reynders, Gabor Gunyho, Sami Lilja, and Ari Tikka. Current and past IPA LT members (and reviewers), especially Tero Peltola and Lü Yi.

Bas thanks the support of Sun Yuan through another year of writing and traveling. Without her support there would be no book. And thanks Craig for tolerating all the discussion and feedback and... more debugging of Bas's writing. No more "rubber chicken" on this book, what's next?

Craig thanks Albertina Lourenci for the healthy food so that he could write well-nourished, and Tom Gilb for his apartment in London so he could write well-sheltered.

Thanks to Louisa Adair, Raina Chrobak, Chris Guzikowski, Mary Lou Nohr, and Elizabeth Ryan for publication support.

(An Early) Colophon

Layout composed with FrameMaker, diagrams with Omnigraffle.

Main body font is *New Century Schoolbook*, designed by David Berlow in 1979, as a variant of the classic *Century Schoolbook* created by Morris Benton in 1919—familiar to most North Americans as the font they learned to read by, and from the font family required for all briefs submitted to the Supreme Court of the USA.

Chapter

- Thinking & Organizational Tools 2
- No False Dichotomy: These are only Experiments 2
- No Best Practices—and no Fractal Practices 4
- Limitations 5
- Onwards 6

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

INTRODUCTION

*Nobody will ever win the battle of the sexes.
There's too much fraternizing with the enemy.*
—Henry Kissinger

The earliest large-scale software-intensive product development was the Semi-Automatic Ground Environment (SAGE) system; created in the 1950s, it involved hundreds of people.¹ In retrospect, what did a senior manager think of the development strategy?

One of the directors of SAGE was discussing why the programming had gotten out of hand. He was then asked, “If you had it to do all over again, what would you do differently?” His answer was to “find the ten best people and write the entire thing themselves.” [Horowitz74] (emphasis added)

This echoes the opening suggestion in the companion book.² Build ‘big’ systems by building a small group of great people that can work in teams, and co-locate them in one place. Only grow when it really hurts, taking the time to hire extraordinary new talent.

But, we know that especially in existing large companies and product groups, that is not going to happen—at least not soon. People are *still* going to do large-group, multisite, or offshore development—usually based on beliefs such as “big needs big” or that offshoring is better value. Rather than debate if so many people are needed, we try to support people to improve their development with agile and lean principles so that at some point it becomes clear to the group that they have too many people in too many places.

-
1. It was way over budget and partly outdated when finally delivered.
 2. The companion book is *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*.

THINKING & ORGANIZATIONAL TOOLS

This book focuses on practices or *action tools*. To be effective and take root, these seeds are best cast on fertile ground—and that is the soil of *thinking* and *organizational-design* tools covered in the companion book. It takes an understanding of *systems thinking*, *queueing theory*, *feature teams*, *requirement areas*, the impact of organizational policies (such as incentives and budgeting), and more, for these practices to flower into a beautiful environment.

Without that foundation, what is likely is ritualistic application of shallow practices—a *cargo-cult*³ adoption—causing disruption with little benefit, the belief that “the adoption is finished,” and the impression that all this is just another management fad.

And then eventually... “Agile doesn’t work here. Let’s try X.” Where X is PMI certification, kanban, CMMI, next-generation lean, ...

The good news is that with a little investment in learning and redesign, these action tools have a powerful positive impact. Some years ago, a well-respected manager at a group we had started coaching sent us an email: “We actually tried *everything* you suggested. It worked!” As consultants and coaches it is a great joy for us to hear this (recognizing it is the *ideas*, not us, that help)—and to see people delighted by tangible improvement and enjoying their work more.

NO FALSE DICHOTOMY: THESE ARE ONLY EXPERIMENTS

A key chapter in the companion book was *False Dichotomies*. It emphasized that right/wrong dichotomies are ill-advised with respect to practices. Practices are context dependent; as such, we are not prescribing what to do. For example, on balance, *feature teams*

3. A *cargo cult* is a religious practice in a (relatively) primitive society that attempts to get the same wealth (the cargo) of a technically advanced society through ritualistic practices that superficially reflect the behaviors of the advanced group. “Cargo-cult process adoption” is a term suggesting shallow adoption of practices but not the deeper intention or principles. For example, holding a daily Scrum meeting to report status to a manager.

usually have more pluses than minuses and they deliver value faster, but we know of organizations where—at this phase in their adoption and in the context of the particular people, learning challenges, and politics—some component teams are still needed.

Yet, on the other hand, there is the “Avoid...Being agile/lean without agile/lean practices/tools” section on page 379. It is easy to misuse the recognition that practices are contextual as an excuse for not changing. We meet groups that say, “Oh, we are unique⁴ so we don’t do that—practices are valid only in a context.”

False Dichotomies is also so named because adopting practices does not have to be framed as a binary choice of accept/reject. Adoption can be along a *continuum* from less to more. For instance, organizations can have both some feature teams and some component teams—and their ratio may shift over time.

Watch out for false-dichotomy thinking and speaking; *computer people*—and that includes us—can get a little too *binary*.

And more broadly, both Scrum and lean thinking encourage inspect and adapt, and kaizen mindset, rather than formulas or cookbook recipes for workplace practices and processes.

All that said, we do have opinions based on experience of what is worth considering for a trial to improve. Therefore, the tools in both books are presented as a series of *experiments* that start with *Try...* or *Avoid...* to suggest only experiments—nothing more. As a suggestion, *Avoid...X* means “experiment with shying away from X and observe what happens.” It does not mean “never do X.”

Another implication of these experiments is that they can be useful for a while, but then dropped if they limit further improvement.

4. It is singularly noteworthy how many groups claim *uniqueness*—and yet how similar they are in terms of symptoms and causes!

NO BEST PRACTICES—AND NO FRACTAL PRACTICES

A variant of false-dichotomy thinking is the notion of “best practice.” But in research and development (R&D)...

There are no *best* practices—only adequate practices in context.

In a review of R&D practices and outcomes, *Organization of Science and Technology at the Watershed* [RS98], the editors conclude:

...there is no best practice [in R&D], since the use of tools depends on the specific context and situation of the enterprise.

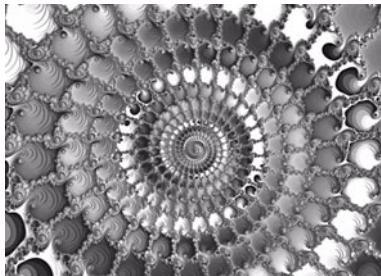
In *Managing the Design Factory*, a similar point is made:

...the idea of best practices is a seductive but dangerous trap. ... The great danger in “best practices” is that the practice can get disconnected from its intent and its context and may acquire a ritual significance that is unrelated to its original purpose. [Reinertsen97]

Since so-called best practices are ‘best,’ they also inhibit a “challenge everything” culture and continuous improvement—a pillar of lean thinking. Why would people challenge ‘best?’ Mary Poppendieck, co-author of *Lean Software Development*, reiterates this point and draws the historical connection from best practices to *Taylorism*:

*Frederick Winslow Taylor wrote “The Principles of Scientific Management” in 1911. In it, he proposed that manufacturing should be broken down into very small steps, and then industrial engineers should determine the ‘one best way’ to do each step. This ushered in the era of mass production, with ‘experts’ telling workers the ‘one best way’ to do their jobs. The Toyota Production System is founded on the principles of the Scientific Method, instead of Scientific Management. The idea is that no matter how good a process is, it can always be improved, and that the workers doing the job are the best people to figure out how to do it better... Moreover, even where a practice does apply, it can and should always be improved upon. **There are cer-***

tainly underlying principles that do not change. These principles will develop into different practices in different domains... [Poppendieck04]



This last emphasized point raises a connection to this book's cover by the fractal-artist Ken Chil-dress.

All our cover art symbolizes some point—usually unexplained. Yet this book's *fractal* cover art could be misinterpreted and warrants clarification: It hints at a creative tension: that *principles* scale “self similar” or fractally, but practices and processes may not—they are context sensitive. “Fractal practices” that apply at all scales has a seductively neat charm to it—compelling to those that yearn for simple solutions for complex problems.

similar” or fractally, but practices and processes may not—they are context sensitive. “Fractal practices” that apply at all scales has a seductively neat charm to it—compelling to those that yearn for simple solutions for complex problems.

Consider the daily Scrum meeting: Answering the three questions is an excellent way to share information needed for a *team* to take a *shared responsibility* and *manage themselves* in a complex environment. But when you do not have a team-shared responsibility (common in a “higher level” organizational group), will that practice still be useful? Perhaps...and perhaps not.

The lean *principle* of continuous improvement, and the Scrum principles of transparency, and frequent inspection and adaptation, these—perhaps—scale “fractaled up” from one person to larger systems. *Situational-appropriate practices can be generated consistent with fractal principles*, but what is practiced for one team may not work at the level of the enterprise.

Do not assume the executive group need a *Sprint Backlog*.

LIMITATIONS

We visited a client, sharing our experiences and coaching. When we left, the client thanked us and said they had learned a lot. We

thanked them in return and said we also learned a lot. They responded quite surprised...not realizing that we learn something new every day and every time we work with large product groups.

There is still much for us to learn in these areas of large, multisite, and offshore product creation and delivery. We welcome other stories, insights, and advice from our readers, especially in the areas where you (the reader) feel that we were limited.

Some experiments in this book are relevant to general-purpose product development, but most of our experience is in software-intensive products that include specialized hardware, including factory automation, ship control systems, printers, and telecom equipment. Consequently, the bias is toward software-oriented practices that may help large-scale agile or lean development.

Finally, we apologize that we are not skilled enough to make this book about big development...smaller.

ONWARDS

The last major chapter in the companion book was *Large-Scale Scrum*. This is a bridging chapter that connects both books. So, we start with a review of frameworks for Scrum when scaling...

This page intentionally left blank

Chapter

- Frameworks for Scaling 10
- Try...Large-scale Scrum FW-1 for up to ten teams 10
- Try...Large-scale Scrum FW-2 for 'many' teams 15

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

LARGE-SCALE SCRUM

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.
—Bertrand Russell

(An expanded version of this was also the last chapter in the companion book. It is a bridge that connects both books.)

Large-scale Scrum is Scrum.

It is not “new and improved Scrum.” Rather, it is regular Scrum, an empirical process framework that within an organization can inspect and adapt to work in a group small or large. **Large-scale Scrum** is a label—for brevity in writing—to imply regular Scrum plus the set of tips that we have experienced and seen work in large multiteam, multisite, and offshore agile development. These are experiments to...experiment with, in the context of the classic Scrum framework.

Be dubious of messages such as “Scrum 2.0,” “Scrum++,” “Scrum#,” “UnifiedScrum,” “OpenScrum,” or “new and improved Scrum that should replace regular Scrum.” They may miss the point of empirical process¹ and the implications of Scrum. To quote Ken Schwaber, the co-creator of Scrum:

There will be no Scrum Release 2.0...Why not? Because the point of Scrum is not to solve [specific problems of development]... Scrum unearths the problems caused by the complexity and lets the organization solve them, one by one, over and over again. [Schwaber07b]

1. Based on transparency, inspection, and adaptation.

Regular Scrum is a simple framework that exposes problems. It is a mirror. We are not suggesting that new ideas cannot arise and improve the framework. But attempts to ‘improve’ it are most often (1) avoidance of dealing with the weaknesses exposed when regular Scrum is really applied, (2) conformance to status quo policies or entrenched groups, (3) belief in a new silver bullet practice or tool, (4) fuzzy understanding of Scrum and empirical process control, or (5) an attempt by the traditional consulting companies to sell you a process—“Accenture Scrum/Agile,” “IBM Scrum/Agile,” and so on.

See “Try...Lower the waters in the lake” on p. 407.

Large-scale Scrum, as regular Scrum, is a framework for development in which the concrete details need to be filled in by the teams and evolved iteration by iteration, team by team. It reflects the lean thinking pillar of continuous improvement. It is a framework for inspecting and adapting the product and process when there are *many* teams.

FRAMEWORKS FOR SCALING

The following descriptions only emphasize what is noteworthy in the context of scaling. Regular Scrum elements are not explained unless we felt that reiteration was useful.²

For large-scale Scrum we suggest two alternative frameworks. One is for up to about ten teams. The other goes beyond that—scaling to at least many hundreds, if not thousands, of people.

TRY...LARGE-SCALE SCRUM FW-1 FOR UP TO TEN TEAMS

The first framework is appropriate for one (overall³) Product Owner (PO) and up to ‘ten’ teams. ‘Ten’ is not a magic number for choosing between framework-1 and framework-2. The tipping point is context

-
- 2. See the online *Scrum Primer* and *Scrum Guide* for basic concepts. Terminology point: This chapter (and book) uses *iteration* rather than *Sprint* because of the former’s familiarity and use in other iterative and agile methods.
 - 3. See “Try...Map different scaling terms” on p. 134.

dependent; sometimes less. At some point, (1) the PO can no longer grasp an overview of the entire product, (2) the PO can no longer effectively interact with the teams, (3) the PO cannot balance an external and internal focus, and (4) the Product Backlog is so large that it becomes difficult for one person to work with. When the PO is no longer able to focus on high-level product management, something should change.

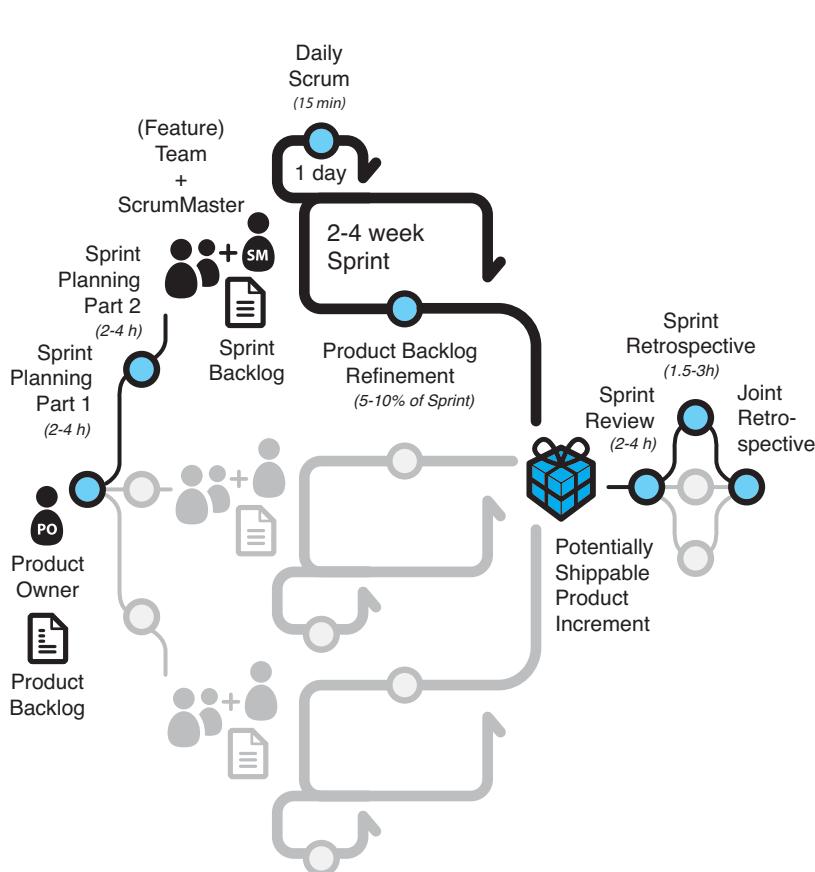


Figure 2.1 large-scale Scrum, FW-1

Before switching to framework-2, first consider if the PO can be helped by (1) delegating more work to the teams and/or (2) identifying PO representatives—who are usually within teams. Encourage teams to directly interact with real customers to reduce handoff and reduce the burden on the PO. Most project management should be

See “Try...Product Owner representative (supporting PO)” on p. 138.

done by the teams. The PO does not need to be involved in low-level details; the PO should be able to focus on true product management.

Roles

- Product Owner
- (Feature) Teams
- ScrumMasters

Product Owner—The Product Owner role and responsibilities are the same as in regular one-team classic Scrum. What are those? There is some confusion, so it may be worthwhile to review...

[The Product Owner] owns the vision for the total product portfolio, the business plan, the road map, and the dates. They are accountable for the revenue stream... [and are] business-focused on the product, so there is not a one-to-one mapping to teams.
[Sutherland08]

The Product Owner's focus is return on investment (ROI).
[Schwaber04].

The PO needs more support from the teams as the number grows.

For more on the PO role, see the *Product Management* chapter.

(Feature) Teams—These are the normal teams in Scrum that take whole customer-centric features and complete them. They are self-managing and cross-functional teams. Because they are *feature teams*, there should be a reduced need for the teams to interact or coordinate, except at the level of integration of code. And that is resolved through continuous integration; see the *Coordination* and *Continuous Integration* chapters for more. That said, multi-team requirements and design coordination are required in a large system; see the *Test, Requirements & PBIs*, and *Design & Architecture* chapter for experiments when scaling.

Teams in Scrum are explored in the *Feature Teams* and *Teams* chapters of the companion book.

For large groups that have many *component teams*, see the “Try...Transition from component to feature teams gradually” section on page 391.

ScrumMasters—These are regular ScrumMasters that (1) act as Scrum coaches for their teams and the Product Owner, (2) help their team become a *real* team by facilitating conflict resolution and removing obstacles, (3) help the Product Owner, (4) remind the team of their goal, and (5) bring change to the organization so that overall product development is optimized and maximum ROI is realized.

In the context of scaling and multiteam development, there are many opportunities for a team to require a representative at meetings. *Avoid* designating a ScrumMaster as team representative. Why? See the *Coordination* chapter for details.

Artifacts

- Product Backlog
- Sprint Backlogs
- Potentially Shippable Product Increment

Product Backlog—Some scaling discussions advise that each team have its own “Product Backlog” or “Team Product Backlog.” This is not correct. As the *Scrum Guide* [Schwaber09a] explains⁴:

Multiple Scrum Teams often work together on the same product. One Product Backlog is used to describe the upcoming work on the product. (emphasis added)

See the *Test, Requirements & PBIs*, and *Product Management* chapters for suggestions on content and priority and for analyzing and splitting large requirements.

Sprint Backlog—Each team has its own regular Sprint Backlog.

4. The *Certified ScrumMaster* course [Schwaber05] also asserts one Product Backlog for many teams.

Potentially Shippable Product Increment—One perfection challenge in Scrum is that the output of each iteration is a potentially shippable product increment. This is not a difficult goal in a small product group, but requires a multiyear journey of improvement in a gargantuan group that has institutionalized weaknesses. See the *Planning* and *Inspect & Adapt* chapters for improving the *Definition of Done* and other things over time, until it is *really* potentially shippable.

Is *test* the same in large-scale Scrum? No. Its role changes from just *verifying* to *prevention* by concurrent engineering with both acceptance- and unit- test-driven development—and that blurs the distinction between test, requirements analysis, and design, so that...testing is no longer testing. See the *Test* chapter.

Large-scale design issues for the shippable product are covered in the *Design & Architecture* chapter.

Releasing a large product is often so laborious that many special release activities are necessary; see the *Planning* chapter for more.

Note that the product increment is not per team. Rather, all teams need to integrate their output into one potentially shippable increment—within the iteration. This means the teams need to continuously integrate their code and coordinate in any other way required. These issues are explored in the *Continuous Integration* and *Coordination* chapters.

Events

- | | |
|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <input type="checkbox"/> Sprint Planning
<input type="checkbox"/> Daily Scrum
<input type="checkbox"/> Product Backlog Refinement | <input type="checkbox"/> Sprint Review
<input type="checkbox"/> Sprint Retrospectives
<input type="checkbox"/> Joint Retrospective |
|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|

Sprint Planning—For scaling, see tips in the *Planning* and *Product Management* chapters.

Daily Scrum—This is the usual Scrum event. The *Coordination* chapter has scaling-relevant experiments.

Product Backlog Refinement (also called backlog ‘grooming’ or ‘refactoring’)—This is the normal Scrum activity of refining the Product Backlog, taking five or ten percent of each iteration for the team, often in a focused workshop. See the workshop suggestions in the *Test* and *Requirements* chapters. For *initial* Product Backlog refinement, see *Planning*.

Sprint Review—See the *Inspect & Adapt* and *Coordination* chapters for experiments when scaling.

Sprint Retrospectives—Each team has its own individual retrospective. See the *Inspect & Adapt* chapter for relevant tips.

Joint Retrospective (optional but recommended)—This is useful for improving the organization *as a whole*. See the *Inspect & Adapt* chapter for more.

Other Elements

Definition of Done (DoD)—The DoD applies to all Product Backlog items for all teams. See *Planning* for DoD and *Undone Work* tips.

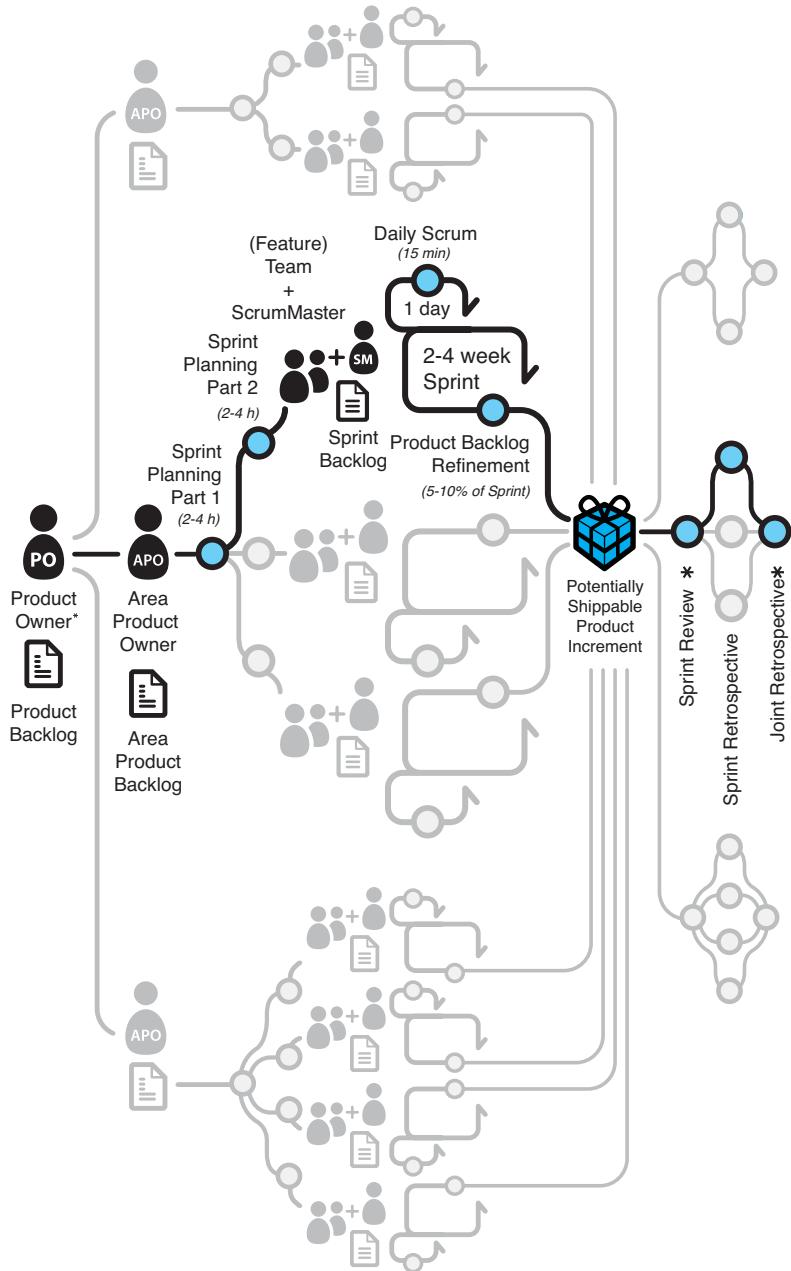
TRY...**LARGE-SCALE SCRUM FW-2 FOR ‘MANY’ TEAMS**

Large-scale Scrum framework-2 builds on—rather than replaces—framework-1. In essence, it is a set of framework-1 sub-groups.

Beyond ten teams (or even fewer), the Product Owner cannot effectively work with all the teams or all the details in the Product Backlog. At this point it is useful to identify the major **requirement areas** and then define the Product Backlog with separate *views* called **Area Backlogs**, each with its own **Area Product Owner** (APO) and its own dedicated Teams. This is explored in the *Requirement Areas* chapter of the companion book, the *Feature Teams Primer* in this, and the *Product Management* chapter.

2 — Large-Scale Scrum

Figure 2.2 large-scale Scrum FW-2



*see Feature
Teams Primer
chapter*

Consequently, framework-2 of large-scale Scrum introduces some new terms: **Area Product Owner**, **Product Owner Team** (all APOs and the Product Owner), and the **Area Backlog**. To be precise, the Area Backlog is not a separate backlog or new artifact; it is simply a *view* onto the Product Backlog for one area.

There are some changes to events in framework-2:

Sprint Planning—There is separate Sprint Planning for each requirement area. See “Try...Scaling Sprint Planning Part One” on p. 163.

Sprint Review—There is a separate Sprint Review for each area. Each involves the Area Product Owner and teams. The Product Owner may attend particular reviews that he or she is especially interested in. It is otherwise the same as framework-1.

(Joint product-level) Sprint Review (optional, recommended)—To focus on the overall product and increase visibility of overall progress, a joint Sprint Review for the entire system is possible—and recommended. See *Inspect & Adapt* for more.

Joint Retrospectives (optional but recommended)—These may happen at the area, site, and/or overall product level.

CONCLUSION

Framework-1 of large-scale Scrum involves a wide variety of practices expanded throughout this book, including experiments in *Test, Design & Architecture, Multisite*, and many other chapters.

Framework-2, for bigger groups, builds on the practices applied in framework-1 and adds *requirement areas* as the key organizational unit for larger assemblies. Framework-2 is essentially a set of many framework-1 units for each requirement area.

The remaining chapters—*Multisite, Offshore, Contracts*—provide suggestions related to these common contexts for large-scale Scrum.

All these practices build on the *thinking tools* and *organizational tools* explored in the companion.

RECOMMENDED READINGS

- The companion book, *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, focuses on foundations supporting the practices in this book.

This page intentionally left blank

Action Tools

This page intentionally left blank

Chapter

- Thinking About Testing 24
- Customer-Facing Test 42
- Developer Testing 72
- Example: Robot Framework 83

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

TEST

*Two things are infinite: the universe and human stupidity;
and I'm not sure about the universe.*

—Albert Einstein

“If we advocate *cross-functional* teams, then we ought to have a cross-functional adoption team,” said the manager of a centralized process-improvement group in a company with perhaps *twenty thousand* engineers. She put her money where her mouth is and formed a team consisting of an agile development expert, a process architect and CMMI coach, a program management adviser, a programmer, and a testing specialist who was well-versed in test automation and TPI¹ assessments.

The shift in thinking between traditional and agile development was perhaps the most difficult for the testing specialist. He knew *everything* about testing, yet when it was discussed in an agile perspective, it appeared like a foreign language to him. He *spoke* fluent test automation; nevertheless, test automation in an agile context sounded alien.

For the first few weeks, he tried to map agile concepts to his traditional frame of reference. But after a couple of months he said, “I don’t believe anymore in what I had been teaching for all those years.”

His experience is not unique. Changing to agile and lean development powerfully alters the way to *think* about testing and how to *do* testing. As a result, this chapter is one of the largest and comprises these sections:

-
1. Test Process Improvement (TPI) is a model for assessing test processes. It can also be used as a road map for improvements [KP99].

- thinking about testing
- customer-facing tests
- developer tests
- Robot Framework example

THINKING ABOUT TESTING

This section covers topics related to testing in general, such as terminology, assumptions, and organizational issues.

Avoid...Assuming *testing* means *testing*

Confused? We can imagine! The purpose of testing used to be fairly clear—“Testing is the process of executing a program with the intent of finding errors” [Meyers79]. This changes when adopting agile and lean development.

Concurrent engineering necessitates parallelizing work. Dedicated cross-functional teams encourage single-specialists to broaden their expertise. These cause the purpose of conventional development activities—such as test—to shift.

At the code level, practices such as (unit) test-driven development *blur* the division between *test* and *design* as is made explicit by agile leader Ward Cunningham’s statement:

“Test-first coding is not a testing technique.” [Beck01]

Acceptance test-driven development *fuzzes* the distinction between *test* and *requirements analysis*. In their IEEE Software article, “Test and Requirements, Requirements and Test: A Möbius strip,” Martin and Melnik argue...



... for early writing of acceptance tests as a requirements-engineering technique. We believe that concrete requirements blend with acceptance tests in much the same way as the two sides of a strip of paper become one side in a Möbius strip. In other words, requirements and tests become indistinguishable, so you can specify system behavior by writing tests and then verify that behavior by executing the tests. [MM08]

This blurring of boundaries is fraught with fallacies. Adopting (unit) test-driven development as a *testing* technique misses the point and drives superficial adoption. Likewise, we regularly need to clarify to testing groups that *they* cannot adopt acceptance test-driven development without involvement of others.

Testing is no longer testing.

Try...Challenge assumptions about testing

As touched upon, testing discussions are rife with assumptions. Challenge these! To be clear, we are not saying that these assumptions are *false*, but that leaving them unchallenged *will* limit thinking and the ability to improve. Deeply rooted in the Toyota culture is a pillar of the Toyota way: *continuous improvement* by challenging everything. Taiichi Ohno (a founder of lean thinking) said:

see Lean Thinking in the companion book

If you're going to do kaizen continuously... you've got to assume that things are a mess. Too many people assume that things are all right the way they are... Kaizen is about changing the way things are. If you assume that things are all right the way they are, you can't do kaizen. So change something!

What assumptions? Some of the beliefs we *bump into*:

- testing must be independent and thus separated from development
- testing cannot start before coding is finished
- testing follows the sequence of (1) test case design, (2) test case execution, (3) test case reporting (a test waterfall)
- there must be a separate test department
- there must be a *test manager*
- testing must be done at the end
- testing must be “well planned”
- there must be a “testing strategy” and a “master test plan”
- 100% coverage is too expensive
- 100% test automation is too expensive
- testing requires a sophisticated test-management tool
- testing must be done by ‘testers’

Leaving these assumptions unchallenged retains *in-the-box* thinking. As long as you believe “Testing can only start after coding is finished,” you will never consider innovative ways of doing testing earlier. But once you are conscious of your assumption, you can question it and ask, “Is there any way I could work differently so that testing starts before coding is finished?”

Avoid...Complex testing terminology

A question we enjoy asking big product groups is, “What do you all need to do before you can ship your product?”²

Long ago, we learned that we need two columns: one for ‘normal’ activities, and a larger one for *testing* activities. The first column fills up with items such as coding, creating the users documentation, developing hardware, pricing, and training sales personnel. The sec-

2. Or, “What does potentially shippable mean?” since the outcome of every Scrum iteration is called a *potentially shippable product increment*.

ond column comprises test *activities*, test *levels*, or test *classifications*. Common entries in the second column are shown below:

unit test	module test	developer test
functional test	system test	integration test
stress test	stability test	regression test
interoperability test	compatibility test	reliability test
load test	traffic test	performance test
installation test	security test	capacity test
monkey test	exploratory test	usability test
documentation test	acceptance test	user-acceptance test

Elaborate terminology is not harmful by itself but it often leads to test-level specialists located in test-specialist departments. For example, the integration-test specialists in the integration team, and the performance-testing specialist in the performance-testing team. These specialist groups cause organizational constraints and department suboptimizations.

See “Avoid...A complex requirements meta-model” on p. 233.

Of course, all of these tests are *probably* compulsory, but complicated classification is occasionally confused with comprehension and capability. As Nobel Prize winner Richard Feynman observed:

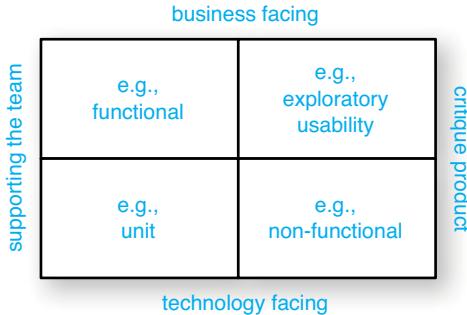
*You can know the name of a bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird... So let's look at the bird and see what it's doing—that's what counts. I learned very early the difference between **knowing the name of something** and knowing something.*

Try...Simple testing classifications

Straightforward terminology inspires intelligent behavior. Brian Marick, an Agile Manifesto author and testing authority, created the simple test categorization shown in Figure 3.1 [Marick03].³

3. Variants exist in most agile testing-related literature [CG09, Poppdeck06]; [Meszaros07] extends it to six categories.

Figure 3.1 Marick's test categories



Marick defines two dimensions:

- *technology versus business facing*—tests done from end-user perspective are *business facing*, whereas tests concerning the implementation are *technology facing*.
- *supporting the team versus critiquing the product*—tests that aid the development by, for example, discovering the requirements or driving the design are *supporting the team*, whereas tests done with the conventional purpose of breaking the system are *critiquing the product*.

These two dimensions lead to four quadrants (see Figure 3.1); we added an *example* in each quadrant. The quadrants are useful for *thinking about testing* because each quadrant has a distinct purpose and characteristic. For example, technology-facing tests that support the team are normally done by programmers during coding, while customer-facing tests that critique the product are usually done by a person other than the original author and are executed right after some user-functionality is implemented.

Our classification is even simpler! Two groups:

- developer test
- customer-facing test

Developer test—these are usually created by the person who is implementing. The purpose is to check whether the code is doing what the programmer wants. If the tests pass, it means that the sys-

tem does what the developer intended—but this does not necessarily mean it does what the customers wants.

Customer-facing tests—these test whether the requirements are fulfilled. They are frequently implemented and executed by a person other than the one who wrote the code. In this grouping, non-functional tests are classified as customer-facing tests because non-functional requirements for large systems are typically explicit and the most important.

Avoid...Separating development and testing

Bill Hetzel, the organizer of the first software testing conference,⁴ defined in *The Complete Guide to Software Testing* six principles of testing. The sixth principles—test independence—is a common theme throughout the history of software testing. Glenford Meyers, author of the first book⁵ on software testing, stressed the independence of testing in *Software Reliability*:

Testing should always be done by an outside party who is somewhat detached from the program and project... System testing should always be done by an independent group such as a separate quality-assurance department. [Meyers76]

Why is separation important? Some frequently stated arguments:

- Programming is constructive whereas testing is destructive—thus, programmers cannot test.
- If programmers test their own code, then they will change the test according to the implementation.
- When testing is done by the same group as implementation, then they can meet their deadline by skipping testing.

4. *Computer Program Test Methods Symposium*, organized at University of North Carolina in 1972.

5. *The Art of Software Testing*. In fact, *Program Test Methods* [Hetzell73] actually was the first but was a collection of papers and is therefore often forgotten [GH88].

The first two arguments assume single-specialist teams rather than cross-functional teams. The last argument suggests a quick fix for the much larger problem of quality-destroying shortcuts when pressuring developers.

In these arguments, test *independence* is equated to test *separation* from development. However, Hetzel clarifies the principle:

The requirement is that an independence of spirit be achieved, not necessarily that a separate individual or group do the testing. [Hetzel88] (emphasis in original)

This point is reiterated in *Agile Testing* in which the authors also point out the suboptimization created by separating testing:

Teams often confuse “independent” with “separate.” If the reporting structure, budgets, and processes are kept in discrete functional areas, a division between programmers and testers is inevitable. Time is wasted on duplicate meetings, programmers and testers don’t share a common goal, and information sharing is nonexistent. [CG09]

Test independence does not mean independent testers.

How to achieve test independence in *spirit* without separating testing? By writing tests before implementing code. The test cannot be influenced by the implementation, because it does not exist yet. This way, test-driven development achieves the *spirit* of independence without separation of departments.

Avoid...Test department

In Scrum, the Team is cross-functional, consisting of *at minimum* developers and testers.

We sometimes work with organizations where the test department ‘gives’ the testers to the team toward the end of the iteration. Not recommended.

Alternatively, some organizations have a matrix organization where ‘resources’ are ‘allocated’ to a Scrum project. When finished, the ‘resources’ are returned to their traditional functional organization—the pool. The tester is full-time on the ‘team’ but will return to the test department. This can work though is not recommended.

see Organization in the companion for more on matrix organizations

Having testers return to their test department often inhibits them from broadening their skill and learning different non-test specializations. It leads to *testers being testers* on the team—the waste of working to job title—instead of *team members with their main specialization being testing*.

Avoid having a test department. Dissolve the test group and merge with the development department to create a “product development” department consisting of permanent cross-functional teams. Also: See “Avoid...Separate analysis or specialist groups” on p. 234.

A product group we coached in India had two separate testing groups—an “end to end” testing group and a non-functional one. When adopting Scrum, they dissolved the end-to-end testing group and merged them into the cross-functional teams. However, even after six months, they were still unable to disband the non-functional testing group, because of its narrow specialization, interrelated work, and lack of automation. Last time we visited the product group, they were automating the non-functional tests and doing pair testing to broaden their skills; they estimated it would take another six months before they could dissolve the non-functional testing group.

See “Try...Product-level Definition of Done” on p. 170.

Integrating all testing into Scrum teams is a gigantic step for many big product groups. They do not yet have the capability to take that step for example, because they do not have any test automation. In this case, they might temporarily keep the test department for the testing that is not yet included in their definition of done—the “undone unit.” As the organization improves—the Definition of Done expands—this department will gradually disappear.

Every now and then we hear, “We cannot integrate *our* testing with the development!” Organizations should be able to *at least* integrate their ‘functional’ testing with the development teams when starting Scrum. We *do* promote incremental improvement, but integrating

development and testing is the minimal baby-step an organization should take for their journey to a lean and agile development.

Avoid...Test department

In Scrum, the Team is cross-functional consisting of *at minimum* developers and testers. Déjà vu? These are frequently recurring topics. We would like to repeat them. Goto p. 29.

Avoid...TMM, TPI, and other ‘maturity’ models

See “Avoid...Believing CMMI appraisal or certification means much in creative R&D work” on p. 489.

“*Maturity Goal 3.1: Establish a Test Organization.*” [Burnstein02]

An organization without a separate testing department is not a very mature organization—according to the Testing Maturity Model (TMM). The Test Process Improvement (TPI) model of assessing organizational maturity also assumes a separate test function. A truly cross-functional organization would be immature? Wrong.

These ‘maturity’ models invariably measure a complex system by using a simplistic model⁶ and therefore provide a limited perspective. But can these models not be used for uncovering improvement ideas? Yes, they can. However, they by definition consist of so-called “best practices”⁷ and rarely novel ideas—therefore, for improvement ideas, look for other, non-“maturity-assessment” testing literature.

Avoid...ISTQB and other tester certification

We were giving an *introduction to agile development* at a client in Poland. Most people appreciated the ideas we introduced but there was an *unusually strong resistance* from the testers—which puzzled us. At the next-day workshop we had the opportunity to dig deeper

-
- 6. The models are often very complex, yet in comparison with the overall development system and potential development contexts they are simplistic.
 - 7. The second principle of the context-driven school to software testing is “There are good practices in context, but there are no best practices” [KBP02].

into the resistance and found one difference between them and other groups...they were ISTQB-certified testers.

We promote learning better testing skills. However, a problem with the ISTQB⁸ test certification is that it seems to assume a traditional environment. For example, *“For large, complex or safety critical projects, it is usually best to have multiple levels of testing, with some or all of the levels done by independent testers”* [ISTQB07]. It also seems to promote narrow role definitions. For example, *“The responsibility for each activity [debugging and testing] is very different, i.e. testers test and developers debug.”*

Try...Testers and programmers work together

Separating testing from development often leads to a conflict between programmers and testers. Testers—hunting for bugs—try to prove that part of the program is faulty. Programmers—with their ego in their code—defend themselves, their code, and the program. Probably everyone who has been in the role of a tester in a test department has experienced this.

In a Scrum team, ‘*testers*’ are no longer testers but ‘simply’ members of the team—with testing as their *primary specialization*. ‘Programmers’ are *any* members of the team who can code. Every member of the team has a shared goal and is held—as a team—accountable to that goal. Team members with different primary specializations have to cooperate in order to reach that goal.

Try...Testers not only test

Specialization is good—it increases depth of knowledge, productivity and pride in workmanship. *Single* specialization is harmful—it creates constraints, silos, waste of handoff, and mental communication barriers.

see *Lean in the companion for more lean wastes*

8. ISTQB stands for International Software Testing Qualifications Board. Information can be found from www.istqb.org.

The tasks for a team never *exactly* map to the specialization of its members. There might be fewer testing tasks than testing specialists and the tasks will not be balanced over the iteration.

The “person with testing as a main specialization” *could* become a part-time member of the team or could just wait for testing work to become available. *Not recommended*. Instead, he picks up a non-testing task and gradually broadens his specialization. For example, he could pair-program with other team members—pairing with a test specialist is likely going to increase the code quality. Or, he might support the Product Owner or “the technical writer.”

Try...Technical writer tests

See “Avoid...Separate analysis or specialist groups” on p. 234.

“Can a technical writer be a part-time member of multiple teams?” we are occasionally asked. We typically reply that it is possible, but we suggest they have a dedicated technical writer⁹ on each team.¹⁰ “But, there is not enough writing work for a full-time writer on each team” is the predictable answer.

Technical writers usually work from a customer viewpoint. This perspective is especially useful when discovering requirements and creating tests. Use their viewpoint and make them dedicated team members who, like other team members, can broaden their specialization. We sometimes joke that it’s easier to teach a technical writer to test than to teach a tester to write proper English.

Try...Educate and coach testing

Good testing skills come with deliberate practice and time. Unfortunately, especially in large organizations, testing skills are not respected. “*Everybody* can do that” is their belief, so they offshore it to a company that grabs people randomly from the street and assigns them to test. Random people hired to *bang away* on an applica-

-
- 9. With “technical writer” in this section, we mean “person with technical writing as a primary specialty.”
 - 10. This is not a novel idea. In fact, it is similar to the Mercenary Analyst organizational patterns described in *Organizational Patterns of Agile Software Development* [CH05].

cation routinely see their job as a temporary stage they need to go through before advancing to a “real job.” They do not bother deepening their testing skills and so contribute to the false belief that testing is trivial.

Testers who don't bother to learn new skills and grow professionally contribute to the perception that testing is low-skilled work. [CG09]

Falling into the “testing is trivial” trap is costly. Support testing mastery by providing self-study material, education, and coaching. We have listed some general testing-skills literature in the recommended reading section.

Of course, providing education and coaching in testing is also important to traditional environments. In cross-functional teams, this becomes even more relevant as testers at times feel marginalized. Not having a testing functional organization may impact the feeling of career progression and their interest in testing. And this is exacerbated if all education and coaching is related to development or management practices. High test turnover during an agile transition is not uncommon

They split up their test organization... However, they put the testers into the development units without any training; within three months, all of the testers had quit because they didn't understand their new role. [CG09]

Similarly, in an agile transition we worked with, many testers left to different products because they felt they had lost their identity and did not know how to work in a Scrum team. Prevent this by developing the team's testing expertise.

Try...Community of testing

Education and coaching are not the only ways to grow expertise. Open discussion and experience-sharing foster learning. One purpose of a functional unit—a test department—is to enable this learning. Without it, other means for discussion and sharing experiences are needed. For instance, by establishing a Community of Practice for testing. People interested in testing—not only those with testing

see Organization in the companion for more on Communities of Practice

as their main specialization—meet every now and then to learn from each other or discuss via a mailing list or wiki.

Test managers can play an important role in this community. They can use their expertise in testing and management and become CoP coordinators—in accordance with the lean principle of manager-teachers.

Rather than keeping the testers separate... [think about] a community of testers. Provide a learning organization to help your testers ... share ideas and help each other. If the QA manager becomes a practice leader in the organization, that person will be able to teach the skills that testers need to become stronger and better able to cope with the ever-changing environment. [CG09]

Try...Recognize project test smells

A *smell* is an indication that something is not okay. In *xUnit Patterns*, Gerard Meszaros defined a set of *project test smells*:

- *buggy tests*—defects are found that should be detected by automated tests. They were not found due to mistakes in the tests.
- *developers not writing tests*—no automated tests are added while the developers are implementing functionality.
- *high test maintenance*—a lot of time is spent maintaining the tests. And, when new functionality is implemented, most of the effort goes to updating the automated tests.
- *production bugs*—many defects slip through the testing.

Meszaros calls these “project smells” because they are at a high level and are easily recognized by the management. Smells signal that something is wrong—they are not the cause themselves.

We should look for project-level causes. These include not giving developers enough time to perform the following activities

- *Learn to write tests properly.*
- *Refactor the legacy code to make test automation easier and more robust.*
- *Write the tests first. [Meszaros07]*

The causes of these smells can be discovered with root-cause analysis using tools such as Five Whys or Ishikawa diagrams. Alternatively, *causal loop diagrams* are a great technique for exploring system dynamics.

see Lean Thinking and Systems Thinking in the companionasdfm

Avoid...Separate test automation team

We advise organizations to invest in test automation and create a safety net of regression tests around their legacy code so that they can gradually work themselves out of the mess. They listen, and then create a separate test automation team.

see Legacy code chapter

Sometimes the test automation team tries to solve *all world problems* with their testware, and the effort produces only a lot of paper. But, sometimes we encounter a more pragmatic test automation team that actually creates testware such as an automation framework. They release every couple of months and everyone is impressed with the results.

What happens then? New functionality is implemented. Interfaces change and the automated tests fail. The development teams are upset and tell the test automation team to fix the tests. Or, the development teams comment out the tests because they do not understand them. Or, the testware is handed over to the development teams who discover it is unusable or incomprehensible and ignore it. Or... we have experienced a dozen different scenarios in organizations. It never worked.

Why? The assumption that is the *creation of the testware* is the difficult part and the most important thing. But other important aspects are under-appreciated:

- Creating testware requires deep understanding of the product.
- Maintenance and evolution is more effort than initial creation.
- Insights obtained during testware creation is perhaps more important than the testware itself.
- Creating testware without *using* it leads to complex and unusable testware.¹¹

Considering these aspects, a separate automation team causes additional complexity, the wastes of handoff, and knowledge scatter. No wonder it so often fails.

Test automation should be the responsibility of the cross-functional development teams—just as testing is also their responsibility.

There is no shortcut to learning how to automate; a separate automation team is a *quick fix*—and harmful in the long run.

Try...Feature team as test automation team

A separate test-automation team has many drawbacks but also some advantages. They can create the initial test framework, produce training material, and support the teams.

How to get these benefits without the drawbacks?

A feature team can temporarily take on the role of test-automation team. Advantages:

- They have a deep understanding of the system.
- They can take a small feature so that the automation is concrete and realistic.
- The learning created during test -automation will not be lost.
- There is visibility into test automation as the items go on the Product Backlog.

Try...All tests pass—stop and fix

Test fails?

Stop and fix it!

11. The same is true for creating reusable components without having used them.

“What about your automated tests?” we ask product groups when we visit them. Sometimes they reply, “We have 800 automated tests of which 200 are failing right now.” This is a huge queue and causes a complete lack of transparency in the development. When automated tests fail, fix them immediately.

see Continuous Integration chapter

Avoid...Using defect tracking systems during the iteration

Fix bugs discovered in the work underway for the iteration immediately. If it takes a lot of work, create a task on the Sprint backlog. However, there is usually no need to log this bug in the defect-tracking system. Logging it would only create another work queue and more delay—a waste.

On the other hand, defects found outside the iteration—by an ‘undone’ unit or the final users—are normally tracked in a defect-tracking system.

See “Avoid...Defect items in the Product Backlog—unless few” on p. 225.

Try...Zero tolerance on open defects

Why do people insist on creating defects? They spend effort to insert a defect, then they need to search for it, prioritize it, and finally fix it. Not creating the bug in the first place would be a lot less work.

We *do* believe it is possible to write *bug-free code*. We do *not* believe it is easy or common. Still, focus on preventing defects.

“Zero tolerance on open defects” is a guideline used by one of our clients. If they find a defect, they fix it as soon as possible. This prevents

- effort spent on tracking many defects
- effort spent on prioritization
- delaying the learning that happens when fixing a defect
- spending extra time on fixing because the developers do not remember the code anymore

Delaying the fixing of bugs is a false economy inasmuch as they need to be fixed anyway and the cost will be higher. Moving bugs from queue to queue is fooling yourself—they are still there!

Avoid...Commercial test tools

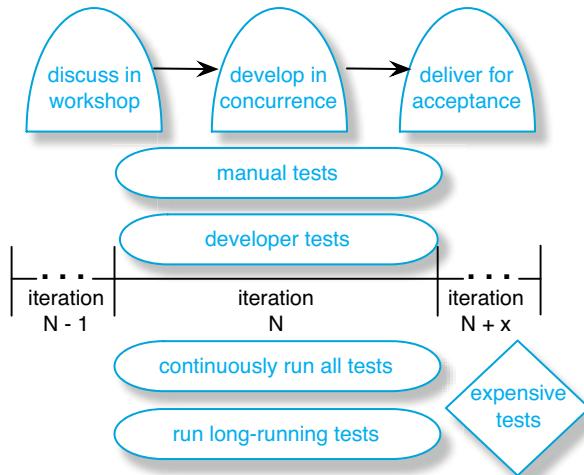
We once coached at a company building a commercial “automated testing” tool—a GUI testing tool. The requested coaching? To learn how to do automated testing for developing their automated testing tool...

A gazillion commercial test tools are available. We rarely meet people who are actually satisfied with any of them. Most are overly complex and focus more on reporting and ‘management’ than on robust test automation. Favor free and open-source tools—made by developers solving real problems—over commercial tools.

Overview of testing in an iteration

What are the test-related activities in an iteration? This section provides an overview of these activities and a road map for the rest of the chapter (see Figure 3.2).

Figure 3.2 testing activities in an iteration



Testing activities in a typical iteration:

Before the iteration

- The Team and the Product Owner clarify the requirements by writing example tests in a requirements workshop.
- After the workshop, a team member moves the examples on the wall to the team's wiki. The team might already distill tests out of these examples and write them in their A-TDD tool.

Sprint Planning

- Additional requirements clarification may happen during Sprint Planning part one, resulting in new examples and tests.
- The tasks for implementing the examples/tests are created during Sprint Planning part two. Tasks are created for
 - distilling tests out of the workshop artifacts
 - creating more automated tests for unanticipated scenarios
 - implementing glue code between the A-TDD tool and the system under test
 - manual tests that cannot be fully automated (yet), such as usability tests or expensive tests
 - timeboxed sessions for doing exploratory testing

During the iteration

- Example tests are the driver for implementing requirements.
- Glue code between the A-TDD tool and system under test is developed.
- Tests that pass are added to the continuous integration system. Long-running tests are also continuously executed—though in a longer cycle.
- Manual tests—such as exploratory or usability testing—are done right after the requirement is implemented.

Sprint Review

- ❑ The examples and tests created during the requirements workshop are executed and demonstrated to the Product Owner and other stakeholders.

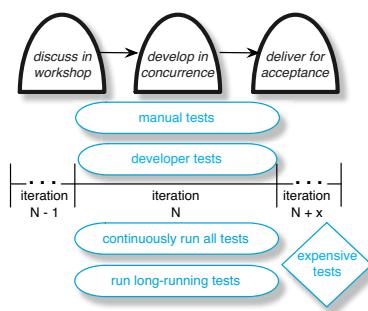
Before release

- ❑ Expensive tests that could not run frequently are executed in a final test run before the release. There *should* be no surprises anymore during this test run, because the risks have been tackled during the iterations.

CUSTOMER-FACING TEST

This section covers testing focused on whether the product fulfills the customer requirements: customer-facing tests. Most experiments in this section are also applicable in single-team development but, from our coaching experiences, these are especially relevant for large organizations with many people involved in the development.

Try...Acceptance test-driven development



Acceptance test-driven development (A-TDD)¹² is a collaborative requirements discovery approach where examples and automatable tests are used for specifying requirements—creating *executable specifications*. These are created with the team, Product Owner, and other stakeholders in requirements workshops.

12. Acceptance test-driven development [Hendrickson08] is also known as agile acceptance testing [Adzic09] or story test-driven development [Reppert04].

A-TDD integrates some major ideas:

- tests as requirements, requirements as tests
- workshops for clarifying requirements
- concurrent engineering
- prevention instead of detection

Tests as requirements, requirements as tests—In *Exploring Requirements: Quality before Design*, authors Gause and Weinberg investigate the link between requirements and tests, “*one of the most effective ways of testing requirements is with test cases very much like those for testing the completed system*” [GW89]. Melnik and Martin extend this further and claim, “*As formality increases, tests and requirements become indistinguishable. At the limit, tests and requirements are equivalent*” [MM08]. Tests must be precise in order to be automatable. A-TDD exploits this formality and formulates requirements by writing automatable tests.

Workshops for clarifying requirements—The sixth agile principle reminds us “*The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*” Face-to-face requirement clarifications in workshops have been used since the invention of Joint Application Design (JAD) [WS95]. And these are also used in Rapid Application Development (RAD) [Martin91] and the agile method DSDM [Stapleton03]. A-TDD similarly exploits face-to-face conversation by using workshops for formulating requirements-as-tests.

See
“*Try...Requirements work-shops*” on p. 240.

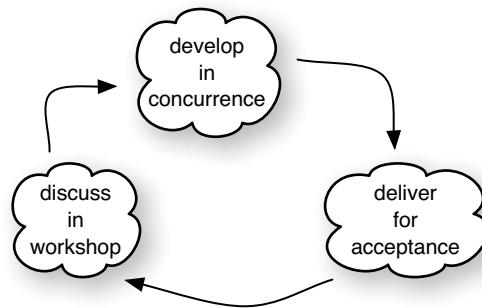
Concurrent engineering—The authors of *Concurrent Engineering Effectiveness* define concurrent engineering as follows: “*There is a tight link between participants in the product development process, such that they can perform much of their work at about the same time*” [FL97]. The main driver of concurrent engineering is shorter cycle times in development. Two-week iterations are fast and therefore the team needs to conceive a way to work concurrently—sequential development in a short iteration does not work. We have seen teams *invent* A-TDD again and again simply because they had to answer the question: “*How can we perform our work at the same time?*”

See “*Try...Two-week iterations to break waterfall habits*” on p. 394.

Prevention rather than detection—In one of the first studies of Toyota, *A Study of the Toyota Production System*, Shigeo Shingo writes “*The purpose of inspection must be prevention; however, for inspection to have that function, we must change our way of thinking*”¹³ [Shingo89]. Similarly, in “The Growth of Software Testing,” the authors identify five periods in the evolution of software testing. They call the latest period “*The prevention-oriented Period*” and state, “*Asking test-related questions... early is often more important to software quality and cost-effective development than actually executing the tests*” [GH88]. This is exactly what A-TDD strives to do. When including people specialized in test in the requirements workshop, they can ask the test-related questions, and in that way improve the requirements and *prevent* defects. The Total Quality movement—an influence to Toyota and lean development—also promotes prevention over detection.

How does A-TDD work? Figure 3.3 presents an overview.

Figure 3.3 A-TDD overview



A-TDD consists of three steps:

1. Discuss the requirements in a workshop.
2. Develop them concurrently during the iteration.
3. Deliver the results to the stakeholders for acceptance.

13. In manufacturing the term ‘inspection’ is used instead of test.

Discuss—Requirements are discovered through discussion in a requirements workshop¹⁴. Participants of a workshop are the cross-functional team, the Product Owner or representative, and any other stakeholder who potentially has information about the requirements. A common question to ask during such workshops is “*Imagine the system to be finished. How would you use it and what would you expect from it?*” Such a question results in examples of use, and these examples can be written as tests—the requirements. The workshop focus ought to be on discussion and discovery of requirements more than on the actual tests.

Develop—At the end of the workshop, the examples are *distilled*¹⁵ into tests and all activities needed to implement the requirement are done concurrently. These include

- making the glue code between the tests and the system under test (“test libraries” and “lower-level tables” in Robot Framework or ‘fixtures’ in Fit)
- implementing the requirement so that the tests pass
- updating architectural and other internal documentation according to the working agreement of the team
- writing customer documentation for the requirement
- additional exploratory testing

The exact list depends on the product, context, working agreements, and the Definition of Done.

See “*Try...Product-level Definition of Done*” on p. 170.

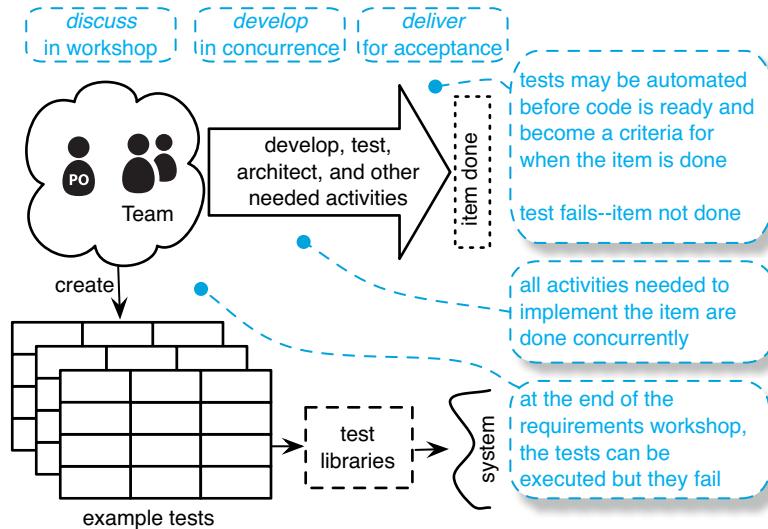
Deliver—When the tests pass, the requirement is reviewed with the Product Owner and other stakeholders. This might lead to new requirements or a change in the existing tests.

A more detailed way of describing A-TDD is shown in Figure 3.4.

14. Gojko Adzic calls these *specification workshops* [Adzic09].

15. [Hendrickson08] considers *distill* a separate step in A-TDD.

Figure 3.4 A-TDD
in more detail



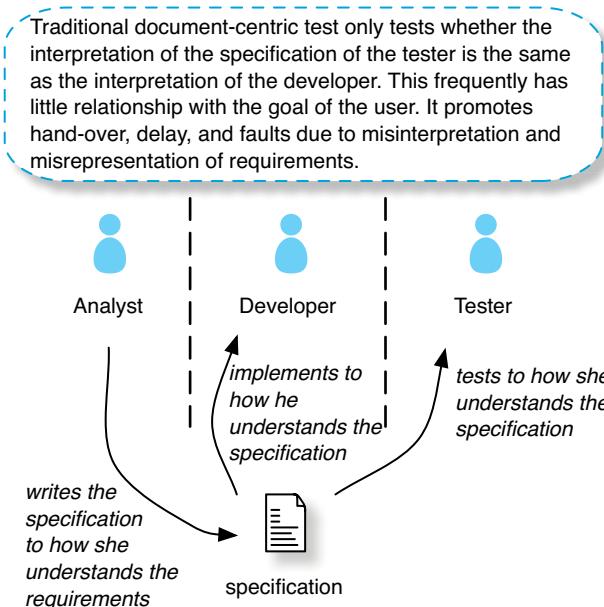
Avoid...Traditional requirement handoff

The collaborative style of discovering requirements in A-TDD is contrary to conventional serial development—where an analyst clarifies requirements by herself, documents them in specifications, and hands these off to a developer and tester.

The developer implements the software according to his understanding of the specification. Afterwards, the tester tests whether her understanding of the specification is the same as the developer's understanding—which often has nothing to do with the *real* wishes of the customer (see Figure 3.5).

The amount of waste—handoff, delay, partially done work, and knowledge scatter—in this document-centric way of development is *extraordinary*. Avoid it.

Figure 3.5 conventional document-centric style of requirements clarification



Avoid...Thinking A-TDD is for testers

“Our testers do A-TDD” we sometimes encounter at clients. Testers cannot “do A-TDD” because it is a whole-team technique—including people with testing as their primary specialty. If not the *whole* team, including the Product Owner or representative, is involved, then whatever they are doing might be useful—but it is not A-TDD.

Avoid...Confusing TDD and A-TDD

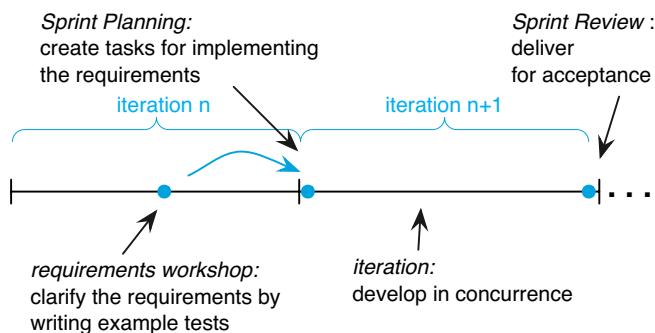
Test-driven development is a developer technique that drives the design by a microcycle of test–code–refactor. Acceptance test-driven development is a whole-team technique that drives the requirement discovery by a cycle of discuss–develop–deliver. Both write tests first, but their goals are unlike. Don’t confuse them.

Both names unfortunately suggest testing techniques. Yet, even though many¹⁶ have attempted to change the names, these are the most commonly used terms—and thus we decided to use them.

Try...A-TDD match the iteration flow

The steps in A-TDD map nicely to the Scrum iteration cycle¹⁷ (Figure 3.6).

Figure 3.6 A-TDD steps mapped to Scrum iteration



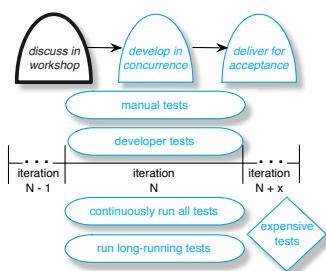
Discuss in workshop—Before the detailed Sprint Planning¹⁸, the team, Product Owner, and other stakeholders clarify the requirements collaboratively in a workshop.

Develop in concurrence—Tasks for implementing the tests/requirements are created in the detailed Sprint Planning and implemented during the iteration. All activities happen “at about the same time.”

Deliver for acceptance—The working product increment—the passing acceptance tests—are delivered for acceptance to stakeholders and discussed together in the Sprint Review.

-
- 16. For example, behavior-driven development [North03], story test-driven development [IXP04], agile acceptance testing [Adzic09], example-driven requirements [Shore03], or example-driven development [Marick03]
 - 17. The Scrum iteration cycle and the A-TDD cycle are variants of the Deming/Shewhart PDCA cycle [Deming82].
 - 18. In Sprint Planning part one or Product Backlog refinement.

A-TDD—Discuss in Workshop



The workshop-related experiments are strongly connected to those in the *Requirements* chapter. This section covers A-TDD-oriented topics; the *Requirements* chapter covers requirements workshops in more detail.

Try...Discuss in workshop during Product Backlog refinement

The team and Product Owner ‘inspect’ the Product Backlog during the Product Backlog refinement to ensure it is in a good shape. This activity includes the following:

- Estimate* and clarify newly added Product Backlog items.
- Split* large items into smaller ones so that they can be selected for implementation.
- Clarify* the *imminent* items so that the team understands them well enough to implement them.

The clarification of *imminent* items can be done through A-TDD-style requirements workshops. Just to be clear, Product Backlog refinement is *not* only a A-TDD requirement workshops but it can be part of the refinement activity. Other activities include estimation and splitting.

See “Try...Split Product Backlog items (such as stories)” on p. 247.

Try...Clarification over writing tests

A-TDD is for collaboratively clarifying requirements. The emphasis is on communication, collaboration, and learning through examples and tests. Increased understanding is the goal, tests are the means of getting there. The appropriately titled book on this subject, *Bridging the Communication Gap* stresses:

[Acceptance-test driven development] is not a programming technique: it is a communication technique that brings people involved in a software project closer. [Adzic09]

People are often so preoccupied with the tangible outputs of a workshop—the tests—that they forget about the intangible outcomes—the learning. Understanding and clarity of the requirements is the key output of a requirements workshop; the tests are an expression of these.

see False Dichotomy in the companion

This is not a false dichotomy. The tests *are* important and this technique is called acceptance-test-driven development. Without tests, it would be just a requirements workshop—but avoid confusing means with ends.

Try...Use examples

“Can you give me an example?”

This question can suddenly transform a vague and abstract discussion into a clear and concrete one. When discussing new products, people tend to end up talking in concepts and abstract terms. They talk past each other without understanding—they are stuck. Asking for examples brings the discussion back to reality.

For example, we hear assertions such as “The system needs to recover from error situations.” This is vague, so we ask for examples that transform the discussion. This could be “When we unplug the cable, the system should not crash”—which is concrete and understandable. Examples are also used for further clarification, such as, “How should the system recover if we remove a unit from the system while it is running?”

Examples are not just useful for clarifying requirements but also for *clarifying ways of working*. “We could never automate all our tests!” is something we would follow up with “Can you give me an example of a non-automatable test?” and that moves the discussion away from principle and into practice.

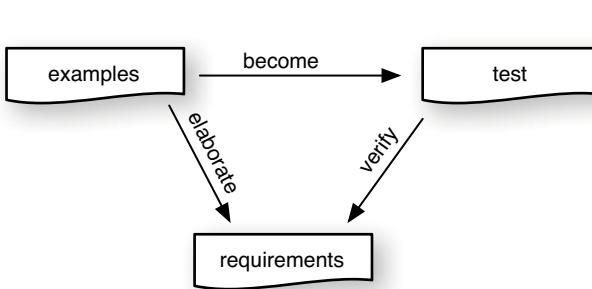


Figure 3.7 relationship between examples, tests, and requirements

Figure 3.7¹⁹ shows the relationship between examples, requirements, and tests²⁰. During requirement workshops, use examples to elaborate requirements and transform these into tests.²¹

Try...Product Owner writes tests

Examples and tests express the requirements, and their leading author ought to be the person who knows these requirements best—usually the Product Owner, representatives, or future users. Not someone specialized in testing.

Should people specialized in testing *not* be involved? On the contrary—they play an essential role. They are experts at critiquing the system, which is exceptionally useful during requirements clarification. They are asking the “what if” questions that the Product Owner might not have considered—and that way *prevent* building the wrong system.

Avoid...‘Optimizing’ the requirements workshop

When we were discussing A-TDD with a large product group, they noted, “We *improved* the A-TDD workshop. Only three people partic-

19. Diagram from [Adzic09] who credits Jennitta Andrea for creating it.

20. In this chapter, we use the words test, example, and requirement almost interchangeably.

21. Talking about examples also might help when you encounter people who say, “I will not write tests, I’m a Product Owner.”

ipate: the Product Owner, the ScrumMaster, and a specialist in the team.” We asked them how the other team members would understand the requirements so that they can implement them and the answer was, “The specialist will tell them.” They ‘optimized’ the workshop by reintroducing traditional analyst-team handoff.

Avoid...Computers and projectors in the workshop

*See
“Try...Requirements workshops” on p. 240.*

Computers suck the lifeblood out of a workshop. They become the center of the discussion. Other than reference checking, avoid the need to ‘optimize’ the workshop by typing directly into the computer. Instead...

Try...Condense workflow in business rules²²

Requirements clarification in workshops often starts with abstract concepts, and then when examples are put forth, it moves into workflow discussions—“When using the system I’d do step one, then step two, and then expect X.”

These workflow examples may end up being similar with only a slight variation in one or two steps. The workflow tests contain hidden business rules, which can be extracted and put into a data-driven test. This centers the discussion on domain clarification and reduces complexity by removing irrelevant details.

The Robot Framework example section in this chapter shows some examples of business-rule tests.

Try...Test the walls

The rightful home of tests is on the wall—well, with a whiteboard between the tests and the wall. Big whiteboard spaces promote collaboration—the purpose of the workshop. The whiteboard sketches are captured with photos. After the workshop, the tests may be distilled and written in a tool.²³ See Figure 3.8.

22. We use the term “business rules” to be consistent with existing literature [MC05, Adzic09, Evans04] though the term is awkward and not frequently used in the context of some product development.

Customer-Facing Test

Figure 3.8 tests on the wall



Try...Use table format

Expressing business rules in tables makes them more comprehensible and assists in finding missing cases. Tables inspire clear thinking. Influential computer scientist David Parnas is a long-time promoter of tables for documenting requirements.

-
23. There is a trade-off between doing this immediately—when the information is fresh—and delaying till near the actual implementation—avoiding potential waste.

[When documenting requirements,] writing, understanding, and discovery go on at the same time... Tabular notations are of great help in situations like this one. One first determines the structure of the table, making sure that the headers cover all the possible cases, then turns one's attention to completing the individual entries in the table. [JPZ96]

Try...Workflow tests

Extracting business rules and using data-driven tests is not always possible or desirable. Some requirements are just better expressed in a workflow (multi-step scenario) example. Also, data-driven business-rule tests can often be complemented with workflow examples that, in a way, link them.

Caution: When most of your tests are workflow tests, then you *probably* missed some business rules.

The Robot Framework example at the end of the chapter shows an example of a workflow test.

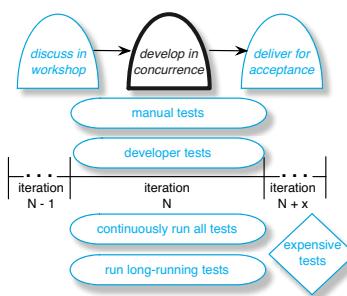
Try...Typical workshop agenda

How is an A-TDD requirements workshop structured? We frequently use the following agenda:

1. *Introduction*—the Product Owner welcomes everybody to the workshop and explains its purpose.
2. *Selection*—the team and Product Owner pick the items from the Product Backlog that will be worked on.
3. *Overview*—the Product Owner or representative gives a short overview of the selected requirements.
4. *Diverge*—the team splits into two or three subgroups that each pick an item and start writing examples on whiteboards. The Product Owner and representatives rotate between subgroups.
5. *Merge*—the subgroups combine and one at a time share their work with the whole group.

6. *Repeat*—the diverge-merge cycles are usually 30–45 minutes long. One workshop contains multiple cycles.
7. *Conclude*—the Product Owner summarizes the work. Then, there is a brief reflection on the workshop.
8. *Distill*—participants take photos of all the work and put them on the wiki. They might already distill some tests and document them in their A-TDD tool.

A-TDD—Develop in Concurrence



After the requirements are clear, they need to be implemented. The different activities required for implementation are done in concurrence.²⁴ The team extends the tests while at the same time implementing the code, writing the documentation, updating the design description, and so forth.

Try...Distill the tests

Many examples are created during the requirements workshop. Not all of these become tests—only the essential parts of the requirements are distilled into tests. The nonessential or duplicate parts are discarded—they have served their purpose for *learning* during the workshop.

How to distill tests from examples? Some techniques:

- **Duplication**—Remove duplication among examples by writing the tests in a different form. For example, a set of workflow tests might be combined into a business-rule test. Most of this should have happened during the workshop.
- **Equivalence class**—Some examples are part of the same equivalence class and therefore it is enough to keep one test.

24. “In concurrence” has a double meaning: 1) at the same time 2) in agreement.

People with a testing speciality are especially valuable since equivalence-class partitioning is a classic testing technique.

- **Acceptance**—Since not all examples are of equal importance, ask the Product Owner, “What set of examples do you want to see running at the end of the iteration?”

Avoid...Multiple requirement descriptions

The tests act as the *living* documentation of the requirements. Additional documents for the same requirements are probably redundant waste. They are a duplication of the same description in a different format, leading to additional cost for maintaining them.

At the risk of being *redundant*: When the requirements are clarified by tests, then additional specification is a redundant waste. Avoid other requirements specifications when there are tests—*executable specifications*.

When additional clarification of the requirement is needed, then A-TDD tools such as Fit and Robot Framework support adding clarification text and images around the tables. This clarification is not in a separate document but is stored along with the test.

Try...Use A-TDD coaches and facilitators

A-TDD is easy to do, and hard to adopt. It requires challenging deeply rooted assumptions and changes in habit. An (external) coach with experience in A-TDD and organizational change is frequently needed for this. Find a coach.

It is important to realize that the *skills of a good A-TDD coach are different from those of a good TDD coach*. TDD coaching is more technical, and focused on individual developers, whereas A-TDD involves the whole team. In addition to technical skills, a good A-TDD coach has excellent workshop-facilitation skills.

Try...Fit or FitNesse

Fit is perhaps the most widely used tool for A-TDD. It was developed by Ward Cunningham in 2002. Fit tests consist of HTML tables that are executed by a piece of glue code—called a fixture. Micah and Bob Martin created a extension of Fit called FitNesse in which the tables are written in a wiki. Fitnesse also includes Slim—a *slimmer* execution model that offers better portability and the flexibility to explore new test syntaxes. More information can be found in the recommended readings at the end of this chapter.

Try...Robot Framework

Robot Framework originates from Nokia Siemens Networks and was developed by Pekka Klärck. It was open-sourced in 2008. Robot has some similarities to Fit, such as tabular structured tests and glue code between the tables and the system. However, it also has unique features such as layering tables (user keywords). The last section of this chapter shows examples of Robot Framework tests.

Try...Other A-TDD compatible tools

Robot Framework and Fit are not the only tools suitable for A-TDD. Any tool that allows for customer-understandable, executable specifications is okay. Most such tools are open source...

For example, Cucumber²⁵ uses the “behavior-driven development”-style of specifications—*given X when Y then Z*. Cucumber tests are readable sentences in plain text. Concordion²⁶ reads specifications written in HTML that are instrumented with tags. These tags are used by the fixture classes to execute the specification.

Avoid...Conventional test tools for A-TDD

Some product groups we worked with try to use their conventional test tools such as Lisp-based scripts or TTCN²⁷ for A-TDD. This

25. Cucumber can be found at <http://cukes.info>.

26. Concordion can be found at <http://www.concordion.org>.

invariably fails. Why? *A-TDD-style tests are created so that the Product Owner or user can read and understand the tests.* But the test format—scripts—of these conventional tools are created for testers and are thus unsuitable for documenting requirements. It is nearly impossible to involve the Product Owner in writing examples using such tools.

Try...Wrap conventional test tools under an A-TDD tool

Should you throw away all conventional test tools when adopting A-TDD? Perhaps not.

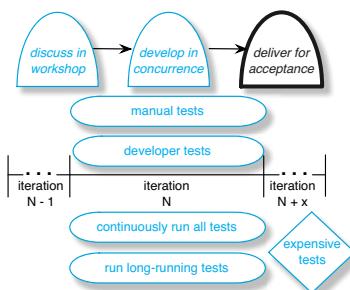
A graphics product group we worked with had spent *years* building a scripting language for automating their tests. It would take an additional few years to develop the glue code (test libraries or fixtures) between the A-TDD framework and their system under test—most of it is the same work they did with their scripting language.

An alternative is to let the glue code wrap their test scripting language and reuse their earlier work. Conventional test tools are not necessarily bad tools, but they just provide the wrong format—the wrong language—for executable specifications.

For example, Robot Framework has wrappers for Selenium and Jemmy²⁸.

-
27. TTCN—testing and test control—is a programming language used for testing protocols and web services.
 28. Selenium is a web application testing system; more info at seleniumhq.org. Jemmy is a Java UI testing tool; more info at jemmy.dev.java.net.

A-TDD—Deliver for acceptance



The code is implemented and all the tests pass. What's next? The A-TDD cycle, like Scrum, contains an inspect–adapt cycle where the results are delivered to stakeholders, who inspect the outcome using the tests and decide how to proceed—which requirements to implement next.

Try...Show tests in Sprint Review

In a Sprint Review, the team demonstrates visible progress to the Product Owner by showing the output of the iteration.

We worked with some groups that *defined the demo steps* during the Sprint planning. The team would spend an inordinate amount of time in *demo preparation*.²⁹ A complete waste.

During Sprint Planning, an alternative is to define the examples that need to pass and show the progress by *using these tests* in the Sprint Review.

Avoid...Confusing acceptance and user-acceptance test

In the ideal situation, acceptance test and user-acceptance test are the same, but...

Agile development literature uses the term “acceptance tests,” where we use the term “customer-facing tests.” However, in traditional development, “acceptance test” often means “user-acceptance test,” which might be different. Avoid misunderstandings.

29. Scrum actually defines no more than one hour of preparation time for the Sprint Review [Schwaber04].

Figure 3.9 UAT is a subset of acceptance tests

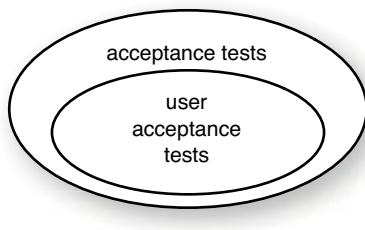


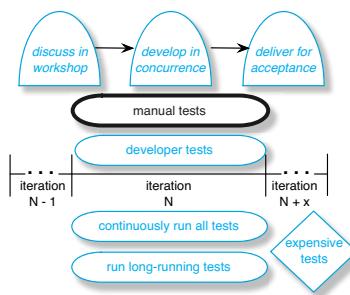
Figure 3.9 expresses the relationship in a diagram. User-acceptance test (UAT) is a part of acceptance testing in agile development. But acceptance test might also include non-UAT tests such as traditional functional or system test created by the team.

See “Try...Product-level Definition of Done” on p. 170.

See “Try...Manual (if you must) UAT each iteration” on p. 463.

Ideally, all the acceptance testing—including UAT—is done within the iteration. However, getting the UAT in the iteration may be difficult because it requires active end-user involvement and not all customers are ready for that. In that case, UAT is excluded from the Definition of Done until the product group improves their relationship with the customer so that they can expand their Definition of Done.

Manual Tests



Agile developers emphasize the importance of automated tests. With short cycles, manual regression testing is nearly impossible. Does that mean there is no manual testing at all? No. Some manual testing is still recommended, though such testing differs from the traditional script-based manual testing.

Try...Automate all tests

Product groups often claim “It is impossible to automate tests related to a lost network connection” or “You can’t automate tests related to hardware failure” Our answer usually is “No, it is not” or “Yes, you can.”

Elisabeth Hendrickson, the author of the mini-book *Exploratory Testing in an Agile Context*, dares to state that:

I do think that if you can write a manual script for a test, you can automate it. [Hendrickson09]

It may be difficult to automate a test in *exactly the same way* as it would be carried out manually. For example, it is nearly impossible to remove the network cable automatically in a connection-lost test case.³⁰ Therefore, the automated test is usually done in a different way. Instead of the cable being physically detached, the automated test instructs the driver to respond *as if* the cable were removed.

Is automating all tests worth it? According to Hendrickson:

If it's a test that's important enough to script, and execute, it's important enough to automate. [Hendrickson09]

Why is this? Iterative and incremental development implies that code is not frozen at the end of the iteration but instead has the potential to be changed each iteration. Therefore, manual regression testing would mean rerunning most of the manual test—every iteration. Automating the tests therefore pays back quickly.

Especially in large-scale development with feature teams and shared code-ownership, the safety net provided by automated tests is of paramount importance. Automating tests is well worth the effort.

Try...Manual tests

Automating *all* tests might not be worthwhile or even possible. These tests may need to be done manually:

- *Tests requiring human opinion and creativity*—A person is needed to judge whether the interface looks good—usability testing. Exploratory testing by definition needs someone to decide the next step to explore.
- *Tests requiring physical movement*—For example, tests with the system in different physical configurations. These can be

30. This is not true. It *is* possible, but requires ingenuity because it would involve robots. For example, a test lab at Xerox had a robotic arm that removed paper from a printer.

automated with simulation, but the real configuration might be needed for the final test run.

- *Expensive tests*—Running capacity tests on the production environment may be too expensive and is therefore done only once or twice. This delays risk. These risks should be tackled early with cheaper tests—for example, running capacity tests on a simulated environment—so that running the expensive test is merely a final check.

No false dichotomy: Try to automate all tests, but do not forget to do the manual tests when needed.

Try...Write “A-TDD tests” for non-automatable requirements

Tests that cannot be automated (yet)—because they are expensive or require physical movement—can still be clarified using tests and examples in an A-TDD requirements workshop. They can *also* be written as automated tests in an A-TDD tool. The only difference between these and fully automated tests is that the glue code between the test and the system under test is not (yet) implemented. Partially automating tests has a few advantages:

- These difficult tests are no different from a requirements-clarification perspective and therefore are treated the same.
- All tests, automated or not, are stored and managed in the same way.
- They can easily be automated once someone discovers how.

Caution: This is not an excuse to avoid automating tests “because we are rushed.”

Try...Exploratory testing

*Testing can be used very convincingly to show the presence of bugs,
but never to demonstrate their absence.
—Edsger Dijkstra*

In *Perfect Software and Other Illusions about Testing*, Gerald Weinberg calls it, “*The Exhaustive Testing Fallacy, that it’s possible to test everything!*” [Weinberg08]. It is not. The number of possible scenarios to test is infinite and therefore automating *all* tests means infinite automation effort. Instead, automate all the anticipated tests and spend time as efficiently as possible on manual exploratory testing to find unforeseen cases.

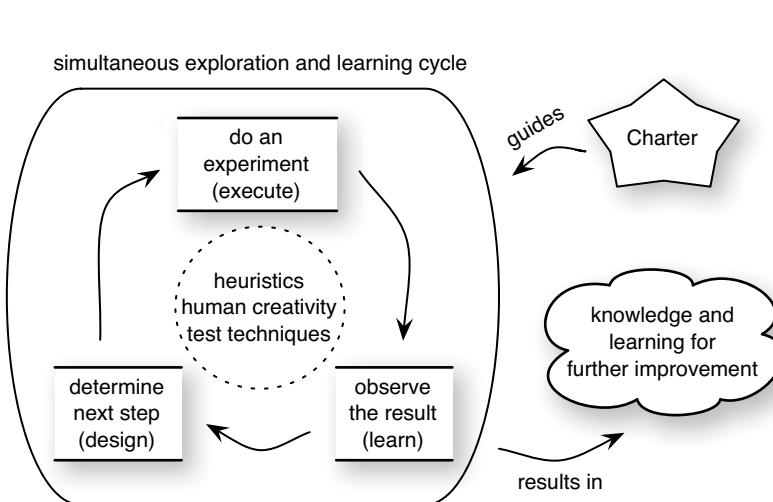


Figure 3.10 overview of exploratory testing

What is exploratory testing? “*Simultaneous learning, test design, and test execution*” [Bach03]. This is in contrast to traditional *scripted* testing (see Figure 3.11³¹) where test-case design and execution are separated and sequential steps—first design then execution. *Exploratory* testing³² aims at fully utilizing human creativity during test execution, using feedback and observations rather than mindlessly following a script. In exploratory testing, the tester is exploring the system, learning about it and using *that* information to make test-design decisions (see Figure 3.10). It is best explained by an example.

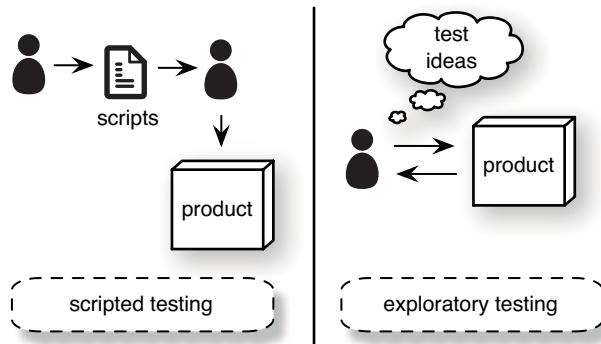
31. This figure is based on James Bach and Michael Bolton’s *rapid software testing* course material [BB09].

32. The amount of exploration during testing is a scale from blindly executing a script to just exploring. We use the term *exploratory testing* for exploring the system guided by a charter.

Imagine that Gita is testing a 2D modeling application. First, she defines the goal of her test session—in exploratory testing this is called a *mission* or *charter*.³³ Her charter is “Explore changing shapes by dragging the control points.” She takes a shape, drops it on the canvas, and creates a couple of control points on it. She drags one of them and observes what happens. Based on this observation (learning), she determines the next step (design) and performs it (execute). The shape takes its new form, though she notices—while dragging the control points—that the shape temporarily took a form that it probably should not have. Therefore, she continues dragging it and moving it around until she can reproduce the accidental transformation.

In the example, there is no detailed preconceived script or test case but instead an area of focus—a charter. The first step is to observe the system, and the next action is determined from that observation—this is *test design*. All traditional test techniques and heuristics are applied during this design step.

Figure 3.11 difference between scripted and exploratory testing



Try...Plan and time-box exploratory test sessions

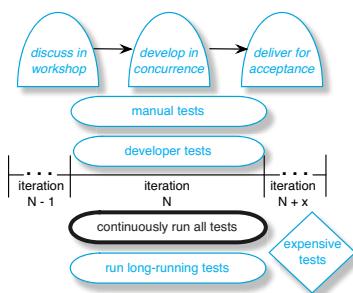
If it is possible to explore a system indefinitely, then how plan *exploration* in an iteration?

33. To be clear, exploratory testing does *not* mean aimlessly banging on a keyboard.

Exploratory testing can be planned and managed through *sessions* [Bach00]. An exploratory testing session is a timeboxed period, guided by a session charter, for exploring the system. In Sprint Planning the team answers the question, “How much time do we want to invest in exploratory testing?” and then creates session tasks accordingly.

When a team member picks up an exploratory testing task, he explores the system until the time is up. Then he decides whether it was enough or not. If more time is needed, he adds a new exploratory testing task—with timebox and charter—to the Sprint Backlog.

Continuously Run All Tests



Q: When to run the automated tests?
A: All the time.

This section covers experiments related to automated test execution and several connected ideas such as maintaining tests and traceability.

Try...Continuous Integration System

If the automated tests need to be run manually, that means they are not automated. Automated tests ought to run automatically. This is exactly what a continuous integration system does. CI is a big topic and has a dedicated chapter: *Continuous Integration*.

Try...Maintainable tests

“Automated tests will increase our test maintenance load” is a common objection we hear. Test maintenance will cost some effort, but a few straightforward techniques can minimize this cost:

- remove duplication in and between tests

- delete tests
- do not test through the UI
- run tests frequently

The next sections dive into more detail.

Try...Refactor tests

Avoid...Duplication in and between tests

Code duplication causes extra complexity, obscurity, and defects—resulting in extra maintenance effort. This is as true for test code as it is for production code. Avoid this by removing the duplication.

Workflow tests are a common cause of duplication. They often consist of one *mother* scenario and multiple derived cases with only slight variations in them. When one step changes, all these workflow tests need to be updated. This duplication can be avoided by data-driven tests that focus on business rules or by separating the duplication into test libraries or fixtures.

We coached a team that made a common mistake—they delayed their test automation until the end of the iteration. Four days remaining and only automation tasks left. In the previous iterations, these tasks were done by the test specialist, but now they *had to* be done by the whole team.

They started with a one-day workshop in which the one specialist coached the other team members. After that, they split into two pairs and one triplet working in parallel on automating the tests. Something interesting happened: The team members who were experienced in programming complained about the extra effort needed because of the duplication in the tests. Previously, none of them had noticed it and the test specialist—who did not have much programming experience—never cared. Now that the whole team was involved, they cared and the quality of the tests improved immensely.

Try...Delete tests

Tests serve multiple purposes. They act as requirements, as verification, and as a safety net preventing system regression.

When an existing test is not needed anymore—because it is a subset of another test—then delete it. Leaving unnecessary tests brings no benefit but still increases maintenance effort and lowers test execution speed.

Avoid...Test through the UI

User interfaces (UIs) tend to change frequently. Running your tests through the UI makes them vulnerable to these changes—even when there is no change in test logic. This increases the test maintenance effort.

Therefore, avoid testing through the UI and instead call the application logic directly through an API. Another advantage of doing this is that it speeds up your test since testing through the UI is slow.

Try...Run tests frequently

Long ago, we worked with a large group following waterfall development. Traditional test automation advice is to select and automate only the most important cases—with a separate automation team—after the release. So they did. At the end of the next release, they executed the tests and... they all failed. Updating them would take much time, so they decided to do all testing manually.

Executing tests once or twice a release seems efficient—fewer CPU cycles are wasted—but much will have changed and therefore many fail and cause a large batch of maintenance work. Alternatively, executing tests frequently—using a continuous integration system—uses more CPU cycles but results in less maintenance work since the effort to fix failing tests is small and straightforward. If you have a high test-maintenance load, chances are you are not executing the tests frequently enough.

see Continuous Integration chapter

Avoid...Traceability

“What about traceability?” we are asked every now and then. Traceability³⁴ seems a *universal good thing*—like happiness—though few are able to explain *why* they *must* have traceability.

When they *can* explain why, we then explore alternatives for reaching the same goal. A common goal of traceability is to “discover which tests need to be run when a certain requirement changes.” An alternative is to automate all tests so they can *all* be executed—eliminating the need to know which one must be run.

Traceability is a solution to a problem, and there may be simpler—less effortful—solutions.

Try...Traceability

Traceability is not a *universal bad thing*—like pollution—either. Avoiding it may not be an option, because it is sometimes required by standards or contracts. The key problem in achieving traceability is that keeping the needed information up to date costs a lot of work. Automation can help here.³⁵

Once tests and requirements are the same thing, then traceability between them is trivial. Traceability to code is more difficult but can be automated with tags in the tests or in the SCM system.

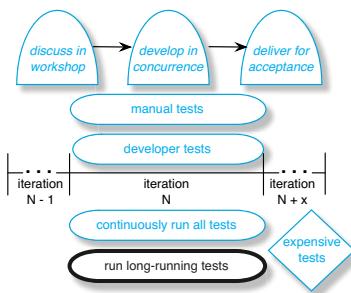
One of our clients builds nuclear-power control systems. They were audited by NUPIC (a USA nuclear agency) because they were the first project using agile development in the nuclear industry. The audit result: “no findings” (somewhat rare in a nuclear audit). NUPIC even used them as an example to others of how development should be done. Traceability was a must in this industry. How did our client achieve traceability? Requirements-as-executable-tests was a key solution; the auditor understood that this solved most traceability problems. Our client also tagged every acceptance test and unit test with its associated requirement ID.



-
- 34. The many definitions and types of traceability make the topic difficult to discuss. Here, traceability means being able to trace tests to code and requirements, and vice versa.
 - 35. Automating traceability is the goal of many requirement management tools. However, they often lead to more time spent rather than less.

Then, as a part of the continuous build, they created a table showing the mapping from requirements to their tests, the test state, and related the code coverage metrics.

Run Long-Running Tests



In large-system development, the non-functional requirements are frequently the most important, and implementing them takes the majority of the development time.

The non-functional tests are expensive and often take long to run. This section covers some experiments related to that issue.

Try...Treat non-functionals the same as functionals

Automating and continuously running non-functional tests is essential. Delaying them to the end means moving one of the biggest risks to where they hurt most. For example, if the system needs a certain performance level, test early to reach it early, and continuously run the test while new functionality is added to ensure that the system does not regress from its performance target.

Non-functionals are often treated *exotically*—people believe they cannot be specified and cannot be tested. This is unfortunate. In a requirements workshop, non-functionals can be considered the same as functionals, and example tests are created for clarifying them.³⁶ Figure 3.12 shows two examples of non-functional tests.

36. This might seem simplistic, but the highest-level A-TDD-style test cases are often that simple. The complexity of automating the non-functionals is hidden in the glue code between the test and the system under test.

Figure 3.12 simple capacity and stability test

Test Case	Action	Arg	Test Case	Action	Arg
System capacity under normal operation			Stability test with a random operations		
	Startup and init system			Operation sequence	random
	Amount of users	1000 users		For duration	2 weeks
	Maximum delay	1 second		Run operations and check	
	Connect users and check			Is system stable	
	Is system stable				

Try...Requirement area for non-functionals

The main requirements for a network element in a radio network are *capacity* and *reliability*. Radio networks are not an exception; it is common for non-functional requirements to be major requirements for large embedded products. How to deal with that?

see Requirements Areas in the companion

See “Try...Split Product Backlog items (such as stories)” on p. 247.

Dedicate a requirement area to certain non-functionals. For example, a *performance* requirements area deals with system performance. The teams in this area take large-system performance requirements, split them, and implement them incrementally.

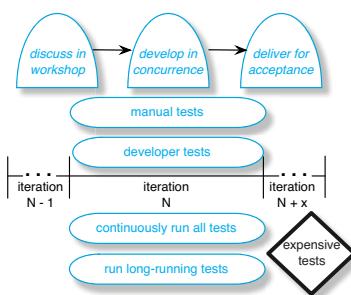
Is there any difference between non-functional requirements areas and regular ones? Not in how they are organized. The key difference relates to the type of work. Implementing non-functionals usually involves a broader part of the system and a larger focus on study and test activities. For example, the performance area use the majority of their time in creating tests, profiling the system, and creating spike solutions—tasks focused on finding bottlenecks and improving performance. Similarly, the stability area create long-running stability tests, tests for different hardware configuration, and implements system recoverability.

Try...Continuously run long-running tests

Non-functional tests frequently cannot be run in the normal continuous-integration-system cycle because they may take too long to execute—a stability test might take two weeks. Some product groups delay them until near the release—creating a delayed feedback cycle. Not a good idea.

Run the long-running tests all the time in a slower continuous-integration-system cycle. Treat them as any other tests. When they fail, inform all people who checked in code. After they pass, get the latest build and immediately run them again. This way the feedback cycle is as short as it can be.

Expensive Tests



Some tests are expensive to run. Maybe they must be run on a production environment, maybe they require expensive (and often *shared*) testing equipment, or perhaps they require an expensive configuration—such as a whole nuclear power plant or ship.

How do these tests fit into iterative development?

Avoid...Expensive tests

Whenever possible, avoid expensive tests completely. How? Analyze the tests to find out what makes them expensive, and come up with a way of doing them differently—cheaper. Some typical examples:

- **Use virtualized hardware**—When it is the hardware itself that is expensive, look for virtualization technology. A product group we worked with had proprietary hardware and so the test environments were sparse. To solve this, they ran the tests on virtual hardware.³⁷
- **Create simulators**—When the test environment is expensive, create simulators. For example, one of our clients builds ship control systems. It is much cheaper and more convenient for them to simulate the ship systems rather than...

37. For example, Virtutech.com provides virtual hardware for embedded software. See also en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines

- **Create similar environments**—When there is only one real environment and it is not available—the one and only production environment—create a low-cost similar one.

Try...Expensive tests

The cheaper tests provide quicker feedback but they are not *exactly* the same as the expensive ones—which are still needed, though they ought to be ‘merely’ a final check. Run them as often as possible—unfortunately, that may mean only once before the release.

Try...Automate expensive tests

An expensive environment may be only available for a limited amount of time, or the cost increases over time. Automating these tests optimizes the time used on the expensive environment.

However, the expensive environment is sometimes needed to automate the tests. This may make automation not worth it because the tests are run infrequently. An alternative is to automate the tests on the cheaper environment instead—then run them on the expensive environment.

DEVELOPER TESTING

This section covers testing done by and for developers. These focus on whether the system does what the developers intend it to do.

Try...Unit testing

Unit testing is not a new concept. But, to our surprise, we regularly visit companies that are not doing any unit testing. This leads to problems because unit tests provide a safety net around the code that gives developers the confidence to make changes and refactor. Without refactoring, the quality of the code and design gradually degrades.

Unit tests are written by a developer while he is implementing functionality. They are usually in the same programming language as the production code and are, by definition, automated. Michael Feathers, author of *Working Effectively with Legacy Code*, created a useful set of rules for unit tests [Feathers05]. A test is not a unit test when...

- it talks to the database
- it communicates across the network
- it touches the file system
- it can't run at the same time as any of your other unit tests
- you have to do special things to your environment (such as editing config files) to run it

Usually, unit tests are written in an xUnit framework such as JUnit for Java, CppUTest or GoogleTest for C/C++, or NUnit for .NET.³⁸

*Try...CppUTest
for C and C++*

Avoid...Unit testing by a separate person

Implementation and unit testing are not separate phases—they are done at the same time. Avoid separating unit testing from implementation.

Unit testing is done by the developer who does the implementation while he does the implementation. Separating them would lead to many small handoffs—enormous waste.

Try...C++ xUnit framework for C

Use a C++ xUnit framework, such as CppUTest³⁹, for testing C code. These frameworks use static initialization for automated registration of tests. This makes adding new tests easier. C xUnit frameworks either require changes at three places for adding a test or a script for generating that code.

Avoid...CUnit

38. JUnit — www.junit.org; CppUTest — www.cpputest.org. Search the Web, and en.wikipedia.org/wiki/List_of_unit_testing_frameworks

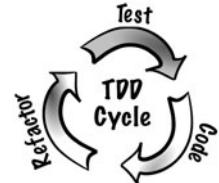
39. CppUTest uses a restricted C++ set making it portable even to old embedded C++ compilers.

The only reason for using a C xUnit framework is when the cross-compiler does not support C++ and you wish to run the unit tests on the target environment.

Try...Test-driven development

Test-driven development is a development style that drives the design⁴⁰ by tests developed in short cycles of:

1. Write one test.
2. Implement just enough code to make it pass.
3. Refactor the code so it is clean.



In a language such as Java, this cycle is as short as five minutes. In older languages, with slower compilation and less automated refactoring support, this cycle is longer—perhaps twenty minutes.

Is test-driven development different in large product development? No. It is an individual developer practice and the number of people in the development does not matter.

The amount of legacy code, old technology, and embedded development does have an impact on unit testing and test-driven development. Therefore, most experiments in this section are related to these.

Try...Use TDD coaches

When a client of ours reviewed a draft of the companion book, he mentioned that we ought to stress coaching more. “One of our mistakes is that we didn’t provide enough coaching,” he said. Though we agreed with him, we pointed out that since we are both consultants

40. It does *not* drive the design by tests alone. Developers who use TDD still have design discussions around whiteboards to talk about the overall design structure and to create a common understanding. See the *Design* chapter for more.

and provide such coaching, this advice would not be very credible. We might as well add an experiment “Try...Hire us.” Thus, we minimized the advice related to hiring coaches.

But related to test-driven development, we cannot stress strongly enough: Hire coaches! Adopting TDD means unlearning traditional programming and relearning how to design and code. We rarely meet people who were able to adopt this by self-education. Most developers need a coach to pair-program with them for days or weeks. The coach constantly reminds them to write the tests first and to *really* clean up the code—including the *test* code. He helps them apply TDD and refactoring to their real code.

Test-driven development might be the hardest agile practice to adopt, but it is also one of the biggest opportunities for improving the quality of the design and code. Hire coaches!

Try...Internal and external coaches

External coaches are needed when adopting TDD because the competence does not yet exist inside the company. But, over time, growing internal coaches reduces the dependence on externals and the cost of coaching.

That said, we have seen several attempts fail to develop internal coaches. Some reasons:

- No structure was in place to decide when and with which teams to work.
- No time was reserved for coaching. Instead the internal coaches were asked to do *normal* development.
- Developers were less eager to learn from internal coaches...you are never a prophet in your own land.
- Coaching skills were not appreciated and further developed. The result is that skilled internal coaches often leave to be an external coach.

Choose both internal and external coaching. Depending on either of them alone is risky but combining them can lead to good results.

Avoid...Write your own xUnit framework

“Why reuse if you can write it yourself?” seems to be the motto of some large product groups.

We have encountered several product groups that write their own unit test framework. At best, this is a waste of time. But often, the framework was buggy, slow, and led to skepticism toward unit testing. Avoid this and instead use an open-source framework.

Try...Use a unit test framework in a compatible language

Try...Write your own xUnit framework

A group we worked with had a 30-year-old codebase of which large parts were written in their own proprietary programming language. The code in this language could be called through a C interface and therefore it was possible to write unit tests in C. But it was uncomfortable for developers to write their unit tests in a different programming language. Instead, they built their own xUnit framework—when none is available, build your own.

Try...Dual targeting

See “Try...Create a low-level hardware abstraction layer (HAL) API” on p. 320.

Design your code to target multiple hardware platforms—dual targeting [Grennings10]. How? Separate hardware-dependent code, third-party library calls, and inline assembler. When the code runs on multiple platforms, then you can...

Try...Run tests on the development environment

The development environment and the *target* environment are different when you are creating embedded software. Code is cross-compiled and executed on either a simulator or the real hardware—the execution speed is slow.

To speed this up, run the unit tests on the development environment instead of on the real hardware. This speeds up the development cycle, which in turn makes test-driving your code more enjoyable.

We coached a team that developed digital signal processing (DSP) software for a Texas Instruments DSP processor. We suggested they run their tests on Linux. They proclaimed *we were nuts*. For two

days we worked with them to separate their hardware dependencies and they were able to run their unit tests on their development environment.

Another advantage of running tests on the development environment is that it allows more control over the environment. This makes it possible to test failure cases that are difficult to produce in the real hardware, such as hardware failure or out-of-memory tests. One of the first tests we wrote for the DSP processor was an out-of-memory test where we let *malloc* return NULL. The software crashed. We asked the team whether this scenario could ever happen in real life on the real hardware. They answered, “All the time...”

Try...Run tests on the real hardware

Running tests on the development environment can give a false sense of security that everything works. Therefore, also run the tests on the real environment to discover these differences...

- *Size of data types*—The size of basic data types is not always the same on different hardware. This leads to range overflows and problems when overlaying memory.
- *Endian*—The byte-ordering is frequently different between a development environment and the real hardware. This might result in defects when data is extracted from a byte-stream message—a common action.
- *Compiler differences*—Compilers generate different code. Usually this does not create problems but... compilers have bugs, especially small-market embedded compilers. These differences—especially those due to bugs—create nasty defects that can cause days of debugging. *Extra caution:* Legacy code might depend on a compiler bug and stop working when ported to the development environment or when the compiler is upgraded.
- *Memory models*—Intel processors are *excellent* in backwards compatibility—they support many memory models. Legacy code might still use a segmented memory model with different pointer sizes and different ways to access memory.

- ❑ *Data alignment*—Compilers pad structures in order to align data access on processors word boundaries for faster memory access. Differently aligned memory can cause problems when raw memory is converted into data types.
- ❑ *C library*—The C library is perhaps the most standardized software in the world. Still, different C libraries behave differently. For example, CppUTest stopped working on Apple’s Mac OS Snow Leopard while it worked on the previous release. On the new release, the vsnprintf behavior was slightly different.

Developers can find many of these incompatibilities by running the unit tests every now and then on the target hardware. This running of unit tests on the real hardware can be done automatically by the continuous integration system so that developers are informed quickly when they check in code with a portability issue.



Doing this sometimes requires extra work. For example, the target hardware usually has less memory than the development environment. Linking all unit tests into one executable might cause them to use too much memory. In this case, the tests need to be split into multiple executables—painful but worth doing.

Try...Function-to-function-pointer refactoring

It is more difficult to add tests to C legacy code. It is all tangled up. The C language does not have a way of overriding methods—as used in object-oriented languages for breaking dependencies. But the same dynamic flexibility is possible in C by applying the function-to-function-pointer refactoring.

This refactoring changes a function to a function pointer so that it can be replaced at run time. The refactoring can be done safely without other changes because a function-pointer call is syntactically the same as a function call and the function pointer can be initialized by the default implementation.

For example, this header file

```
int send_message(int number, message_t* msg);
```

and the related C file

```
int send_message(int number, message_t* msg)
{
    /* do something */
}
```

After the function-to-function-pointer refactoring is applied, the header file becomes

```
extern int (*send_message) (int number, message_t* msg);
```

and the related source code becomes:

```
int send_message_impl(int number, message_t* msg)
{
    /* do something */
}

int (*send_message)(int, message_t*) = send_message_impl;
```

The rest of the code is not impacted and now the function call can be replaced at run time.

One danger in replacing the function at run time is forgetting to restore it. As a result, the fake function would be called in all the following tests. There are solutions; for example, CppUTest can set pointers in the setup such that they are automatically restored to their original value in the teardown.

```
void setup()
{
    UT_PTR_SET(send_message, send_message_fake);
}
```

This simple but powerful refactoring makes it easier to gradually cover legacy C codebases with automated unit tests.

Try...*Learning* tests

Learning tests are tests written with the sole purpose of learning about the system. We sometimes see developers stare at code for hours wondering what it does, while they could discover this by writing a small test, running it, and observing the result.

One time, we programmed with a developer who needed to use a new XML parser with an awkward interface.

The previous days he had read the documentation and the code, but he still could not understand how to use it. He random-programmed⁴¹ for a while but could not get the code to work.

After observing him for a few hours, we suggested that he write learning tests for the XML parser.

He answered, “It’s not my job to test the XML parser.”

We clarified that learning tests are *not for testing but for learning*.

He reluctantly agreed to try it out. After one hour of writing learning tests, we learned enough and felt confident enough to switch to using the parser in the production code.

Should you throw away learning tests after you learned?

Though they served their primary purpose, it might be beneficial to keep them. Running them checks that all you have learned about the system is still true.

For example, when upgrading the XML parser, run the learning tests to discover what assumptions have changed.

Try...Learning tests for hardware

Hardware documentation is often incomplete and hard to understand.

Embedded-software developers sometimes base their work with hardware on trial-and-error—call the hardware and observe what happens. This takes a lot of time, and the learned behavior is not documented after the discovery.

41. Random-programming—Randomly trying out things without thinking, in the hope that they will work.

Learning tests are excellent for learning about the hardware. Instead of experimenting with the hardware in production code, try writing learning tests for the hardware.

A major benefit of this is that you can rerun the tests when you receive a new version of the hardware.

This is important: traditionally, embedded developers are often hesitant to upgrade because the new hardware might contain bugs they must chase for days.

Now they can run the learning test to check whether all the assumptions they have about the hardware are still valid.

This creates confidence that results in more frequent software-hardware integration—eventually becoming continuous hardware integration.

Try...Refactor tests

“Ouch, that code does not look very clean,” we said to a developer at the start of a pairing session. “Yes, but that’s test code!” he replied.

In response, we suggested:

Test code is as important as production code.

The same standards of cleanliness need to be applied.

Therefore, refactor test code as much as production code. We often encounter groups spending an extraordinary amount of time maintaining the tests. This sometimes results in the developers not updating them anymore.

The cause: ugly, messy, dirty test code. *Treat test code well.*

Duplication

Taiichi Ohno considered overproduction the worst of all manufacturing wastes—it hides the other wastes [Ohno88]. In software development, duplication is the worst of all the code smells. According to software craftsman Bob Martin, “Duplication may be the root of all evil in software” [Martin08].

Duplication is the number one trigger for refactoring. Learn to see duplication. It does not just mean the copy-paste-modify variety (probably the most common way of producing code in many companies) but also means the duplication in logic. Two pieces of code can do the same thing in different ways and that would still be duplication.

Some things to look for:

- similar values loops of the same length
- similar variable names duplication between test and production code
- similar code structures similar error handling clauses

This is not an exhaustive list. As Toyota teaches “eyes for waste,” so should you develop “eyes for duplication.”

Try...Small tests that test only one thing

Tests should test one thing.⁴² A unit test is usually not longer than 10 lines of code (LOCs). Larger tests probably contain duplication or are multiple tests written as one.⁴³

“Wow!” we exclaimed while looking at a 5000-LOC test file while pair-programming with a developer. Most individual tests were larger than 150 LOCs—they were huge! Worse, they were so unreadable that we had no clue as to what they were testing. We decided to clean them up.

42. This does not necessarily mean having only one assertion. See [Meszaros07] for further discussion.

43. An awkward unit test framework is sometimes the cause. For example, CUnit requires a change at three places (source, header, registration) to add test. The result is that developers make larger tests.

Two days later: There were 1500 LOCs left in the file. Several tests turned out to test the same thing but nobody had noticed because of the complexity and duplication. Two tests did nothing—we deleted them. The average LOCs per tests was seven lines and most importantly, we could understand them—and maintain them.

Avoid...Slow unit tests

Slow tests cause developers to stop executing them. Therefore, make sure the tests run in seconds.

We noticed the developer we paired with did not run his tests frequently. The reason: they took thirty seconds. We persuaded him to find out why they were so slow and quickly discovered that a three-megabyte structure was allocated in all setups and freed in all teardowns, yet most tests did not even use it. After refactoring, the tests ran in eight seconds—still too long—but the developer started running them again in the compile cycle.

Pay attention to the speed of the tests. Frequently, test runs can be optimized with just a little extra effort—and the payback is enormous.

EXAMPLE: ROBOT FRAMEWORK

This section presents a brief example of Robot Framework that uses a system we worked on years ago. The original development did not use A-TDD or Robot Framework—the example is new, the system is old. The system is medium-sized,⁴⁴ yet the example demonstrates the key points that are also valid in larger development.

Robot Framework is a *keyword-driven*⁴⁵ test automation framework created by Pekka Klärck⁴⁶ [Laukkanen06] at Nokia Siemens Net-

-
- 44. Larger products tend to come from a more complex or unfamiliar domain—making it hard to use as an example in a few pages.
 - 45. Keyword-driven test frameworks use keywords in data to determine the action to take on the data [FG99]. Keywords are sometimes called action words [BJP02].
 - 46. Formerly known as Pekka Laukkanen.

works in 2004. One of its early design goals was A-TDD support. It was open-sourced in 2008 and is available at www.robotframework.org.

The product used in this case study is a conference information system built for a convention center. Access to the system is available at “information pillars” distributed throughout the convention center. Visitors can use it to find information about the current conference or the convention center, such as

- Where can I find the booth of vendor X?
- How do I get there?
- Where is the nearest restaurant?
- What did other visitors have to say about the conference?

These pillars are maintained and controlled centrally. The information inside the system *must* come from the *existing* conference preparation process. Not much additional preparation work is allowed when the system is updated for a new conference—the new system must adapt to the existing systems and ways of working. And, of course, the system must be *flashy* and *shiny*—sound, graphics, and other bells and whistles.

Example One: The Vendor List

The first example is simple but demonstrates some key points. The customer requested the ability to list all the vendors and display them in “a nice list.” After abstract discussion about what “a nice list” means, *we asked for an example*. The first example was a workflow, one (Figure 3.13) with three vendors in the database shown in an alphabetically ordered list with three columns.

Example: Robot Framework

After asking for more examples (not shown), we discover that the provided data in the supplied database is not consistent—the information system will have to compensate for that because we have no control over the database. For example, there can be duplicate entries with minor differences (with and without a logo).

Describing all the data compensations with *workflow examples* leads to myriad similar tests, so we switch to *data-driven tests* for these particular *business rules* by asking, “What data in the database will lead to what nice vendor list?” The results are shown in Figure 3.14 and include examples for alphabetical sorting, removing duplications, and preferring logo over no-logo entries.

Figure 3.13 workflow example for selecting vendors

1. Request Show all Vendors
When 3 Vendors, Apple, IBM, MS

Logo	Name	Stand#
Apple	Apple	A1-01
IBM	IBM	B2-03
MS	MS	A1-07

Figure 3.14 data-driven test for vendor lists

Logo	Name	Stand#	Logo	Name	Stand#
Apple	Apple	A1-01	Apple	Apple	A1-01
IBM	IBM	B2-03	IBM	IBM	B2-03
MS	MS	A1-07	MS	MS	A1-07
IBM	IBM	B2-07	Apple	Apple	A1-01
APPLE	APPLE	A1-01	IBM	IBM	B2-03
MS	MS	A1-07	MS	MS	A1-07
Apple	Apple	A1-02	Apple	Apple	A1-01
Apple	Apple	A1-01	Apple	Apple	A1-02
IBM	IBM	B2-03	IBM	IBM	B2-07
100 vendors					
Apple	Apple	A1-01	Apple	Apple	A1-01
Apple	Apple	A1-01	IBM	IBM	B2-03
IBM	IBM	B2-03			

Intermission—Robot Framework overview

The next step is to *distill* these examples into Robot Framework tests. But how does Robot Framework work?

Robot tests are written in tables with HTML, TSV, reST⁴⁷ or plain text. HTML is the most commonly used format whereas Robot Framework uses only the tables and ignores all the additional text—which can be used for documentation. There are four types of tables:

- ❑ *Test case tables*—contain the actual test cases. The first field of these tables must contain “Test Case” or “Test Cases.”
- ❑ *Keyword tables*—contain lower-level user keywords that can be used to construct test cases. The first field must contain ‘Keyword’ or “User Keywords.”
- ❑ *Settings table*—allows for importing files and defining metadata. The first field must contain ‘Setting,’ or ‘Settings.’
- ❑ *Variable table*—declares variables containing global data that can be used elsewhere. The first field must contain ‘Variable,’ or ‘Variables.’

Figure 3.15 coffee table test

Test Cases	Action	Arg
Drink coffee	Drink coffee in liters	10
	Is physical health	OK
Drink coffee over capacity	Drink over the max amount of coffee	
	Is physical health	NOK

Figure 3.15 shows a test case table with two test cases in it—the first column. The second column contains the keywords, and the remaining columns are for passing arguments to these keywords.

Robot Framework has two types of keywords: *user keywords* and *library keywords*. A user keyword is implemented in a keyword table, whereas a library keyword is implemented in a piece of glue code between the test and the system under test—called a **test library**. Executing Figure 3.15 fails because it contains three undefined keywords—Drink coffee in liters, Is physical health, and Drink over max amount of coffee.

47. TSV are tab-separated-values files. ReStructuredText (reST) is a markup language commonly used for documentation in Python projects.

Example: Robot Framework

Figure 3.16 implements the “Drink over the max amount of coffee” user keyword as drinking one liter more than the maximum. \${MAX COFFEE} is a variable defined in a variable table (not shown).

Keywords	Action	Arg
Drink over the max amount of coffee	Drink coffee in liters	\${MAX COFFEE}
	Drink coffee in liters	1

Figure 3.16 keyword table for max coffee

Executing Figure 3.16 fails because two keywords are undefined. These are low level and implemented as library keywords in Java (or alternatively, in Python).

```
public class CoffeeTestLibrary
{
    Human humanUnderTest = new Human("Bas");

    public void drinkCoffeeInLiters(Integer liters) {
        humanUnderTest.drinkCoffee(liters);
    }

    public void
    isPhysicalHealth(String expectedHealth) throws Exception
    {
        if (!expectedHealth.equals(humanUnderTest.checkHealth()))
            throw new Exception("Health problem. Expected health: " +
                expectedHealth + " but actual health was " +
                humanUnderTest.checkHealth());
    }
}
```

Figure 3.17 is an overview of Robot Framework. Test case and user keyword tables are fed to Robot Framework. The framework calls the test libraries, and they call the system under test. More information can be found in the user guide [Robot09].

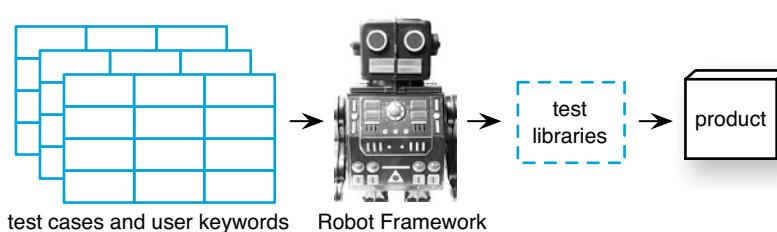


Figure 3.17 Robot Framework architecture

Continuation—List the vendors

We *distill* the test from the example on the whiteboard. The ‘normal’ example is removed because it is of the same *equivalence class* as the ‘sorted’ test.⁴⁸

These test cases are executable—but they fail. Robot Framework complains that the keywords “Stand input,” “Is stand output,” and “Has No Extra Stands” are undefined.

Figure 3.18 lists all vendor test case table

Test Case	Action	Logo	Name	Place
Sorted				
	Stand input	x	IBM	B2-03
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	IBM	B2-03
	Is stand output	x	MS	A1-07
	Has no extra stands			
Same vendor, different place				
	Stand input	x	Apple	A1-02
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	Apple	A1-02
	Is stand output	x	MS	A1-07
	Has no extra stands			
Duplicate entry, logo plus non-logo version				
	Stand input		Apple	A1-01
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	MS	A1-07
	Has no extra stands			

48. The keyword-driven nature of Robot Framework is evident from the repeating keywords in data-driven tests. In Fit this would not be necessary. This will be enhanced in the future versions.

Example: Robot Framework

The keywords can be implemented as user or library keywords. We chose to implement the “Is stand output,” and the “Has No Extra Stands” as user keywords (Figure 3.19).

Keyword	Action	Arg	Arg	Arg
Is stand output	[Arguments]	<code> \${expected_logo}</code>	<code> \${expected_name}</code>	<code> \${expected_place}</code>
	<code> \${actual_logo}=</code>	Get current logo		
	Should be equal	<code> \${expected_logo}</code>	<code> \${actual_logo}</code>	
	<code> \${actual_name}=</code>	Get current name		
	Should be equal	<code> \${expected_name}</code>	<code> \${actual_name}</code>	
	<code> \${actual_place}=</code>	Get current place		
	Should be equal	<code> \${expected_place}</code>	<code> \${actual_place}</code>	
	Increment stand index			
Has no extra stands				
	<code> \${stands_left}=</code>	Stands left		
	Should not be true	<code> \${stands_left}</code>		

Figure 3.19 list all vendors user keyword table

The first row declares the three arguments of the “Is stand output” keyword. The next rows assign the output of the “get current logo” keyword to the `${actual_logo}` variable and compare that with the `${expected_logo}`. The same steps are repeated for the other expected values. At the end, we increment the stand index.⁴⁹

When we now run the tests, Robot Framework complains about five undefined keywords: stand input, get current logo, get current name, get current place, and stands left. These keywords will be implemented in a test library.

Our system under test is written in C. We can call it through the user interface (not recommended), or we can call C code directly. We chose the latter and implemented the test library in Python because we can easily call C code directly from Python by using the `ctypes` foreign library.⁵⁰ The test library code:

49. This is a side effect in the keyword implementation which should be there but is there to keep the higher-level table simpler

```

from ctypes import *

class conferencekeywords:

    def __init__(self):
        self.conf = cdll.LoadLibrary("stands.so")
        self.conf.init()
        self.stand_index = 0
        self.logo_mark = [" ", "x"]
        self.conf.get_place_at_index.restype = c_char_p
        self.conf.get_name_at_index.restype = c_char_p

    def increment_stand_index(self):
        self.stand_index = self.stand_index + 1

    def stands_left(self):
        return self.conf.stand_output_at(self.stand_index)

    def stand_input(self, logo, name, stand):
        has_logo = logo == "x"
        self.conf.add(has_logo, c_char_p(name), c_char_p(stand))

    def get_current_logo(self):
        logo = int(self.conf.get_logo_at(self.stand_index))
        return self.logo_mark[logo]

    def get_current_name(self):
        return self.conf.get_name_at_index(self.stand_index)

    def get_current_place(self):
        return self.conf.get_place_at_index(self.stand_index)

```

The code is trivial—the only thing it does is call the C interface.

The system under test is linked into a shared library and loaded by the *cdll* ctypes call. The output is assigned to the *conf* variable and is used to call the C functions in the shared library. The additional code specifies the parameter and return types; this must be done when they are not integers—the default.

When we run the tests, they will pass—if the functionality is implemented.

Example Two: Importing AutoCAD Files

The floorplan department uses AutoCAD to create conference layouts. These contain the location of the stands and everything else related to the conference—including weird symbols and electrical

50. Works excellently for C. For C++ you may prefer to use SWIG or Boost Python.

outlets. These CAD drawings, however, *were made for humans to read.*

A requirement for the conference information system was to show a conference map. Visitors can select a location on the map and the system will show them the vendor. The convention center required the system to import the AutoCAD DWG files that were created by the floorplan department. Reading AutoCAD files is slow, so the files are converted to an internal format that stores only the needed information—it discards *noise* such as electrical outlets.

The DWG contains lines that form shapes. A shape with a number written in it is probably a stand. The system needs to read the file, form the shapes, recognize the stands, and discard irrelevant information. This is not too hard... except that the data is inconsistent—not made for computers. The shapes are not always *exactly* closed, the number is not always *exactly* in the stand, and so forth.

In a requirement workshop, we discussed the different possible inconsistencies that the system should support. We started with *abstract descriptions* and moved toward *examples*, as shown in Figure 3.20.

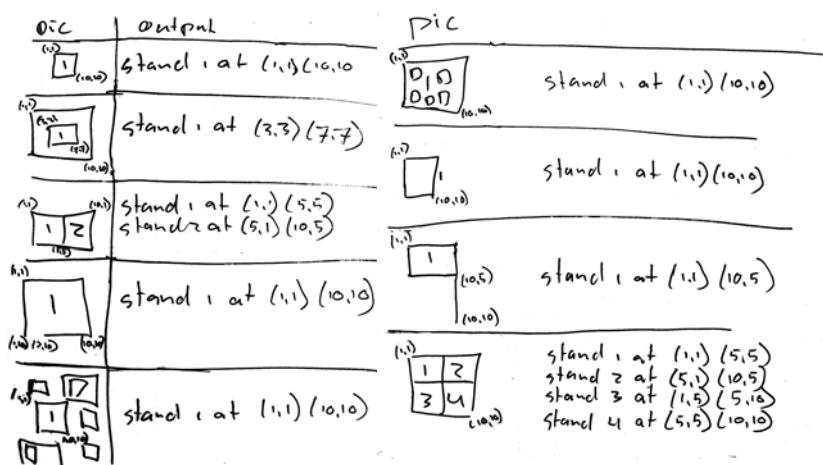


Figure 3.20 examples of rectangular stands

The examples are drawings of shapes. The *discussion* was about distinguishing stands from noise and dealing with inconsistencies in the drawing.

The tests *distilled* from the examples are shown in Figure 3.21. The tests contain an AutoCAD input file and the expected output file. We added the pictures to the tests for documentation. Robot Framework ignores all information except for the text inside the table. This allows us to add documentation to our specifications tests.

Figure 3.21 tests for stand recognition

Test Case	Action	Original	Result
Recognize rectangular stands	Detect and Check stands	simple_rectangle.dwg	simple_rectangle.fpm
	Detect and Check stands	rect_in_rect.dwg	rect_in_rect.fpm
	Detect and Check stands	two_rects.dwg	two_rect.fpm
	Detect and Check stands	non_closed_rect.dwg	non_closed_rect.fpm
	Detect and Check stands	rect_with_noise.dwg	rect_with_noise.fpm

The “Detect and check stands” keyword is implemented as a user keyword shown in Figure 3.22.

Example: Robot Framework

Keyword	Action	Arg	Arg
Detect and Check stands	[Arguments]	\${dwg_drawing}	\${fpm_result}
	\${fpm_output_file}=	Convert to FPM	\${dwg_drawing}
	Compare FPM files	\${fpm_output_file}	\${fpm_result}

Figure 3.22 stand recognition user keywords

The test library calls the C interface of the system under test (not shown).

Example Three: Leaving Messages

This last example is an example of a workflow test. Visitors need to be able to leave messages for one another on the conference information system. During a *discussion* in a requirement workshop, we moved from the abstract description into *workflow examples*. The results are shown in Figure 3.23.

1. select leave message
 2. insert name "Bas"
 3. insert topic "Great Conference"
 4. insert message "Party afterwards?"
 5. save
 6. show saved message
-
1. select leave message
 2. save
 3. error message "you should insert name"
-
1. select leave message
 2. insert name "Bas"
 3. save
 4. error message "you have no topic"

Figure 3.23 examples of messaging workflow

The *distilled* tests ended up the same as the wall-workflow examples. They are shown in Figure 3.24.

Figure 3.24 workflow example

Test Case	Action	Arg
Leave a message succeeds	Select leave a message	
	Insert name	Craig
	Insert topic	Great conference
	Insert message	Party afterwards?
	Save	
	Is message	\${NORMAL EXPECTED MESSAGE}

These actions were each implemented as user keywords so that they can be reused for future workflow tests. This minimizes the duplication between workflow tests and reduces the maintenance effort. We skip the user keywords and the test libraries; they are similar to the previous examples.

Robot Framework conclusion

These examples showed some core features of Robot Framework. Some other features worth mentioning are

- Ability to classify tests by using tags. These can be used for reporting, statistics, or selective test runs.
- Clear logs and reports that make it easy to discover what happened.
- Easy integration with other systems such as SCM systems.
- An IDE for developing the tests—RIDE.

CONCLUSION

This is a long chapter, but testing is not what it used to be. The barriers between testing and programming have to be demolished. Testing and programming are two sides of the same activity done by the same people. The purpose of the tests changes from finding defects to preventing them by writing the specifications as tests; and from checking the implementation to driving the design. This fundamen-

tally different perspective leads to vital changes in the way people work—and work together.

We categorize testing as customer-facing and developer tests.

The focus of *customer-facing tests* changes from traditional testing to collaboratively clarifying the requirements with techniques such as A-TDD. These techniques are used not only for simple requirements but also for complicated non-functionals—common in large-scale development. These automated tests become the executable specifications. Nobody is perfect—people make mistakes. Techniques such as exploratory testing harness creativity and are a more *human* approach for finding defects than the traditional design-a-script-then-manually-run-the-script approach.

The focus of *developer tests* changes from traditional testing to driving the design. The distinction between testing and implementation disappears. Test-driven development turns traditional development upside down and uses the tests to incrementally drive the design. These techniques are widely used in modern programming languages, but are just as applicable and valuable for large legacy code products with proprietary programming languages and embedded C software. The practices are slightly different, the cycle is slower, but the concepts are exactly the same.

Robot Framework is a promising open-source tool that supports an A-TDD style of requirements clarification. Worth trying out.

RECOMMENDED READINGS

A vast number of books and articles exist on the subject of testing in agile development. We grouped recommendations into similar categories as the structure of this chapter.

Some recommendations related to “thinking about testing”:

- *Agile Testing*, by Lisa Crispin and Janet Gregory. A great overview of the role of testing in agile development. It covers the challenges organizations face when adopting agile develop-

ment and also describes the concrete role of testing during the iteration.

- *Lessons Learned in Software Testing*, by Cem Kaner, James Bach, and Bret Pettichord. This book describes the lessons learned from decades of experience in testing and also introduces the context-driven school of thinking in software testing.
- *Agile Testing Directions*, by Brian Marick. A series of blog posts wherein Brian Marick introduces the agile testing quadrants.

Some texts about general testing techniques for improving testing skills. These are unrelated to agile development:

- *A Practitioner's Guide To Software Test Design*, by Lee Copeland. An easy-to-read catalog of test design techniques.
- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen. A thorough coverage of different test design techniques. Starts off with mathematics for testing.

Not very much has been written on the subject of A-TDD and related tools. The following books are recommended, though:

- *Bridging the Communication Gap*, by Gojko Adzic. At this moment, Gojko's book is the only book purely on the subject of A-TDD (which he calls agile acceptance testing). It has a strong focus on requirements clarifications and workshops.
- *Acceptance Test Driven Development: An Overview*, by Elisabeth Hendrickson. A blog post and related paper providing an overview of A-TDD by giving a detailed example of using Robot Framework.
- *Fit for Developing Software*, by Rick Mugridge and Ward Cunningham. This book has a strong focus on improving the communication of requirements by means of Fit tables.
- *Robot Framework User Guide*. Does not cover A-TDD but does provide an excellent introduction to the Robot Framework tool.
- *Test-Driven .NET Development with FitNesse*, by Gojko Adzic. This book has less emphasis on A-TDD and more on FitNesse. But it does a good job in describing the tool.

Exploratory testing is also a topic rarely covered in literature. Some recommendations related to this:

- ❑ *Exploratory Testing Explained*, by James Bach. An article available on the web; it is the classic reference related to this subject. Definitely worth reading.
- ❑ *Exploratory Testing in an Agile Context*, by Elisabeth Hendrickson. A freely available mini-book related to the role of exploratory testing in agile development. Easy to read.

A lot has been written on the subject on developer testing and test-driven development. Our recommendations:

- ❑ *Test-Driven*, by Lasse Koskela. A well-written thorough book on the subject. It uses Java and also covers A-TDD.
- ❑ *Test-Driven Development*, by Kent Beck. A classic and one of the first books on the subject.⁵¹ It uses Java.
- ❑ *Test-Driven Development in Microsoft .NET*, by James Newkirk and Alexei Vorontsov. A good introduction to TDD in .NET.
- ❑ *xUnit Test Patterns*, by Gerard Meszaros. More than you ever wanted to know about xUnit.
- ❑ *Test-Driven Development in C: Modern C Programming for Embedded, Mobile, Open Source and You*, by James Grenning. Does this work for embedded software? Yes. James discusses how to use TDD when developing embedded software. Not yet published.
- ❑ *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce reinforces the value of evolving design based on feedback from tests.

51. The first book about test-driven development was in German, written by Johannes Link and published at the end of 2001 [Link01].

Chapter

- Thinking about Product Management 100
- Product Owner 120
- Many products 128
- Many Teams 132
- Prioritization 139
- Customers and R&D 145
- Change and Improvement 147

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

PRODUCT MANAGEMENT

He taught me housekeeping; when I divorce I keep the house.
—Zsa Zsa Gabor

Sun streamed in through the windows while about 300 people at a client settled in for an introduction to large-scale Scrum and lean thinking. We discussed how these change the opportunities not only for R&D but also for business as a whole, including *product management*. Afterwards, one of the in-house agile coaches was summoned before the person “responsible for the product-management process.” The process ‘owner’ threatened to have the coach fired if “agile people” mentioned such ideas again. “I am in charge of how product management works, and there should be no interference between departmental processes!”

That silo mentality does not best serve a business or its customers; agile and lean principles and practices can be applied not only in development, but also in product management—and beyond.

Chapter scope—This is not a primer on product management or even agile product management; most—not all—topics are at the intersection of product management, scaling, and Scrum. For more, see the recommended readings at end of chapter.

Terminology—In this book—and this chapter—*product management* (P-M)¹ refers to one of

- when it exists, true product management
- a marketing group that fulfills P-M responsibilities

1. ‘P-M’ is also used to denote *product manager*.

- when Scrum is used for internal products or applications, someone from the business unit that represents the customers or users as Product Owner

Research and development (R&D) refers to the organization usually called R&D, engineering, development, or IT.

On to the first experiment...

THINKING ABOUT PRODUCT MANAGEMENT

Try...Exploit *business* advantages of Scrum & lean thinking

Distinct functional departments are the norm in big organizations. A P-M group separate from R&D may assume that when R&D starts adopting Scrum, that it is for them and has little or no relevance to P-M...“You guys are welcome to Scrum. We hope you’ll be more efficient.” However, there is a bigger picture...

Although an oversimplification, a summary of the purpose of lean thinking is this well-known quote by Taiichi Ohno:

All we are doing is looking at the time line from the moment the customer gives us an order to the point when we collect the cash. And we are reducing that time line by removing the non-value-added wastes. [Ohno88]

There are twelve agile principles; the first is

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*

These phrases are business themes: *order, cash, satisfy the customer, early, valuable*. Some people, as in the chapter-opening story, believe lean and agile principles are solely aimed inward. Not so.

Lean thinking and agile principles are customer-focused for business success; they are not “development processes.”

These principles intersect with P-M, development, operations—and more as explored in *Organization* in the companion book. They reflect systems thinking and cross-functional integration.

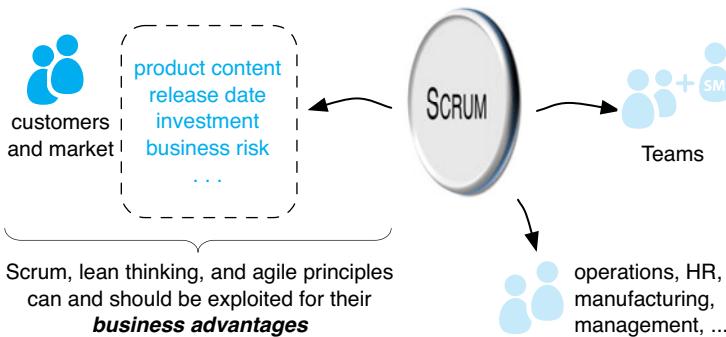


Figure 4.1 business-facing change opportunities when adopting Scrum

There are many sides to the Scrum coin and to lean thinking (Figure 4.1); they intersect with business factors and offer new kinds of control—arguably increased control—over these. This control and flexibility comes from these key Scrum elements:

- Short iterations increase transparency and control.
 - replaces sequential life-cycle development
- Adaptive planning (inspect and adapt) increase flexibility.
 - replaces predictive planning: the Contract Game, traditional project planning, ...
- Product management steering development increases business-centric control, and increases R&D teams' appreciation of customer needs.
 - replaces the Contract Game and layers of indirection



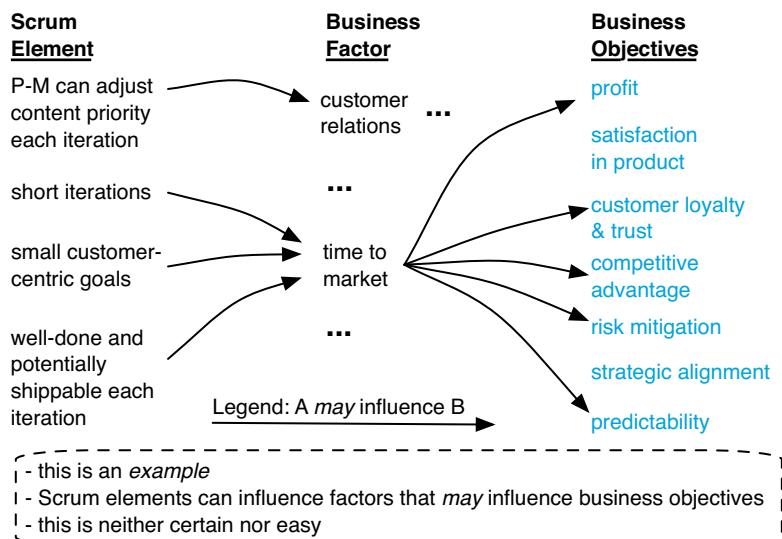
Agile product management is akin to driving an all-terrain vehicle to explore new uncharted territory: P-Ms have the steering wheel of product creation in their hands, are driving a vehicle with a clear view and responsive steering, and are adapting as

they drive and discover new easy paths or roadblocks. Plus, although they originally planned to stop at *that* big mountain, they discovered *this* mountain along the way—and it satisfied their latest goals of tranquility, fish, and cold beer.²

Magic-free zone

We do not mean to imply a simple relationship between Scrum elements and meeting business objectives; there are few guarantees to success in a complex system. That said, these elements can influence various business factors, and maybe—just maybe—contribute to major objectives (Figure 4.2).

Figure 4.2 Scrum may contribute to business objectives



2. Traditional development is like [product management] being the passenger on a bus tearing down a serpentine mountain road while being driven by a drunk. [Schwaber07c]

Areas and opportunities

What areas may intersect with Scrum? Here is some of the territory of P-M, as summarized in *Product Management* [LW05]:

product vision	business plan	customer relations	profitability
sales support	ideation, innovation	market testing	risk management
marketing planning	financial analysis	market research	product review
requirements	pricing	product launch	forecasting
channels	advertising	supply chain	competitor analysis

The following table considers some of these areas, asking where the flexibility and control offered by Scrum may have a positive impact.

product vision, ideation	business plan, market testing
<ul style="list-style-type: none"> • Early demonstration and field testing either validate the vision or provide feedback for changing it. Adaptive planning supports adjusting the product vision. • Vision improved by increased collaboration with teams, including during initial creation. 	<ul style="list-style-type: none"> • The vision may be ‘great,’ but what revenue will it actually pull? Is the business plan realistic? Early release or early market testing provide data to support or invalidate a business thesis.
customer relations	profitability
<ul style="list-style-type: none"> • Customers or potential customers can provide early, repeating feedback, and see the results of their feedback in future demos. This may support positioning, satisfaction, and buy-in. • Increased transparency, responsiveness, and collaboration can build trust. Transparency improves by progress measured and demonstrated in terms of customer-centric features (that they care about), not technical tasks. • Early confirmation of problems that will impact customers (such as delay) offers more degrees of freedom: timely internal experiments to improve, or prompt communication with customers to seek alternatives. 	<ul style="list-style-type: none"> • Fine-grained re-prioritization occurs each iteration to front-load the product with items that help drive profit for the least cost. These adjustments are based on estimates revised each iteration and on customer feedback. • Low-impact items become low-priority items; so it may be possible to avoid their creation—steering clear of the cost of their creation and the cost of their long-term <i>care and feeding</i>. • Time to market improves since the product may ship any iteration (from the first), which could support earlier revenue or capturing market share^a.

sales support	risk management
<ul style="list-style-type: none"> Suppose that closing a major deal is aided by either the existence or soon-existence of a specific feature or theme of features. Raising their priority is simple. 	<ul style="list-style-type: none"> High-risk items increase in priority and are implemented early, attacking the risk <i>before it attacks</i>.

- a. For embedded-software products adopting Scrum, the constraint of release date will depend on the agility of hardware engineering and manufacturing. And, operational or field support... One of our clients builds ship-control systems; even if the system is ready, an installation crew may not be. But in real cross-functional spirit, this client is exploring R&D members joining installation crews on the ship to increase learning and reduce bottlenecks.

Exploiting the business advantages means some changes in behavior for product management. What are these?

Try...Understand the changes with Scrum & lean thinking

“We don’t have time to meet with teams; we meet customers.”

This was said to us by one P-M group we worked with. Customers are important, and yet they might be even better served by directly connecting them and teams, and by P-M collaborating regularly with teams. These are example adjustments when adopting Scrum.

Predominately, the changes involve an increase in transparency, collaboration, inspect and adapt, and control. Plus, working or making decisions in small batches and short cycles. Concretely, what are *some* changes in behavior or responsibility for P-M? See Table 4.1.

These changes imply on a larger scope that the product management and R&D relationship shifts from *contract negotiation* to *customer collaboration*, as explained in the next two suggestions:

- “Avoid...Product management negotiating a “release contract” (scope & date) with R&D” section on page 106
- “Try...Product management collaborates with R&D each iteration, adapting release scope or date” section on page 116

Table 4.1 changes

product vision, ideation	customer relations
<ul style="list-style-type: none"> Broadly, an increase in testing and adjusting the vision, based on customer, market, and team feedback, or field testing. For instance, customers can participate more frequently in concrete product evaluation, since the product is shippable from the first iteration. Based on their feedback, P-M re-prioritizes and strengthens outstanding <i>themes</i> in the backlog. Or, learns to quickly abandon a deeply flawed vision (“fail fast”) and start afresh. Include the cross-functional development teams in early vision workshops, rather than only engaging them later. 	<ul style="list-style-type: none"> Re-prioritize items based on customer feedback each iteration. Ask for earlier pilots at client sites. Investigate with clients how to reduce the impact of more-frequent pilots or upgrades. Invite customers to Sprint Reviews. Invite teams to collaborate with clients. Increase transparency: Show customers the current Product Backlog priority and Release Burndown. Alert early when a problem (such as delay) is identified. Ask for feedback. Increase transparency: Ensure the Product Backlog contains customer-centric features (not technical tasks), so that progress is measured in terms relevant to customers.
release date	profitability
<ul style="list-style-type: none"> Since product is potentially shippable each iteration, P-M adjusts release date to learn faster^a, or to fit changing company, market, or competitor conditions. Attention shifts from “Is it all ready?” to, “Is now the right time to ship?” and “What to add next to get closer to minimal marketable feature set (minimum viable product)?” 	<ul style="list-style-type: none"> As the product is always shippable, a recurring question is “Is now the time to start generating revenue and/or to learn about the revenue streams and profit potential?” Each iteration, revisit the Profit Drivers and their weights in the Product Backlog, and update these. Feedback from Business Plan tests (revenue tests, ...) is salient input. Each iteration, re-prioritize the backlog to drive more early profit. Each iteration, ask the teams to refresh the Product Backlog estimates of cost for high-priority items.

marketing	team relations
<ul style="list-style-type: none"> Marketing plans and materials start earlier, in a small batch. Feedback is used to evolve them incrementally in short cycles with small sets of decisions. 	<ul style="list-style-type: none"> Invite teams to collaborate with clients. Meet with teams each iteration to (1) explain iteration goals, (2) refine the Product Backlog, and (3) review the product.

- a. “By far the dominant reason for not releasing sooner [at the startups I worked at] was a reluctance to trade the dream of success for the reality of feedback.” [Beck09]

Avoid...Product management negotiating a “release contract” (scope & date) with R&D

This idea starts with the “story of the traditional game of large-scale product development.”

(Backing up, there is often a pre-game involving customers, Sales, and Product Management, in which the latter get their own ‘contract’ of “customer promises” pushed on them from Sales. For more, see the “Avoid...Fixed content with unrealistic deadlines” section on page 335 and “The Problem” section on page 111.)

This traditional game is described in a lighthearted tongue-in-cheek way; we do not really mean to imply in the upcoming story-telling that things are black-and-white or to point fingers.

Plus, “not blaming” is especially noteworthy in the context of systems thinking and lean thinking: In the *Systems Thinking* chapter of the companion book, we shared the “laws of systems thinking” described in *The Fifth Discipline* [Senge94], and shown in Table 4.2.

Why no blame? Because people—and the policies created by them—behave in ways shaped by the system they work within. This ‘law’ does not literally imply no personal responsibility, but reminds us to be slow to judge; in a workplace system there are complex dynamics behind what appears...odd.

We are not always successful at applying this advice ourselves—we have a lot to learn, too.

Table 4.2 laws of systems thinking

- Today's problems come from yesterday's 'solutions.'
- The harder you push, the harder the system pushes back.
- Behavior will grow worse before it grows better.
- The easy way out usually leads back in.
- The cure can be worse than the disease.
- Faster is slower.
- Cause and effect are not closely related in time and space.
- Small changes can produce big results...but the areas of highest leverage are often the least obvious.
- You can have your cake and eat it too—but not all at once.
- Dividing an elephant in half does not produce two small elephants.
- **There is no blame.**

In this vein, W. Edwards Deming wrote, in his well-known *14 points for management*, that, "...the bulk of the causes of low quality and low productivity belong to the system and thus lie beyond the power of the work force." [Deming82].

And blaming is arguably waste, both for the previous reason and because it just sheds more heat than light on problem solving.

All that said, "poetic license" usually makes a better, direct story! On to the traditional game...

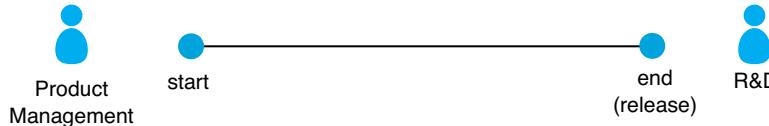
The traditional game of development

In game-theory terms it is a *two-person competitive game*. The players:

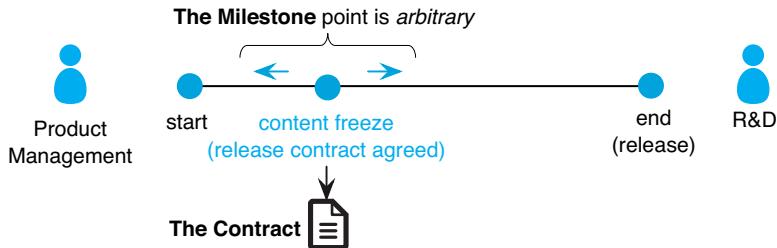


R&D

The two key events—both fixed—*start* and *end*:



Then, there is a third event between start and end—an *arbitrary* point in time—called **The Content Milestone**³ (or *milestone*) at which time agreed scope and date details are recorded in **The Contract**.⁴



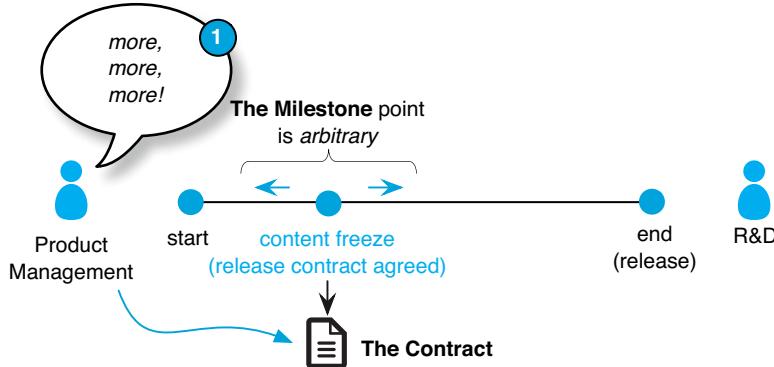
The Contract is not a commercial contract, rather, it is an *internal* agreement between P-M and R&D.

The Content Milestone is often a fixed arbitrary date decided at The Start: *Thursday is an excellent day for a milestone!* Sometimes the milestone is a sliding point defined by some arbitrary presumed state; for example, *The PRD is 80% complete* or *Janet is anxious*.

The first move is played by product management. Because they have one big chance to (1) *squeeze* as much as they can from R&D and (2) establish means to deflect blame—also known as “assigning responsibilities”—if the customers are unhappy, there are therefore various fears and odd behaviors at play (whose exact nature we leave a mystery). So, the first move is to push as much content as possible into The Contract—and hence, product release.

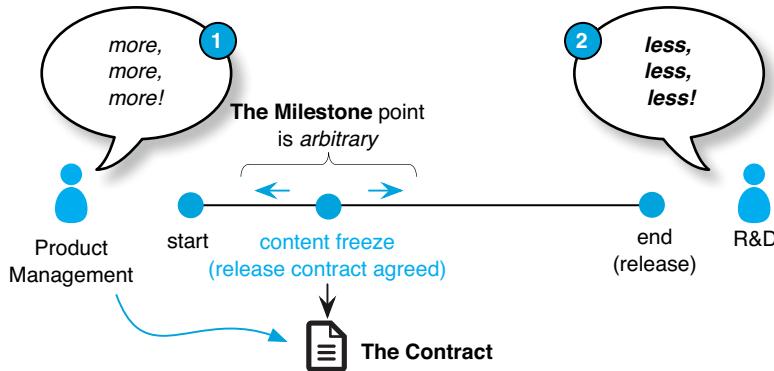
-
- 3. *The Content Milestone* is also called *content freeze, release agreement, product definition done, or sign-off*.
 - 4. *The Contract* is also called *program contract, project contract, the specification, the product definition, the product requirements, or a whole alphabet soup of document acronyms: MRD, PRD, SRS, FSD, and/or PSD*. None of these are used in their traditional way in agile development because all assume (1) (big) define it, (2) negotiate it with R&D, (3) ‘freeze’ it, and (4) develop it.

(After The Content Milestone, change is still possible, but involves hard work.⁵⁾ This first move is called *this is your last chance to spit it out*. Here's how it's played:



This is also called *local optimization*.

The second move is played by R&D. Since they have to 'commit' to do what they negotiate in *The Contract*—after, change is possible, but involves hard work—here's how it's played:

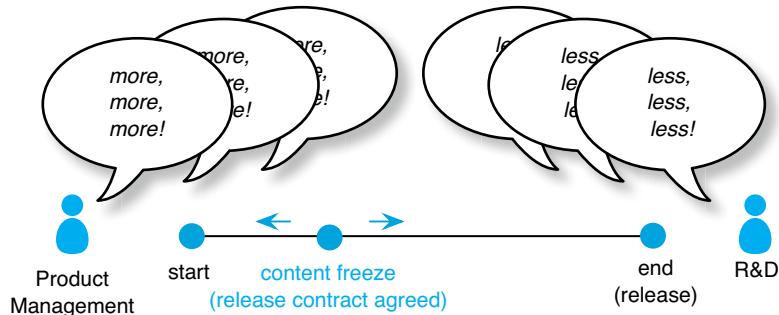


This is a second local optimization.

These moves repeat until (1) *Thursday!* (the random date), (2) the PRD is (believed) 80% 'complete,' (3) the players think they have suf-

5. Also known as *change management*; it too includes many acronyms: CR, CCB, ... The traditional approach to change management is not necessary in agile development.

ficiently protected themselves in case something goes wrong, (4) the executives (referees) decide “it’s enough, move forward,” or (5) they are really tired:



At this point they declare *successful release contract agreement* and The Contract is *finalized*. In some versions of the game, there is a ceremony: signing rituals or declarations of “*we commit*.” Sometimes a talisman is offered: the Bonus If You Meet the Contract.

This is called *the belief in magic*.

No one is to blame in this game; actors are stuck in the system.

Key Point: After The Content Milestone, P-M has finished their content-and-date move in this game, so they go away. They have won poker chips, and wait to cash them in at The End, to get the prize.

The next move is played by R&D, to implement The Contract during the *Development Phase*. P-M waits (increasingly anxiously) for The End. Note that they have money and hope—and fear—invested in The Contract, but little control.

Note: Delivery—and the path to delivery—is in the hands of R&D.

Worse, visibility of progress is low, especially if a sequential life cycle is used, and this exacerbates a lack of effective business control.



Finally, The End. P-M cashes in the chips they won. Did the prize description match what they really got? Sometimes yes.⁶ And sometimes no—in which case the final moves in the game are



The Problem

It is a silly story, but so what? Since it is a *competitive* game, both players are locally optimizing. The focus is not on optimizing the system to quickly deliver the best customer value, in the context of learning and a changing world. Rather, the focus is on departmental goals, avoiding being “assigned responsibility” for problems, and perhaps getting rewards, *even though these practices are established with good intentions*.

6. Delivering *The Contract* is not necessarily related to business success or customer satisfaction, as is explored next.

There are several core assumptions behind this traditional game:

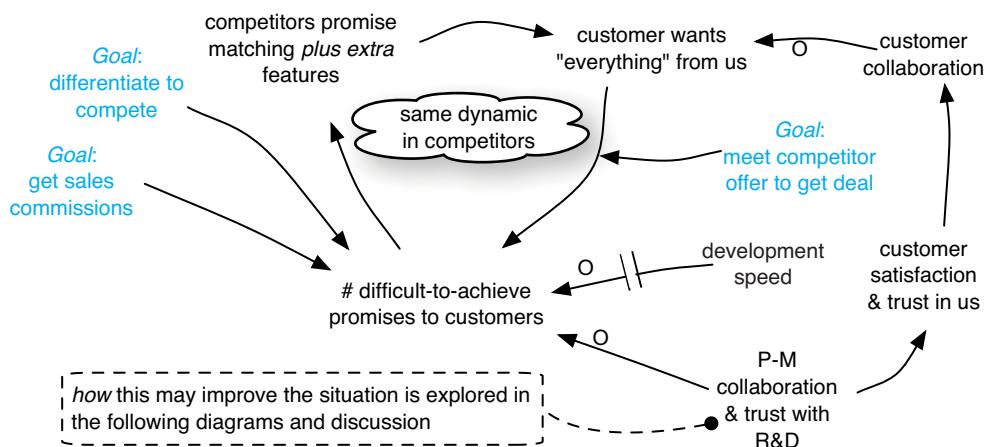
see Queuing Theory in companion book

- It is not efficient, *given the way we are currently working and organized*, to release early with a subset of features, or to manage fine-grained features; to be efficient (in our current system), it is important to work in big batches.
- The whole solution needs to be “thought out first,” and then big customer features need to be split into *architectural tasks*.
 - one consequence of the prior assumption is that there is no satisfying measure of progress from the customer or P-M perspective; R&D ‘progress’ is related to technical tasks, not customer value, and so there is lower transparency
- As a result of these assumptions (and others), there is *no relevance in product management directing development*—that is the job of R&D management (for example, a project manager).

System Dynamics

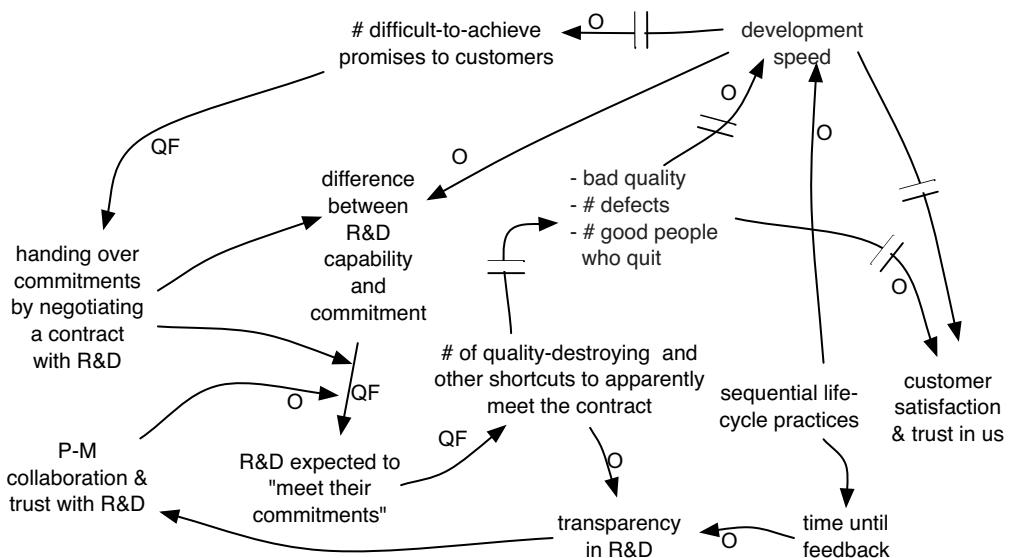
The system dynamics and assumptions behind this game are fuel for a firestorm of problems, so understanding the system is important.

It often starts with the pre-game mentioned earlier. Many product groups participate in a competitive features race that leads to an increase in difficult-to-achieve promises:



Notice variables that can influence the central, painful feedback loop: *development speed, customer satisfaction and trust in us, and the P-M collaboration and trust with R&D.*

Faced with the challenge of meeting all these ‘promises,’ and the belief that the key problem is the speed of development—which is the responsibility of R&D—compounded with a lack of trust and poor visibility or transparency into the work of R&D, a quick-fix response to the situation is to negotiate The Contract with R&D, *shifting the commitments to them:*



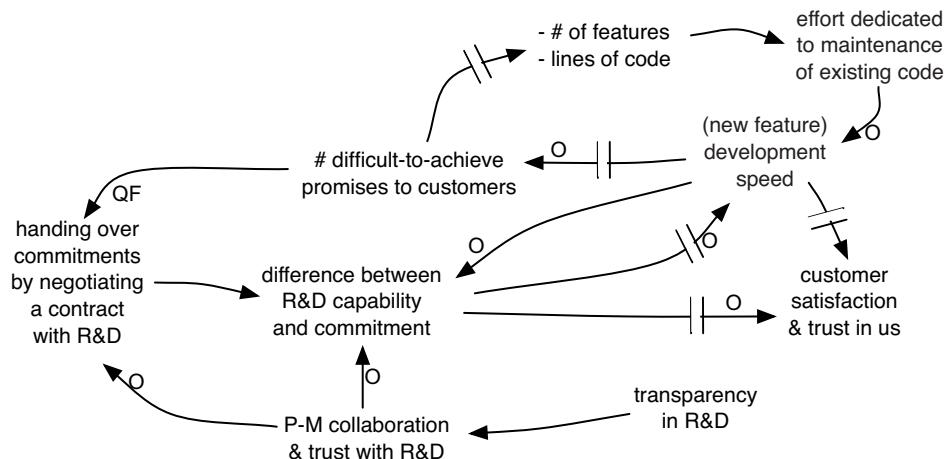
As illustrated, interesting dynamics then ensue, which boil down to this: development and the product get worse, and customer satisfaction and trust degrades further.⁷ This leads to a vicious cycle of graceful degradation. But since there are significant delays in these causal links, the frog does not notice it is being slowly boiled to death.⁸

7. Plus, other quick fixes to ‘solve’ the problem then ensue, as explored in the *Systems Thinking* chapter of the companion book: Adding more people, leading to an increase in cost, leading to a quick fix to hire more low-cost people, leading to offshore outsourcing and multi-site development, leading to more delay and reduction in quality.

*See
“Avoid...Fixed
content with
unrealistic
deadlines” on
p. 335.*

And there are other problems with boiling the frog this way. The cheapest system to build *and maintain* has no code; beyond that it gets worse. One metasurvey of maintenance studies summarized that *at least 50%* of cost is not during original development, but during later maintenance [Koskinen03].

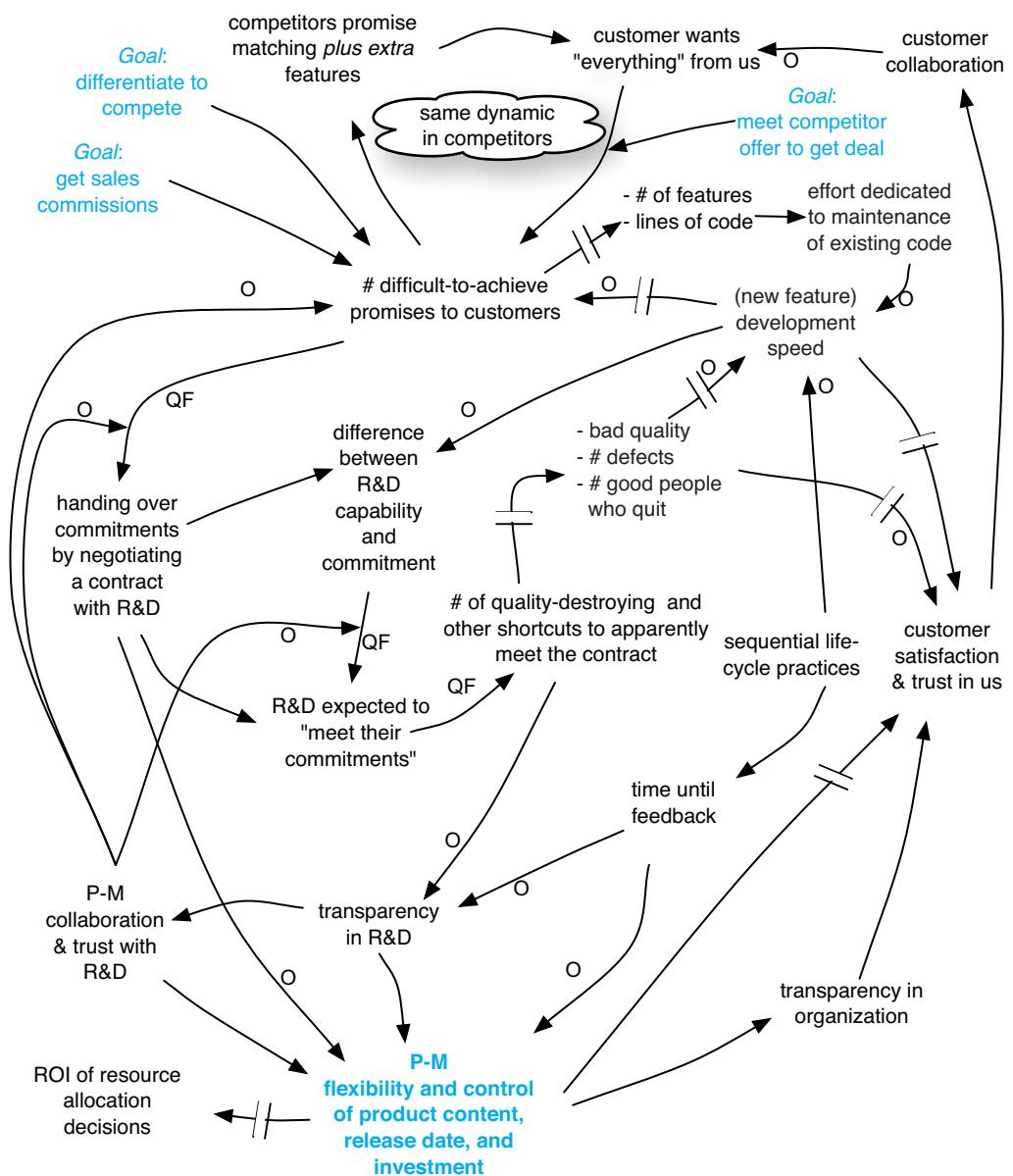
Consequently, the waste of overproduction of extra features in the traditional Contract Game not only has short-term but also long-term downside because making more stuff today probably means making *less* new stuff next year, due to “care and feeding” costs. Plus, to meet the content-heavy deadline, opaque (in the short term) quality-reducing actions occur within R&D. Worse, sometimes those extra features were only requested by a few customers—adding to the sub-optimization...



There are ways to cool the waters and save the frog. Figure 4.3 illustrates key dynamics in the market and within one company—for good or ill. A product manager who grasps the implications of Figure 4.3 holds one key to unlocking improved long-term ROI, better product management, and better relationships with customers.

8. For another source of aggravation, see “Avoid...Short-term product managers or focus” section on page 123.

Figure 4.3



*If R&D can potentially ship a well-done product each iteration, and if P-M (rather than R&D) adaptively steers the choice of features to develop each iteration (with fresh insight based on learning and feedback) and enters into a collaborative and transparent relationship with customers and R&D, then there is a *Goldilocks* solution—a product management balance neither too hot nor too cold.⁹*

Keys to the improvements possible in Figure 4.3 come from...

**Try...Product management collaborates with R&D each iteration,
adapting release scope or date**

The third agile value is *customer collaboration over contract negotiation*.¹⁰ This refers not only to commercial contracts, but also to the internal ‘contracts’ between parties within the product-creation company, as explored in the previous topic.

In Scrum, the voice of the customers to the teams is the Product Owner from product management,¹¹ and they enter into a

two-person cooperative game of invention and communication

as explained by Alistair Cockburn in *Agile Software Development: The Cooperative Game* [Cockburn07]. This is a paradigm shift; a new game for P-M and R&D. The Product Owner (PO) and R&D participate in the game of Scrum, and the PO adaptively steers R&D each iteration by re-prioritizing the Product Backlog and by collaborating with teams during Sprint Planning and Sprint Review.

-
- 9. We recognize that this solution makes an assumption that itself may be flawed: that the product management and R&D leader are *separate people*. Although that may remain true in Scrum, at least separation and us-them mentality is reduced by including the Product Owner in the Scrum Team. See also the lean idea: “Try...Overall product manager is chief engineer” section on page 128.
 - 10. No false dichotomies. Recall the agile values postscript: *That is, while there is value in the items on the right, we value the items on the left more.* It is common in Scrum to start with a target Release Backlog, with a specific long-term release date and product content in mind; however, this evolves every iteration.
 - 11. Or for internal applications, the business area.

Further, the PO and teams enter into a cooperative game of *ongoing invention and communication* with the customers or users that the PO represents to the teams.

The system dynamics in Figure 4.2 illustrate some of the deeper forces at play in the collaboration game. Tip: By *looking backward through the causal links* from *customer collaboration*, P-M and R&D can experiment with changes to strengthen the system, starting with understanding the dynamics that influence *customer satisfaction and trust in us*. For instance, that is influenced by *increasing transparency in the organization*.

The Product Owner has business responsibility—The PO is responsible for the product profitability, has the business view, and talks to real customers. So, with a clear understanding of the traditional Contract Game and its problems discussed in the previous section, it should be clear why the Product Owner cannot be someone from within R&D (unless that person had business responsibility). A critical rule and dynamic in this game is that the steering of development is not in the hands of R&D, but in the hands of product management—those responsible for profit, release content, and talking to the customers.

Try...Challenge traditional product-management assumptions

A new product manager may have been exposed to books or education that convey traditional assumptions. Know and question these:

Assumptions: Life cycle and contracts

In *The Product Manager's Handbook* [Gorchels06], the “product development process” is defined with sequential stages. Stage four is *Definition*, whose definition is

Establishment of clear functional specifications... [R&D] Team agreement on the requirements for the product.

The *Handbook* explains “the new product project” in more detail:

Freeze the product concept (i.e., obtain commitment on what benefits it will provide the customer) after the concept development phase. Freeze the product specification after successful prototype development.

In *Product Management* [LW05], step six in the “product management planning process” is *Negotiate Final Plan*.

The last stage in the *Handbook* is *Project/Process Evaluation*: Only after launch, consider improvements “for future projects.”

All this assumes the traditional Contract Game and sequential life-cycle development.

Assumptions: P-M does not direct product development

The quotes above, and other conventional literature on P-M assume or assert that the product manager does not drive development of the product:

...the product manager's control over the internal groups [R&D, marketing research, ...] may be less than over an external group because she lacks direct authority. [Gorchels06]

Outward—Part of the rationale behind this assumption is good: the product manager’s (Product Owner’s) major focus should be *outward* to the market, not inward, and include acting as a cross-functional leader between R&D, manufacturing, and so forth. For instance, *Product Management* by Lehmann and Winer contains not a single word on the subject of *developing* the product; it addresses the customer, market, and channels. We agree with that outward focus—and have a section emphasizing it later in this chapter.

Outward and inward—However, that does not necessitate a false dichotomy in which P-M cannot drive development. It boils down to the degree of time and attention. Balance. Scrum responsibilities do not demand all of the Product Owner’s time; there is Sprint Planning on the first day, Sprint Review on the last, and some time in between, but certainly not all the time.

And this involvement is vital; in *Product Strategy and Management*, which surveys the success and failure research into successful product management and development, the view of P-M well divided into a separate silo apart from R&D is rejected:

...all the performance indicators in new product development point to the need for [cross-] functional integration. [BH07]

Assumption: P-M is not chief engineer

Finally, an unspoken assumption in organizations is that the overall product manager and the chief engineer are separate people. This is discussed further in the “Try...Overall product manager is chief engineer” section on page 128.

Challenging these assumptions: New trends in product management

Some product managers, aware of the change in P-M assumptions and behavior with the introduction of Scrum, are challenging old assumptions. For instance, in the product management guide *Inspired*, Marty Cagan (VP of product management at eBay) writes

R.I.P. PRD: I think the product spec is long overdue for renovation. Some would argue that agile methods accomplish this by doing away with the spec altogether.¹² While there are other issues with that, in many respects I think they are on the right track.

Replace heavy PRDs and functional specs with prototypes and user stories. [Cagan08]

In fact, *Inspired* includes the chapter *Succeeding with Agile Methods*. Another sign of changing times in product management is *The Art of Product Management* by Rich Mironov [Mironov08], devoting

12. This is a common “false dichotomy” misunderstanding of agile development by Mr. Cagan. The second agile value is *working software over comprehensive documentation*, which is quite different from “doing away with specs altogether.” His point still holds.

twenty pages to the changing assumptions, behaviors, and business opportunities when adopting agile methods. For example:

I continue to see that Agile delivers more and better software; that product managers are an irreplaceable part of that improvement; and the P-M function needs to be champion of business improvement.

PRODUCT OWNER

Try...Product Manager is Product Owner

Avoid...Product Manager is *not* Product Owner

In early Scrum practice and writings, there was no special title *Product Owner*, and the common title *product manager* or *product marketing manager* was used:

Only one person prioritizes work. This person is responsible for meeting the product vision. The title usually is product manager, or product marketing manager. [BDSSS99]

*Don Roedner, the first Product Owner, did not move to VMARK. I specifically took the best person out of **Product Marketing** at Easel and assigned him [Don] the team and he had total control of the backlog. [Sutherland09c] (emphasis added)*

We sometimes visit groups adopting Scrum where the separation between P-M and R&D is so ingrained that no one could *conceive* that the real product manager can serve as the real Product Owner—interacting with the teams and directing the priority of development. Rather, the so-called Product Owner role is delegated to a project manager or program manager within R&D who is not responsible for the product profitability.

This is a misunderstanding;¹³ the Product Owner has *business* responsibility, not “project execution.” In Scrum, the Product Owner has these responsibilities [Schwaber05]:

13. An exception to this is the lean development practice of entrepreneurial chief engineer in which the chief engineer and product manager with business ownership are one and the same.

- ❑ decides on release date and content
- ❑ is responsible for the profitability of the product (ROI)
- ❑ prioritizes features according to market value
- ❑ can change features and priority every iteration
- ❑ accepts or rejects work results
- ❑ defines product features

And in *Agile Project Management with Scrum* [Schwaber04]:

The Product Owner's focus is return on investment... The Product Owner represents all stakeholders [customers, funders]

These are the remit of *product management*. Therefore, the *Scrum Guide* [Schwaber09a] suggests:

For commercial development, the Product Owner may be the product manager.

If there is a fake “Product Owner” who does not have this authority, a vital essential dynamic of Scrum is missing: that business steers product direction each iteration, not R&D, and that there is an end to the Contract Game; rather, there is a shift from *contract negotiation* to *customer collaboration*.

Avoid...Fake Product Owner

Avoid...Business manager is *not* Product Owner

This is a retelling of the previous idea, expressed in terminology used for internal products or applications (as in a bank). To quote the *Scrum Guide*:

For in-house development efforts, the Product Owner could be the manager of the business function that is being automated.

Once again, the real Product Owner is not a project manager from IT or engineering; rather, he or she is a business person from the area the application is created for. If the so-called Product Owner was a project manager from IT, the Contract Game remains; and nothing has really changed for the better in the relationship and

dynamics between business and IT, because “nothing needs to change.”¹⁴

*Try...Product
management
owns the product*

Try...Product Owner owns the product

When we coach, we may ask the product manager(s), “Are you responsible for the ROI of the product, and do you have independent authority to make decisions related to this responsibility?” “Do you choose the release date and content?” Sometimes—especially in very large product companies—the answer is “No.” A common situation is *responsibility without authority*—a recipe for problems in most contexts. For instance, perhaps they are responsible for the ROI but do not decide the content, or they have proposals approved or amended by a committee.

Independent of Scrum, this is a problem because it often reflects

- no unity of vision or constancy of purpose
- sub-optimizing goals pushed by competing factions: Sales, Marketing, and more
 - these two points lead to products that are less likely to meet the highest-priority business objectives
- someone else—often Sales or Marketing—imposing the Contract Game onto both product management and R&D

The name “Product Owner” was coined by Jeff Sutherland (the co-creator of Scrum). His intention, while working in a commercial product company, was straightforward: that the Product Owner *really own the product* and have responsibility and decision-authority related to profit, business case, pricing, release date, and content [Sutherland09a].

14. This is a shallow Scrum adoption, in which only the surface words and actions appear to have changed (holding a daily stand-up meeting, having a backlog, ...), but the underlying dynamics that are the root causes of the problems have not been tackled. This happens because “it’s easy”—the status quo organizational design is not challenged or significantly improved.

Avoid...Short-term product managers or focus

Sometimes the pivotal role of product management is unfortunately viewed as a short-term engagement that potential executives *step on* during their climb up the ladder. Or, product management (and quality management) is viewed as a place where people go when it is not clear where else they should work. This weakens the system because people may not be strongly motivated by a desire to excel in product management, or they may make short-term decisions without facing the outcome of those in future releases, and do not develop deep skill over many years. For example, quality may be sacrificed and there is low motivation to invest in improving.

(This same short-term-view problem exists when *program* or *project* managers are responsible for delivering a release—since they usually are finished when the ‘project’ is finished. Scrum eliminates this problem with a Product Owner from business driving development.)

Try...Fake Product Owner

Sometimes the person who should rightfully serve as Product Owner (such as the product manager with the business view) is not yet interested in participating in Scrum. Or, that person may be interested, but the teams are new to Scrum and expect awkward Scrum-adoption learning in the first few iterations—and there is valid concern that on seeing this awkwardness, the new PO would be scared off. In either of these cases, a *pseudo Product Owner* is a temporary option.



When there is no interest (yet) by the rightful PO, teams can work with the pseudo PO, and invite the potential PO to attend Sprint Reviews to gain informal feedback and to attract him or her to participate. By demonstrating skill and responsiveness, the team—and the advantages of acting as PO—may become attractive. Or when working through extremely awkward learning, a pseudo PO can collaborate with the teams for a few iterations until the teams are fluid.

*Try...Business manager is Product Owner***Avoid...Believing Product Owner is just an analyst role**

There are misconceptions and incorrect statements that have been written about the role of Product Owner. Here is one:

Product Owners sit with their development teams full-time, elaborating user stories, managing sprint-level backlogs, ...

This is not correct; it is inconsistent with the definition [Scwhaber05, Schwaber09a]. The Product Owner meets with the Team during Sprint Planning on the first day, during Sprint Review on the last, and some time in between, but not as described above. Also, the PO does not manage the Sprint Backlog, which is solely created and used by the self-managing Team.

More broadly, the PO is responsible for product profitability and for deciding release date and product content—business-oriented product management responsibilities.

Most incorrect descriptions relegate the role of “Product Owner” to some minor variation of business or requirement analyst. That might only happen as a temporary stop-gap measure, as described in the “Try...Fake Product Owner” section on page 123.

Avoid...Believing Product Owner must attend the Daily Scrum

We encourage Product Owners to help reduce us-them culture and to practice Go See by observing Daily Scrums—at least sometimes. However, a variant of the misunderstanding that “Product Owner is just an analyst role” is that he is *required* to attend the Daily Scrum. Not true [Schwaber09a]. The incorrect notion that the Product Owner must do so is a reflection of a larger problem to avoid...

*Try...Product Owner product manager focuses outward to the market and channels***Avoid...Too ‘inward’ product management & Product Owners**

“Before we adopted Scrum I never met the customer and never saw the product. Now that we use Scrum, at least I see the product every iteration.” This was said by a “product manager” in a group we were coaching. We felt sad to hear this, but happy that things were improving.

Product Owner

We sometimes work with a product management group that spend almost all of their time writing PRDs, clarifying specifications, attending Product Backlog refinement workshops with teams, and interacting with other internal departments. While these activities are within the remit of P-M, and the product manager has a central role as cross-functional leader between different departments, it is too inward.

A great product-manager Product Owner also pays attention *outward* to the customers and market. Consider this partial list of major responsibilities, described in *Product Management* [LW05]:

product vision	business plan	customer relations
competitor analysis	ideation, innovation	market testing
channels	pricing	market research
profitability	risk management	product review

The Product Owner meets with the Team during Sprint Planning on the first day, during Sprint Review on the last, and some time in between; there is—and there must be—time to focus on market- and customer-facing outward activities.

Avoid...Too ‘outward’ product management & Product Owners

“Our product management is not involved in Scrum because they talk to customers” is a refrain we sometimes hear. It reflects no relationship with teams and high levels of handoff. It leads to the Contract Game and/or other issues explored in “The Problem” section on page 111.

The role of Product Owner implies a balance between outward and inward focus, in which the Product Owner commits to *at least* spend some time each iteration with teams.

Avoid...Us-Them: Product Owner versus Team

We were facilitating a group’s first requirements workshop that included both development team members and product management. We were at a whiteboard with one team, listening while they

discussed an unclear requirement. The product managers that (everybody knew) could clarify were two meters away, sitting on a sofa and wondering why they were in the workshop. The team members never thought to walk over and ask them—they had never talked to product management before. And the product managers never thought to join the team members at the walls to collaborate.

Especially in a large organization, departmental silos with us-them culture is common. When Scrum is adopted in such an environment, it is predictable that attitudes will, at first, remain the same... *I am the Product Owner, not involved with teams. We are the team, not involved with the Product Owner.*

ScrumMasters in large-scale Scrum adoptions need to pay special attention to breaking down the old walls. A change in title does not result in a change in behavior.

Avoid...“Product Owner”

For *commercial products*, the term *product manager* is long-established, and arguably the new term *Product Owner* is not mandatory in this context if the product manager truly fulfills the PO role. Plus, as already mentioned, in the early Scrum literature, the term *product manager* (not ‘PO’) was used:

Only one person prioritizes work. ... The title usually is product manager, or product marketing manager. [BDSSS99]

(The Product Owner term is, in general, useful because it signifies a special role with Scrum behaviors and responsibilities and because Scrum is also applied to internal products or applications, where there is no “product manager.”)

For a group with a bona fide product manager, a critical point of Scrum is to end the Contract Game, and have the existing person with real responsibility for product profitability, release content, and release date take on the *behavior* of Product Owner, steering development. Unfortunately, the introduction of the term “Product Owner” sometimes creates confusion...maybe the Product Owner is just some requirements analyst, maybe the Product Owner can be delegated to an R&D project manager, or other misunderstandings.

In that sense, if you are an agile coach, experiment with avoiding the term *Product Owner* (if it is adding confusion) and instead focus on what is really important: *the change in behavior that the product manager needs to make in the role of Product Owner*.

A product manager acting as Product Owner *does* have expanded Scrum-related responsibilities; this experiment is not about avoiding these, it is about being sensitive to the impact of terminology.

In *Agile Project Management with Scrum*, Ken Schwaber makes a similar point:

It isn't always necessary to make a big deal out of the role of Product Owner. Sometimes it makes sense to lowball the whole thing and propose something casual instead, like getting together and talking about what to do next. People are often suspicious of new jargon and new methodologies—and not without reason. [Schwaber04]

Try... “Product Owner”

For *internal* applications, people often—unfortunately—think of *projects* with a beginning and end. In contrast, product companies understand their *product* may last for years, and it benefits from a consistent team of product manager and developers that support it. Problems with “project mindset” were considered in the *Organization* chapter in the companion book. The same point is reiterated by Roman Pichler, author of *Agile Product Management with Scrum*:

Having one person in charge across releases ensures continuity and reduces handoffs, and encourages long-term thinking. [Pichler10]

In reality, internal *applications* made at a bank or energy company are products—*internal* products. They do *not* just disappear when the first release is deployed. So what? Overlaying a short-term *project* view on a *product* introduces various sub-optimizations. For instance, after *project* release, the *project* team is disbanded; when a new release is required, a new project team is formed—with many wastes from loss of consistency and knowledge handoff. Or, since

each project release is led by a different project manager, they locally optimize for this release and sacrifice long-term quality.

In this situation, introducing a long-term *product* paradigm to replace the short-term project culture is a good thing. As a coach introducing Scrum, emphasize the notion of a stable *Product Owner* from business, managing a long-term *Product Backlog*. Call your systems *internal products*. Experiment with removing or downplaying *project* terminology and mindset.

Try...Overall product manager is chief engineer

Both Toyota and Honda apply the lean development practice of *entrepreneurial chief engineer* (“large product leader” in Honda terms). *One* person with technical excellence as a master engineer who also has business acumen (and education) is the leader of product management, market research, and product development. Common in small start-ups, this is all-too-rare in large establishments.

It is most appropriate for products that are strongly technology-driven, such as a new car or a software-intensive system. For customer-driven products where technology is a minor player, a separation between product management and R&D is less a problem.

MANY PRODUCTS

Portfolio management was discussed in the *Organization* chapter of the companion book. This section covers more specific experiments.

Avoid...Platform group with a “shared infrastructure” backlog

Try...Add and do a cross-product common goal

When we start coaching a big organization, one group we are guaranteed to find is...the Platform Group. The conventional idea is to collect requests for shared components or services from across several products, which are then implemented by the Platform Group “for efficiency.”

Alternatively, sometimes the idea is that the Platform Group speculatively create the Grand Frameworks (the “business object frame-

work,” ...) that are pushed on to product groups, who are forced to use what frequently turns out not grand, but grandiose.¹⁵

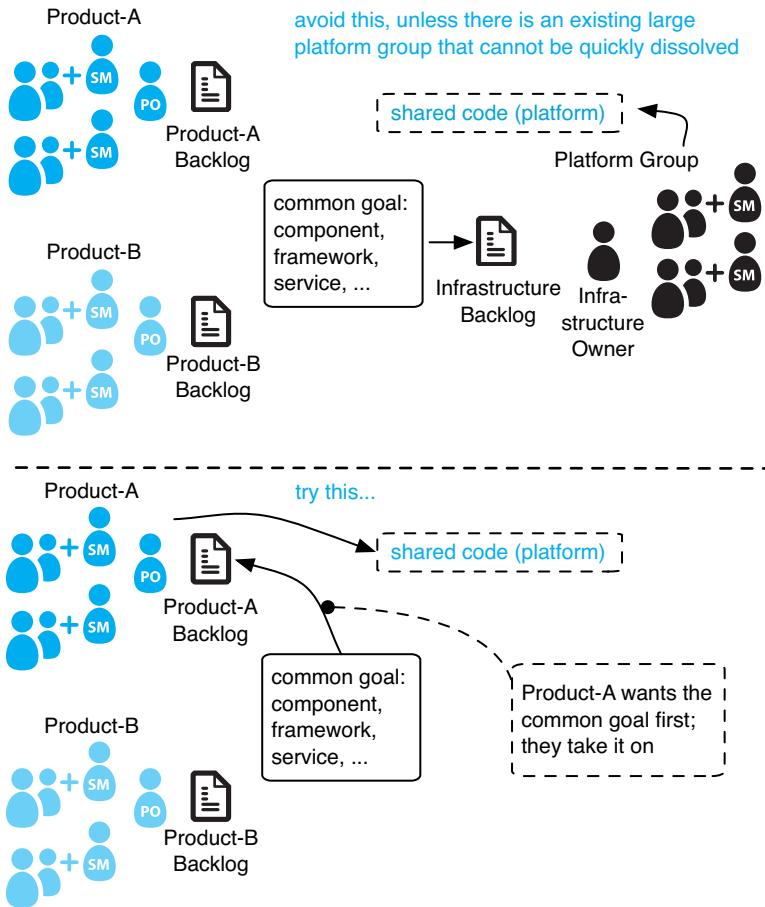


Figure 4.4 take on a common goal rather than create a platform group

When an organization with a platform group adopts Scrum, a common notion is to create a “shared infrastructure” backlog for the group, fed from the products. The Product Owner from business needs to understand the implications of this common—but expen-

15. Because the designers are not involved in using the frameworks to build real product features—*eating their own dog food*. We recommend, “No reuse before use.”

sive and value-delaying—organizational design within large-scale R&D, challenge the assumption it optimizes the system, and guard against its weaknesses.

In contrast, this experiment means that a Product Owner from a real product—not a platform group—will be taking on a shared infrastructure goal (Figure 4.4).

What are the weaknesses associated with a platform group and shared infrastructure backlog? A platform group is one large *component group*, and all weaknesses with component teams identified in the *Feature Teams* chapter¹⁶ of the companion book exist at a macro scale: delayed delivery, handoff, lack of visibility of R&D work to P-M, delayed integration and testing, and more. To be clear, in this design there are at least two “component teams” involved in delivering customer value: the product and platform groups.

The conventional thinking behind a platform group, “for efficiency,” is based on local optimization thinking—that a group specialized on one component produces the best design and is fastest¹⁷—rather than delivering value fast to customers. In short—to repeat the opening metaphor of the *Lean Thinking* chapter in the companion book¹⁸—it was created by a focus on the *runner* rather than the *baton*.

The alternative? See Figure 4.4. Avoid platform groups; instead, the Product Owner of the product (Product-A) that most quickly wants the common goal (item-X) adds it to their Product Backlog and creates the shared code in the common platform. The Product-A Scrum feature team that implements item-X collaborates with stakeholders from other products to learn and account for cross-product support. Item-X is created quickly by the group that wants it most, and other products benefit from their effort.

Furthermore, the people *creating* item-X are the ones also *using* it. Good things come from *eating your own dog food*—in software, policies, and processes.

16. The *Feature Teams* chapter is available on the web.

17. Assumptions not even guaranteed true with component teams.

18. A close variant of the *Lean Thinking* chapter is available as the *Lean Primer*, at www.leanprimer.com.

Platforms, not platform groups—Just as it is possible to have components without component teams, it is possible to have a platform of shared code without a platform group. Plus, a high-quality platform is supported with a *design community of practice* and *joint design workshops*.

See “*Try...Plan infrastructure items by regular teams*” on p. 168.

Transitioning from a large platform group—At the clients we work with, merely the platform group itself (below the product groups) may be 200 to 1,000 people. It will not disappear quickly. In this case experiment with this transition strategy:

1. Temporarily maintain the platform group, but gradually (and permanently) move members into product groups.
2. Create a shared infrastructure backlog; apply Scrum in the platform group with this backlog.
 - note: items in this backlog are not customer requirements
3. Do some common infrastructure goals in the platform group, and some in product groups—with coaching and review from current- and ex-platform group members; this also implies abolishing private code ownership.
4. Gradually reduce the platform group as the product groups learn to effectively take on more shared platform goals.¹⁹

See “*Try...Joint design workshops for broader design issues*” on p. 298.

Try...Product Owners work together to maximize *company ROI*

The previous experiment demonstrated one Product Owner taking on work that benefits other products. Systems thinking suggests a perspective and actions that maximize ROI and sustainability for the company, not just one product. How can Product Owners work together to support this?

- Shift teams (not individuals) to the products that currently have the most to gain from an increase.
 - avoid frequent shifts; there are overhead costs: learning, ...
-
19. Platform groups are often an internal cost center funded by the product groups. As it shrinks, so does the cost to product groups.

- Do an important item for another product when your product group has the skill or slack.
- Do a common item that benefits several products.
- Challenge one another's business cases; reduce wishful thinking by encouraging other products to get earlier feedback.
- Share improvement experiments—for P-M, not only for R&D.

MANY TEAMS

Try...One and only one Product Backlog

Some “scaling Scrum” descriptions advise that each team have their own Product Backlog. This is not correct. There is only one Product Backlog for the overall product, regardless of the number of teams. The *Scrum Guide* explains:²⁰

*Multiple Scrum Teams often work together on the same product.
One Product Backlog is used to describe the upcoming work on
the product. (emphasis added)*

This is necessary for focus on the whole product and to optimize the overall priority of features for the product—to avoid local sub-optimizations by separate teams.

Avoid...Fake team-level “Product Backlogs”

In product groups with many component teams or single-function teams (such as design or testing), these teams do technical tasks—partial work for a feature—rather than a complete feature. Sometimes, rather than transitioning to Scrum feature teams (that do complete features), these existing teams have team-level so-called Product Backlogs that contains these tasks. Avoid that.

We visited several companies that had team backlogs and the result was always the same: The team and their local PO always focused on their separate work, causing problems for the product as a whole

20. The *Certified ScrumMaster* course [Schwaber05] also presents one Product Backlog for many teams.

and an uncooperative attitude across teams. The local optimizations were not hard to detect.

For more, see

- ❑ “Avoid...Try...‘Easy’ agile or lean adoption” section on page 386
- ❑ “Avoid...Technical task ‘requirements’ (PBIs)” section on page 237
- ❑ “Avoid...Technical task PBIs in team-level “Product Backlogs”” section on page 238

Try...Area Product Owners when many teams

Area Product Owners (APOs) are applicable to large-scale development involving *many* teams, such as 50 teams for one product. Briefly, the (large) Product Backlog is subdivided into **requirement areas**; these are major areas from the customer perspective, such as color workflow, transaction printing, and security. Each area has its own **Area Backlog**²¹ and APO, and is served by many teams that specialize in that one requirement area. An APO (and requirement area) has many teams, *never* (or at least extremely rarely) only one.

*see Feature
Teams Primer
chapter*

Is an APO the same as a supporting Product Owner for one or two teams? No. When there are many teams, one *overall* Product Owner and several *supporting* POs (or “PO representatives”) can help, when there is too much work or too many people for a single person to act effectively as Product Owner towards all teams. However, the overall APO responsibilities and focus are different than a supporting PO. Table 4.3 shows some of these differences. Note: they are not mutually exclusive—both may be useful. But, in most product groups we have worked at, a supporting PO was not needed and would be unnecessary overhead. Challenge the need of supporting POs and avoid them if an overall PO plus APOs are enough.

21. An *Area Backlog* is not a separate Product Backlog; it is a view into the one Product Backlog, filtered by requirement area.

Table 4.3 Area Product Owner and supporting PO

Area Product Owner	Supporting PO
(similar to the overall Product Owner) is part of Product Management and has an ROI focus	probably in R&D, though may be part of Product Management; does not have ROI responsibility
main focus is on customer-centric requirements, not on teams	main focus is toward the team(s)
the number of teams per area is dynamic; it changes over time but not at every iteration	stable for team(s)
works with 4–10 teams	works with 1–2 teams
overview across many teams, focus on product-level priorities	focus only on the team's work; overview often lost

This is an important scaling idea, worth closer study and experimentation if applicable.

Try...Product Owner Team

The overall Product Owner and Area Product Owners may be called a **Product Owner Team**; they collaborate to deliver the product.

Try...Map different scaling terms

When scaling to a Product Owner Team—because the product needs a team of product managers, and there are too many teams for one person to act as their Product Owner—it is common (not required) to distinguish a *single* ‘overall’ Product Owner who has overall vision, and final say on prioritization, to prevent sub-optimization. In fact, there is no Scrum rule that an overall PO exist; the Product Owner Team can collaborate to globally optimize product development.

Still, an overall PO is common. For example:

The chief product owner guides the other product owners. The individual ensures that needs and requirements are consistently communicated to the various teams, and that the project-wide progress is optimized. This includes facilitating collaborative decision making as well as having the final say if no consensus can be reached. [Pichler10]

Different groups have explored scaling Scrum, *in parallel at different organizations*. Thus, alternative but equivalent terms emerged. Additionally, there are some nonequivalent terms. Here is the basic mapping, followed by comments:

	[LV08]	various [e.g., Cottmeyer09]	[Schwaber07a]	[Pichler10] / [Cohn09]	[Eckstein10]
<i>overall</i>	Product Owner (PO)	PO	overall PO	chief PO	lead PO
<i>supporting</i>	PO representative, Area PO ^a	PO proxy	PO	PO	PO

- a. Area POs are a unique concept, different from other kinds of supporting POs.

Note that some groups have *scaled down* the terminology, and others have *scaled up*, each with the same underlying intentions. For instance, most of our clients use “Product Owner” to mean the one overall Product Owner and then scale down the terminology to “PO representative” or “PO proxy” for the many supporting POs. Other groups use “Product Owner” for the supporting POs and scale the name up to “lead PO” or “chief PO” for the one overall PO.

Overarching principles when scaling:

- ❑ *Who does one Team meet with?*—Regardless of name, a Team needs a consistent person to act in the role of Product Owner.
- ❑ *Who does one PO meet with?*—One PO (or *supporting* PO) may be able to work with more than one team.
- ❑ *Who ensures overall product vision?*—If many *supporting* POs have different top priorities or product visions, there will be conflict and sub-optimization. Although the Product Owner Team can in theory collaborate to avoid this (without a top per-

son), it is common that a single overall PO ultimately crafts overall product vision and prioritization.

Area Product Owners are different

As explored in the *Feature Teams Primer* chapter, an **Area PO** is different from the simple concept of a sub-PO that meets with a team. Area POs are only used in relatively large-scale development, with perhaps 30 teams or more, and are tied to the idea of **requirement areas** and **Area Backlogs**.

If there is one Area PO for one team, there is misunderstanding; one Area PO is responsible for an area served by *many* teams.

Area Product Owner is not another name for a supporting PO.

Try...Better behavior over ‘better’ PO scaling definitions

Q: What’s the difference between a terrorist and methodologist?
A: You can negotiate with a terrorist.

Shared understanding of method terms reduces the friction of misunderstanding. But we suggest not spending energy on the trivia of different Scrum scaling terms. What *is* important is a change in behavior—from the Contract Game between product management and R&D to a cooperative game of invention and communication.

Avoid...Try...“Product Owner Team”

See “Try...Large-scale Scrum FW-2 for ‘many’ teams” on p. 15.

In a very large-scale product group with requirement areas, the Product Owner Team includes the (overall) Product Owner and Area Product Owners. A team is needed because there are too many Scrum teams and too much product management work for one person to handle as Product Owner. *This suggestion is not about having or avoiding such a team, it is about terminology.*

Product Owner Team [LV08] is an unofficial but common phrase used in Scrum scaling contexts.²² Avoiding the phrase “Product

Owner Team” reflects the same terminology-confusing dynamics as “Avoid...Product Owner” (p. 126) but at scale when a team of product managers is needed. *Embracing* the phrase “Product Owner Team” reflects the same dynamics as “Try...Product Owner” (p. 127) when building internal products or applications (such as at a bank).

Avoid...Too inward-focused Product Owner Team

This suggestion reflects the same point as the “Avoid...Too ‘inward’ product management & Product Owners” section on page 124. Whether it is called the product management team or the Product Owner Team, their attention needs to include an outward-to-the-customer and business focus.

See “Avoid...Product Owner Team as separate analysis group” on p. 236.

Suspicious *too-inward-activity smells* include²³

writing specifications	designing user interfaces	defining architecture
------------------------	---------------------------	-----------------------

Suspicious *too-inward-composition smells* include²⁴

business or requirement analysts	UI designers	architects or system engineers
----------------------------------	--------------	--------------------------------

These activities and people are meant to be within the cross-functional teams in Scrum. Scrum is not a sequential process in which the Product Owner Team hands off requirements to the Scrum teams for implementation; rather, the regular teams (that contain analysts, UI designers, architects, and system engineers) are involved in this work—usually during ongoing Product Backlog refinement—in collaboration with guidance from an outward-looking product-management “Product Owner Team.”

22. The term was first used by Scrum co-creator Jeff Sutherland in the 1990s at IDX [Sutherland09b].

23. ‘Smell’ does not assure a problem; it is a signal worth investigating.

24. People *previously* working in these roles may make great P-Ms (or not); the point is that *product managers are product managers*.

Try...Product Owner representative (supporting PO)

Consider this example:

- ❑ a single product group with 100 teams and 30 people in product management (common, for example, in telecom products)
- ❑ one product manager playing the role of ‘overall’ Product Owner
- ❑ fifteen *requirement areas*, each with an Area Product Owner from the product managers

On average, each Area Product Owner is then served by seven teams. That one person cannot effectively spend time with each team acting in the role of Product Owner—giving full attention to each team during Sprint Planning, helping to refine all the backlog items, and so on. Someone else needs to serve as Product Owner (*supporting Product Owner or Product Owner representative*) in relation to the team. This PO representative may rotate over time and may help multiple teams.

A PO representative is either a product manager or team member, and is distinguished from a general subject-matter expert in that she *can make fine-grained decisions regarding requirement details*. For instance, if team members ask, “Should we consider this rare edge case?” Then, a PO representative can decide. However, PO representatives do not make larger decisions, such as priority.

In some multiteam Scrum implementations, this person is simply called the “Product Owner.” We avoid this because it is inconsistent with the true Scrum role of Product Owner; for more, see

- ❑ “Avoid...Product Manager is not Product Owner” section on page 120
- ❑ “Try...Product Owner owns the product” section on page 122
- ❑ “Avoid...Believing Product Owner is just an analyst role” section on page 124

PRIORITIZATION

Try...Value

For a long-lived big product such as a ship-control system or printer, the Product Backlog contains thousands of items.

Fact: During initial Product Backlog creation (when Scrum is first adopted), the Product Owner is responsible for prioritizing many of these items *in some way* to try to improve ROI or to improve delivery of ‘value.’ That suggests a measure of ‘value’ for each item.

Problem: What is ‘value’ and how to estimate it quickly—especially when there is a *pile* of items?

A deeper discussion of value is deferred until the following topics. Yet, with *relative value points* (RVPs) as a lightweight proxy for ‘value,’ use *planning poker* to experiment with *relative value points* (RVPs) and their estimation. For example, a scale 1–7 of relative value points assigned by business people (the Product Owner, other product managers, marketing, ...).



It is fun and simple to estimate RVP with planning poker:²⁵ The technique looks as shown in this photo. The details are beyond the scope of this introduction; ask a ScrumMaster.

Prioritization—When finished, all items have an RVP. They also have an *effort or cost estimate* that was provided by the teams. This somewhat guides prioritization: All other things being equal (they never are), the Product Owner increases the priority of items with high RVP and low effort estimate—more *bang for the buck*.

25. Planning poker is a technique most ScrumMasters can teach to the Product Owner. A subtle but important dynamic in planning poker is that it stimulates *conversation and learning* among people.

Trade-offs—This relative point estimate, a proxy for ‘value,’ reflects an informal collage of factors in the minds of estimators: revenue attraction, current market desirability, strategic alignment, and more. It is a fuzzy technique, but with the singular merits of simplicity, speed, and fun—planning poker is a surprisingly enjoyable way to estimate and to learn-through-conversation while doing so. The Product Owner quickly makes progress at prioritization, at the expense of fidelity or insight into the underlying “value factors.”

Other techniques—This is but one of several lightweight methods. ScrumMasters should know—or should be able to quickly find—other agile value estimation methods. These include *bidding with limited capital*, *distributing 1000 points*, and more.

Since RVP is so fuzzy, it might be better to...

Avoid...Value

‘Value’ is not a simple attribute or number—with context, the term is almost meaningless. And value—as with ‘quality’—is in the eye of the beholder. Therefore, advice such as “prioritize the Product Backlog by business value” feels warm and fuzzy, but has no teeth—just like Grandma.

Furthermore, concrete value-oriented measures for *one specific item*, such as an estimate of one item’s total life-cycle profit contribution, are wicked hard and mighty expensive to estimate. The contribution of one item independent of others is seldom clear, and even if it were, the market analysis cost to obtain an estimate is non-trivial—and the thesis behind the estimate will be quickly invalidated.

RVP can be one input, but consider moving beyond the simplistic notion of ‘value’ to...

Try...Prioritize with multiple weighted factors

The product manager of a big product has many factors to weigh and balance when prioritizing (and re-prioritizing) the Product Backlog. These include [Hohmann08]:

- stakeholder preferences
- strategic alignment
- relative points (value and effort)
- drive profit
- risk

These can be modeled with weights in a backlog spreadsheet...

Stakeholder preferences—“All customers are equal, but some are more equal than others.” Plus, there are internal stakeholders...Production support is interested in some items. And the ‘system’ itself (voiced by a technical leader) has a special stake in some:

Item	stakeholder weighted sum	customer-1	customer-2	production support	the ‘system’
	<i>weight >>>></i>	50	30	10	30
M	80	1	0	0	1
C	30	0	1	0	0

Strategic alignment—Alignment with major objectives—often established by the leadership group. For instance:

Item	strategic weighted sum	new regulatory compliance	reduce customer cost	touch-based interface	transaction-fee based
	<i>weight >>>></i>	100	40	30	60
M	100	0	1	0	1
C	130	1	0	1	0

Drive profit—This includes *revenue attractors*, *expense repellors*, and *expense attractors*.

*Try...Include
total life-cycle
cost of an item*

Expense attractors lead to a related suggestion: Consider the life-cycle cost of building an item, and model it in the Product Backlog. For instance: extraordinary maintenance or licensing fees, service preparation, change in manufacturing, or operational delivery.

Item	<i>profit weighted sum</i>	motivates upgrade	hot!	annual OPEX reduced > \$1M	annual extra costs > \$100K
	<i>weight >>>></i>	50	20	100	-50
M	100	1	0	1	1
C	20	0	1	0	0

Risk—This includes uncertainty related to business risk and technical risk [Reinertsen97]. The weighting typically reflects two elements: *Probability multiplied by Impact*. Noteworthy categories of *impact* include loss of money, loss of life ...

There is no point including risk attributes unless something is to be done. The well-known quote by Tom Gilb applies: *If you do not actively attack risks, they will actively attack you* [Gilb88]. Thus, increase the priority of heavily weighted risks, implementing (attacking) them early rather than late. Fail fast. For example:

Item	<i>risk weighted sum</i>	new technology	difficult performance targets	lack of consensus on meaning of item	very uncertain market reaction
	<i>weight >>>></i>	30	30	80	60
M	170	1	0	1	1
C	30	0	1	0	0

Relative Points—Items in the Product Backlog have effort estimates, often expressed as *story points (relative effort points)*. All other things being equal, items of higher effort or cost should be lower pri-

ority—they need more *bucks* for *bang*. And, if relative *value* points are used, high value points raise priority. For instance:

Item	<i>points</i> weighted sum	effort ^a	value
	<i>weight >>>></i>	-20	20
M	-120	8	2
C	20	5	6

- a. copied from the “new estimate of effort remaining” for each item in the Product Backlog

Other—Input from other product-feature prioritization systems, such as Kano model and Quality Function Deployment, can likewise be captured in this approach.

Prioritization—The separate weighted sums are added. In this example: M = 330, C = 230. As a first-order approximation, backlog priority reflects overall score. However, prioritization requires judgement and manual adjustment by the Product Owner to balance soft factors not captured in this model.

Trade-offs—This technique takes more effort but offers more fidelity. Watch out for illusions of accuracy in the weightings and numbers. Precision and accuracy are not the same thing... “Pi equals 3.14158265359” is precise; it is also inaccurate.²⁶

There's no sense being exact about something if you don't even know what you're talking about.—John von Neumann

Avoid...Feature priority categories

We joined a meeting with product management and some R&D teams in a group starting Scrum adoption. “Can you tell us how you prioritize features?” we asked. “Well, we have ‘mandatory’ features,” someone answered. “Okay, so your top-priority items are classified

26. See [Cockburn97] for more on this illusion.

mandatory, right?” “No, no,” they replied, “We also have *absolutely mandatory*.” Everybody laughed—we wondered how much more *mandatory* than mandatory can get. “So, your top-priority items are classified *absolutely mandatory*, is that correct?” They all smiled, “No, that’s not it. The top priority is *deal breaker*!”

What is wrong with the following feature-priority categories? [A, B, C], [*mandatory*, *absolutely mandatory*, *deal breaker*], [*breakable*, *unbreakable*], or MoSCoW.²⁷ All these schemes are based on big category groups, and frequently associated with the Contract Game in which a big feature set is pushed to R&D for the Development Phase... “Well, maybe at least we will get all the ‘A’ categories items at the end.” These schemes (1) reduce visibility of progress, flexibility, and control because of their coarseness, (2) they are based on big batch thinking with big queues, and (3) as in the case of [*mandatory*, *absolutely mandatory*, ...]²⁸ they are inherently flawed—there could always be something more important. They reflect a belief in magic or—in lean thinking terms—the waste of wishful thinking.

Item	Order
M	1
C	2
T	3
K	4
...	...

Traditional product management use priority categories, so when they shift to Scrum and first create the Product Backlog, this categorization (inappropriately) continues. The novice Product Owner is not yet thinking in the *fine-grained* control model of Scrum: items ordered [1, 2, 3, 4, 5, 6, 7, ...]²⁹.

Nor is the new Product Owner thinking of continually re-prioritizing the backlog and applying adaptive iterative planning. This is a significant shift in mindset and behavior.

27. MoSCoW—Must have, Should have, Could have, Will not have.

28. This group’s scheme did not start out like this.

29. The *Queuing Theory* chapter in the companion book discussed why it is not necessary to prioritize *all* items in the Product Backlog, but only those in the *clear-fine* subset of the Release Backlog.

CUSTOMERS AND R&D

Avoid...False dichotomy yes/no answers to customers

When a customer asks, “Can we have everything ready by May of next year?” it seems to beg a yes/no answer—given far too frequently. He wants to hear ‘yes’ as he expects that solves his problems, but if ‘yes’ ignores reality, it does not really solve them—and ‘yes’ introduces new ones that in the worse case devolve into a lose-lose situation for both customer and product company. This is the *waste of wishful thinking* in lean.

There is a lot in between ‘yes’ and ‘no.’ In a probabilistic world, an answer based on probability is more realistic and transparent: “We estimate there is a 40% risk of not doing everything in this goal. Here’s our reasoning and numbers.”

In addition to answering with probabilities, steer toward more collaboration, including more fine-grained prioritization for ‘partial’ solutions of the most important elements in their major goal. For example, rather than discussing coarse-grained groups such as “must have,” move the discussion to a force-ranked 1–N priority list of more fine-grained goals.

Try...Involve real users or customers in Sprint Review

Since every Scrum iteration ends in well-done features, there is opportunity for early and ongoing customer involvement, every Sprint Review from the first. Try that! This leads to increased feedback about what is important and attractive, new and better feature ideas, earlier testing of the business case, better insight about the usability, improved customer engagement, and more.

After attending, one customer said, “The Sprint Review provides an excellent occasion to share opinions and influence the products to be closer to what we really need.”

This suggestion may seem obvious, but in traditional development (with long-delayed or unstable features) some product managers were not used to this—and could be hesitant to try.

Avoid...*Product management or Product Owner between teams and users*

The advantages of hands-on development people directly interacting with real users are numerous... (1) handoff waste is dramatically reduced, (2) the team's appreciation of the customer context improves—leading to better judgement about solutions, (3) customers see increased engagement and responsiveness—leading to higher levels of satisfaction, and more. A bonus for product management is that some work previously done by them shifts to the teams.

In traditional large-scale development, a team seldom does complete end-to-end customer goals. For example, a component team only programs a fragment; a test team does not do requirements analysis. Consequently, the notion of putting a development team in direct contact with customers is unfamiliar. All this changes with the transition to feature teams that do complete end-to-end work. In this case, what a team does is profoundly relevant to real users, and by connecting teams and users, good things ensue. Try it.

Avoid...*Multi-level P-M indirection from customers to teams*

We encourage product management to connect teams directly with real customers; that said, P-M is a valid voice of the customer to the teams. However, when acting as “the voice,” stay away from more than one level of indirection. For instance, avoid

1. product manager Jack talks to customers
2. product manager Jill talks to Jack
3. Jill talks to teams

Jack needs to talk *directly* with teams!

Try...*Shift R&D language toward P-M and user language*

Encourage a shift in the R&D mental model and language towards P-M, business, and real customers. How?

- ❑ Write Product Backlog items that are understandable and meaningful to users, if they read them.
- ❑ Write items in a format that includes *user motivation*.
- ❑ Ensure split, smaller items remain customer-centric.
- ❑ Connect teams with users.
- ❑ During Sprint Planning and Sprint Review, encourage the Product Owner to share customer and business perspectives.
- ❑ Promote feature teams that do end-to-end customer goals.

See “Try...Split Product Backlog items (such as stories)” on p. 247.

See “Try...Ask, “Would users understand every PBI?”” on p. 238.

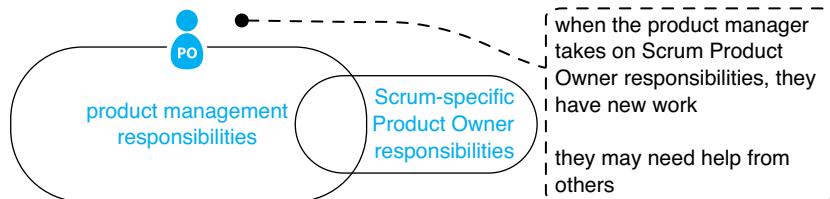
By these actions, teams can relate better to customers, and make decisions and suggestions that are more appropriate for the business and users—improved R&D alignment with business.

Why bother? Traditionally in large product groups, R&D people speak a very different—and technical—language because of their education, lack of direct engagement with users and P-M, splitting of tasks into architectural components, and more.

CHANGE AND IMPROVEMENT

Try...Extra help for product-manager Product Owner

The positive value of product management directly steering development (in their role as Product Owner) through their time with teams has been established. It brings with it one key complication:



Before Scrum, product management was already busy, and now they have additional responsibilities. Where can they get help, if needed?

First, *ask the Teams*. The pivotal notion of a cross-functional team is part of a larger picture of *cross-functional integration* across many traditionally separate departments: Marketing, Sales, and more. Challenge the assumption that no Team could learn requirements directly from customers, or help with marketing—you might be surprised. Challenge the assumption that no Team should help with sales. And if P-M connects teams directly with customers, teams may be able to take on some work previously done by P-M.

Second, *ask the ScrumMasters*. But do not ask them for help with the hands-on work—because that is not what ScrumMasters do. Rather, ask them for help analyzing why there is overburden, help in understanding Scrum, and help in creating experiments to improve.

Scaling—The need for help is proportionate to the number of teams that one Product Owner (or supporting PO) serves. For instance, one Product Owner collaborating with five teams needs support from several *subject matter experts* (SMEs) so that during Sprint Planning or Product Backlog refinement, the five teams are not blocked waiting for feedback or clarification.

Where do subject matter experts come from?—The Product Owner may find other product managers to serve as SMEs. Or, a member of a regular team for supporting PO *John* may be an expert, and temporarily help supporting PO *Jill* as an SME with her teams.

Avoid...*SMEs not talking to customers*

Caution: People outside of regular Scrum teams—A subject matter expert who is not a product manager or Product Owner should be part of a regular cross-functional Scrum team, not part of a specialist separate group. And watch out for SMEs who are disconnected from talking to customers.

Try...Product Management inspect and adapt

R&D and production workers, especially workers in software development, are *relatively* avid students of improvement ideas, and tend to experiment frequently. In Scrum and lean environments, frequent kaizen is the norm; Scrum teams hold a formal improvement workshop at the end of every iteration.

In contrast to this, we observe fewer improvement experiments among product management groups.

Ask a ScrumMaster to facilitate an improvement retrospective for product managers. And don't stop: As Scrum teams do, hold a retrospective to inspect-and-adapt improvement experiments regularly.

Try...Product management education

This experiment is related to the “Avoid...Too ‘inward’ product management & Product Owners” section on page 124. Some commercial product managers we meet have not been educated in the subject, and are not aware of P-M books, learning resources, or communities. Partly as a consequence of this, some product managers focus on inward activities such as writing product specifications.

In the case of internal products or applications, almost certainly the Product Owner (for instance, a business-area manager) has not been exposed to product-management education. While there are aspects of P-M that are not relevant to them (such as pricing and channel management), some aspects are, including product vision, ideation, and (internal) customer relations.

See the recommended readings for suggestions.

Try...Product Managers study Scrum & attend a course

Since P-Ms are entering the two-person cooperative game of Scrum, they need to understand the game rules and deeper dynamics. It is not enough have a vague notion of Scrum. At the most successful Scrum-adopting groups we have coached, product managers actively read Scrum books and attended a ScrumMaster course. Product Owner courses are also available, but we observe that things go best when the product management and development people are *learning together* in the same ScrumMaster course.

Try...Product managers Go See

A central lean principle is Go See at *gemba* frequently. Rather than sitting at a desk and looking at reports or emails, managers in a

lean enterprise “look with their feet” at the place where hands-on product-creation value work is being done. In an interview, Toyota’s chief engineer quoted Taiichi Ohno, who insisted on managers practicing Go See at gemba:

Don’t look with your eyes, look with your feet... people who only look at the numbers are the worst of all. [Hayashi08]

This applies to product managers—and all other management in a lean organization.

Try...Senior product managers coach

The foundation of lean is manager-teachers who are experts in the work and in lean thinking and who coach and mentor others. When we work with product managers, it is rare for us to observe a culture of mentoring or pair work, or of long-term senior product managers focusing on coaching. Try more of that.

Try...Invite displaced people to join product management

When moving to Scrum, certain roles are no longer needed, such as project managers. Product management is often chronically under-staffed and can benefit from more people—especially those already with knowledge of the product and organization. We have seen successful transition of displaced (valuable, experienced) people moving into product management.

Toyota emphasizes stable employment—somewhere in the organization—as an important part of continuous improvement. Without that, people avoid improvements that may eliminate positions. And it is simply an element of respect for people.

Caution—People joining product management need to act as *product managers*—and need genuine interest in this crucial area. If an ex-project manager joins and attempts to continue doing project management, there is misunderstanding.

CONCLUSION

Scrum, lean thinking, and agile principles involve product management—and many other business functions.

In traditional organizations there are systemic barriers created between P-M and R&D: from a business perspective there is a lack of visibility, *real* control, and flexibility. Both parties locally optimize in a *competitive* Contract Game that inhibits building the right thing and building the thing right—although the policies and practices behind these barriers were added with good intentions.

Key to dissolving the barriers between P-M and R&D and collaborating to deliver value to customers is this: end The Contract Game and move to a *cooperative game of invention and communication*.

This collaboration means that the product manager or business-area manager—in the role of Product Owner—and teams are directly interacting during the Scrum events. In that sense, the Product Owner is spending some time focusing inward—towards the teams. But a skillful Product Owner balances this with focusing outward—towards the customers and market. And they directly connect hands-on team members with real customers, encouraging the Scrum feature teams to help with some product management work, so that teams learn what is important to customers, and so that the Product Owner is not overburdened—one of the sources of wastes in lean thinking.

When scaling Scrum to multiple teams, there is only one Product Backlog per product; this is important to focus on optimizing the overall system, increasing visibility, and avoiding a fake or “waterfall Scrum” in which the old, slow team structure remains the same—merely overlaid with agile words. Scrum implies cross-functional teams that do complete end-to-end customer features; that has a non-trivial impact on most existing organizational designs—and it is critical for the Product Owner to grasp this point.

Finally, when scaling Scrum to a multi-hundred person product group, experiment with Area Product Owners and Product Owner representatives. But do not fixate on terminology; far more impor-

tant than choosing *overall Product Owner*, *chief Product Owner*, or alternative titles, is choosing to end the competitive Contract Game.

RECOMMENDED READINGS

- The *Product Development and Management Association* (www.pdma.org) offers online and printed learning resources (such as *The PDMA Handbook of New Product Development*), and an online list of classic P-M literature. Some of the material assumes the traditional Contract Game or sequential life cycle development, but much is still worth investigation.
- *Innovation Games* by Luke Hohmann emphasizes simple, creative, and collaborative techniques—applicable in workshops—for customer-focused product definition.
- *Agile Product Management with Scrum* by Roman Pichler explores envisioning a product in the context of Scrum development, the role of the Product Owner, and more.
- The text *Product Strategy and Management* is written by researchers with long in-depth study into product management and development. It contains many solid suggestions, and a vast number of references to the major papers and researchers in this field. This book is an excellent window into the breadth and depth of P-M-related research.
- *Product Management* by Lehmann and Winer is a solid introduction to market analysis, product strategy, pricing, distribution channels, and more.

This page intentionally left blank

Chapter

- Early days 155
- Iteration (Sprint) planning 163
- Done 170
- Estimation 181

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

PLANNING

If everything is going to plan, something somewhere is going massively wrong.
—anonymous

This chapter has planning experiments related to large-scale Scrum.

EARLY DAYS

Try...Kickstart large-scale Scrum with *one initial Product Backlog refinement workshop*

For a small group, setting up a new Product Backlog is relatively simple work—maybe a day. For a larger product—for example, a ship-control system or medical device—involving 500 people with new hardware and software, one day is not going to do.

In this case, consider holding *one and only one* relatively long, highly structured, and intensive **initial Product Backlog refinement (or creation) workshop** when an existing large product group first transitions to Scrum. This takes several days.

This activity is also known as **release planning** but we call it ***initial Product Backlog refinement***¹ for three vital reasons that *distinguish it from conventional release planning*, to communicate that...

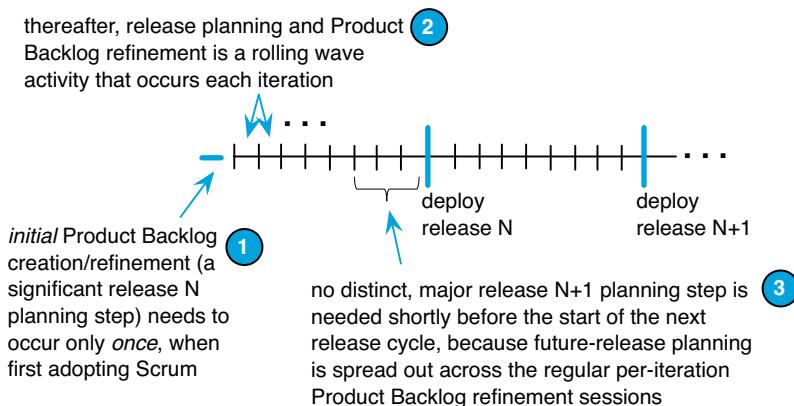
- ❑ *Initial Product Backlog refinement* has the same activities as the ongoing per-iteration Product Backlog refinement.²

1. It is not wrong to call it *release planning*; this is the name used in the *Scrum Guide*.

- ❑ The activity of release planning is never-ending in Scrum—it is part of ongoing Product Backlog refinement each iteration; naming it *release planning* suggests (strongly) to a traditional group that release planning happens *only once* for each release.
- ❑ *Initial Product Backlog refinement* needs to happen *once and only once for the lifetime of the product when the group first adopts Scrum*; thereafter, Scrum “release planning” is a *rolling wave* that happens every iteration.

This last point (and see Figure 5.1) is a significant change because big traditional product groups usually think (1) shortly before the next release cycle, do a major release-planning activity, (2) develop toward the release. This mindset is a variation of “order the meal, then wait for delivery of the meal”—which is associated with the traditional Contract Game, and inconsistent with the *cooperative game of invention and communication* in agile planning.

Figure 5.1 *initial Product Backlog creation and distinct, major release planning needs to happen once and only once*



-
2. Also called Product Backlog refactoring or *grooming*—an evocative phrase to a *native-English* speaker, but we have learned (working frequently in Asia and Europe) that ‘grooming’ is an unfamiliar word, and so use the familiar ‘refinement.’

Try...Continuous product development rather than projects

This suggestion is emphasized in greater detail in the *Organization* chapter of the companion book, in *Avoid...Projects in product development* and other experiments.

'*Projects*' are often assumed as the best or sole way of organizing work; for instance, big product groups assume long projects (or programs) for big releases: (1) major release planning at the start, (2) a long development period, (3) release. But a *project*-orientation has several drawbacks, including a short-term focus in which long-term improvement and quality is sacrificed.

Projects can be reduced or eliminated in Scrum, replaced with the simpler model of **continuous product development** (Figure 5.2). See the *Organization* chapter of the companion book for more detail.

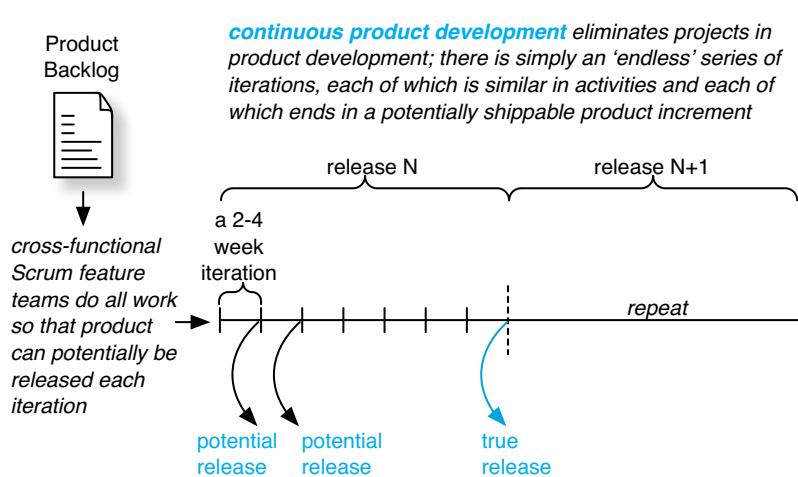


Figure 5.2 move to continuous product development rather than large *projects* that plan, execute, and release

Continuous product development does not mean there is no release goal or no Release Backlog. Those still exist, but a release cycle is not treated as a special distinct project.

There are exceptions to this continuous model; for example, game development tends to be single-project oriented.³ However, most products (including internal products) are long-lived and go through

series of evolving releases; *continuous product development*—rather than projects—applies in these cases.

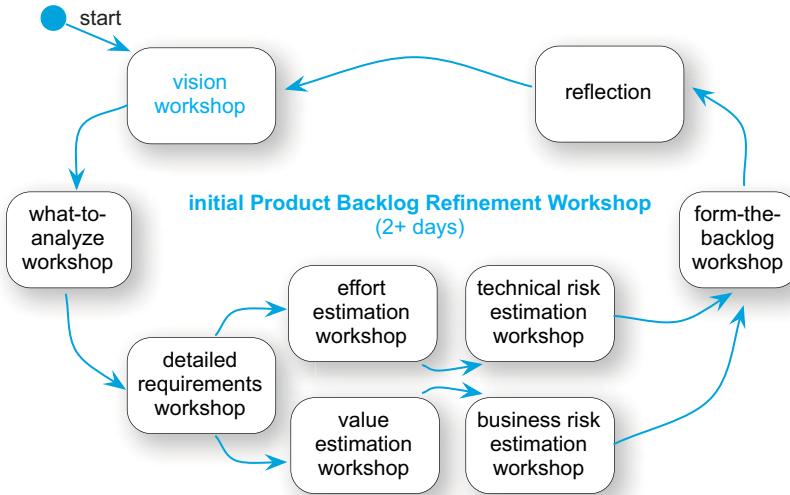
Try...Initial Product Backlog refinement workshop

See
“Try...Requirements workshops” on p. 240.

How to do *initial* Product Backlog refinement/creation? We have facilitated these events over the years, and have suggestions on participants, environment, tools, order, and activities.

Participants—Include the Product Owner Team and all team members, or representatives if too many people; workshops with more than 50 people are unwieldy—even this size is only effective with a skilled large-workshop facilitator.

Figure 5.3 sample activities in kickoff workshops



Workshop facilitator—A facilitator is critical for a multi-day, intensive structured workshop with many people.

Format—*Facilitated workshop* (with active hands-on activities), not a meeting (where people sit and listen, or give presentations).

-
- 3. There is a subset of core technology (such as a game engine) that may evolve across releases, but much is replaced and new.

Environment—A *big room* (see the *Multisite* chapter for variations).

Tools—Lots of simple, tangible tools for creativity: paper, cards, whiteboards, flip charts, and so forth. Digital camera for pictures of tangible things is useful. Use a wiki to store pictures and any detailed text. Include several computer projectors so that a subgroup can easily see the wiki material or other on-line resources—but in general, avoid using computers or projectors.

Order and activities—The overall workshop will contain a series of sub-workshops, each lasting from a few hours to several days. Consider the following sub-workshops (Figure 5.3):

Vision Workshop: Envision business case, strategy, high-level list of features (major scope), and constraints for the release.



Figure 5.4 agile vision workshop—brainwriting and mind mapping



What-to-Analyze Workshop: Identify ten or twenty percent of the features (for example, 20 of 200 items) that deserve deep analysis immediately.⁴ Choose a subset that will yield broad and deep information about the overall release work. There are always some key features that, if deeply analyzed, give you overarching information about the big picture.

These may be the most complex, the most architecturally influential, the highest-priority features, or features with the most obscurity. From the viewpoint of information theory, they represent a subset that, if analyzed, is most likely to have lots of *surprising* information—the most valuable kind [Reinertsen97].

4. If development is for a *fixed-price fixed-scope* project, then this may need to be much larger than 20%. See the *Contracts* chapter.

See “Try...Split Product Backlog items (such as stories)” on p. 247.

See “Try...Learn many analysis skills: user stories, use cases, ...” on p. 268.

Detailed Requirements Workshop: Hold a “deep dive” requirements workshop on the ten or twenty percent identified in the what-to-analyze workshop. This may be a relatively long workshop. It also includes splitting coarse-grained requirements into smaller ones. Myriad analysis techniques are applicable. At the end of this workshop, ten percent (for example) of influential requirements are *well-refined*—better understood in detail and split into smaller sub-items—and ninety percent less refined.

For very large groups, *requirement areas* are identified during the vision workshop or near the start of this workshop. These areas can be used to concurrently analyze requirements in several sub-groups.

Parallel Effort- and Value-Estimation Workshops: In the Product Backlog, all items have both effort and ‘value’ estimates.

Team members head for one end of the room and initiate effort estimation of items identified in the Vision and Detailed Requirements workshops. Typically done with planning poker (see Figure 5.5). These workshops usually take a half- to full-day for a big release.

See “Try...Prioritize with multiple weighted factors” on p. 141.

In parallel, the Product Owner Team heads for the other end of the room and initiates estimation (perhaps with planning poker) of the ‘value’ for items identified in the Vision and Requirements workshops. See “Try...Value” on p. 139. As explored the *Product Management* chapter, ‘value’ is not one simple attribute.

Figure 5.5 parallel value and effort estimation in the same workshop room



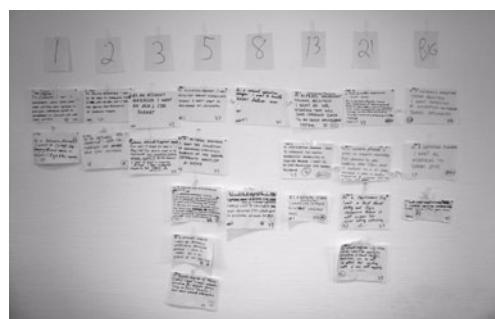


Figure 5.6 user-story cards on the wall, with estimates created with planning poker

Parallel Technical- and Business-Risk Estimation Workshops: Risks can be grouped into those related to the development teams or the Product Owner Team:

- *technical risks*—uncertainty (of outcomes or events) regarding development, technologies or their performance [Reinertsen97]
 - development teams have something to say—about identifying, estimating, and mitigating these
- *business risks*—uncertainty regarding the business case, strategy, market, economy
 - Product Owner Team has something to say

Technical and business risks are inputs into the prioritization of the backlog. Why? Because to either *fail fast* or mitigate risk, work on items that attack probable and costly risks *in early iterations*.



Form-the-Backlog Workshop: At this point, the group has several hundred cards with item summaries, each of which has effort estimates, and a set of ‘value’ and risk attributes.

Separate the Release Backlog⁵ and future backlog subsets—The first step in forming the Product Backlog is to create two groups on the floor: (1) items for the Release Backlog and (2) items for future

5. The Release Backlog is not a separate artifact; it is the part of the Product Backlog for the next release.

releases. How to identify items for the Release Backlog? In short, the choices are influenced by a date-driven or content-driven release goal, and the small-scale agile techniques for either case also apply to larger systems; see *Agile Estimating and Planning* [Cohn05] and *It's All in How You Slice It* [Patton05] for examples, and the “Try...Prioritize with multiple weighted factors” section on page 141.

Some scaling-specific suggestions for the initial Release Backlog:

- Parallelize the initial Product Backlog creation if there are requirement areas and Area Product Owners. The cards are separated into different areas of the room, by requirement area. Each subgroup works in parallel. The overall Product Owner visits each group, and team members visit other areas.
- Large product groups are especially accustomed to playing the traditional development game with The Content Milestone. Old habits die hard, so we see workshop participants over-process the Release Backlog creation, concerned it must be ‘correct.’ The facilitator needs to emphasize that people can relax—it is just a rough approximation.

*See
“Avoid...Product
management
negotiating a
“release con-
tract” (scope &
date) with R&D
on p. 106.*

Separate the clear-fine and vague-course subsets—Once items have been separated into the Release Backlog and future backlog sets, the next step is to further separate the Release Backlog items (on the floor) into two sub-groups: (1) clearly analyzed fine-grained items that are small enough to be done by one team in much less than one iteration, and (2) the remaining vaguely analyzed coarse-grained items.

Prioritization of the clear-fine subset—The Product Owner can do the *fiddly* work of prioritizing the Product Backlog after the workshops are finished. Only items in the clear-fine set need prioritization since they are the only candidates for implementation. An exception to delaying the prioritization is if the iteration starts the following day.

Type it in? A single-site product group can experiment with using only visual management (such as cards on the wall) to record the Product Backlog. A multisite group probably needs a spreadsheet,

for easy sharing. In that case...many hands make light work: Ask everyone to type in a few cards.

Reflection: End with a *short* retrospective on the workshop process itself, to learn and improve.

Many of these activities repeat each iteration: A non-trivial initial workshop is needed to kickstart adoption of Scrum with a well-formed Product Backlog. As a big event, that happens only once, but some or all of these activities repeat each iteration in smaller, ongoing refinement workshops.

ITERATION (SPRINT) PLANNING

Try...Scaling Sprint Planning Part One

We and our clients have experimented with several approaches to scaling Sprint Planning. First, note that there are two distinct steps in Sprint Planning: Part One (SP1) that focuses on *what*, and...wait for it...Part Two (SP2) that focuses on *how*.

Related to large-scale Scrum framework-1 and framework-2, a tipping point in the SP1 scaling-approaches happens at *around* ten teams (for the total group). Ten is not a magic number; it is related to the upper bound of people that can effectively meet together in one common SP1 meeting, and the ability of the Product Owner to focus on the big picture. At some point, the size is unwieldy—although that point is context sensitive. For example, larger groups remain effective with good facilitation, practice, clear requirements, or long-term stable teams.

see the Large-Scale Scrum chapter

Suggestions...

Around Ten or Fewer Teams

In this case, do SP1 for the entire product group in one meeting (unless there are time zone constraints involving multisite development). If there are only a few teams, it is possible for the entire group (such as 14 people) to come. Otherwise, send one or two repre-

sentatives from each team; sending *two* representatives has the advantage of providing multiple perspectives for each team. As discussed in the *Coordination* chapter, *avoid* the ScrumMaster as a team representative—at this and most other meetings.

Start SP1 by viewing the existing baseline product-level Definition of Done, to ensure that this key point is clear.

The Product Owner spreads out wish-list cards that summarize backlog items (and their priority), probably grouped into *epics* or *themes*. She invites teams to volunteer for items. A creative period ensues in which the teams make tentative decisions on the items they will take forward to SP2. Each team physically moves and groups cards they will take forward. If there are grouped items, it is normal for one team to take the group (or manageable subset), so that there is increased consistency and cohesion. Furthermore, it is normal for one team to take new items related—by epic or theme—to old items they did previously, for the same reasons.

At the end of this period, the floor will have groups of cards for each team, and perhaps leftover cards that no team has chosen.

Offerings directed to individual teams?—To encourage self-organization, experiment with teams deciding among themselves—by interest, negotiation, or skill—which teams will pick up which items. This also reduces decision-making effort by the Product Owner. However, the Product Owner has the final decision on which items are matched with teams, so if there is a problem, she decides the distribution. Also, if a high-priority item is not picked up by any team, or high-priority items are not spread across teams (see next point), then the Product Owner can decide to change the distribution.

Spread high-priority items across teams—Potential problem: Assume two teams. During SP1 Team-1 takes items with priority [1, 2, 3, 4] and Team-2 takes [5, 6, 7, 8]. Later, during SP2 or the iteration, Team-1 descopes item-4. Result? A relatively high-priority item (item-4) has been descoped, even though (perhaps) Team-2 could have done it. Solution? Spread high-priority items across teams; for example, Team-1 takes [1, 3, 5, 6] and Team-2 takes [2, 4, 7, 8]; this is not always appropriate due to relatedness or dependency of items.

This seems too easy—If Scrum feature teams are in place, and the Product Owner and teams are applying Scrum with skill, then even with multiple teams SP1 *should* be simple, quick, and without many questions. The items being offered should already—before this meeting—be small, clear, estimated. The Product Owner should already know the priority. And a team should be able to independently do an end-to-end item by itself. If SP1 is complicated and filled with questions (or silent confusion), that signals lack of preparation in the previous iteration. If SP1 raises many cross-team coordination problems, that signals lack of true feature teams.

More than around ten teams

This is more-or-less the tipping point where it is useful to introduce *requirement areas* and *Area Product Owners* (APOs)—each APO served by a maximum of around ten teams. We have seen this scale to groups of about one thousand people.

See “Introduction to Requirement Areas” on p. 555.

Pre-Sprint Planning—Before SP1, in the previous iteration, the APOs and overall PO (if there is one) need to coordinate so that the different Area Backlog priorities reflect product-level themes or other coordination agreements.⁶ For instance, if the theme of the next iteration is *touch interface*, the APOs coordinate so that related items rise in priority in all Area Backlogs.

SP1—Area-level Sprint Planning Part One—Simply, in parallel meetings on the first day of the iteration, each APO and their requirement area teams (or team representatives) meet and hold SP1 exactly as described in the previous section.

Multisite Issues

There are several multisite issues, including (1) the participants in SP1 not working at the same site, and (related) (2) no common time.

For a multisite SP1 meeting, one approach is to organize the display and choosing of items with a software tool, such as Google Spreadsheet, combined with video (such as Skype video) or audio conferenc-

6. For example, one APO taking on a common infrastructure goal.

ing. *Caution:* If the overall Product Owner or an APO is physically in the room with a subset of teams of one site, there is tendency (due to communication ease) for the APO to favor the local teams in some way—in terms of information, clarification, and so forth. Pay attention to that. We have seen an APO travel to different sites over time for Sprint Planning and Sprint Review—this supports balance and building relationships between the APO and teams.

No common time window—For example, some teams are in New York and in Singapore—about 12 time zones apart. Experiment with the Product Owner deciding which set of items to offer to each site and holding separate SP1 meetings—one of which will be in the evening for the Product Owner.

Try...Simple Sprint Planning Part Two

If the scaling practices for SP1 have been applied and the organization has created feature teams, then SP2 is simple: Each separate team individually doing—more-or-less in parallel—their own individual SP2 meeting.

A variation is *asynchronous* SP2 meetings so that some people can observe other meetings, when there is a coordination interest.

Try...Asynchronous or joint Product Backlog refinement

If a team wants to coordinate with another, or learn from another:

- Hold Product Backlog refinement workshops at different times.
 - a few people from other teams can observe or participate
- Hold a joint workshop.

Try...Plan bounded research or learning items

Large product groups, especially those for embedded-software products, frequently have genuine and complex research work. What new color models can be used in printing? What is IEEE 802.11ad? As discussed in the “Try...Genuine research work as PBIs” section

on page 227, genuine research work can and should be identified and planned in Scrum, similarly to regular feature items.

For example, we were once consulting at a site in Budapest; the group wanted to provide “push to talk over cellular.” The international standards document for this is *thousands* of pages. Just to vaguely grasp the topic is a formidable effort.

One approach is to ask a team to “study the subject.” Yet, that is fuzzy unbounded work, so it leads to more variability and a big batch of analysis—not all of it necessarily useful.

An alternative approach—this suggestion—is to plan an effort-bounded goal to learn within the iteration. For example, maximum 30 person-hours on “push to talk” in the next iteration. This item and its limit is clear in SP1; the team knows how much this otherwise unbounded work will impact their SP2 planning, and the Product Owner is making a bounded investment in this research.

Suggestions:

- This research item should not consume all the time of the team for the iteration; balance it with development work.
- Focus research work on *useful output* for the Product Owner, especially identifying new Product Backlog items (PBIs); limit study; quickly prototype and implement.
- Focus research work on *tangible output* for the Product Owner, such as a research report presented at Sprint Review—ranging from an oral presentation to wiki pages. Include advice to the Product Owner on how much to invest in future research.
- Avoid giving research items to special ‘research’ teams; research is done by normal Scrum teams, not separate groups.
 - this reduces the waste of handoff, and increases learning by the teams doing the real development work

To repeat an example from the *Requirements* chapter: Rather than studying in depth a 500-page document of the next PDF specification,⁷ people can

7. Printer-product companies write software to interpret PDF.

1. skim over the document and identify large sections that can be studied in detail later⁸
2. study only a smaller subset in detail
3. create new customer-centric PBIs; prototype; ...
4. start implementation

After the iteration, the Product Owner knows more and can decide to invest more bounded effort in another cycle of research, and can perhaps start implementation of some concrete PBIs to incrementally build up the functionality.

Try...Plan infrastructure items by regular teams

This is a variation of the “Try...Add and do a cross-product common goal” section on page 128 in the *Product Management* chapter. Review that section for issues that apply to this case; consequently, this section is terser than it would be otherwise.

When we start coaching a big group, we often find the *Infrastructure Team* (a smaller variation of a Platform group that serves multiple products). The conventional idea is to collect requests⁹ for shared components or services from across several teams in the product group, which are then implemented by the Infrastructure Team “for efficiency.” And sometimes there is an Infrastructure Backlog, separate from the Product Backlog, perhaps even with an Infrastructure Owner.

In contrast, this suggestion is to put shared goals for common infrastructure on the Product Backlog, done by regular Scrum feature teams, temporarily playing an infrastructure role (Figure 5.7). The motivation is analyzed in the related *Product Management* chapter and the *Feature Teams* chapter of the companion book.

8. Reading is itself a skill; see *How to Read a Book* [AD72].

9. Often called “technical requirements”—which are in fact not requirements, but design solutions.

Iteration (Sprint) planning

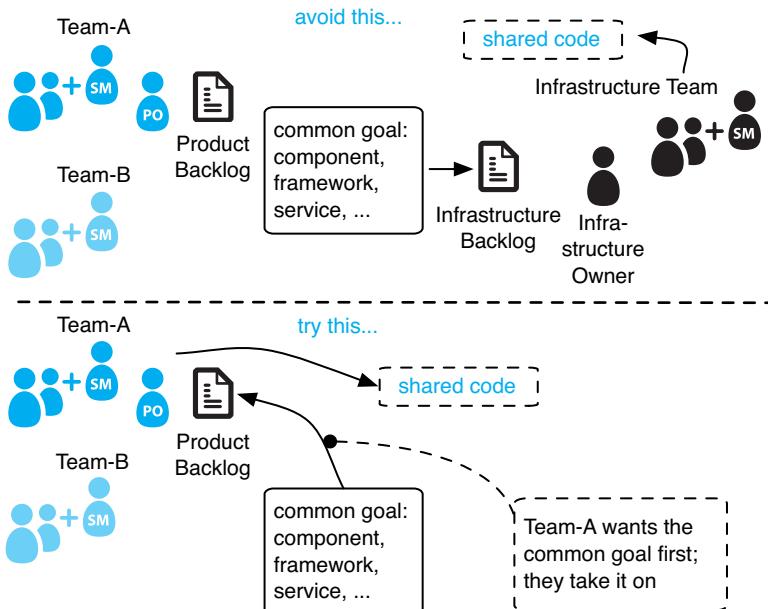


Figure 5.7 a feature team, not a separate infrastructure group, takes on infrastructure goals

Try...Avoid... Fixing defects

One of our clients builds ship-control systems; they have a category of defect called 'sinking.' Those don't wait. How to plan for these surprises? One approach is for teams to include some *bug slack* in their Sprint Planning; in the best case they can use the slack time towards new-feature work rather than for 'predictable' surprises.

Another approach: Large, old systems invariably have a long list of known defects. How to plan? Some of our clients have $<N>$ feature teams move into the role of bug teams for one or two iterations. All defects are handled by these teams, even if they are not the fastest experts—and in this way the teams learn. *Component guardians* may also be involved in helping or review. Team members only interrupt an *über-expert* (in this example, not currently in the bug teams) if the problem is extremely time- or expertise sensitive. Bug teams eventually move back to the role of regular feature teams, and fresh feature teams become bug teams. During the transition back to fea-

"Try...Component guardians for architectural integrity when shared code ownership" section on page 314

ture team, people carry work-in-progress defects into their ‘feature’ iteration, wrapping them up before starting their first feature.

See “Try...Zero tolerance on open defects” on p. 39.

Avoid—As a perfection challenge, there should not be any defects, and so planning for them should not be necessary. Although this is tough in the old systems we work with, we encourage agile coaches to help create the culture and engineering practices for zero tolerance on open defects.

Also: See “Avoid...Using defect tracking systems during the iteration” on p. 39.

DONE

Try...Product-level Definition of Done

Definition of Done

“We are done!” said the team. But what does that mean?

The **Definition of Done** (DoD) is an agreement between the Product Owner and teams about what ‘done’ means. When team states ‘done,’ what will that mean? Does it mean that all testing has been done? How about the customer documentation?

The perfection-challenge output of an iteration in Scrum is called a **Potentially Shippable Product Increment** [Schwaber04], that is “[done] in the sense that it could be sent out to the marketplace” [Schwaber06]. But for a product—the kind we commonly work with—that has a background of *five-year release cycles*, 100 component teams spread across 13 sites worldwide, and many single-function departments...that is an *unimaginable* step.

Therefore, a big group defines their DoD as their current technical and organizational ability—typically starting below the perfection challenge. What are they capable of doing each iteration when starting Scrum? For any group, this includes at least programming and some kind of testing. Over time, as the group improves, their DoD expands until it is equal to truly potentially shippable.

Do not confuse the DoD with *criteria of satisfaction* or *acceptance criteria*. The first are valid for all items in the Product Backlog, whereas the latter are item-specific criteria that evolve in acceptance tests, probably during a requirement workshop with A-TDD.

Work of iteration = Product Backlog items + Definition of Done.

Here is an example (imperfect) DoD from one of our clients building a large product:

Category	Definition of Done	Details and Exceptions
Implementation	Item implemented in SW build.	
	No open faults caused by item.	
	Code peer reviewed & improved	
	Coding style followed.	For changes on top of 3rd party component such as Linux, we follow their existing coding style.
	Code cyclomatic complexity at 'good' level.	Complexity for new code below 15 for all routines, no increase in complexity for code changes based on inherited code.
	Static analysis done; no new warnings.	
Testing	Unit tests done for new and old changed code.	
	Unit tests under version control, peer reviewed & improved, and run automatically by CruiseControl.	
	Integration & functional testing done for items on the latest working platform build.	Integration and functional testing shall be done with the build which was available two working days before the Sprint Review.

Category	Definition of Done	Details and Exceptions
	Integration & functional test cases peer reviewed & improved; under version control.	Use the I&V Checklist in review.
	Integration & functional testing in target environment.	
	... more testing elements ...	
Documentation	Requirements are documented, peer reviewed & improved, and under version control.	Requirements are recorded (1) in English in wiki or (2) as automated test cases in Robot Framework.
	Technical design documentation done, peer reviewed & improved, and under version control	In wiki; see “Design Documentation.”
	API documentation updated, peer reviewed & improved, and under version control.	Automatically generated from source code and comments.
	Customer Doc Inputs done.	See separate input list for each guide-type customer document.

Product-level baseline

For meaningful tracking of overall progress, there needs to be a common product *baseline* DoD that all teams conform to. Our clients usually record this common definition in a wiki page.

In some cases, there is also an intermediate *requirement area* baseline that extends the product baseline.

When to first define it? In a special workshop that must include involvement by the hands-on teams. If there is contention about the original definition, the Product Owner has the final say.

When to evolve it? During Joint Retrospectives.

See “*Try...Joint Sprint Retrospectives*” on p. 403.

Why bother? Without a product-level definition, there is

- reduced visibility into the state of the overall product—it is not possible to track meaningful system-level progress
- less focus on the overall system—there is less discussion and attention to the state of whole product
- increased variability—different features can and will have different degrees of Undone Work
- reduced ability to deliver a potentially shippable product increment each iteration—some features can and will have more Undone Work than others

Avoid...Definition of Done defined by quality group

We coached a group in India that included a quality manager who wrote checklists and wanted teams to follow those. His focus was on centralized defined processes and top-down conformance. When this group started to adopt Scrum, he injected his checklists and centralized conformance into Scrum by defining the DoD. Avoid that—the DoD is an agreement between the *Product Owner and teams*. It is not an agreement with a quality group.

Avoid...Undone Work

When the DoD does not *yet* include all the work needed to ship the product to the customer—common in big groups first adopting

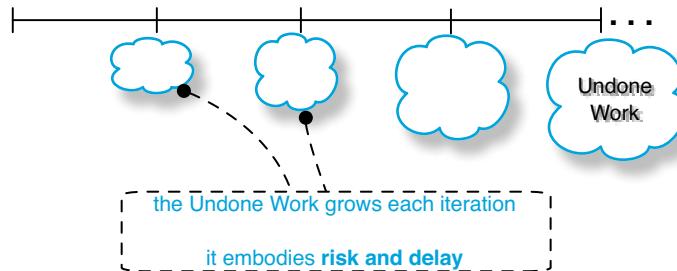
Avoid...Needing a Release Sprint

Scrum—then work is left before the product can ship. This is the **Undone Work**,¹⁰ and it increases each iteration.

Potentially Shippable	
Done	Undone
<ul style="list-style-type: none"> • implementation • unit tests • functional tests • architecture doc update • customer doc 	<ul style="list-style-type: none"> • performance tests • reliability tests • deployment • marketing material updated • support people trained

For instance, if the group cannot (yet) do performance testing each iteration (perhaps it is not automated and is outsourced to Vietnam), then this is Undone Work and the amount of performance testing increases over time, building up (implicitly) as a big batch of work in a queue to be done before release (Figure 5.8).

Figure 5.8 Undone Work grows over time



See
 “Try...“Undone Work” and system-level NFRs as PBIs” on p. 225.

Risk and delay—Since the Undone Work has to be done before shipping, it represents delay for the Product Owner—and therefore also responsiveness. Undone Work also represents risks—performance testing being an excellent example. When it is delayed until near the end of the release and the group discovers performance problems late in the game...bleh!

10. Not to be confused with unfinished work from the iteration. The Undone Work exists by intention or plan.

Also: See “Avoid...Try...Separate “Undone Work” from the Product Backlog” on p. 226.

Hardening—When we first visit a client new to lean or agile development, a word we frequently hear is...*hardening*. Such as, “We are planning to do three iterations, and then do a hardening iteration.” ‘Hardening’—an aspect of Undone Work—is so common that some do not see it for what it is: a *failure*, not a solution. In lean thinking, *defects* and then all that follows (test and fix at the end of a cycle) is considered waste. The lean perfection challenge is to *build quality in*—through *changing the system to* address the root cause problems—so that hardening and similar superficial *quick fix* reactions are no longer needed.

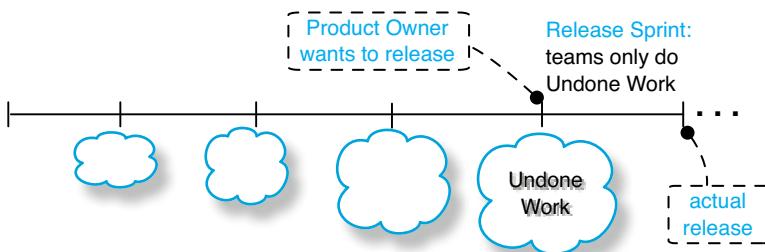
Avoid...Needing
to 'harden'

But, sometimes there still is Undone Work. Then, how to do it?

Try...Include Scrum teams in a Release Sprint

In small-scale Scrum, the standard solution to do the Undone Work is to hold a **Release Sprint** in which the regular Scrum team does all this work—they do not work on new features—and then releases the product (Figure 5.9). Good idea¹¹—especially because it *educates* the team through painful experience about this Undone Work. Since they are “eating their own dog food,” the team is increasingly likely to think of ways to reduce it in the future—to improve the DoD and reduce the Undone Work.

Figure 5.9 Release Sprint



11. The need for a Release Sprint is not good. Rather, if needed, including the teams is good.

As a rule, big groups moving from sequential development have a dedicated “Undone Unit”—the group that handles all the Undone Work such as documentation for field engineers, acquisition of part numbers, system testing, fulfillment of legal or government regulations, and so forth. When there already exists a big *institutionalized* Undone Unit that has done this end-of-release work before, it is mighty tempting to fall back on old habits and simply, once again, hand over the Undone Work to them. And we have seen ‘Undone’ management groups with a stake in keeping the Undone Unit alive—they press the Product Owner (or someone) to give them all the Undone Work.

Resist that temptation.

Rather, (1) hold *one* Release Sprint with some or all of the regular Scrum feature teams. For exactly the same reasons as above—learning through eating their own dog food. Also, (2) mix Scrum team members with the Undone-Unit experts, to reduce handoff problems and improve two-way learning (Figure 5.10). Examples:

- One of our clients is a bank. They have a production operations group that, before adopting Scrum, received a candidate release and tested it in a sandbox. After adopting Scrum, regular Scrum team members physically join with (pair-work with) product operation people during a Release Sprint to do this testing in a sandbox.
 - This is a temporary phase. The longer-term change is for the production operations group to diminish, and some members join regular development Scrum teams. Then, regular Scrum teams will have development knowledge to help with operations.
- One of our clients builds ship-control systems. They have a ship-installation group that installs a control system (wires, hardware, software, ...) in a ship. They are exploring including regular software-development team members on the ship during installation, to help. (This change is complicated by safety and insurance issues).

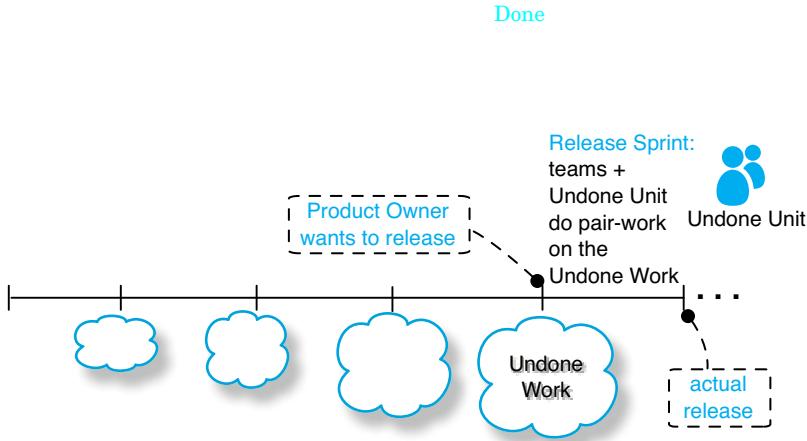


Figure 5.10 teams do a Release Sprint with the Undone Unit

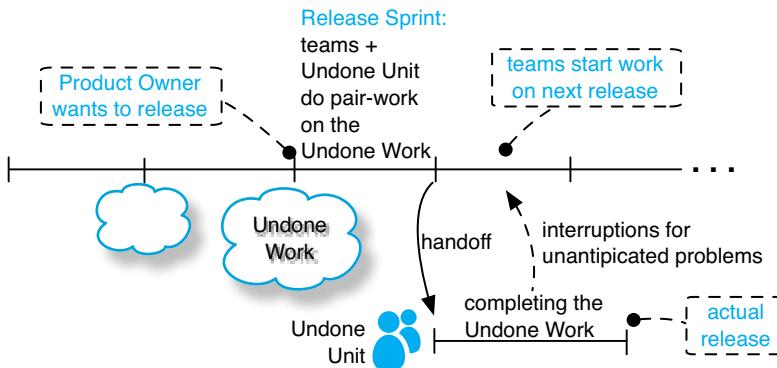
This is a good step, but there is a *mountain* of Undone Work in large embedded-software products new to Scrum. One Release Sprint may not be sufficient to *level it* and to release. Therefore...

Try...After one Release Sprint, hand off remaining Undone Work to the Undone Unit

After one Release Sprint with the teams and Undone Unit working together, if Undone Work remains, then experiment with the Undone Unit taking over the remaining pile. Fortunately, since the teams and Undone Unit have done pair-work in the Release Sprint, the handoff waste will be lower than otherwise. For example, in the prior ship-control system example, after a few weeks, most Scrum team members leave the ship and the expert installation crews remain behind.

Bring teams back to the rhythm of working on new features—at this point, for the next release cycle (Figure 5.11).

Figure 5.11 try this when Undone Work remains after one Release Sprint



Try...Reduce—and eventually, remove—the Undone Unit over time

Try...Expand the Definition of Done

Never lose sight of the perfection challenge: to have the *potential* to release, completely done, at the end of any iteration. That capability extends beyond just the R&D department—it includes product management, marketing, delivery, field support, and more.

Over time, (1) expand the DoD (Figure 5.12), which implies (2) shifting Undone Unit experts into regular Scrum teams so that the regular teams can truly wrap up by themselves in one Release Sprint; this implies an increase in multi-skilling. It also implies the end or a diminished role for a smaller Undone Unit.

In general these are the ways of expanding the DoD:

- automate—for example, performance testing is automated
- expand team cross-functionality—for example, a person with technical-writing skills joins the team

In big complex products, this is a long-term journey; see the “Try...Lower the waters in the lake” section on page 407 for a sense of perspective in this continuous improvement.

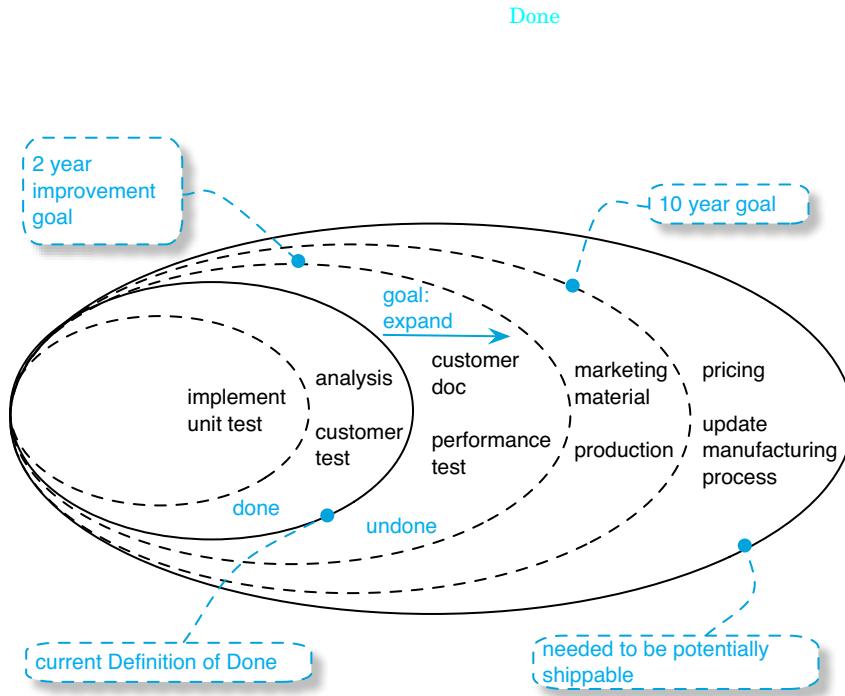


Figure 5.12 expand done over time

Try...Expand team-level Definition of Done

No team should be unable to conform to the product-level baseline DoD, but each is encouraged to expand beyond it, according to their capabilities. This is discussed in team-level Sprint Retrospectives and agreed to with the Product Owner in Sprint Planning.

Try...Avoid...Early and incremental handoff of Undone Work

The previous tips on Release Sprints posit a *false dichotomy*: the only two choices for the Undone Work are (1) do it during a Release Sprint, or (2) get rid of it by expanding the Definition of Done. But there are gradual-improvement alternatives...

Try—Hand off (to the Undone Unit), early and incrementally,¹² Undone Work that the teams cannot yet do themselves (customer

12. Incrementally every *three* iterations, for example. If, on the other hand, it was possible to hand it off *each* iteration, it would be time to *stop* handing it off and, instead, extend the Definition of Done.

documentation) or that takes too long to complete within one iteration (long-running manual stability testing). As another example, internationalization of user-interface text may be handed off early and incrementally. This reduces the wastes of delay and work-in-progress (WIP), and confronts some risks earlier.

Avoid—But on the other hand, this still engenders the wastes of handoff, delay, and WIP. And once there is a ‘successful’¹³ handoff in place (especially once it can be done every iteration), that suggests it should *soon after* be possible to expand the Definition of Done and remove the handoff.

This approach to gradual improvement is used by several of our clients in big embedded-software products, with mountains of difficult-to-remove Undone Work.

Avoid...Try...Planning an ‘agile’ release train

An “agile release train” [Leffingwell07] is an incremental release planning, integration, and system-test (‘hardening’) strategy for managing dependencies between teams—the type created by having component teams. In a release train, there are predefined periodic (such as every four iterations) internal release dates for system-level integration and testing during an ‘hardening’ iteration. The hardening iteration is a mechanism to work on the growing queue of WIP and Undone Work. The internal release dates are inviolate, but component teams have flexibility to decide their functionality and whether to “join the train” or not. And if not, they provide a simple “plan B” component version (such as an old version) to not break the new integration build.

Avoid—Release trains partially solve problems that can be solved more powerfully and fundamentally with other methods, such as adopting feature teams, doing real continuous integration with test automation, and preventing Undone Work by expanding the Definition of Done. Release trains add additional planning, coordination, and management overhead—a “release train management team.” System integration and testing is not delayed until the final end,

13. A ‘successful’ early-and-incremental handoff process does not imply improvement is finished, or the process should remain.

but...there is still some delay in integration and testing, and Undone Work. By retaining component teams—that create dependency and integration problems—system-level delivery of value is still slow, and there are more queues, WIP, handoff, and multitasking. Related to the prior experiment, “Try...Avoid...Early and incremental handoff of Undone Work” section on page 179, a release train could involve some handoff to an Undone Unit, such as a separate system-level testing group.

Try—If, for some reason, the group is unable to adopt feature teams with continuous integration, then a release train approach is at *least* better than a classic sequential life cycle.

ESTIMATION

Try...Estimate with Story Points

In *Agile Estimating and Planning*, Mike Cohn makes a good argument in favor of using *story points* (relative effort points) for estimation. We do not repeat the motivation, but note that this advice is even more compelling in development with many teams. Why? In this case, other—more detailed—estimation techniques (such as work-breakdown person-day estimates) are extraordinarily slow and laborious. Thus, there is disinclination to refresh the estimates—even though *updated estimates* are useful in planning, and are part of Product Backlog refinement each iteration.

But story points, especially when combined with techniques such as *planning poker*, are *relatively* quick and easy to refresh. Especially in large-scale development, that translates to an increased chance that re-estimation will occur each iteration.

Problem: Story points are relative—‘5’ has no absolute independent meaning. Two teams can define ‘5’ differently, which makes it difficult to estimate overall effort or track overall progress. Therefore...

Try...Avoid...Synchronize points and range

If the product group makes a common agreement on the size of story points¹⁴—so that a ‘5’ is the same for all teams—there are benefits:

- a common Release Burndown chart—better view of progress
- a product-level velocity—better prediction of future progress
- a team’s estimate of items that can be done by other teams
- the time together to create a common agreement that builds more shared understanding and cross-team relationships

How to define a common point, and a common range of points? One approach is...

Cross-team Estimation Workshops for Canonical Set—All members—or team representatives—of all teams join in a common estimation workshop and identify items for which they have *common understanding*. Then they estimate these, using planning poker with story points. This *canonical set* of items is a baseline of shared understanding and is used as the baseline of future estimation workshops, including those done separately by individual teams or larger sub-groups. This cross-team estimation workshop occurs not only before the first iteration, but repeatedly in subsequent ones to resynchronize.

There are at least two ways that the group can have common understanding of the *meaning* or *effort* in a canonical set:

- Historical set of completed items is reviewed. Likely the best choice, since the items were done and so have the clearest information.
 - this gives common understanding of effort but not necessarily of meaning
- New* items are first analyzed together by the estimators in a previous requirements workshop.

14. A common agreement also fosters overall product-level perspective among the teams, rather than isolated-team mindset.

Issues—Synchronizing points across teams works *relatively* well, but requires occasionally repeating cross-team estimation workshops,¹⁵ since they drift out of synchronization. Synchronizing also brings with it a few potential dangers:

- ❑ *comparing*—we have seen management groups start to compare the relative ‘performance’ of teams that use a common synchronization, even though such comparisons are in reality meaningless—and worse, harmful.
- ❑ *converting*—one merit to story points is that they have no absolute or independent meaning. They cannot be realistically converted into person-days, for example—and the attempt to do so by management (or others) indicates that the purpose of their use in *velocity* has not been understood. We have seen teams attempt to synchronize by stating, “one point means two person-days.” This is a slippery slope to a deep misunderstanding of story points and velocity.

We recommending trying cross-team synchronization on points with a canonical set. But if these dangers are manifest, there is an alternative: *Each team uses its own idiosyncratic definition of points*. However, this leads to problems in estimating overall effort and tracking overall progress across all teams. One partial solution is...

Try...Combine progress measures

If different teams (or larger requirement areas) use different point systems, then each has its own Release Burndown chart, and there is no common product-level Release Burndown chart and no common velocity. How to estimate progress? Some alternatives:

- ❑ The simplest approach we have used is to hang the separate Release Burndown charts on a wall, stacked directly underneath each other, and ‘eyeball’ the overall scene.
- ❑ Estimate the percentage of progress, based on points, of each group. For example, if team-1 has 100 points in their Release Backlog and have finished 20, they are “20% complete.”¹⁶ Over-

15. These workshops need not be cast as a problem; positively, they foster common understanding and product-level perspective.

all product progress can be defined with different *combination methods*: worst case, mean, or median, depending on context.¹⁷

Try...Avoid...Estimate velocity before iteration-1

Real *velocity* is not estimated, it is measured—it is the total of all story point estimates of the work completed in the last iteration, a historical measure.¹⁸ Yet, sometimes a group wants to estimate it before iteration-1—before it can be measured—to help predict total release duration. For instance, the group may want to estimate release cost, and use duration as one factor. *Agile Estimating and Planning* [Cohn05] offers techniques for its estimation in the small-scale case.

For the multiteam case, one approach we have used is for all teams to hold a *pretend* (planned but not executed) Sprint Planning Part One and Part Two meeting, and assume that the items chosen for implementation were done—or that 75% (for example) were done. The total story points of all these items, across all teams, is the estimate of product-level velocity.

Avoid—If the next release of a product is going to be done anyway, rather than spending a week before iteration-1 *estimating* the velocity, just immediately *do iteration-1*, and measure *real* velocity.

Try...Adjust duration estimate with Monte Carlo simulation

In a tiny release with one team and four months estimated duration, plenty can go wrong to invalidate the estimate. This variability is magnified in the large scale. For some product groups we work with,

-
- 16. This is actually a fiction, because of variability and because it assumes the Release Backlog cannot be descoped.
 - 17. As an example of context and combination method: If team-1 does COBOL features in Beijing and team-2 does Java features in São Paolo and they refuse to talk to each other (and cannot, anyway)...perhaps *worst-case* is the combination method. This is an extreme example; in very big groups, *mean* suffices.
 - 18. Velocity is a kind of *budgeted cost of work performed* (BCWP) in earned value management [FK06]; as a name, ‘BCWP’ emphasizes that it is a historical measure, not an estimate.

that is not a problem—vague confidence in a fuzzy duration estimate is sufficient. In others—especially fixed-price, fixed-scope, fixed-duration outsourced work—it is vital to do the best one can to estimate duration.

Once when we were coaching in Bangalore, a lead developer’s father died, and he had to leave for several weeks. Once in Budapest, a lead developer’s girlfriend left him, and he was pretty useless on the team for a while—until he found a new girlfriend! These are examples of risks and sources of variability. And there are others: scope change, productivity, attrition, snowstorms, and more.

How do scientists and other estimators model stochastic (probabilistic) systems, such as development work? A standard solution is *Monte Carlo simulation* (MCS). With an MCS model for release duration, you can include the impact of many elements of variability or risk, including scope change, attrition—and girlfriends. MCS is especially useful in very large and long-duration development and in offshore, outsourced, fixed-price, fixed-duration projects.

And MCS fits well in the context of *agile estimating* because it is a simple, lightweight, fast technique to apply and adjust.

A classic book on product development risk management and MCS is *Waltzing with Bears* [DL03]; this explains the method of MCS for estimating a more realistic duration. The authors, DeMarco and Lister, also provide a free Excel spreadsheet called *Riskology* that implements an MCS model. Find *Riskology* on the web and try it, to improve the realism of your estimates.

CONCLUSION

Planning large-scale agile development is simpler than traditional approaches—or at least should be. We notice that this simplicity is disconcerting for some, because the traditional paradigm of management is that big complex work needs big complex planning and control by project managers. But there is a different way: the *emergence of order* from self-organizing Scrum feature teams. Top-down planning and control is not particularly effective in systems with variability and discovery—because the plans assume something

relatively static or deterministic—and the approach grows even less effective as these *non-linear* systems grow larger.¹⁹ Complex systems on the boundary of chaos and order (*chaordic* systems [Hock99])—and that includes big development groups—cannot be truly planned or controlled from above.

No false dichotomy regarding “no planning” versus “top-down planning”...In Scrum, the group *does* start by creating a Release Backlog for a future goal, identifying and estimating the product features. But after that starting point, agile planning emphasizes continual learning and adapting.

How to do that within a large chaordic system? By (1) encouraging self-organization and bottom-up emergence of order, (2) increasing transparency and feedback, and (3) making it easy to frequently inspect and adapt. This is precisely what agile approaches such as Scrum offer. Consequently—in contrast to those that assumed agile development was for *small* groups—agile planning is especially useful for the *large* scale.

RECOMMENDED READINGS

- For envisioning and vision workshops, two books already recommended in the *Product Management* chapter are relevant: *Innovation Games* and *Agile Product Management with Scrum*.
- For planning with small or large groups, *Agile Estimating and Planning* by Mike Cohn is an excellent, practical resource.
- *Waltzing with Bears* by DeMarco and Lister is informative and entertaining; it emphasizes iterative—rather than sequential—development as a key risk-management practice, and explains how to apply Monte Carlo simulation in estimation.

19. This was explored in *Queueing Theory* in the companion book.

This page intentionally left blank

Chapter

- Thinking About Coordination 189
- Coordination Techniques 200
- Centralized Coordination—Coordination Meetings 200
- Decentralized Coordination 206

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

COORDINATION

The best argument against democracy is a five-minute conversation with the average voter.
—Winston Churchill

A next-generation flexible platform was at the top of the list of strategic technology developments at a client. Two products were built on top of this prestigious new platform. One of the product groups was constantly complaining about the platform's *bloatedness*. The other group? They had *stabbed out* the entire platform so that it would not affect their work. Something had gone seriously wrong. Coordination is hard.

This chapter contains two major sections:

- ❑ *Thinking about coordination*—This section looks at the organization, product, and team perspectives. Who should be involved and why?
- ❑ *Coordination techniques*—This section dives into the how. What systems should be put in place to ensure coordination happens.

THINKING ABOUT COORDINATION

Most of this chapter focuses on coordination within the “development department.” However, as the organization transforms this department to a broader one by employing true cross-functional teams, the line between the functional departments (such as marketing and operations) becomes blurry. The increase in cross-functionality goes hand in hand with the expansion of the product’s Definition of Done. For most starting organizations, when your Definition of Done is still modest...

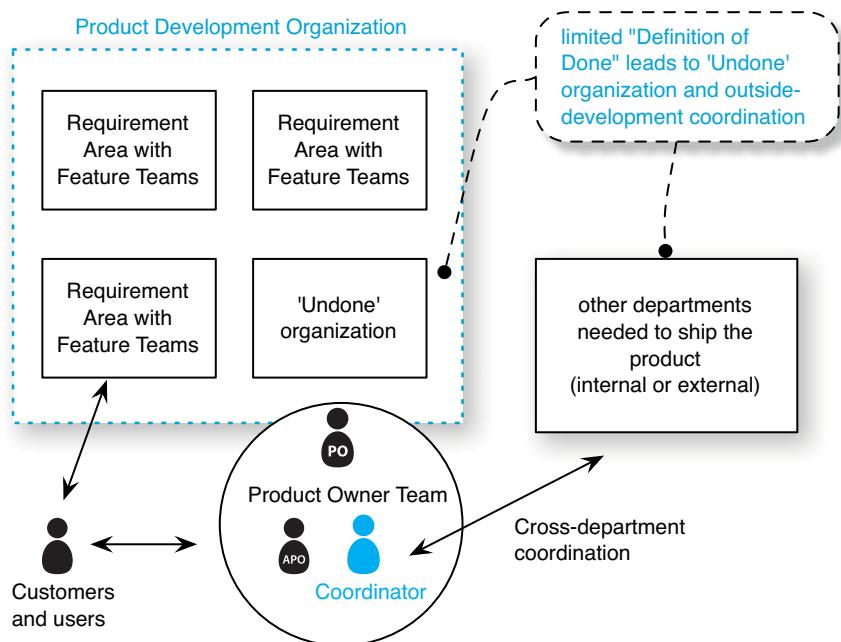
See “Try...Product-level Definition of Done” on p. 170.

Try...Avoid...Cross-department coordinator

See “Try...Product Owner Team” on p. 134.

Cross-department coordination and synchronization occurs through department interfaces based on agreements, commitments, and informal or formal ‘contracts.’ To maintain visibility and to shield the department (or work streams) from cross-departmental interrupts, one person is typically assigned to be the interface or coordinator—the single point of contact for issues *between* departments. This role is often called project manager, program manager, or stream manager, but we prefer to call it **coordinator**. This cross-department coordinator is part of the Product Owner Team and works with the Product Owners on the prioritization of the Product Backlog. He does not have final decision-making power nor is he involved in the development itself; his focus is purely on cross-department synchronization (see Figure 6.1).

Figure 6.1 role for coordinator between departments



A similar approach to cross-department coordination described earlier in the NextGen platform story demonstrated a major *drawback*: separation, that is, having separate departments focused on their own goals and their dependencies through contract/commitment. That led to local optimization because of the individual department's target-setting and organizational politics: *most requirements* came from its own platform teams rather than from its users. Result? A platform that was beautifully engineered, flexible, and useless.

see Feature Teams in companion book

In a variation of a cross-department coordinator, other departments send a representative as a member of the Product Owner Team.¹ This multiple-coordinator approach is also used in Toyota's product development; the managers of the functional departments meet with the Chief Engineer in the *obeya*² [LM06b].

see Systems Thinking in companion

An alternative way to deal with cross-department coordination is to *evaporate* the functional departments and increase cross-functional integration inside the teams by...

see Organization in companion

Try...Integrate all functions into the teams

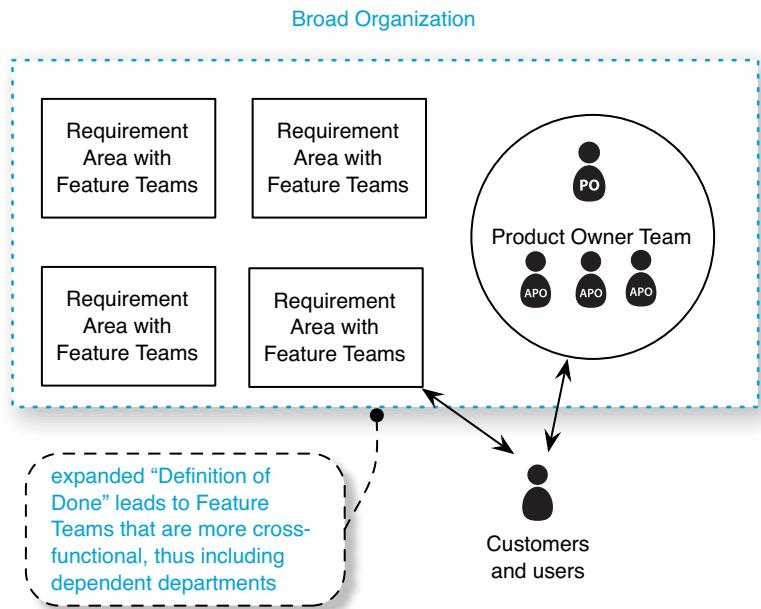
Organizations that improve and expand their Definition of Done integrate departmental dependencies by increasing the teams cross-functionality. In the ideal case, all functions are integrated within a broader product organization and all coordination happens within this department (see Figure 6.2). See also *Teams* in the companion.

See "Try...Product-level Definition of Done" on p. 170.

If the NextGen platform had taken this approach, then the people from the platform group would have been integrated into the two products to form true feature teams.³ The platform department would *evaporate*. To be clear, the platform *department* dissolves but the shared *platform* itself still exists.

-
1. Doing this creates a cross-functional project management group. The drawbacks of such an approach were covered in the *Teams* chapter in the companion book.
 2. Big Room where they sit and work together.
 3. Note that a separate platform department is just a detached component team, engendering all the weaknesses of that organizational design. See the *Feature Teams* chapter in the companion.

Figure 6.2 full cross-functional integration



see Feature Teams in companion

See “Try...Joint design work-shops for broader design issues” on p. 298.

See “Try...Plan infrastructure items by regular teams” on p. 168.)

People sometimes ask us, “Won’t the team size be too large when we integrate all these functions into the teams?” They will not. How come? Two reasons. First, not every function is present in every team, and requirements selection is based on team skill. Second, the specialization boundaries become fuzzier over time.

As an example of the second point, an organization we worked with decided to integrate the customer documentation into their feature teams. This required a sizable change because the documentation department was centralized, outsourced, and offshore and had to be decentralized, insourced, and localized. The new team member, who was previously a ‘writer,’ was 100 percent dedicated to one feature team. When someone asked her if there was enough documentation work to keep her busy, she answered, “No, but I am also learning and doing *testing* and writing of the *developer documentation*.” While she kept her *primary* specialization, she also started developing secondary specializations and helping out her team members by broadening her skill set.

In our work, we have seen organizations gradually move from the coordinator solution to more integrated feature teams. However, we have not yet seen any development organization of hundred-ish people reach the ‘ideal’ of integrating all functions into the teams. Most are in between the two extremes, with the goal of gradually including more and more functions in the teams by expanding their Definition of Done.

The rest of the chapter focuses on coordination within one department. Multisite specific coordination tips are covered in the *Multisite* chapter.

Try...Focus on the overall product

A team triumphantly finishes their Sprint and proudly demonstrates their work to the Product Owner who is thrilled until...he later notices it is not integrated and does not work with the work of the other teams.

In any practice, focus on the overall product and avoid locally optimizing individual teams. This prevents individual results being produced by individual teams and leads to one working whole product. Focus on the total product also impels the teams to coordinate their work. This tip is the general theme behind most other tips in this chapter...

see *System Thinking in companion*

Try...Coordinator, ambassador, and scout activities

MIT professor Deborah Ancona spent twenty years of her team research on “boundary spanning” activities—a team’s external activities. She identified three categories of outward activities:

- ambassador
- task coordinator
- scout

Clarified in her own words [AC92]:

Ambassadorial activities provide access to the power structure of the organization as members promote the team, secure

resources, and protect the team from excessive interference.

Task-coordinator activities provide access to the workflow structure; they are aimed at managing horizontal dependence. Through coordination, negotiation, and feedback, these activities allow for a tighter coupling with other organizational units, often filling many of the gaps left by formal integrating systems. **Scouting** activities provide access to the information structure; they are aimed at adding to the expertise of the group. These activities allow the group to update its information base, providing new ideas about technologies and markets.

Ambassador activities—relate to ScrumMaster responsibilities [Schwaber04, James07], though in a mature Scrum team these responsibilities are shared among team members.

see Lean Thinking in companion for more on yokoten; and the Organization chapter for more on CoP

Scout activities—create learning and keep the team up-to-date about their environment. Toyota’s yokoten—spreading knowledge laterally—and participation in Communities of Practice are examples of scouting activities. According to Ancona’s research, too much scouting leads to low-performing teams—they end up in analysis paralysis.

Task-coordinator activities—focus on coordination needed to get the product done, and are the focus of this chapter.

See environment mapping session on p. 211

Framing external activities with these categories is worthwhile for analysis of a team’s coordination needs. For example, someone might during an environment mapping session say, “Did we think about all the scout activities of the team for the next Sprints?”

Try...Team is responsible for coordination

see Teams in companion

A high-performing team pays as much attention to the outward team activities as to its internal team dynamics.

It is not enough anymore to work well around a conference table or in the laboratory; it is not enough to know how to build internal consensus and effective team decision making; it is not even enough to work well as a cross-functional team on concurrent engineering or horizontal processes. Unless you know how to manage within and across the new organization, you lose. The

only way to survive is to manage the external environment.
[AKSMW09]

Teams must be aware of their context and actively manage their boundaries by undertakings such as synchronizing work on shared code or clarifying cross-team requirements. Organizations need to make it crystal clear that the teams themselves are responsible for managing their boundaries and that they should take this into account when planning and executing their work. In *Succeeding with Agile*, author and Scrum expert Mike Cohn puts it like this:

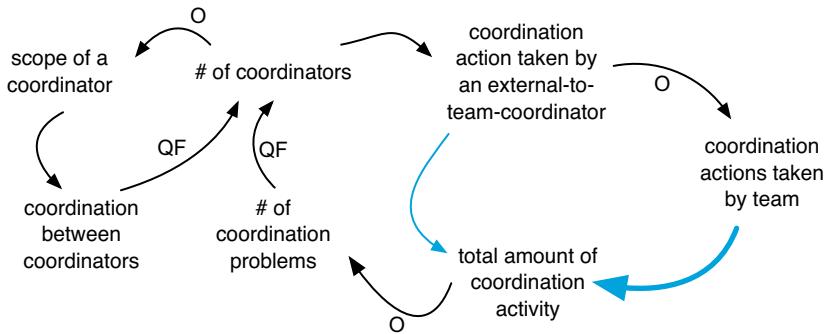
My favorite questions are the ones starting with, “Who is responsible for...” It doesn’t matter how someone ends that question, my answer is always the same: The team. [Cohn09]

Avoid...External-to-team coordinator

It never ceases to amuse us that many organizations behave in the same way. For example, organizations attempt to solve problems by creating new roles and assigning the problem to that role. If there is a ‘coordination’ problem, then they create a “coordination role” and assign it the responsibility for “doing the coordination.” The person in this role might be called ‘coordination’ manager, but is often called “program manager” or “project manager.” Some organizations were told that Scrum does not have a project manager role, but instead of changing their behavior, they merely renamed the role! We have encountered “feature manager,” “item owner,” “feature coordinator,” “project coordinator,” and similar variations.

Why avoid such a role? Is creating a role with clear responsibilities not a good thing? Sometimes. However, a problem is that making one person responsible for something often leads to other people (the Team) *not* being responsible for it (see Figure 6.3). So, a project manager doing the coordination leads to the team not doing it—resulting in handoff of knowledge and delay.

Figure 6.3 effect of team-external coordinator role



This effect was evident in a study done by Harvard Professor Amy Edmondson—an accomplished team researcher concentrating on psychological safety and team performance. In her study, she noticed, to her own surprise, that the external activity of the team *decreased* when the team leader took a more active role in coordination. She concluded

Thus, it is possible that frequent external relationship activity on the part of the team leader requires the team as a whole to engage less frequently in such activity, such that team leaders' extra-team activities lead to less boundary spanning by others on the team. [Edmonson99]

Avoid...Project managers

Avoid...“Fake Scrum” by renaming the project manager role

Traditionally, project managers are responsible for coordination along with other things. It is a coordinator role. Avoid team-external coordinator roles and avoid the project manager role completely.

When a role is removed, the conditions that originally required the role do not go away. Thus, removing the project manager role does not remove the need for coordination and project management. Who does this work? In Scrum, the project management work is distributed over the three Scrum roles. The Team is responsible for cross-team coordination.

When we work with organizations, we ask lots of questions to uncover potential problems—we look for “organizational smells.”⁴ One of these questions is “Do you have a project manager?” If the answer is yes, then one problem that often occurs is that the teams focus on “their parts” and do not take an overall product responsibility—including coordination.

see Teams in companion

Avoid...ScrumMaster coordinates

ScrumMasters are frequently enthusiastic people who sincerely want to help their teams. When the team has coordination worries, the ScrumMaster “removes the obstacle” for them and does the cross-team coordination. Time and again he gradually transforms into a project manager dubbed ScrumMaster. For more, see “Avoid...Fake ScrumMasters” in the *Organization* chapter of the companion. And: See “Avoid...Scrum of Scrums being a ScrumMaster meeting” on p. 203.

Try...Facilitation (rather than coordination) by ScrumMaster

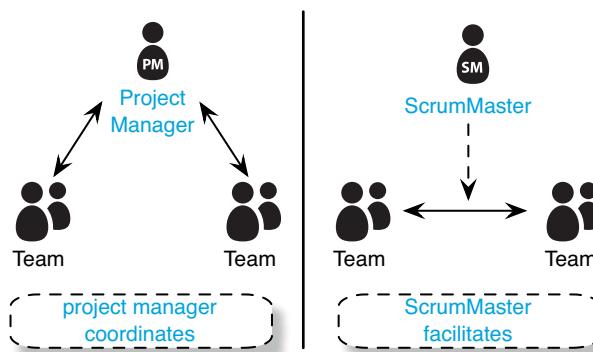


Figure 6.4 one difference between project manager and ScrumMaster role

A good ScrumMaster rarely *does* the coordination and instead *facilitates* coordination (see Figure 6.4). How? *He reminds* the team that their focus is the overall product. *He ensures* that the team members of different teams know each other and have practices and means in place to synchronize their work. *He teaches* and *facilitates* the team’s

See “Try...Scrum-Masters acting as and encouraging matchmakers” on p. 435.

4. The word ‘smell’ is sometimes used as a signal that something might be wrong—something smells.

problem-solving and root-cause analysis activities so that they recognize their coordination hurdles. And so forth.

see Organization in companion for more on measurement dysfunction

“You get what you measure” states an old management maxim.⁵ What is measured and especially what is rewarded influence people’s behavior. Measuring, setting targets, and rewarding based on individual teams promote individual team performance. Alternatively, overall product metrics such as business-case realization, lead time, value delivered, overall faults and integration problems promote a total product performance.⁶

That said, individual team targets *set by the team themselves* in order to improve their working practices are, of course, encouraged.

Try...Focus on overall product measures

We worked with a networking product group that considered competition between teams a good thing. It would surely lead to increased productivity. Not so. Their management awarded ‘points’ to the teams based on end-of-iteration criteria the management had devised. The teams could ‘buy’ prizes with these points. The result? Intense competition between the teams who paid more attention to the criteria than to the value to the customer. Teams argued with one another about the points awarded. The opposite of an attitude of working together toward one product.

Established team researcher, professor, and author Richard Hackman described this behavior:

“... conflict between groups, once underway, can rapidly escalate and spawn significant and unfortunate consequences for the warring groups, for their members, and for the organization as a whole. ... intergroup conflict is ridiculously easy to engender.

-
- 5. At least it will appear so. But whether it is what you meant...
 - 6. Although *overall* product measures are good, *rewards* based on those will still trigger measurement dysfunction and should be avoided [Austin96].

... [one condition is when] the groups are parties to a zero-sum game (i.e. any gains obtained by one group are directly at the expense of the other) ... These conditions are far from uncommon in organizational life. Indeed, they are sometimes deliberately created by managers in the hope, which I believe to be misguided, of fostering collective motivation and productivity by setting organizational groups into competition with one another.” [Hackman99]

Try...Myriad coordination methods

Cross-team coordination and inter-team communication is a hard problem for which there is no single or simple solution. In the classic paper on team-external activities, “Bridging the Boundary: External Activity and Performance in Organizational Teams,” the authors remind us that

New-product teams face a highly uncertain and complex task. There may be periods of creativity alternating with times when efficiency is the primary outcome of interest. Therefore, most of the interaction with other groups is not clearly programmed in standard operating procedures and routines but evolves to meet task demands. [AC92]

There is no “best way” of coordinating teams. Different teams, at different times and, in different contexts, require different solutions. Thus, instead of seeking the cross-team interaction technique that solves the problem, *experiment with multiple approaches and adapt.*

Why *change* approach? Jutta Eckstein, author of the first book on scaling agile development, concluded from her experiences that

There must be a law stating that as soon as a communication path works, it will be abused until it doesn’t work anymore...Therefore, you should also be agile and flexible with the communication. Use various modes of communication that address different persons differently...Change the communication channels from time to time. [Eckstein04]

COORDINATION TECHNIQUES

Coordination techniques can be classified into centralized and decentralized strategies.

Centralized techniques—often consist of meetings at which people from multiple teams assemble to coordinate their work—coordination meetings.

Decentralized strategies—let teams figure out their dependencies by themselves and expect them to resolve these in a decentralized, networked manner.

CENTRALIZED COORDINATION—COORDINATION MEETINGS

We have experimented with several different approaches to coordination meetings:

- Scrum of Scrums
- Open Space
- Town Hall
- Joint Scrum meetings
- Joint Sprint Review bazaar

Try...Scrum of Scrums

The Scrum of Scrums is the usual team coordination meeting when Scrum is used. It is a meeting at which representatives of the different teams⁷ gather to discuss and explore cross-team topics. Its form often resembles the Daily Scrum except that it does not need to be daily—two or three times a week is enough [Cohn07]. However, achieving an efficient Scrum of Scrums can be tricky.

7. A Scrum of Scrums commonly is held at Requirement Area level.



Figure 6.5 Scrum of Scrums

When we met up with a local agile coach working with a multimedia product developed in Budapest, he complained that their Scrum of Scrums was not working well. The team representatives felt the meeting was boring and useless. We applied Go See and joined their Scrum of Scrums. It was instantly clear why their meeting was ineffective. They used *exactly* the same structure as the Daily Scrum. This meeting form works great for teams with shared responsibility and a common work product. But in their Scrum of Scrums there was no shared task list and every team worked on a different part of the system. Reporting status as “my team worked on requirement X” resulted in a boring and useless meeting.⁸

see Lean Thinking in companion for Go See

How can the Scrum of Scrums become interesting and effective? One technique is to use slightly different questions than those asked at the Daily Scrum. Experiment with the following:

- What did your team do since the previous meeting that is relevant to some other team?
- What will your team do by the next meeting that is relevant to other teams?
- What obstacles does your team have that affect other teams or require help from them?

Try...Use different questions for the Scrum of Scrums

These questions encourage the representatives to share the most relevant topics and result in a more effective Scrum of Scrums.

8. When a team-level Daily Scrum is boring and useless, it similarly suggests the ‘team’ might be just a group of individuals doing “their own tasks” rather than a true team working on a common work product and sharing responsibility.

For example, “My team changed the interface of the messaging component to fulfill the new performance requirement. You guys probably want to know about that. For the last couple of weeks, my team has developed some performance improvements on a separate branch. Yes, we know we should not branch, but we could not find a better way to do this. Today and tomorrow, we will be merging this branch to the main line, so there will be a lot of changes. We have no obstacles other than the branching.”

Try...Two-part Scrum of Scrums

This tip is also applicable to normal Daily Scrums.

The Daily Scrum meeting form has a no-discussion rule. Its purpose is to avoid long, dreary, boring meetings and instead convey the needed information as efficiently as possible. But some topics require deliberation. How to balance between the efficiency and the need for discussion?

Try a two-part Scrum of Scrums. It consists of

1. the normal Daily Scrum-style: 15-minute-timeboxed, answer-the-three-questions part followed by
2. self-organizing followup meetings where team members (and perhaps observers) self-organize around the most pressing problem. This part is voluntary; team members who are not interested leave.

The topics for the followups often surface during the first part, when a team member might say, “I want to talk with you about that in the followups.”

Avoid...Scrum of Scrums being a status meeting to management

An organizational smell we often encounter is the “status-reporting Scrum of Scrums.” This is where the ScrumMasters meet each other and report their team’s progress to a program manager or a similar role.

A Scrum of Scrums—like the Daily Scrum—is a synchronization meeting and not a management status-reporting meeting.

Avoid...Scrum of Scrums being a ScrumMaster meeting

Another smell: Scrum of Scrums is a meeting of ScrumMasters at which the ScrumMasters take responsibility for the cross-team coordination. In every case in which we saw this, the ScrumMaster *mutated* into a project manager within two iterations.

To be clear, we are not saying that ScrumMasters should not meet with each other. In fact, we recommend they meet in a ScrumMaster Community of Practice where they discuss their experiences, share the obstacles they faced, and learn from each other. However, such a CoP is not intended for cross-team coordination, though obstacles to cross-team coordination could be discussed here.

*Try...CoP for
ScrumMasters*

Try...Rotate Scrum of Scrums representatives

Who should be the team's representative in the Scrum of Scrums? Of course, this decision should be made by the team members themselves. Sometimes, nobody wants to be the representative, or everybody does, or the selected representative has lost interest.

Avoid...Frequently rotating representatives

Rotating representatives works well unless you rotate the representative too frequently; then there is discontinuity and often chaos. Experiment with changing the representative every two or three iterations.

*see Organization
in companion*



Try...Open Space

Open Space Technology is a method for exposing “burning issues” and self-organizing around them [Owen97]. Exactly what is needed for team coordination! A team coordination Open Space is held perhaps weekly, and its form is the same as that of a normal Open Space. Members of all teams⁹ get together, create topics related to coordination, and organize around them.

Open Space may also *replace* the Scrum of Scrums. We were coaching at a client with about 30 teams on a product. They were dissatisfied with a SoS; we suggested trying mini-Open-Space meetings instead and facilitated the first meeting, which was attended by team representatives:

1. Five minutes to fill the *space-time* board with burning issues.
2. One hour total for sessions. The *spaces* were corners of the main meeting room plus some smaller rooms. The *times* were three 20-minute sessions. They usually had three parallel sessions, for a total of nine.

		Space	1	2	3
		10:00			
		10:20	(component) change impact	...	
		10:40	...		

X. I'll be hosting a session in space-1 at 10:00.”

In an SoS, through the “three questions,” someone might say, “Our team is about to release a significant change for X and I’m concerned about the impact on the teams.” In Open Space, the representative might say (and write on the space-time board), “I’ve got a burning issue about the release of

-
9. All teams could mean all teams in one requirement area (see *Requirement Areas* in companion book).

In fact, this occurred at a multisite Open Space meeting across two cities. See “Try...Multisite Open Space to replace Scrum of Scrums” on p. 430.

Try...Town Hall meeting

A Town Hall meeting is a public gathering to which everyone in the product development is invited—though participation is voluntary. It offers the chance for participants to speak up and raise any issue. To get the Town Hall meeting started, representatives from each team could start with a Scrum of Scrums-like meeting.

At the Scrum Gathering 2005, Bob Schatz, the VP of Engineering at Primavera at the time, shared a story: During their early-days Scrum adoption, a fake “Scrum of Scrums” private meeting was arranged by project managers. There was minimal self-organization, and top-down management remained a dominant mindset. Bob suggested replacing it with Town Hall meetings, which made an improvement in raising and dealing with teams’ issues, and with fostering more self-management [Schatz05].

Try...Joint Scrum meetings

Most Scrum meetings can be with multiple teams together—either the full team or team representatives. The *Planning* chapter describes how to scale Sprint Planning Part One using a joint meeting. The *Inspect & Adapt* chapter discusses joint meetings for Sprint Review and Retrospective.

An important advantage of joint Scrum meetings is that they foster coordination. For example, in a joint Sprint Planning Part One, team members of different teams learn about the work of the other teams and can agree on coordination practices. One particular joint Scrum meeting deserves special notice...

Try...Joint Sprint Review bazaar

See “Try...Joint Sprint Reviews” on p. 405.

In a joint Sprint Review bazaar, everybody assembles in a large room where each team has its own area to present its work to anyone who visits. Members of each team are encouraged to stroll around and discover what others did—promoting a focus on the whole product. During a bazaar, contacts are made that form the basis of future cross-team coordination.



DECENTRALIZED COORDINATION

Try...Prefer decentralization solutions over centralization ones

Decentralized coordination strategies are applied by teams independently. For these strategies there is no cross-team agreement or centralized place to meet; instead, each team decides its own coordination practices. Decentralized coordination is easy to scale up because it does not have a bottleneck, which is a key reason decentralized solutions are preferred over centralized ones. The drawback of decentralization is that no one person has an overview of the teams' decentralized coordination.

Try...Send chickens to Daily Scrums

See “Try...Environment mapping” on p. 211.

A simple decentralized coordination method is for each team to send a chicken (a team representative) to the Daily Scrum of other teams they are interested in. Either the same day or the following day, the chickens report back to their team so that their team can determine the needed coordination actions.

Before teams choose their chickens, they need to discover their dependencies—for example, at an environment mapping session. Afterwards, they desynchronize their Daily Scrum with the depen-



dent teams so that they do not happen at the same time. A team chooses its chicken and off he or she goes.

Try...Travelers

We worked with a product group that had a couple of experienced technical experts. This group created feature teams with dedicated members but could not decide in which team to add the (scarce) experts whose knowledge was useful to all teams. So the experts became *travelers*—each iteration they would join a different team for the entire iteration.

Swedish Scrum trainer Mikael Lundgen also recommends travelers but calls them “free agents.” Why? He noticed that in many of the companies where he worked, one or two persons were not comfortable working inside one team for a longer time. So instead he “set them free”—invited them to be travelers—hence “free agents” [Lundgren08].

The intention of travelers is to coach team members and share their knowledge with the team. A team can request a traveler to join them because his experience and expertise is needed. Although travelers are not specifically created for coordination, by joining different teams they create or strengthen a broad network, which is exactly what is needed for informal coordination channels. Component guardians might make good travelers.

see Feature Teams in companion, for component guardians

Try...Communities of Practice

A Community of Practice is a group of volunteers who share an interest or topic and have the passion to deepen their knowledge through discussion and interaction with peers. Examples of common CoPs are the ScrumMaster community, the architecture community, the TDD-coaches community, and the continuous-integration community. CoPs create cross-team interaction and are therefore well suited for solving coordination matters or for creating the channels through which informal coordination occurs.

see Organization in companion for CoP

See “Try...Internal open source with teachers—for tools too” on p. 315.

A telecom company we worked with has a proprietary programming language¹⁰ and needed to create their own xUnit framework. The developers of the framework also created an “internal open source” community for maintaining and extending the unit test framework. The community mailing list provided support for people new to the framework and quickly grew to be a popular tool.

Try...Communication CoP

Coordination and communication itself is an excellent topic for a CoP. Large-scale development expert Jutta Eckstein recommends a (virtual) communication team [Eckstein04]. This CoP discusses communication topics such as usage of common language and culture, and collaboration channels.

Try...Increase shared space

Alistair Cockburn, author of *Agile Software Development*, uses the term osmotic communication for people who *accidentally* “take in information without directly paying attention” as a result of being near someone [Cockburn01]. *Accidental* cross-team communication transpires from dependent teams being placed next to each other or from shared space such as fitness rooms or Ping-Pong tables [Weinberg71].

While we were coaching test-driven development to a team, two other feature teams shared the same office space with us. These teams both worked on the same component and they were *constantly* working together merely because of their co-location. It was hard to identify the team boundaries when watching them.

In China, companies often have after-work ‘clubs’ where people with a common interest or activity meet (similar to CoPs). While working there we encountered the badminton club, the automobile club, and even the baby club. Obviously, these clubs are not intended for coor-

10. In the telecom industry, most companies have a proprietary programming language. It usually stays within the company, with the Ericsson Language *ErLang* [Armstrong07] an exception.

dination between teams, but they do result in people knowing each other and thereby creating informal communication channels.

Multisite shared space

Creating shared physical space is impossible in a multisite environment. In that case, create a shared space as close to a physical shared space as possible. For example, at a Valtech project that I (Craig here) worked on, we installed always-on “room cam” web cameras in each team room (Bangalore and Paris), plus a monitor that showed the other (remote) room. There was a 3½ hour time zone difference, and thus people could often see each other live across the sites. This helped create a sense of shared space.

*see more on
shared space in
the Multisite
chapter*

A third-best alternative is to create *virtual shared space* using tools such as wikis, forums, iirc, instant messenger, and so forth.

Virtual worlds and “second life”-style technologies are gradually becoming mature enough for creating shared spaces [VH09]. For example, the realXtend open-source virtual world platform project uses Scrum in their development. They decided to “eat their own dog food” and hold their Daily Scrum meetings in their virtual world. However, in one news article they reported “everyone looked the same—our first woman avatar, which was later called a ‘crack-slut madonna’ by someone, she had just white underwear and no hair... voice was working through team speak software” and they abandoned using a virtual world for their Daily Scrum [Cybernews09]. Perhaps we still need to wait a couple of years.

Try...Break cubicles and other barriers

If we were to vote for the worst invention in the history of humankind, then cubicles would be high on our list. They and related communication barriers prevent osmotic communication. Perhaps the best thing about cubicles is that they tend to be easy to dismantle and then assume a less harmful form. Bas is well known for his hidden cubicle-disassembling skills.

Figure 6.6 barrier—a wall

A team working for a telecom operator had a problem. Their retrospective revealed communication problems with their Product Owner. The team was in a room and the Product Owner was in the room next to them... with a wall in between (Figure 6.6). For every question to the Product Owner, they had to 1) leave the room, 2) walk to the Product Owner's room, 3) enter, 4) ask the question, 5) wait for the answer, 6) leave the room, 7) walk back to the team's room, 8) enter, 9) continue their work. Not efficient.



Figure 6.7 solution—a door



How did they solve this problem? They made a door! (Figure 6.7). As a team, you do not want physical barriers such as walls to obstruct your communication. It is relatively easy to pick up the phone, call a contractor and ask him if he could please punch a hole in the wall.

The product group consisted of multiple teams, each in a different room. They realized the same communication problems were happening between the teams. So, what did they do? They knocked a hole in all walls (Figure 6.8). One team member remarked, “After punching the holes, you can walk directly from one room to another. Or shout to the next room. It's a bit too far to shout from the first room to the third, though.”



Figure 6.8 more doors

Try...Communicate in code

Feature teams touch the same code at the same time and thus need to coordinate their effort. For many developers, communicating in code and tests is highly effective. What might take hours to explain in English can sometimes be described in code in minutes. Code and tests are a clear and unambiguous communication medium. Applying continuous integration without branching results in people sharing each other’s work all the time, leading to plenty of cross-team communication through code and tests.

Try...Communicate in tests

Try...Environment mapping

With whom should the team coordinate? A simple technique called **environment mapping** helps the team explore their dependencies. They get together in a room with a flip chart and draw themselves in the middle of the flip chart and ask, “Who do we need to coordinate with?” Around them they draw their dependencies and connect them with lines on which they describe the nature of their dependency.

this is known as “project community” in Industrial XP [IXP04].

Figure 6.9 mapping environment



When to do this? The first mapping session can be useful during a new team kick-off meeting. But when working in long-lived feature teams, experiment with the exercise and see how often it needs to be repeated in order to be kept up-to-date.

Try...Coordination working agreements

See “Try...Joint Sprint Retrospectives” on p. 403.

With or without mapping their environment, the team needs to decide how to then do coordination. These decisions are part of the team’s working agreements (revisited during their Retrospectives). Working agreements can include things such as who joins the Scrum of Scrums, how frequently to change that, who will join other teams’ Daily Scrums, and so forth.

CONCLUSION

Coordination between departments and between teams is a challenging problem for which there exists no single solution. A thorough Definition of Done and cross-functional feature teams ensure that a lot of coordination is *within* the team or within the broad product department. However, cross-team coordination is still needed.

Decentralized coordination is preferred over centralized coordination because it scales better and well suits the self-organization principles of agile. That said, any serious-sized product group probably needs to try a combination of the experiments described in this chapter... and invent some more.

RECOMMENDED READING

Cross-team coordination seems to be an infrequently covered area in existing literature. Therefore, we do not have many recommendations; but these may help:

- “Bridging the Boundary: External Activity and Performance in Organizational Teams,” by Deborah Ancona and David Caldwell. One of the early research articles that explored teams within their context and how external activity—boundary spanning—relates to team performance.
- *Leading Teams*, by Richard Hackman. Still one of the best references on teams and self-managing teams. It also covers how teams manage their boundaries.
- *Succeeding with Agile*, by Mike Cohn. Scrum coach Mike Cohn covers some team coordination topics in his book.

Chapter

- **Organizing and Managing** 215
- **Team Organization** 234
- **Analyzing and Modeling** 236
- **Tools** 273

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

REQUIREMENTS & PBIs

My formula for success is rise early, work late, and strike oil.
—J. Paul Getty

“We cannot possibly fit our requirement into a two-week iteration, and there is no way it can be made smaller,” said a manager to us—actually, said *hundreds* of times to us by different people. It is the standard view in the world of big embedded systems. Fortunately, it is not true.

This chapter covers experiments related to requirements *analysis, modeling, or management*, most connected to scaling. More broadly, it considers all kinds of Product Backlog Items—PBIs or ‘items.’

It starts with suggestions related to organizing items and requirements information, follows with team structure, then takes a deeper dive into analyzing items, and ends with tool tips.

ORGANIZING AND MANAGING

Try...Group items into requirement areas

When there are *many* feature teams in a product group, organize related items (PBIs) in the Product Backlog into a **requirement area**, and group-related teams to work in that area. A requirement area is customer-centric; it is *not* an architectural subsystem. It is a set of items that are strongly related from the customer perspective. For example, *color workflow, security, or network management*. The subset of the backlog for one area is an **Area Backlog**—a view into the one Product Backlog whose items can be managed independently by an **Area Product Owner**.

See “Introduction to Requirement Areas” on p. 555.

Try...Group items into themes

Group items together into a *theme*, such as “PDF support.”

A theme is *not* one big item split into sub-items; rather, it is a varied collection related by a... *theme*. One item may be a member of many themes; for example, “automatic sleeping” may be a member of the themes *energy savings*, *heat reduction*, and *safety*.

Themes are useful to define release or market strategy; for instance, “Let’s focus on touch-screen items next release.” Sometimes, themes are useful for testing—they can inspire broad *integration tests* across a family of related features.

Themes are similar to *requirement areas* in that both are groupings. However, a requirement area (for instance, *color workflow*)...

- is motivated when there are *many* teams and the desire to organize a subset of teams into a family each with its own Area Product Owner
 - in contrast, themes are useful regardless of group size
- constrains a PBI to one and only one requirement area
 - in contrast, a PBI may be a member of zero or more themes, and a theme may span several requirement areas
- tends* to be more stable over time than a theme
 - in contrast, themes are sometimes transitory, such as the theme “*competitorX-matching features*”

Add the ability to ‘tag’ a PBI with a set of themes. For instance, if the Product Backlog is saved in a spreadsheet, add a ‘theme’ column. Each cell can contain a list of themes, such as “safety, energy.”

Avoid...Feature screening for PBIs

Historically, **feature screening** is a filtering activity to decide *early* to include or exclude a feature in a product release. The idea is coupled to traditional product- or requirements management in which decision points are defined, at which time large sets of decisions are

made (once) to let some features beyond the decision point and on to the next phase. Features pass through—or are removed from—a narrowing funnel. And once a feature is out, it's gone.

The *goals* of the technique are valid, including (1) early feedback to customers if their request will be in the next release, and (2) manageability—not being overwhelmed by a mountain of requirements. Yet, the goals can be achieved differently—with a prioritized backlog.

Feature screening is decide-early, coarse-grained decision making applied to large batches of decisions on a long queue, and to individual requests that arise sporadically. The paradigm is rooted in the assumption of big releases, “better get it right early,” and traditional slow development, rather than the agile alternative.

This old practice was often established long before a group adopts Scrum. Then, when a new feature idea comes along, rather than placing it on the Product Backlog with a low priority, a product management group inculcated in feature screening will revert to habit and make a big upfront decision: *Should this feature be promoted to the Product Backlog or not?* Avoid that.

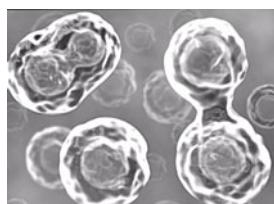
Lean development includes the principle to *decide as late as possible*. Reflecting this, Scrum offers more fine-grained, flexible decision making by using a prioritized Product Backlog. New ideas can be simply added to the bottom; if their priority does not rise, no immediate investment in analysis or decision making is required.

Try...Prune an overgrown backlog

Avoiding feature screening does not imply accumulating a gargantuan backlog of thousands of low-interest items. Some backlogs do accrete *junk*, and at some point the Product Owner Team needs to consider pruning it to reduce information scatter and management.

Try...Prefer cell-like splitting over treelike splitting

When a large item is split, it is the **ancestor** of new **sub-items**. How to think about the relation between ancestors and sub-items?



During mitosis when a cell splits, the ancestor cell ‘disappears.’

This is also one approach to item splitting.

An alternative model is tree-branch splitting; the ancestor branch *remains*.

When splitting large items, *prefer the cell-like model*. First an example, then motivation. Example:

Suppose the item ***decryption services for network traffic*** can be split into the sub-items:



- detect encrypted traffic*
- decrypt encrypted traffic*

And ***decrypt encrypted traffic*** can itself be split further:

- decrypt BitTorrent-encrypted traffic*
- decrypt Blowfish-encrypted traffic*

Cell-like splitting

With cell-like splitting, the ancestors disappear: The Product Owner¹ deletes the ancestors:

Item
detect encrypted traffic
decrypt BitTorrent-encrypted traffic
decrypt Blowfish-encrypted traffic

1. Here, and throughout the book, Product Owner may alternatively imply *Area Product Owner*.

This backlog does *not* record ancestors or ancestor-relationships.

Advantages of discarding the ancestor and all links to it include *simplicity*—a style that asks, “What is the *minimal* or *simplest* approach in the backlog that is useful?” rather than “What should we add because it may be *potentially useful?*”² A second advantage is an increased *sense of item independence*: One or the other sub-item may be discarded—some cells may die. And quite importantly: The new items are clearly prioritized *independently* from one another and without reference to their ancestor.

A sub-item *does not inherit the priority* of its ancestor.
This is a key mindset change for product management.

A *disadvantage* of the cell-like model is that relationships and ancestor information are lost; whether this is a *real* or *speculative* problem is context dependent. In any event, treelike splitting remains an option...

Treelike Splitting

Treelike suggests maintaining a record of direct ancestor requirements and links to them from sub-items. Why? Ancestor data may...

- provide big-picture context, aiding comprehension or decisions
 - alternate solution: *themes*
- be a source of inspiration for new sub-items
 - “For *decryption services*, what does the customer want?”
 - alternative solution: *themes*
- link *Area Backlog* items to the overall Product Backlog

2. This latter mindset—what *might possibly* be useful—is unfortunately all too common not only in backlog management but also in *product features* and *process definitions*, creating many wastes: overprocessing, inventory, and more.

- see “Try...Maintain three levels when using Area Backlogs” section on page 221

If treelike splitting is done, the Product Owner will maintain ancestor links (if any) in the Product Backlog.³ In this next example, the *direct* ancestor is maintained.

Item	(Direct) Ancestor
detect encrypted traffic	decryption services for network traffic
decrypt BitTorrent...	decrypt encrypted traffic
decrypt Blowfish...	decrypt encrypted traffic

Note one disadvantage: more information to manage—especially when splitting deeper and deeper.

The advantage of “strong item independence” found in pure cell-like splitting is also *theoretically* possible with this treelike organization, but we notice a subtle disadvantage: Product Owners are sometimes *influenced by explicit relationships to ancestors*, tending to incorrectly think of sub-items not as independent but rather...

- as a family that should be prioritized or implemented together
- that all sub-items must be done
 - “The item is only done when *all the sub-items* are done.”

That attitude is undesirable: it reduces flexibility.

Try...Maintain at most one ancestor—direct or indirect

To reiterate, before we can no longer see the forest for the trees: Prefer cell-like splitting, discarding ancestors completely. But when ancestors are important...

3. Additionally in either cell- or tree-splitting, a sub-item’s wiki page can link to an ancestor wiki page.

The prior treelike splitting example showed a link to *direct* ancestors in the backlog. This means more work maintaining data about *all* ancestors at all branching levels. If the tree of split items branches five or six levels deep, then that is a *lot* of ancestor information—and how would you manage all that?

Another option is to combine aspects of cell-like and treelike splitting: *Maintain at most one meaningful⁴ ancestor; it may be direct or indirect along the path back to the ultimate root item.* Less work, while still providing some context. For instance:

Item	(any meaningful) Ancestor
detect encrypted traffic	
decrypt BitTorrent...	decryption services for network traffic
decrypt Blowfish...	decryption services for network traffic

For the first item, the Product Owner decided to discard ancestors completely. The second two maintain the *root* ancestor—a common case but not required.

If *user stories* are the model of backlog items, an ancestor may be the ‘epic’ behind the user-story sub-item.

Try...Maintain three levels when using Area Backlogs

The previous example—with at most one meaningful ancestor—maintains *two* levels when items are split. When Area Backlogs are also involved—only suggested for very large groups—it is useful to maintain *three* levels, providing a link between the Product Backlog and Area Backlogs. For instance, the *Security* Area Backlog:

Item	Ancestor
detect encrypted traffic	decryption services for network traffic

4. Meaningful in terms of the advantages described in the “Treelike Splitting” section on page 219.

Item	Ancestor
decrypt BitTorrent...	decryption services for network traffic

Notice that the *ancestor* in the Area Backlog is an *item* in the overall Product Backlog.⁵

Item	Ancestor	Requirement Area
decryption services for network traffic	cryptography	security
encryption	cryptography	security

Avoid...Maintaining more than three levels of split items

Over the years we noticed that groups that maintain *many* nested levels of split items fall into the trap of not defining customer-centric requirements but defining *fake requirements*—technical activities or tasks. Keeping only three levels of split items seems to help avoid this. Also, see “Avoid...Technical task ‘requirements’ (PBIs)” section on page 237.

Try...Use special terms for size of items

Product management uses *special terms* for large items because doing so clarifies some context. For instance, “Let’s decide the market strategy for *major features* of the release.” Also, if one of the things people mean by *story* is that it is small enough to implement in an iteration, and by *epic* that it is not, these size-related terms are informative. And if three levels of items are in the Product Backlog and Area Backlogs, it is clarifying to have phrases for each level.

5. The Area Backlog is ideally only a *view* into the Product Backlog, not a separate document with duplicated information.

A Stories Story

First sponsored in 1993 by Kent Beck and Grady Booch, the *Hillside Group* (with Ken Auer, Jim Coplien, Ward Cunningham, Hal Hildebrand, and Ralph Johnson) met to explore *patterns* and their *generativity*. Ward invented the *wiki*—in part—to support ongoing discussion. At a subsequent Hillside Group workshop, Bruce Anderson raised the topic of *stories* (as in tales) and their power to connect with people.

The implications for development work evolved in Ward's *Episodes* patterns (notice that 'episodes' relates to 'stories'), especially in the *Implied Requirement* pattern; Ward wrote, "*I chose that name because the story only suggested the need to the degree that the developers and customers could talk about it.*" The implications also evolved in Kent's *stories*, articulated as part of his—influenced by Ward—agile development method, Extreme Programming (XP), whose first XP book cites *Episodes*. Kent wrote, "*I imagined a user grabbing another user in the hallway and saying, 'I gotta tell you about this wonderful new thing the application does...' Stories are the stories customers wish they could tell about the system but can't (yet).*" (continued...)

For these reasons, establish terms related to the size of items. There are no correct phrases in Scrum, it is only important that people have a common language. Terms we have seen used together:⁶ [huge item, big item, small item], [major feature set, feature set, feature], [epic,⁷ feature, user story⁸], [feature, epic, user story].

Try...Defer or ignore implementation and analysis of sub-items

One estimate in 2004 was that around 35% of all Internet traffic was for BitTorrent. True or false, it is substantial, and so some telecommunication operators are interested in throttling network bandwidth for it; some are also interested in identifying if copyright violations are involved. Therefore, one Product Owner we worked with had the following item in his backlog:

detect and decrypt BitTorrent traffic

6. Abstractly, these are names in a *requirements meta-model*.

7. *Epic* is only used with *user stories*.

8. Implies a specific approach to analysis.

(continued...) On the original XP stories, Kent wrote, “*I don’t recall specifying format. The salient properties I asked for were (1) testable with automation, (2) small enough so that several should fit in an iteration, (3) quick and easy to write, (4) estimatable with some confidence.*”

Interestingly, the phrase **user story** was not coined by Kent; he wrote, “*I never say ‘user stories’, because I think they belong to the whole team.*” However, “user story” was used in the first XP project—the Chrysler C3 project—though no one can remember precisely who coined it. Chet Hendrickson (on C3) speculated, “*It is possible we started to call them ‘user’ as a nod to ‘Use Cases’ and to differentiate them from ‘technical’ stories. We had not yet discovered the evil that is technical stories.*”

We suggested to split this as follows:

- detect BitTorrent traffic*
- decrypt encrypted BitTorrent traffic*



The teams estimated relative **effort**, and the Product Owner Team estimated relative **value** (on a scale of 1 to 7, 7 being most valuable). Here were the results:

- detect... effort=3, value=6*
- decrypt... effort=21, value=2*

When the Product Owner saw this, *his* lights went on. He realized that by splitting big requirements, he could identify valuable and independent sub-items that could and should be delivered early and independently, and other parts that could be deferred. To reiterate:

A sub-item does not inherit the priority of its ancestor.

Once, we were coaching in Poland with a group *starting* Scrum adoption. The Product Owner Team—separate from the development

teams—had split many big items into many smaller items and analyzed (and documented) them all in detail. Waterfall ‘Scrum.’

Returning to the original story...In contrast to the group in Poland, the BitTorrent-story Product Owner realized that not only could the ‘decrypt’ item be deferred for implementation, *it could be deferred for requirements analysis*. The wastes of overprocessing and more WIP inventory of detailed requirements could be avoided by lowering the sub-item’s priority.

This is an important shift in product-management mindset and behavior; traditionally, product management would have pushed a large batch of coarse-grained requirements onto the R&D work queue for implementation, and not be involved in finer-grained investment, splitting, or decision making.

Avoid...Defect items in the Product Backlog—unless few

A standard suggestion in Scrum is to record customer-reported defects as PBIs. Excellent advice when there are only 10 defects; but when there are 7214 defects, forget it. The latter case is unfortunately typical for long-lived embedded-software systems. Then, use a separate defect-tracking system.

See “Try...Avoid... Fixing defects” on p. 169.

On the other hand, *out of sight, out of mind*; these defects and their estimated effort should be visible to the Product Owner so that he understands the defect situation and responds. Some of our clients make coarse-grained *placeholder PBIs* at the top of the backlog, such as “1000+ category-A defects,” “3000+ category-B defects” and so forth, each with an overall estimate.

Try...Add a single placeholder PBI for all defects—when many

From a lean thinking perspective, *Stop and Fix*, the product should never have gotten to 7214 defects...but...it happens. Large-product groups commonly move to Scrum many years after formation.

Try...“Undone Work” and system-level NFRs as PBIs

Assume that a system should start up within five seconds. The work and proof involves at least a team writing and deploying acceptance tests. And possibly they are also doing performance-tuning work to

See “Avoid...Undone Work” on p. 173.

achieve the goal, if not already met. Record this, and all system-level non-functional requirements (NFRs) as PBIs.⁹

And there may be *Undone Work*¹⁰ such as educating the sales people or license reviews. Record Undone Work as PBIs.

Avoid...Try...Separate “Undone Work” from the Product Backlog

It might not yet be feasible to include the teams in the Undone Work... or it might not be feasible to include them in *all* of it. Where do you put the Undone Work then?

See
“Try...Include Scrum teams in a Release Sprint” on p. 175.

Avoid separating it, and keep it on the Product Backlog (as suggested in the prior experiment)... because that increases the visibility towards the Product Owner, and otherwise there is the problem of “out of sight, out of mind.” He can keep track of all the Undone Work and that helps him to make release decisions. This also promotes teams getting involved with the clearly-visible Undone Work, and learning from the experience. Finally, definitely avoid separating it if there is no Undone Unit and the regular teams eventually do this work, such as in a Release Sprint.

Try separating it from the Product Backlog (when there is an Undone Unit)... because an Undone Unit does not work off the backlog or apply Scrum—there is no such thing as an Undone Unit Scrum; for example, a so-called *test team Scrum* is a contradiction in terms. And by separating the Undone Work it is independently managed by the Undone Unit (as a traditional project or however they work) and they provide the Product Owner with a duration estimate. In that way, the Product Owner can still decide on release dates. Keeping the Undone Work separate can help during the *early* days of agile adoption as it does not require immediate change from the existing Undone Unit. Finally, putting release-2 Undone Work in

-
9. Some use the term NFR to mean *all* requirements and tasks other than functional goals; for instance, including educating sales staff. We limit the term to *qualities of the system*, such as stability.
 10. In Scrum, *Undone Work* is the difference between Sprint Definition of Done and Release Done. The perfection challenge in Scrum: no difference and therefore no Undone Work.

the Product Backlog can be confusing when the teams have already moved on to work for release-3.

Of course, neither way of handling the Undone Work solves the problems that an Undone Unit creates. *Risk* and *delay* are hidden in the Undone Work and it will (eventually) generate feedback that will interrupt the teams—and that will impact focus and productivity, and increase friction between groups.

In the end, the only way to ‘manage’ the Undone Work is to not have any—by expanding the Definition of Done.

Try...*Genuine research work as PBIs*

Some *extraordinary* things need *unusual* research before implementation. One of our clients built a financial trading system that involved choosing third-party software, integrating it, and also developing software in-house. The work to identify some candidate software should be identified as a *research item* PBI.

See “Try...Plan bounded research or learning items” on p. 166.

Moving up the innovation scale: One of our clients develops ship-control systems. How to apply artificial intelligence to increase their autonomy? Perhaps a research PBI is needed. Another example: Someday, version 2.0 of the PDF standard will be released; printer engines need to interpret it. Teams will need to digest at least some non-trivial information before they can start some implementation. Another research PBI.

Moving from *recording* research items to *doing* them: *Doing* research should lead to *recording* new customer-centric items.

Focus research work on useful output for the Product Owner, especially identifying new PBIs. Limit study; quickly prototype and implement.

For example, rather than studying in depth a 500-page document of the next PDF specification,¹¹ people can

1. identify large sections that can be studied in detail later
2. study only a smaller subset in detail
3. create new customer-centric PBIs; prototype; ...
4. start implementation

Try...Research items quickly lead to customer-centric PBIs

This suggestion expands a point made in the prior one. When a research item is worked on, it should yield useful results for the Product Owner, especially the identification of new customer-centric PBIs for the backlog. It is not necessary to read all 500 pages of the new PDF specification before identifying some useful new features.

Avoid...Fake research items: regular analysis, ...

Avoid...Giving research items to separate 'research' groups

We coached a group in the USA that included a team of interaction designers. They resisted becoming bona fide full-time members of regular Scrum teams; they wanted to do “*their* interaction design work” separately and hand it off to the Scrum teams when finished. When they heard about the idea of research items in Scrum, they said, “Great! We will do the interaction design *research work*, and then deliver it to the teams.” That is not research.

More not-research is *deciding on and documenting a design solution*.

All this is *fake research*—merely normal exploration and discovery. Avoid that. A research item is for truly extraordinary discovery or study *far outside* the familiar.

Finally, avoid giving research items to separate ‘research’ teams; *research is done by normal Scrum teams, not separate groups*.

11. Printer-product companies write software to interpret PDF.

Toyota Big Room and Visual Management of a Release



In Toyota, in the Big Room (*obeya*) where the chief entrepreneurial engineer sits and meets with others, they use a variety of visual tools to show new-vehicle release progress. The walls are covered with big visible charts, color tokens to signal status, and more.

Try...Visual management for the Product or Release Backlog



Visualize the Product or Release Backlog, probably on a wall with tangible cards. (See also the Toyota Big Room topic-box.)

The photo does not show it well, but this Release Backlog used color papers to signify items that were added after *initial* backlog refinement. This was helpful so that the Product Owner could see the changes.

Try...Traceability with executable requirements as tests

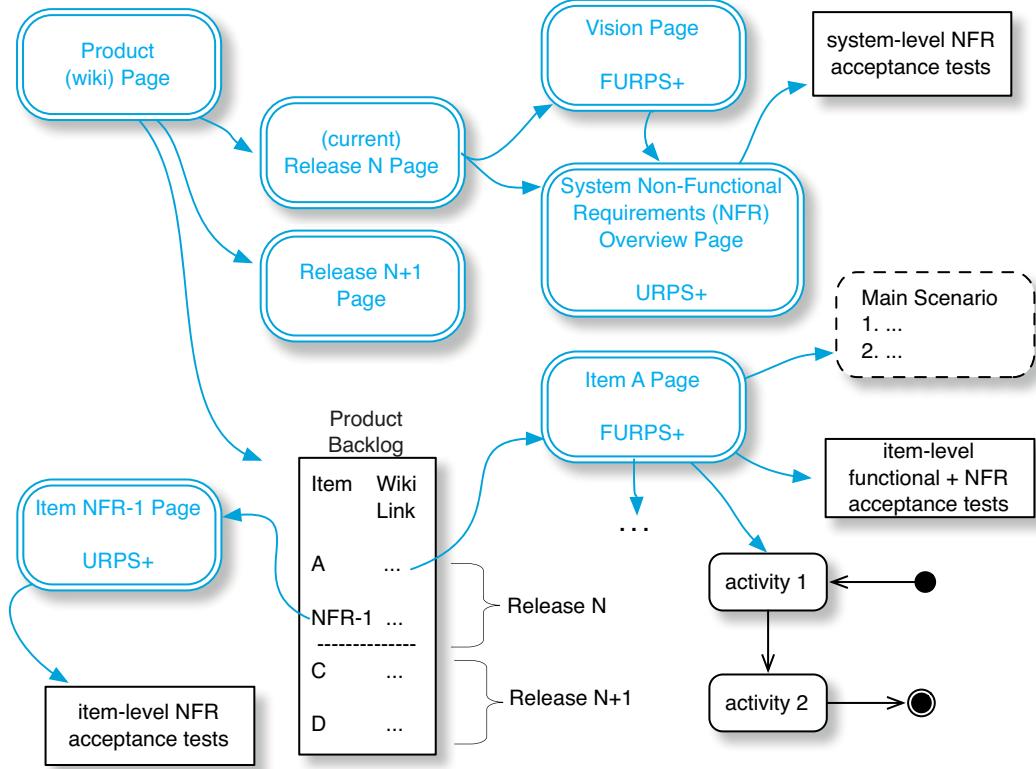
When traceability of requirements is truly needed, *cut the Gordian knot* on this problem by originally writing the requirements as executable tests rather than in a natural language (such as English).

See "Try...Acceptance test-driven development" on p. 42.

Try...Organize requirement artifacts to include...

Figure 7.1 illustrates sample requirement artifacts and relationships. This figure assumes wiki pages.

Figure 7.1 artifacts



Product page—a product-portal page that, among other things, points to release pages.

Release page—a portal to information of one release, including

- ❑ *Vision page*—a summary of the release business case, probable high-level FURPS+ analysis (see topic-box), and more.
- ❑ *System-level Non-Functional Requirements¹² (NFR) Overview page*—a summary of system-level *NFRs* and *constraints* to help

12. For good reason, some dislike the terms “functional” and “non-functional” requirements [BCK98], but they are very widely used. We are using these terms for familiarity, rather than our agreement.

Try...FURPS+

Whatever requirement approach is taken, from user stories to acceptance tests to use cases, it is easy to miss stuff—often ‘non-functional’ stuff—when hundreds of people are involved. Want a technique to miss less? Try remembering and teaching other people to remember FURPS+ [Grady92, Larman04a], a simple, easily remembered mnemonic with the following meaning:

- **Functional**—features, capabilities, security.
- **Usability**—human factors, help, documentation.
- **Reliability**—frequency of failure, recoverability, availability, predictability.
- **Performance**—response, throughput, accuracy, resource usage.
- **Supportability**—adaptability, maintainability, internationalization, configurability.

The “+” in FURPS+ indicates ancillary and subfactors, such as:

- Implementation—resource limitations, languages, tools, hardware ...
- Interface—constraints imposed by interfacing with external systems.
- Operations—system management in its operational setting.
- Packaging—for example, a physical box.
- Legal—licensing and so forth.

Use FURPS+ as a mental checklist. Apply it when analyzing the system or any item—when writing acceptance tests, writing a use-case text, drawing a diagram.

people grasp the big picture. Some details may be expressed as acceptance tests. This material surveys probable “URPS+” qualities and constraints of the system.

Agile implies *responding to change over following a plan*. With re-prioritization of items each iteration, there is no guarantee of following an early *Vision* or *System-level NFR Overview*.

A *Vision* and *System-level NFR Overview* are speculative overviews. As high-level tools for envisioning or learning they are useful, but

they are *not* the list of requirements. They serve as *inspiration* for the Product Backlog items and *item pages*.

In Scrum, the Product Backlog lists the real product requirements, not other documents.

Item page—a portal to the details for a specific item (functional or NFR) in the Product Backlog. It may point onwards to anything deemed useful for elaboration: use-case text wiki pages, photos of activity diagrams, acceptance-test pages, ancestor items.

Avoid...‘Solving’ requirement problems with a documented meta-model

A *requirements meta-model* describes types of requirements and their relationships. For instance: a *feature is described by use cases*, and so on. This topic seldom comes up in small groups where there is little time or money for waste. In big groups with process engineers or document-writing quality managers, it is different...

We were invited to a meeting on how a group could better “manage the requirements.” It devolved into discussions by the managers on their existing requirements meta-model and proposals for a new one. No one in the room was involved in the hands-on development work—including us. But we had spent time applying Go See, and knew there were deep problems at the team level where the real work was being done—isolated specialists, handoff, ‘requirements’ that were technical tasks. We suggested that they could not solve their problems with a new model...to no avail. Months later, they had a new documented meta-model. But nothing had changed.

This behavior is part of a pattern in organizational waste. It starts with these steps:

1. A writer writes a document for policy-X.
2. People do not read the document.
3. People do not follow the policy.

Upon observing the situation, what does the policy writer do? Rewrite or add more documents for policy-X! There is no Go See behavior; rather than going to the real place of work and inquiring with Five Whys, there is the assumption that documents are the problem and documents are the solution.

Avoid...A complex requirements meta-model

A requirements meta-model has its uses. For instance, when someone asks, “What are the *epics*?” or “What are the *user stories* in this *epic*?” people need common understanding of these words and their relationship. This is especially helpful in a group adopting new ideas and terminology, such as agile development. However, the model should be so simple that it does not need a document—people should be able to grasp it quickly.

Avoid...Describing a simple meta-model in a complex way

We have been involved in relatively complex (several thousand people, hardware and software) products and never did the group need a UML diagram or document for their requirements meta-model.

If diagrams or documents are created for what should be a simple meta-model—these days for an “agile requirements meta-model”—there may be a deeper problem. What might that be?

- The model is too complex.
- It is simple, but people do not take the time to learn even simple ideas, so the ‘solution’ is to document the meta-model.
- It is simple, but there are people who fill time creating unnecessary documentation, either because they believe it solves real development problems, or they only do overhead work.
 - this is often process engineers or quality managers

Michikazu Tanaka, a student of Taiichi Ohno, said of his teacher:

[One characteristic of Ohno] was his dislike of written material. He wanted us not to waste time producing useless documentation. He insisted that we could convey information better by

showing people the workplace than by turning out documents.
 [SF09]

The tenth agile principle: *Simplicity—the art of maximizing the amount of work not done—is essential.* This suggestion does not mean to avoid a clear meta-model; it is to avoid waste and focus on simplicity and the *real work*.

Especially in large groups, there are myriad examples of waste when more process-related documentation is introduced. For instance, at one of our clients adopting Scrum, an “agile requirements meta-model” was formally introduced with a diagram to describe...a *feature* is composed of *epics*, and an *epic* is composed of *stories*. Some months later we were coaching at one of their sites in India and we discovered the management had now introduced two backlogs: There was a *Feature Backlog* and a *Product Backlog*! (We’re still waiting for the *Epic Backlog*.) When we asked why, they explained it was because of their new ‘agile’ meta-model. Now there was more waste of information scatter and more complexity, but even worse, *behaviors, roles, and responsibilities had changed*: In this relatively small group, one person was responsible for the Feature Backlog, and another for the Product Backlog...they were going back toward old traditional habits, just dressed up with new agile terminology.

Their *real* problem relates to lack of learning and careful thinking.

TEAM ORGANIZATION

Cross-functional teams were explored in the companion book. The following is reiteration—on purpose, because we noticed the following change needs to be stated explicitly...

Avoid...Separate analysis or specialist groups

Avoid...Separate systems-engineering group

Big traditional development has separate analysis groups—the *business analyst* team, the *interaction design* team, the *architecture* team, the *systems engineering* team, the *requirements engineering* team, and more. When the organization moves to Scrum, a typical response among these groups is “We understand that cross-func-

tional teams are for *them* and *them*, but of course *not for us!*” Yes they are.

Or, we are asked, “How do we do Scrum in the [analysis/architecture/systems-engineering/interaction design, …] team?” You don’t.

The proper question flips: Rather than, “How do we do Scrum in the analysis team?” it becomes, “How do we do analysis in Scrum?”

All these separate groups dissolve and members disperse into real cross-functional Scrum teams doing hands-on development work.

Avoid ‘fake’ team members—We visit customers where someone—usually an analyst, interaction designer, architect, or documentation person—says, “Oh yes, I’m a member of the Scrum team!” What this really means is that they *visit seven* different teams, collect a queue of work requests from them, do *their special work*, and then hand off the results back to the waiting teams. Stay clear of that.

Another version of fake team member is a person allegedly on the team but separately analyzing for the next iteration…“Sanjit is the analyst on our Scrum team. He’s busy figuring out and writing the specs for the next Sprint.”

In the case of separate analysis groups or ‘fake’ team members, there is handoff from that separate group or person to the Scrum team, iteration by iteration (Figure 7.2). Not a good idea.

Avoid...Separate interaction design group

see Teams and Feature Teams in companion

Avoid...Separate architecture group

Avoid...Fake team members

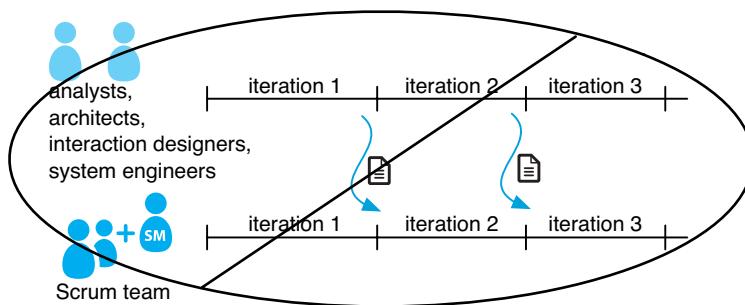


Figure 7.2 avoid separate analysis groups and fake team members that hand off requirements

Avoid...Product Owner Team as separate analysis group

See “Avoid...Too inward product management & Product Owners” on p. 124.

The Product Owner or Product Owner Team is focused outward, on customer analysis, marketing, branding, and so on—solid product management. If the majority of their time is spent on inward activities such writing specifications and then handing them off to the Scrum teams, there is the waste of handoff and mini-waterfalls, and the Product Owner is not sufficiently outward-oriented.

People with skills in business or requirements analysis are meant to be part of the normal cross-functional teams in Scrum, not part of the Product Owner Team. The regular teams (that contain analysts, UI designers, architects, and system engineers) are deeply involved in this work—usually during ongoing Product Backlog refinement—in collaboration with guidance from an outward-focused product-management “Product Owner Team.”

ANALYZING AND MODELING

Try...Write customer-centric requirements (PBIs)

Items on the Product Backlog should be requirements in the domain of the customer or user—addressing their needs, in their language, and of value to them.¹³

Test of a good Product Backlog?
Your customers immediately understand every item.

This reflects the first agile principle, *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*—lean thinking, and avoiding sub-optimizing on secondary goals such as the work or convenience of particular teams. This suggestion seems obvious—even trivial—but a recurrent problem we see in large, multisite, or offshore development is that it is not followed. In big groups, people become increasingly isolated from real

13. One exception in Scrum is major internal *improvement goals*.

customers and absorbed in their technologies because (1) sites are added or moved, (2) people are organized into component teams that only see work focused on their component, and (3) it is hard to grasp an overview of a giant system.

Avoid...Technical task ‘requirements’ (PBIs)

During coaching we see “technical user stories” such as, *As a systems engineer, I want an interface for X...* This “user story” is a contradiction in terms, and in reality is a technical task. The point of *user stories* is that they are for *users*. This contradiction is a sign of phony ‘requirements’ that do not belong in the Product Backlog.

We also see tasks in the Product Backlog when groups are organized into component teams rather than feature teams. No one team does complete ‘vertical’ customer-value work; the goal is split into technical tasks across teams. For example, if there is a UI and DB team, a ‘fake’ requirement (a task) will appear in some guise such as

As the UI Team, we want the database to have a RECIPIENT table with a SECOND-ADDRESS... The UI Team is not a real user, and this has been written to coordinate tasks with the DB team.

As the DB Team, we want the database...

As a Shipper, I want the database...

Another variation is a technical task added simply to fulfill a customer requirement. For instance, at a client in Singapore, we saw the following prioritized Product Backlog:

1. change ProductB APIs so that ProductA can integrate with it
2. ProductA works with ProductB

Item-1 is a technical task, not a customer-centric requirement or goal. Item-2 does properly belongs in the Product Backlog, but item-1 should be removed. Later, as a *task* it might appear in a *Sprint Backlog* when a team works on the customer-centric goal “ProductA works with ProductB.”

see the discussion of component teams in the Feature Teams chapter of the companion book

Avoid...Technical task PBIs in team-level “Product Backlogs”

Another situation we see: inappropriate technical tasks as PBIs...

Usually in Scrum there is one Product Backlog for the product—regardless of the number of teams—so that the focus is on overall product goals and avoidance of sub-optimization at the team level. Large groups beginning Scrum adoption sometimes ignore this and create a “product backlog” for each existing traditional (often single-function or component) group because “then you don’t have to change much” when adopting Scrum. Said another way, rather than an organizational redesign toward a flow of value to the customer implied by adopting real Scrum teams (that do end-to-end customer-centric requirements), some groups adopt Scrum terms while keeping the old structure—with the old limited value throughput.

Then, the existing single-function and component teams remain, each doing only part of the overall customer feature. Each has its own “product backlog.” Teams (usually via managers) coordinate work by placing *technical tasks* on the so-called product backlogs of other teams.

The result is a continuation of traditional development, with essentially the same levels of handoff, WIP, delay, and coordination and integration problems, all merely overlaid with ‘agile’ terminology.

Try...Ask, “Would users understand every PBI?”

As a thought experiment to determine if inappropriate technical tasks have been weeded out of the Product Backlog, ask, “If we show the Product Backlog to real users or customers, will they understand and relate to every item?”

Try...Prefer goal-oriented over solution-oriented requirements

A large package-shipping service in Europe is adopting Scrum. They had complaints about usability at their website, and considered writing a requirement as follows:

As a Shipper, I want all the shipping option details on one webpage so that I can find information faster.

This is a *solution-oriented* requirement that presumes a solution to the problem. Stay away from those. Rather, prefer goal-oriented requirements, such as this:

*As a Shipper, I want **to find critical shipping information fast** so that I have more free time.*

In this form, the solution is *unconstrained* (or at least, low-constrained) and the team can do whatever to satisfy the goal. This is a desirable, *powerful* way of working with teams because it increases *creative freedom*. Who knows?—the final team solution may be better or cheaper than one pre-specified, and motivation increases.

see Teams chapter in the companion, for motivation discussion

Sometimes *pure* goal-oriented requirements are not possible; there can be *constraints* on requirements or solutions. For example, a customer does not actually want an *oil-well testing tool* or a *printer* or a *webpage*. They want *subsurface reservoir extraction design* (actually, they want...*money*) or a *paper printout* or *shipping information*—and if *magic* could solve their problem, they would be *happy*. But these days, documents are in *PDF* format and printed via printers—and there's a short supply of *magic*. *Solutions* such as *PDF* (a standard created by engineers) and *printers* have entered the customer domain, and then the customer says, “My *requirement* is a printer that prints *PDF* documents.” This is a requirement in the context of certain constraints—that there are *standards* and *hardware* for printing. In fact, even *paper printout* is usually a constrained requirement or solution to a deeper goal, such as “communicate financial results to shareholders” (perhaps a deeper constraint is that shareholders want a familiar old-style paper report). In reality, ‘requirements’ exist along a continuum from “more goal-oriented and unconstrained” to “more solution-oriented and constrained.”

As far as possible, aim to write *minimally-constrained goal-oriented requirement statements*. Because...

“If you tell people where to go, but not how to get there, you will be amazed at the results.” — General George S. Patton

A precondition for successfully using goal-oriented items is that *the goal and conditions of satisfaction are crystal clear*.¹⁴ Words such as *critical* and *fast* require quantification and clear acceptance tests for this to work.

Try...Requirements workshops

Workshops reduce the waste of handoff and increase collaboration and feedback within and between teams in multiteam development. Experiment with holding *requirements workshops* with the teams and Product Owner (or representatives).

I (Craig here) have been involved in facilitating these since the early 1980s, because an early version called the Joint Application Development (JAD) workshop [WS95] was created and popularized in Canada,¹⁵ where I used to work. On reflection over these years, the essential elements of a good workshop boil down to

- a formally designated facilitator who knows how to guide, and knows tools for creativity, modeling, information organization, and group decision making
- people with varied *perspectives and skills*, people who are *subject matter experts*, people empowered to make tough *decisions*

Requirement workshops are useful in these cases:

See “*Try...Initial Product Backlog refinement workshop*” on p. 158.

- for a *vision workshop* to evolve and communicate the high-level big picture for overall product release
- for a *detailed requirements workshop* after a vision workshop
- for *Product Backlog refinement* each iteration

-
14. Supporting techniques for *clear measurable goals* include acceptance TDD and *goal statements* in Planguage [Gilb05, Larman03].
 15. Requirement workshops extend back to at least 1977 with Chuck Morris at IBM Milwaukee, inspired by the book *How to Make Meetings Work* [SD76]. In 1979, Tony Crawford at IBM Toronto worked with Morris to formalize and popularize this as JAD workshops.

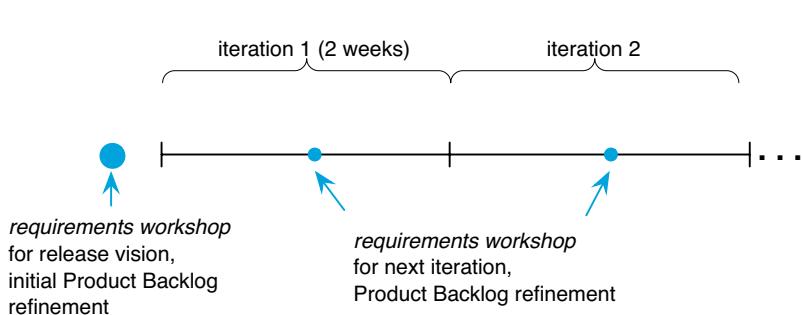


Figure 7.3 timing of requirement workshops

Durations—When Scrum is applied to small products, a *vision workshop* or *detailed requirements workshop* during initial Product Backlog refinement may take less than a day. When a complex embedded system, such as a new medical device or printer, is the product, these workshops will take longer; we suggest “two days to two weeks, with a preference to the shorter.”

Overview—Avoid looking at computer projections, PowerPoint presentations, and so forth. Do not sit around tables passively listening to a speaker. One workshop saying is “If you’re not at the wall or on the floor, there’s something wrong.”

Encourage *activities* rather than large-group discussion or presentation, and organize these at whiteboards, flip charts, and the like.

Avoid...Using computers in workshops

People sitting at tables is often a sign of a *dead workshop*, but table work has its time and place, especially when the focus is on *actions* rather listening or watching, as suggested in Figure 7.4.

Techniques—This section shares techniques for workshops. It is a *sample* of a subject that facilitators need to study in depth. Scrum-Masters should learn these skills and coach others in them.

- **Visioning**—Try building a *Product Box*. People create the cover advertising of a product box, imagining that the box and contents will be sold in a store. For similar activities see *Innovation Games* [Hohmann06].

- **Establishing common vocabulary and concepts**—Try sketching a *domain model* at a whiteboard [Larman04a].
- **Generating ideas (such as features or constraints)**—Try *brain writing* in which many people (separately or in pairs) write ideas, one per card or small piece of paper.

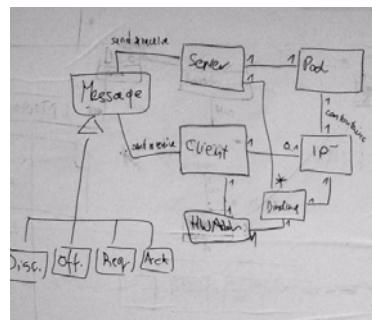


Figure 7.4 brain writing requirements during a multiteam joint requirements workshop; affinity clustering on the floor; mind map at wall



- **Grouping and organizing ideas**—Try *affinity clustering* the *brain-written* cards: Cards are shuffled on the floor into families. Try *mind mapping* [Buzan96]: The cards on the floor have their information organized onto a mind map at the wall. The mind map can then further evolve.
- **Clarify and specify with examples**—See “Try...Specification by example—usually in tables” section on page 245.

Avoid...A large queue of well-analyzed, fine-grained PBIs

Try...Maintain only a small queue of fine-grained PBIs

Figure 7.5 illustrates how to do evolutionary PBI refinement in Scrum. Keep only a small queue (a WIP inventory) of clearly analyzed, finely split, prioritized items. This reduces overprocessing and supports the benefits of small queues, explored in *Queueing Theory* in the companion book.

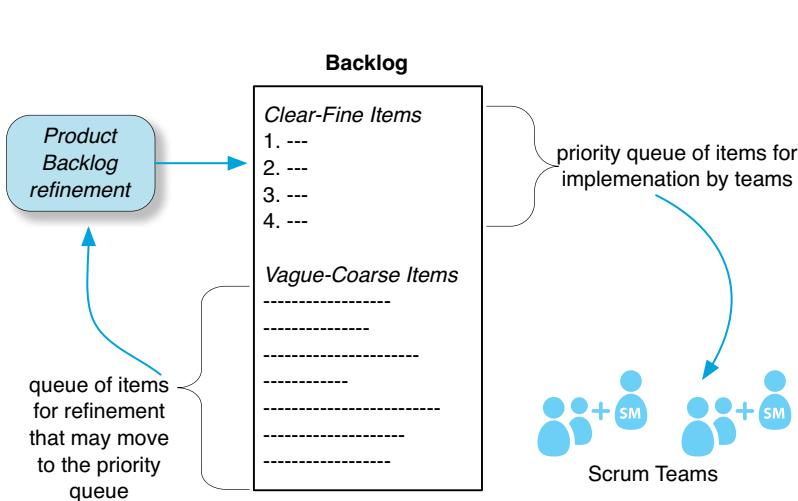


Figure 7.5 refine the backlog incrementally

How to do this ongoing refinement each iteration? ...

Try...Requirements workshops for Product Backlog refinement

Experiment with requirements workshops for refinement, with the team(s) and Product Owner or representatives. Try them half way through each iteration so that people have time to resolve discovered issues (such as clarification) before the next iteration.

Refinement involves splitting, detailed analysis, estimation, and prioritization. The next section is a workshop story focused only on *splitting* and *detailed analysis* activities. It is an example from our coaching, not a recipe. It assumes a single-team case; the subsequent suggestion shares multiteam tips.

Ideally, the setting is the team room itself.

1. Group identifies some big or vaguely analyzed items that are worth refining. A well-prepared Product Owner identifies some of these before the workshop. To avoid computers, the Product Owner brings desired items written on cards, or the workshop starts with card writing.
2. Apply *diverge-merge cycles* for splitting and/or analysis...

3. Diverge—Rather than all eight people discussing and modeling



the same item, diverge into two or more sub-groups (at least two people per group) and work in parallel on different items. Each sub-group is at a different wall of the workshop room, with their own creativity tools. The Product Owner and other subject matter experts do not belong to any group; they rotate between groups, spending a few minutes at one area to clarify and decide and then moving to the next group.

- This approach increases the speed of analysis or splitting, and smaller groups tend to engage *everyone*; in large groups some people become passive and there is more *analysis paralysis* owing to an *overload* of simultaneous issues.

4. Merge—Agile development is based on *whole team together*, not



sub-groups, so it is important for everyone to synchronize, learn, and feed back. After 30 or 45 minutes of separate analysis, people come together and review each sub-group's work. We usually do this as “show and tell” in which the team visits one area, the sub-group presents their results, there are question and answers, and input. This visit is usually finished within 15 minutes; then the team moves on to the next wall area.

5. Diverge and merge repeatedly—The team repeats these cycles multiple times until the items under refinement are sufficiently clear.

- *Set-based development and diverge-merge cycles*—The lean practice of set-based development can also be applied with diverge-merge cycles. Instead of each sub-group working on

see Lean Thinking in companion

a *different* item, each sub-group works in parallel on the same item, while subject matter experts rotate across the groups. This sparks more creative variation and accelerates overall learning or discovery regarding an item.

Techniques—(1) Write *examples* (usually in tables) using pseudo-

C	Action	IN OP ID	IN TIME/DATE	IN OPERATION	OUT NEW TIME	OUT NEW OPERATION
1	MODIFY	100	NOW + 2 MIN TIME + 1 hr	—	NOW + 2H	SAME
2	MODIFY	103	—	NEW MM1	SAME	NEW MM1
3	MODIFY	112	NOW + 5 MIN TIME + 1 hr	NEWER MM2	NOW + 5 MIN	NEWER MM2
4?	MODIFY	151	NOW - 5 MIN	—	?	?

code for executable acceptance tests, and then derive business rules, on whiteboards. (2) Each group maintains an “issues and questions” flip chart. (3) Ask for each item, “Have we considered FURPS+?”

Wrap up and take away?—Abstractly, the workshop output is *more learning and shared understanding*. Concretely, it is cards, flip charts, and whiteboard sketches. If new split sub-items need adding to a Product Backlog in spreadsheet format, the Product Owner enters the information from the cards. Move the flip charts back to the team area. Take digital photos and store them on a wiki page.

Try...Specification by example—usually in tables

This idea reiterates the “Try...Use examples” section on page 50 (and other sections) in the *Test* chapter. Using *examples* (in a requirements workshop) to clarify and communicate is not complicated, but it is surprisingly rare in requirements practice. That is unfortunate, because this is a *wonderfully* helpful technique, and attractive to non-technical experts—as they quickly find it a familiar, concrete way to help bridge the communication gap.

See “Try...Acceptance test-driven development” on p. 42.

See “Example: Robot Framework” on p. 83.

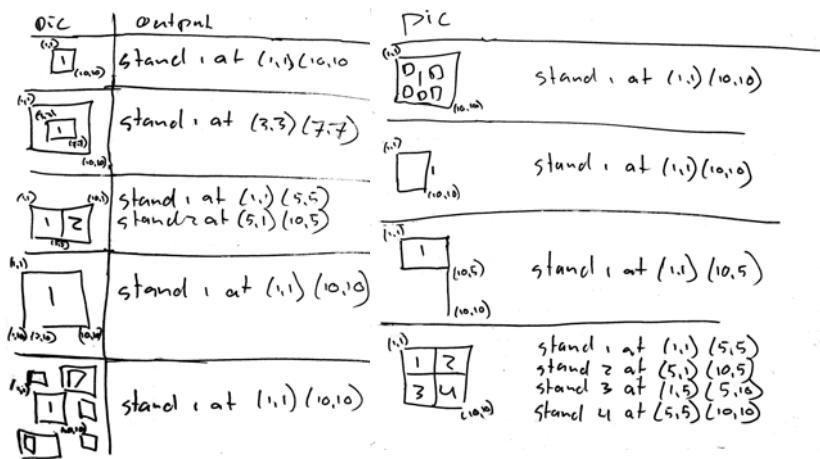
People are so used to discussing and writing specifications for the *general* case, such as in use-case narratives, that the deeper dynamics behind *examples* are not usually grasped until applied.

Although any familiar or natural format for examples is acceptable, prefer a *table* format when possible; tables can improve the ability to understand information and see discrepancies or patterns.

And because several acceptance test-driven-development frameworks (including FitNesse and Robot Framework) use tables, these *table examples* easily map or distill to executable *table tests*.

The recommended reading *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* explains more.

Figure 7.6 using examples to specify, in tables



Try...Joint requirement workshops

Multisite: See
“Interaction &
Coordination” on
p. 423.

With true feature teams the need for a requirement workshop that spans multiple teams is reduced. Nevertheless, there are still times when a **joint workshop** is useful...



- when early visioning
- when teams are working on related features with a common theme or common ancestor

The diverge-merge pattern supports scaling the workshop to multiple teams; the sub-groups are slightly larger and more wall areas are used. Since there are more sub-groups, ensure enough subject matter experts (Product Owner or representatives) rotating across the groups to avoid a group being blocked too long without answers. We have facilitated up to 50 people *effectively* in one workshop with this approach.

As with any workshop, this one may include all members of all teams that want to attend, or if that is impossible because of overall size, then team representatives—although avoid that if possible to avoid handoff waste and information scatter. Figure 7.4 illustrates a joint workshop with representatives from a big development group.

See “Avoid...‘Optimizing’ the requirements workshop” on p. 51.

Try...Stop refining an item once it is fully INVESTed

When is an item refined enough to implement in an iteration? Apply the INVEST test. Bill Wake [Wake03a] advocated that as items—he was referring to *user stories*—are incrementally refined, they ultimately have the INVEST qualities before development...

Independent: The item’s implementation order does not depend on others—desirable but not always achievable, especially for small items split from big ones. **Negotiable:** A user story is not a fixed contract; through conversation a new or simpler definition of acceptance may arise. **Valuable:** A user story has identifiable value to the customer. **Estimatable:** The item is clear enough that the team has confidence that the estimate is not a *complete fantasy*. **Small:** Can meet the Definition of Done within a third or quarter of an iteration by the whole team working together.¹⁶ **Testable:** The definition of acceptance is clear and can be verified with automated tests.

Try...Split Product Backlog items (such as stories)

“We cannot possibly fit our requirements into two-week iterations, and there is no way they can be made smaller.” In response to this,

16. Normally, Scrum suggests even smaller items, but for the originally massive requirements for embedded-software systems, that is not usually desirable because of the overhead cost of so much splitting.

we usually invite the person to tell us their largest, most *impossible* item that could *never* be split into small customer-oriented items, and together at a whiteboard we split it.

For every requirement of every size and type—even those involving hundreds of people for many months—we have been able to split it into small customer-oriented items. We often *hear* that some requirement cannot be split, but we have never *seen* one.

Splitting Overview

This tip spans many pages because there are multiple aspects to splitting, with differing trade-offs. “*We have big requirements. What do we do?*” and “*How do you split big requirements?*” are two of the most common questions we get. Plenty of examples help…

Why split?

- Small items increase control and visibility for the Product Owner. Release goals and customer ‘promises’ are typically made on huge items, but these consist of sub-items with different priority that can—and *should*—be considered separately.
- Customer-centric ‘vertical’ splitting helps divide and parallelize *valuable* work over multiple teams.
 - In traditional large development, splitting is also done, but into *technical tasks* for separate teams. The splitting is along ‘horizontal’ architectural or component lines. Each task is not independently customer-centric, and there is a long delay before the tasks across the components and teams are integrated to finally deliver a valuable ‘vertical’ feature.
- In Scrum, items chosen for implementation should be small. See *Queueing Theory* in the companion book for some of the justification of implementing small similar-sized requirements.
- Splitting supports evolutionary analysis and development.
 - identify smaller items within bigger ones that are worth analyzing and implementing soon, versus parts that can be done later—or never

see the Feature Teams Primer chapter

- deliver quickly a minimal viable product of high-value items
- reduce wastes of over processing, WIP, and inventory
- build, integrate, and *test* smaller items early, increasing feedback and attacking key risks

When?—During (1) Product Backlog refinement workshops, (2) Sprint Planning when an assumed-small item is discovered to be too large, and (3) Sprint Planning when a previously unseen item is offered. (This last case is ill-advised, due to increased variability.)

Who?—Team and Product Owner or representatives.

Tools?—If the entire team is involved in splitting an item, sketch the splitting on a whiteboard. Paper cards are also handy; write each split item on a separate card (Figure 7.7).

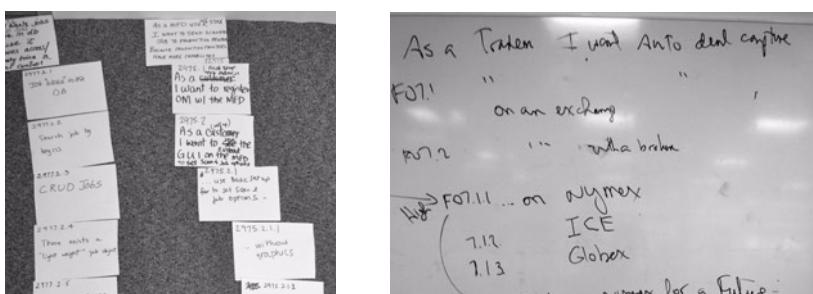


Figure 7.7 tools for splitting items

Splitting Perspectives

The following table lists some *splitting perspectives* we have found useful to help people learn how to split items.

use case	- the major work flows or cases of use	configura-tion	- a varying configuration, such as type of operating system
scenario	- a <i>specific</i> sequence of steps within a use case	I/O channel	- an input or output channel, such as GUI or command line

data part	- the data has many elements; a subset may be useful	data format	- XML, ...
type	- varying types or kinds of things, such as types of trades	role or persona	- novice or power user, administrator or clerk, ...
knowledge	- one sub-item is understood, but others need learning	NFR	- moderate vs. high throughput, with or without recovery, ...
test sub-group	- many acceptance tests exist, fulfill some sub-group	operation	- a system operation, such as HTTP GET
scenario step(s)	- one or more steps in a multi-step scenario	CRUD	- create/retrieve/update/delete use case or operation
integration	- the solution requires integration with an existing element	stub	- a ‘fake’ implementation of something

This list could be simplified. For example, several perspectives are specializations of *type*: type of I/O channel, type of operating system (configuration), and so forth. That said, this fine-grained list is useful for new practitioners because it highlights common categories they do not—at first—identify.

How to use this list? Ask, “What <perspective> are there in this item?” For instance, “What configurations are there?” or “What data formats are there?” Examples are in a following section.

Try...Ask, “What benefit from splitting in this way?”

Alternative splittings for value and risk—There is not one correct splitting. Maybe an item could alternatively be split by scenarios or configurations. How to decide? Use these guidelines:

- early ‘value’: (1) better return on investment, (2) better cost reduction, (3) satisfaction of key customers, (4) alignment with strategic objectives, (5) critical to minimal viable product
- early risk mitigation—business and technical risk
- the sub-items are better INVESTed (p. 247)

When splitting, *understand the benefits of your strategy*.

For example, suppose an investment bank wants to *trade three types of financial derivatives (futures, options, and swaps) on three exchanges (Nymex, ICE, Globex)*. This can be first split by *derivative* or by *exchange*.

Years ago, the development group implemented something else (very different) that involved interaction with Nymex and ICE, but they have never worked with *Globex*. Their experience is that all exchange integration is slow and difficult, involving poor documentation, inconsistencies, and reliance on third-party exchange programmers who are too busy to help. In short, they know that integration with *all* the exchanges will be slow, surprising, and awkward, and will involve coordination with others. And in addition, *Globex* is a black hole of uncertainty.

The Product Owner has no derivative-type preference—not true, but let's pretend. Implementing for all exchanges will be slow, *Globex* is unknown, and there are no short paths to early value delivery. Then, solely by the principle of risk mitigation, the first-level splitting (and priority) is by *integration or I/O channel*:

1. trade derivatives on *Globex*
2. trade derivatives on other exchanges

More realistically, the Product Owner first wants *futures* trading on all exchanges, because that's where the big money is—they hope! *Combining* a strategy for risk mitigation and early value delivery:

1. Trade *futures* on *Globex*.
2. Trade *futures* on other exchanges.
3. Trade other derivatives on other exchanges.

Example: DHCP (Networking)

The first larger example is from the domain of networking: server-side support for Dynamic Host Configuration Protocol (DHCP). Here is the original item:

dhcp.¹⁷ As a Network User, I want DHCP server support¹⁸ so that I can easily get a viable IP address and then use the network.

Split by use case—DHCP support has several main use cases related to the life cycle of IP address leases (obtaining, renewing, releasing, ...). The *dhcp* item can be split by use cases as follows, expressed in user story format:

- ❑ *dhcp.request. As a Network User, I want the server to handle requesting an IP address so that I can easily get a viable IP address and then use the network*
- ❑ *dhcp.renew. As a Network User, I want the server to handle renewing an IP address so that I can easily keep using my current IP address when its lease expires*
- ❑ ...and more

Why split this item by use case?

- ❑ A use case delivers independent customer-centric value.
- ❑ The use cases have different value-delivery priority; *dhcp.request* is critical to a minimal viable product, but the other uses cases are not.
- ❑ Implementing a use case requires ‘vertical’ development, integration, and testing across many components, addressing major risks: (1) delayed feedback or verification of large architectural decisions, and (2) delayed integration.
- ❑ Use cases are natural to this domain;¹⁹ DHCP documentation is frequently organized by these use cases.

Common misunderstandings or misuse of use cases: We work with many groups purportedly “doing use cases” in Scrum and see fundamental mistakes. Understand the following:

- ❑ Use cases are *not diagrams*. Use cases are *text* documents and involve writing, not drawing.

-
17. Notice that the requirement IDs—such as *dhcp* or *dhcp.request*—are informative, in contrast to a conventional legal numbering scheme such as 5, 5.1, or 5.1.1. Try this informative style.
 18. Ideally, items do not indicate technical solutions such as ‘servers.’ However, DHCP is a standard that mandates a server and its behavior; this is a *constraint*.
 19. In fact, use cases were first used in telecommunications.

- In big systems, some incorrectly write use cases for *internal* elements of a system (where one element is the client of another element). This is a significant misunderstanding and misuse. Use cases are meant to be written from the external view of true outside actors, such as a human or ‘robot’ that makes a phone call.

For example, this is a use case: it is text, and written from the viewpoint of actors truly external to the overall system:

Use Case: Request IP Address

Main Success Scenario:

1. Client broadcasts a discover message.
2. System (a DHCP server) responds with offer message for IP address.
3. Client sends a request message to confirm reservation.
4. System responds with an ack message.
5. ...

Evolutionary and partial splitting—From a Scrum perspective it is not necessary to split a big item into all its possible sub-items—at least not all at once, and perhaps never. That creates the wastes of overprocessing and inventory (of WIP requirements).

In DHCP, the `dhcp.request` item is critical, but other parts are optional. Therefore—usually during a Product Backlog refinement workshop—you can split like this:

dhcp. As a Network User, I want DHCP server support so that I can easily get any viable IP address and then use the network.

- *dhcp.request. ...I want the server to handle **requesting an IP address**...*
- *dhcp.full-support. I want **full DHCP support***

Once `dhcp.request` is complete or near-complete, return to splitting `dhcp.full-support` into a few other sub-items by the guiding principles of value and risk.²⁰

20. The splitting need *not* be binary into only one useful sub-item (`dhcp.request`) and one ‘full-support.’ Yet, split into only a few sub-items rather than all sub-items.

In short, *evolutionary and partial splitting*, over time.

In a way, this is simple advice. Yet it is *very* different behavior than traditional requirements engineering. Breaking the “big analysis” habit demands coaching and follow-up.

Placeholder sub-items for “everything else”: The sub-item `dhcp.full-support` is a convenient placeholder that captures an estimate for the remainder of the ancestor.

Split by scenario—A use case is a set of success and failure scenarios—all the specific paths from start to finish. For instance, several scenarios of `dhcp.request`:

- ❑ *dhcp.request.success.main. ...I want a viable IP address when requesting an IP address and free ones are available...*
- ❑ *dhcp.request.fail.none free. ...I want to receive an error message when requesting an IP address and none are free...*
- ❑ *dhcp.request.other: Everything else.*

In many cases, splitting by scenarios is desirable. Why?

- ❑ All benefits of use-case splitting apply to scenario splitting.
- ❑ We guess an average use case has 20 or more scenarios, so scenario splitting dramatically reduces a big item into many much smaller ones, leading to more flexibility and visibility for the Product Owner in terms of fine-grained items.
- ❑ If the ancestor item is a use case, scenario splitting is natural and simple.

In this case, the scenario item `dhcp.request.success.main` is worth early identification and implementation: it is critical to a minimal viable product and it allows early integration testing of existing DHCP clients with this new server under development.

Split by I/O channel; alternative splittings—In the DHCP case we worked on, the customer’s network elements supported both IP over Ethernet, and IP over ATM (asynchronous transfer mode).

These are examples of different I/O channels, and implementing the DHCP server involved different work for these two channels.

So, an alternative splitting of the dhcp item by I/O channel is

- dhcp.ethernet. As a Network User/Device with Ethernet, I want DHCP server support over Ethernet...*
- dhcp.atm. As a Network User/Device with ATM, I want DHCP server support over ATM...*

Is this a useful way to split? The teams knew that implementing the Ethernet case required no Ethernet-specific work; but there was specific ATM work—though only minor tweaks once the basic server was built. It was low-risk wrap-up work.

Since dhcp.ethernet does not represent distinct work (specifically for Ethernet), it is not worthwhile identifying it as a distinct item. In contrast, dhcp.atm is meaningful. A better splitting:

- dhcp.request. handle requesting an IP address*
- dhcp.full-support. provide full DHCP support*
- dhcp.atm. As a Network User/Device with ATM, I want DHCP server support over ATM...*

Split by scenario or data part or type?—This next example illustrates that a splitting may be classified under several perspectives or a mixture thereof. The classification is not really important.

When an IP address is requested, two noteworthy cases are (1) reserve and give the client *any free* address (the common case), and (2) reserve and give the client a specific address that *they provided*.

By scenario—This can be classified as scenarios of the use-case item dhcp.request:

- I want any viable IP address when requesting an IP address and free ones are available...*
- I want a specific IP address, X, when requesting an IP address and X is available, so that I can control my identification*

By data part—The DHCP REQUEST message record that is sent to a server contains many data elements (fields). One particular element is either empty, implying any free address, or filled with an address, implying a request for that one. From this perspective, this is splitting by *data part*.

By type—More abstractly, this is about two types of addresses: generic versus specific. From this perspective, it is splitting by *type*.

Split by simple success scenarios—In a use case, there is one *main success scenario*. How to write an item for a simple scenario that ignores all complexity or concerns?

- *...as simple as possible, I want any viable IP address when requesting an IP address and free ones are available...*

Why do this? Early identification and implementation of simple success scenarios quickly delivers something customer-centric. Also, implementing this engenders “tracer bullet development” [HT99] in which a “walking skeleton” of the system across various architectural elements is designed and tested. This gives early feedback about architecture and integration, addressing these risks.

There is a risk: Have you seen a demo of a “happy path” to product management, and the response is a variation of “Great! We’re nearly finished.” There are still failure cases to handle—and in some systems, doing those is the major effort.

And, sometimes—especially in massive systems—the estimated effort for even a simple success scenario is too high for the team to implement within an iteration. For either of these cases, consider...

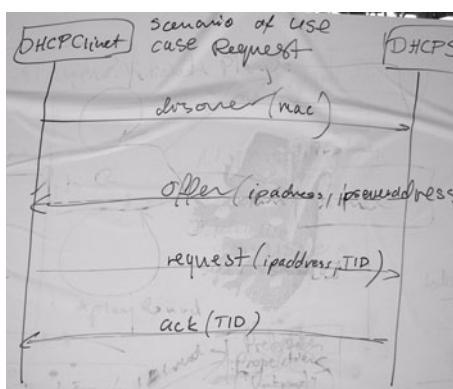
Split by simple failure scenarios—DHCP is not complicated; a simple end-to-end success scenario *can* quickly be done by a team. In contrast, we were involved in implementing the 3G telecom standard HSDPA (High-Speed Downlink Packet Access) in a Radio Access Network. Even a simple success scenario was too much work.

The development team started the splitting by trying to simplify the success scenario. They discussed

- *Make an HSDPA call in the simplest possible network configuration, ignoring all error cases.*

But they quickly discovered that even this was too much to do in the first iteration. Thus, instead of looking at success scenarios, they started splitting from a failure perspective—there are many in a radio network. They first split for the simplest possible failure scenario—no network connection—and then gradually worked down the stack and the protocol, implementing more failure scenarios. After two iterations, the cumulative failure scenarios had put enough *stuff* in place for them to work on the simplest success scenario. Then they could implement more complicated success scenarios, and leave exotic failure cases until the end.

Why is this useful? By splitting on failure cases they gradually build up functionality while still focusing on a customer perspective. Plus, they are addressing some risks early, and increasing learning.



Split by operation—In this context, **operation** or **system operation** means exactly one kind of computer message or signal sent to a computer system, requesting it to do something. The system sequence diagram in this photo illustrates a scenario of DHCP and shows two system operations: *discover* and *request*. This offers another way to split the `dhcp.request` item:

- ❑ `dhcp.request.opDiscover`. Server handles **discover** operations
- ❑ `dhcp.request.opRequest`. Server handles **request** operations

Implementing only the customer-visible DHCP *discover* system operation by itself does not deliver an end-to-end solution, because *request* is also needed for the two-step scenario. However, it may reduce risk or increase feedback while incrementally building up the feature. The “potentially shippable” perfection challenge for each iteration in Scrum does not imply that *all* items must be stand-alone useful, although that is a very worthy goal.

If risk were associated with *discover* or *request*, that could motivate splitting by operation—in our experience not the case with DHCP.

In contrast to *discover*, sometimes *one* system operation is *independently* valuable as a customer solution. For instance, in the HTTP protocol the *GET* operation is independently useful.

Split by CRUD—CRUD is the acronym for create, retrieve, update, delete. When each CRUD case by itself is a set of complex scenarios, then these four common cases are modeled and realized as four separate *use cases*, such as *Create Account* and *Update Account*.

On the other hand, when each case is simple and involves only one signal to a computer system, these can be modeled and realized as four *system operations*, such as the messages *createAccount* and *updateAccount*.

Therefore, when you work on CRUD items, consider splitting by operations or use cases.

Recall: “Try...Defer or ignore implementation and analysis of sub-items” section on page 223. Not all CRUD sub-items must be done.

Split by integration—There are open-source DHCP solutions. Assume the plan is to use one and then modify it with some embellishments—for instance, to improve scalability. Then, for example:

- ❑ *dhcp.integrate. ...Provide a DHCP server by integrating an open-source solution*
- ❑ *dhcp.scaling. ...DHCP server support with a mean-response time (when 100 simultaneous requests) < 0.3 seconds...*

It is inadvisable to embed a technical solution (such as, “by integrating with an open-source solution”) in requirements since they should focus on ‘what’ rather than ‘how.’ But this is arguably a reasonable exception because the Product Owner is involved in this key ROI choice, and then the solution dominates how work is split.

Split by acceptance test sub-groups—Usually, an acceptance test represents a splitting by a perspective already discussed, such as splitting by scenario. Then, it is not useful to think about “split-

ting by acceptance tests” as a distinct strategy. There are exceptions...

One of our customers was implementing DHCP and had found a published public-domain document describing 74 acceptance tests. For example, here is acceptance test TS-5.4 from the suite:

If the selected server is unable to satisfy the DHCP REQUEST message (e.g., the requested network address has been allocated), the server SHOULD respond with a dhcpTypeNack message.

This test represents a scenario; in fact, all the 74 tests represent splittings by scenario, data part, and more. But the point in this example is that the Product Owner may use the list *as the organizing model for splitting*; he may say, “Let’s identify a sub-group of tests that ‘make sense together,’ and satisfy them as one item.”

Example: Automated Derivatives-Trading Deal Capture (Finance)

Some of our work is with investment banks and energy-trading companies. The following item represents a use case:

adtdc. As a Derivatives Trader, I want automated derivatives-trade deal capture (ADTDC) so that I have more free time, there are less errors, and the deal is processed faster.

This was then split by I/O channel because trading on exchanges was more much more frequent than trading via a broker. Also, implementing exchange trading first was more valuable:

- adtdc.exchange. ...when trading via an **exchange**...*
- adtdc.broker. ...when trading via a **broker**...*

There are several exchanges; Nymex was used most frequently, so automating that was most valuable. Again, a split by I/O channel:

- adtdc.exchange.nymex ...trading via **Nymex** ...*
- adtdc.exchange.other...trading via **everything else**...*

There are several derivative types; futures were traded most and generated the most profit. Split by type of derivative:

- adtdc.exchange.nymex.futures ...trading **futures on Nymex**...*
- adtdc.exchange.nymex.other ...trading **everything else**...*

See “Try...Lots of stubs, plus dependency injection” on p. 318.

Split by stub—Another common and useful strategy is to split by *stubs*²¹ for *dependent elements*. This is especially useful when the dependent element is *hardware* or a *remote service* or *remote component* needing network or inter-process communication. For embedded-software systems, a stub is an invaluable tool to decouple from hardware dependencies.

Our client (an energy company) has developers, and Nymex has developers. To implement items for trading, our client had to ask Nymex to make a software change on their side; Nymex software was outside our client’s control. Our client had to wait months before the change was ready.

Introducing a local stub for Nymex solves several problems:

- reduction of the waste of waiting, since we can implement our part of the solution with a stub
- ability to test our solution quickly and in isolation, without the delays and setup complexities of remote communication
- our *assumptions* recorded in *learning tests*

Introducing a stub *could* delay integration with the real dependent element if when the dependent was available, people still avoided integrating with it. Delayed feedback—bad idea. But that is not justification to avoid local stubs, just to also integrate early.

Stubs are good. Early integration is good. Do both.

Therefore, split by stub:

- ...trading **futures on a stubbed Nymex**...*

21. A local stub is useful even when it is a simple fake component whose operations do little or nothing. We sometimes meet people who express a false dichotomy thinking about a stub: If it is not a perfect simulation, it is not useful and should not be done. Not true!

- ❑ ...trading **futures** on real Nymex...

When splitting by stub, always include the complementary stub and non-stub versions as two related items.

Split by I/O channel: API and GUI channels—An *endemic* problem in software development is that developers embed application or functional logic in the GUI²² layer (or more broadly, UI layer) rather than maintaining a separation of concerns and keeping the logic in a distinct layer. We see this problem exacerbated in big, multisite, and offshore groups because there is less concern for hiring master programmers, and no culture of coaching from masters.

This problem inhibits

- ❑ reuse of the logic when other UIs are introduced
- ❑ automated testing because one must test the functionality *through* the GUI layer—difficult to write and to maintain
- ❑ test-driven development because of slow test cycles

All this can be solved by applying the guideline that *whenever an item involves a GUI, split by I/O channel with a version that uses a non-GUI API, and a version that uses the GUI*. For instance:

- ❑ ...when trading futures, **via an API**
- ❑ ...when trading futures, **via a GUI**

When splitting this way, always include the complementary API and GUI versions as two related items.

Implement the API version first. This will require the development team to implement the application logic in non-GUI layers and components, and enable simple fully automated testing of the functionality via the API (rather than via GUI testing).

22. For instance, a browser or Java Swing client. There are, by the way, solutions so that browser-hosted JavaScript can be organized into separate UI and application-logic layers, and developed/tested separately. Probably the most well-known is *Google Web Toolkit*.

Notice that the benefit is related to maintenance and development speed rather than the immediate concerns of the derivatives trader. *Not* doing this will become a concern to business people when development becomes slower and slower because of bad engineering.

Split by NFR—Reducing the interaction steps, complexity, and possibility for errors is important when the user is a derivatives trader, whose per-minute profit or loss capacity exceeds most of our incomes in a lifetime. It is great if we can automate the deal capture; it is even more valuable if we can make the solution extremely *usable*. As with most non-functional requirements, ‘usability’ is not binary but relative; it can be measured and improved over time.

Most NFRs can be split to gradually improve things. For instance:

- ...*I want deal capture when trading futures on Nymex, with 90% of standard deals complete in 60 seconds*
- ...*I want deal capture when trading futures on Nymex, with 90% of standard deals complete in 30 seconds*

Example: Automation of a manual printing process (Commercial Printing)

One of our clients creates solutions that help print operators who do large-volume customized printing (such as marketing leaflets). Print operators set up print jobs that have page templates in which *variable data* (such as customer names or photos) is inserted uniquely on each page. This can involve manual steps.

One of the original items represented splitting by use case:

handleVariableData. As a Print Operator, I want automated preparation and processing of variable data, to have more free time and fewer errors.

Split by scenario step—Often, splitting by *scenario step* delays delivering value because (usually) all steps must be implemented. But in the case of automating an existing manual process, even automating one step or two steps is useful and deliverable...

1. *handleVariableData.associate. (step 1) I want to associate a container and a queue...*

*2. handleVariableData.verify. (step 2) I want to **view and verify** an associated container...*

3. ...

Furthermore, even if *one* customer-visible step did not realize a complete stand-alone solution—as with splitting by operation—implementing it can mitigate risk or provide feedback while incrementally building up the complete scenario.

Example: Data transfer (Geophysical Analytics)

One of our clients analyzes geophysical data to make decisions related to subsurface oil, gas, and water extraction or injection. They transfer large sets of measurement data between their products; for example, one calibrates models and another does simulations. The following item represents a use case:

As a Geophysicist, I want to transfer measurement data between ProductA and ProductB so that I can analyze it.

There were three kinds of data to transfer, in very different formats and with different content. The work was quite different for each type. The following splitting was used; it could be classified as splitting by **type (of data)** or by **data format** or by **scenario**:

- *transfer well data; transfer log data; transfer horizon data*

Well data was the critical case—most frequent and associated with the most money. Splitting this way created smaller, independently valuable items.

When dealing with measurement data and different applications, one can encounter problems related to units and coordinate systems. Sometimes, units, precision, or coordinates have to be transformed for a receiving system. The following splitting was used; it could be classified as splitting by **scenario** or by **data format**:

- transfer well data when no transformation is needed...*
- transfer well data when units and coordinates transformed are needed for the receiving system...*

Transforming *units* is major work distinct from the work to transform the *coordinate system*. This is splitting by **data parts**:

- transfer well data with **units** transformed
- transfer well data with **coordinates** transformed

Split by NFR—Transferring and transforming data takes time. A geophysicist would like the data *before retirement*. It takes significant software engineering effort to go from sluggish to zippy. Split by **non-functional requirements** for incremental improvement...

- transfer well data with units transformed **in less than 60 seconds per megabyte**
- transfer well data with units transformed **in less than 10 seconds per megabyte**

Example: Virus detection (Security)

Several of our clients develop security software. A use-case item:

detect viruses

The following splitting was used; it could be classified as splitting **by I/O channel** or **by configuration (of input channels)**:

- *detect when email; detect when browser; detect when USB stick*

For browsers, the following splitting was used; classifiable as splitting **by type (of browser)** or **by configuration (of browser)**:

- *detect when FireFox; detect when MSE; detect when Safari, ...*

It was also useful to split **by use case (or scenario)**:

- detect when MSE when downloading a file*
- detect when MSE when downloading & installing plug-in*

Split by NFR—When people start a computer, they do not want to wait *hours* while the virus software loads. And they do not want it to consume 50% of processor resources. Suppose that “ten seconds and ten percent” vaguely annoys people but “five seconds and five per-

cent” is a happy customer. It could take plenty of engineering effort to reach that goal. Split **by non-functional requirements** for incremental improvement...

- load in 30 seconds; load in 10 seconds***
- maximum 10% of processor capacity; maximum 5% ...***

Split by role—Software security companies have internal *power users* who want to detect viruses; their job is exploratory observation and probing to see what viruses are infecting systems. They want to be notified and see what is going on in great detail. General computer users also want virus detection, but they (usually) just want silent resolution while they carry on downloading torrents from PirateBay. This leads to splitting **by role**:

- As a Virus Security Analyst, I want virus detection....***
- As a Computer User, I want virus detection...***

Splitting by role is not useful unless it implies role-distinct work.

Split by knowledge (or research)—Suppose your organization is creating a new virus-detection product. There are some viruses (or categories of viruses) people in your group understand and can implement to detect, others that outsiders have described but are not yet understood by your group, and others that nobody understands and that require deep investigation.

This leads to splitting items into families of requirements that are understood and can be implemented, and those needing research.

We see this frequently with our customers. As a different example in the printing domain... “The X-part of the new PDF specification is like stuff we’ve worked on before. But the Y-part is going to take some research before we can implement it.” Y-part requirements will need to be distinct items in the Product Backlog.

See “Try...Genuine research work as PBIs” on p. 227.

Conclusion

With these techniques, we decompose large requirements into much smaller slices that are still fully or partially customer-centric, rather than splitting big requirements only into myriad technical tasks.

Avoid...Adopting user stories because they are ‘agile’

“We are doing agile development, therefore we must apply user stories.” Incorrect. Scrum does not mandate any particular approach to requirements, and there is no guaranteed relationship between writing a user story and *being agile*.

The companion book briefly explored *cargo-cult process adoption*—ritualistic adoption of practices, often without judgment or skill. “Cargo-cult user-story adoption” happens in several situations:

- A group of people *want* to be agile and adopt agile development, but they assume a particular practice is necessary.
- A group—usually a large group in an “enterprise transformation”—does *not want* to adopt agile development, but are pushed to do so by top-down directive. People just follow orders (or do what appears necessary to meet “agile adoption” targets) and do superficial “agile stuff.”

Avoid...Believing *writing* user stories means *user stories*

As a Mobile Device User, I want HSDPA support in the network so that I can download faster and have more free time.

First, writing a user story in the above format does not mean it is a ‘correct’ user story. As originally described, user stories do not require any particular format; they must simply be *very short* and *understandable to the customer*.

Second, and far more important, *writing* a user story is not *applying* user stories. What is?...

Try...Apply user stories with card, conversation, confirmation

Big product groups usually have high ceremony around their requirements work. The culture is *comprehensive documentation over working software*, and *not* talking frequently with customers to evolve understanding. Then, when “do user stories” is *pushed* onto this group, the *intention* and *behavior* is not grasped; what remains

is superficial practices such as writing statements *As a <Customer-Role> I want <Goal> so that <Reason>*.

The concept of user stories comes from Extreme Programming (XP), where the intention and behavior is emphasized in the ‘3Cs’ that are the heart of *applying* user stories [Jeffries01]:

- **Card**—One user story is first written on one index card. The deeper point of ‘card’ is that a user story (1) is a short statement—just a *label* for a requirement that does not contain all information, and (2) it is easy to change or discard.
- **Conversation**—Alistair Cockburn succinctly described a user story as “A promise to have a conversation.” This is a key behavioral change in adopting user stories: Teams increase their conversation and collaboration with customers and the Product Owner rather than emphasizing detailed up-front written documentation. This reflects the third agile value, *customer collaboration over contract negotiation*.
 - No false dichotomy: Written documentation is acceptable but *is the result of ongoing conversation*.
- **Confirmation**—*Before* a user story can be implemented, the Product Owner needs to define the acceptance criteria that confirm the user story is fit for the purpose. This is another key behavioral change in adopting user stories.

To quote Ron Jeffries, one of the original agile thought-leaders, who first described the 3Cs of user stories:

...the writing is not the most important part. Quite likely it is the least important part, far behind the thinking, the communicating, and the testing.

Avoid...User stories good; other models bad

“In Scrum, requirements must be written as user stories, because they are good.” Not true. Scrum is requirements-model neutral—one of its strengths, not a weakness. Officially, the Product Backlog contains *items*—a flexible abstraction. Scrum’s strength and longevity lie precisely in *not* prescribing *how* teams should work.

Avoid false-dichotomy suggestions such as “user stories are better than other models.”²³ A backlog *item* may be explored as user stories, use cases, or anything else...

Try...Learn many analysis skills: user stories, use cases, ...

Expand a team’s detailed-analysis literacy to multiple approaches—each applied in the spirit of **agile modeling** [Ambler02]: simple tools, “barely good enough,” active stakeholder participation. Apply these varied techniques to *small* batches of detailed-requirements inventory (low WIP) in a small queue—detailed analysis will focus on the next few iterations.²⁴ *Evolve* details with conversation.

What techniques to use? Many! See *Figure: Experiment with many analysis techniques*. The graphic is merely hints; see the recommended readings for a deeper dive.



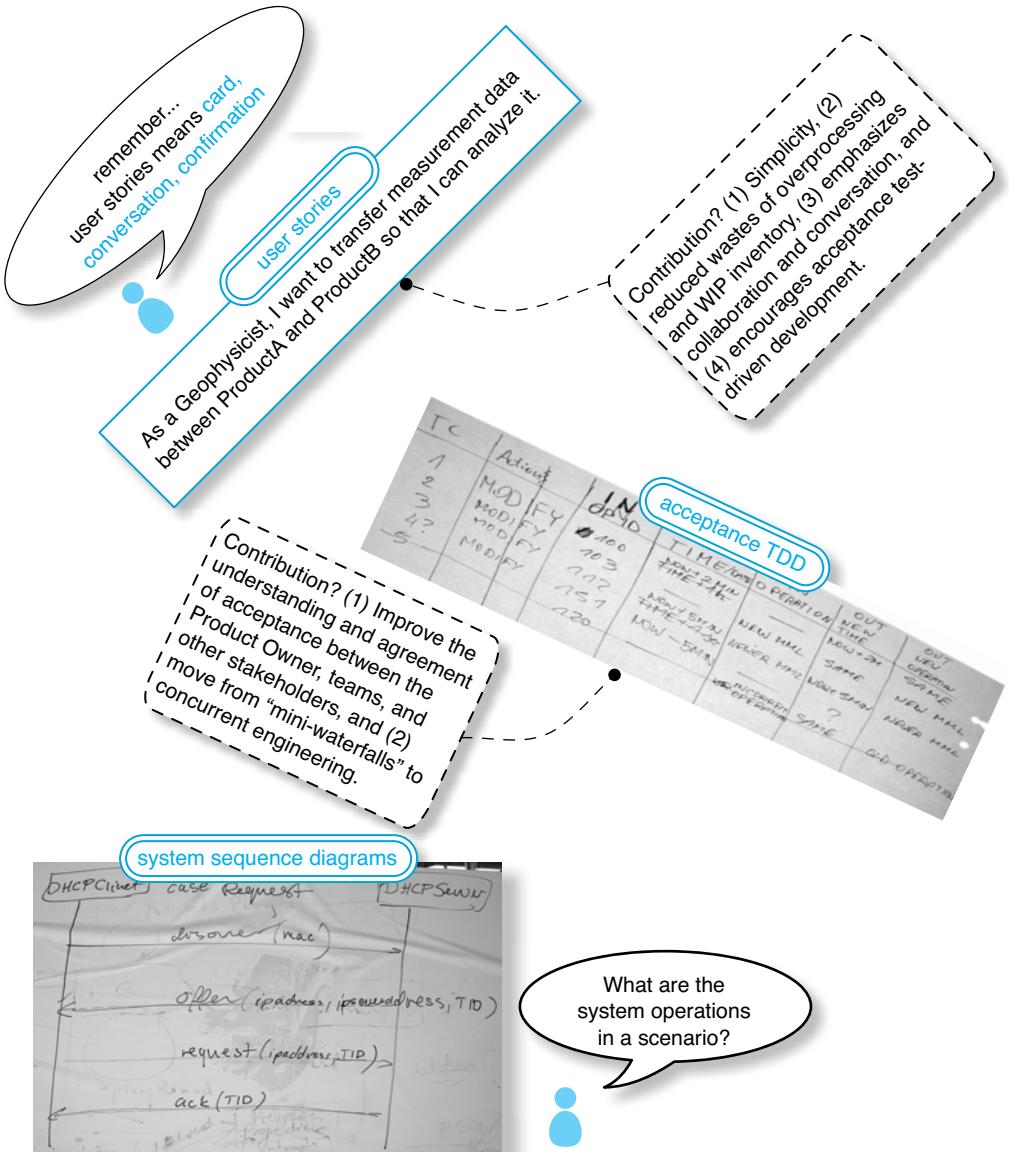
Combine techniques—For example, when we facilitate a detailed requirements workshop, we may first sketch some system-sequence diagrams, followed by an activity diagram, then expand and

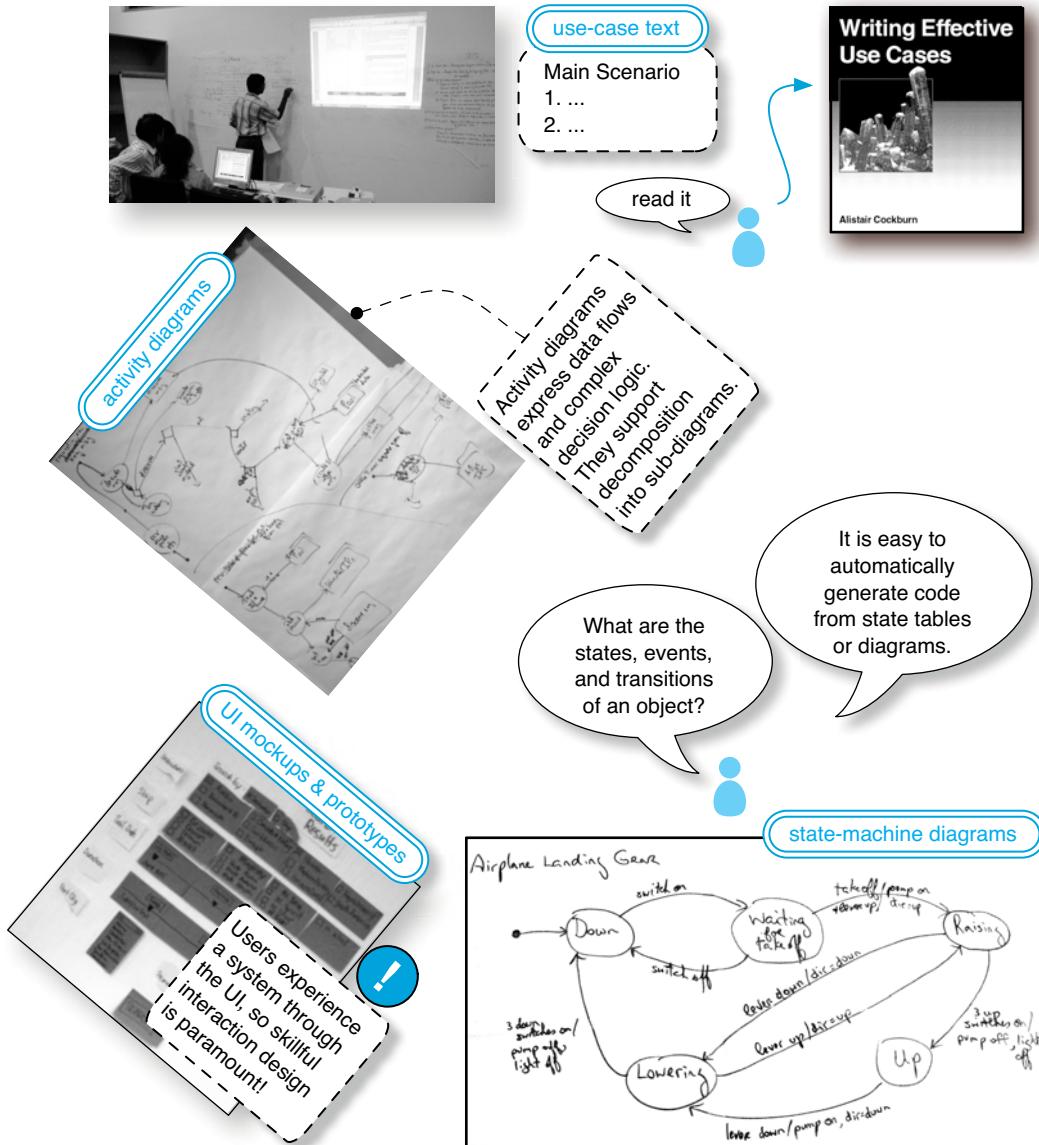
clarify this with use-case text, evolve that into executable acceptance tests, and either before or after create UI mock-ups and prototypes. (This is an example, not a recipe.)

23. User-story experts do not promote this dichotomy; it is from novices.

24. Unless estimating and bidding on a fixed-price, fixed-scope project.

Figure: Experiment with many analysis techniques.





The C-Style User-Story Story

There is a belief that user stories must be written in the *As a...I want...so that...* format, but this is incorrect. Any short form is possible; “Sell a put option” is a story. The *As a...* format was created to solve a particular problem in context, at Connextra (in England, circa 2000). Co-founder John Nolan wrote us, “*The whole team participated in continuous improvement of the format that fitted our process and context. In other places the evolution of story formats has gone in different directions to address particular needs. The ‘As a...’ format is a good place to start for teams but should not be considered an end point.*”

Connextra co-founder Peter Marks first proposed the *I want...so that...* format, with collaboration from the entire Connextra team.^a Peter wrote, “*It took us quite a while to find what was useful to us. The ‘I want to’ clause forced us to focus on the desired consequence of new functionality. It struck the balance between describing a design (too specific) and stating some business requirement (too vague). ‘I want to’ describes something you will be able to do once the feature is implemented that you can’t do now.*” (continued...)

a. Rachel Davies, Tim Mackinnon, and others.

Try...Explore requirements as automated tests

This practice was identified in the previous tip, but deserves highlighting. It reduces the waste of defects and enables concurrent engineering. It applies to both functional and non-functional requirements.

See “*Try...Acceptance test-driven development*” on p. 42.

Try...Prefer PBI titles in C-style user-story format—usually

Requirement items of every size need a short title or name in the Product Backlog. Try this “C-style” user-story format:²⁵ *As a <CustomerRole> I want <Goal> so that <Reason>*.

(continued...) Peter also wrote, “The ‘so that’ clause started off as a way of describing what would happen to information after you entered it, so you might have ‘I want to enter contact details so that they are listed in my address book.’ Quite quickly, this became a business justification for your desired consequence: ‘I want to enter contact details so that I won’t lose them.’ Without this it was very easy to invent things we could build that we didn’t really need. This justification was particularly important as we didn’t have any real customers at the time. The ‘As a’ clause came naturally. As Rachel Davies pointed out to us, it allows you to consider roles and that leads to access rights. However ‘As a’ served another, more fundamental purpose—it allowed us to check that someone would actually use the functionality described.”

Why call this the C-style format? Because of its connection to Connextra, and its widespread popularization by Mike Cohn in *User Stories Applied* [Cohn04].

The innovative Connextra team also created other improvements: *mock objects*, *Gold Cards*, and *heartbeat retrospectives*.

Most requirements can be summarized in this format. For example:

a classic function style (IEEE 830)	use-case-name style	C-style user-story format
The system shall transfer data between productA and productB	Transfer Data	As a Geophysicist, I want to transfer data between ProductA and ProductB so that I can analyze it

Why prefer this format?

- for readers: summarizes who, what, why—provides more context and information
- for readers: consistent format—improves readability
- for writers: when writers think to clarify the role (who) and motivation (why), this can lead to useful learning

25. Why “C-style”? See *The C-Style User-Story Story* box.

Big requirements too!—It is important to notice that gargantuan goals (which will involve thousands of people a long time) can be expressed in this format:

As a Printer Operator, I want color printing so that I can convey more information and influence emotional response to material.

This suggestion does *not* mean to avoid a variety of analysis techniques; it does not mean “avoid use cases.” Even though the PBI is *titled* in user-story format, it may be analyzed and described in detail as a use case, a state machine, and so on.

This format is not always appropriate. For example:

As a Mobile Device User, I want HSDPA support in the network so that I can download faster and have more free time.

Telecommunications people understand the roles and benefits well enough that the sentence above is largely *noise*. It is sufficient if the following *user story* (not in C-style format) is in the backlog:

HSDPA call

TOOLS

Avoid...Requirements management and ALM tools—for N years after agile adoption

The key, first agile value is *individuals and interactions over processes and tools*. Yet an early-adoption question we are often asked is “What tools should we *buy*?” We see attempts to solve the *systemic* problems of requirements with tools—including requirements management tools and application life-cycle management (ALM) tools—rather than addressing the root causes: People, interactions, and organizational design.

This is not a concern with tools; it is a concern with the *quick fix* behavior of believing that systemic problems can be fixed with tools.

Avoid the seductive lure of “tools to solve requirement problems” for at least the first $<N>$ years after starting to adopt agile or lean development, so that *people’s focus* can be where it belongs: on the *system*.

Start with simple tools: cards on walls, spreadsheets, wikis.

After $<N>$ years? Prefer free tools so that the cost of experimenting is low and there are fewer barriers to discarding tools.

Agile expert Ron Jeffries shared a frequently observed result:

I have surveyed a few clients who use [‘agile’ management tool X] and so far have found one individual who actually liked it. [Jeffries09]

Avoid...Old-style, centralized, and hierarchical document tools

Documents are *so* 1980s. We agree that Apple Pages, Microsoft Word, and OpenOffice Writer are great tools for editing a separate text document. But one of the wastes in lean thinking is the waste of *information scatter*. A related one is *delay*. Requirement information is *intensely related*; actually, most development information—status, requirements, design, and tests—is related.

When development information (such as requirement details) is stored in separate documents (on network drives, in document-management tools, or worse...*on Bob’s laptop*), there is...

- more information scatter
- weak, slow navigation between information elements
- difficulty and delay in creating new relationships or links
- delay in accessing information

Contrast the utility and speed of finding and relating information by using the World Wide Web versus a document. Your development project has information needs much more like the Web, *especially* in

developments with many people and sites. Therefore, for lean and agile development, experiment with avoiding

- old-style text documents and document-centric tools
- old document-centric management tools such Lotus Notes and MS SharePoint; these are centralized and hierarchical

Try...“Web 2.0” decentralized, networked tools

A wiki²⁶ is the quintessential example of a “Web 2.0 tool.” Google Wave is another example. Such tools encompass Web 1.0 qualities (webpage-centric, decentralized, a network of hypertext) plus community-oriented content in which many people can easily edit or add webpages, using the browser. To quote Wikipedia, “[Web 2.0 tools facilitate] communication, information sharing, interoperability, and collaboration...” These are qualities aligned with agile values, and especially useful in big, multisite development.

Experiment with Web 2.0 tools (probably a wiki) for requirements and other product information. Similarly, try web-based list-tools for the Product Backlog, such as Google Spreadsheet. For each item, include a URL (such as wiki page link) in the backlog. Item details (acceptance tests, photos of sketches, ...) are accessible through the item’s portal webpage.

Try...Baseline and version-control in your “Web 2.0” tools

Baselining—Some of our clients develop regulated products—such as medical devices—that require more traceability. Then, take (using an automated script) an end-of-day snapshot of the product ‘web’ or wiki site. One client compresses the entire site into one file and saves it daily. Another alternative: Some wikis support tagging all pages within a wiki-name-space with an arbitrary tag.

Version control—It is wonderful that anyone can improve requirement-X wiki page, until... *oops*. Not to worry, many wikis have version control: every version of every page can be recovered. Google

26. Wiki technology was created by Ward Cunningham (also one of the founders of agile development) in 1994.

Spreadsheet also has built-in version control. These have authentication and access control, to identify who changed a page or—not recommended but possible—control who can modify content.²⁷

Avoid...Requirement information in email

Information scatter and delay are heightened when a requirement—or any kind of—clarification is buried in someone’s email. It definitely does not scale. Instead, use a wiki feature in which all emails related to a project or requirement are automatically aggregated, threaded, and displayed on a webpage.

Similarly, if teams use wiki text for all requirement elaboration, use the built-in, blog-like threaded-comment feature on the page for people to discuss or clarify a requirement.

Try...RSS feeds on requirement page changes

With many people in many sites touching or discussing a requirement webpage, a simple way to keep informed is to subscribe to an RSS feed for changes on the page; several Web 2.0 tools support this.

Try...Multiple page labels for a requirement page

Several Web 2.0 tools provide the ability to tag a webpage with multiple labels. For example, a page that contains a digital photo of an activity diagram for an item whose top-level requirement is “PDF 1.7 level-3 printer support” can have the two labels “activity diagram” and “pdf 1.7 level-3.” This is useful for searches and for tool-generated lists based on label, such as all pages labeled “activity diagram.”

CONCLUSION

Reflecting on our observations (while coaching) of requirements work in big systems, a few problem-themes repeatedly dominate

27. This may be important in safety-critical products.

- ❑ handoff from one group doing the analysis (product managers, UI designers, business analysts, architects, ...) to separate development teams
- ❑ ‘fake’ requirements—technical tasks that are mis-labeled as requirements, usually to coordinate work between single-specialist component teams
- ❑ development teams disconnected from the real customers and real requirements
- ❑ a lack of agile modeling skills across all team members

Broadly, many suggestions in this chapter—such as *Avoid...Separate analysis group* and *Try...Write customer-centric requirements*—are aimed squarely at addressing these key problems.

Another major set of suggestions address scaling issues related to a large Product Backlog and large requirements; these include *Try...Group items into requirement areas*, *Try...Prefer cell-like splitting over tree-like splitting*, and the longest tip in the chapter, *Try...Split Product Backlog items*. Why so long? In every large product group we visit, the splitting of big requirements is one of the first questions—people can seldom imagine that they could finish a customer-centric requirement in an iteration. Therefore, we devoted plenty of time to show that it *is* possible and clarify how.

RECOMMENDED READINGS

Basic requirements analysis:

- ❑ *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* by Gojko Adzic was also recommended in the *Test* chapter; it emphasizes acceptance TDD, requirements by examples, and includes a chapter on agile requirements workshops.
- ❑ *Requirements by Collaboration* by Ellen Gottesdiener describes how to organize and facilitate requirement workshops.

- ❑ *Writing Effective Use Cases* by Alistair Cockburn is an excellent book on use cases, and widely considered the de facto standard for this subject.
- ❑ *Patterns for Effective Use Cases* also includes useful tips.
- ❑ A *Use Cases* chapter, based on the Cockburn system, is available in the *Articles* section of www.craiglarman.com.
- ❑ *User Stories Applied*, by Mike Cohn, is a great introduction.

User interface, user experience, and interaction design:

- ❑ In the classic *The Design of Everyday Things* and his more recent *Emotional Design*, Donald Norman emphasizes that human factors need to be front and center.
- ❑ Interaction design is a fast-moving field of publication. Broadly, search for material that emphasizes lightweight modeling, iteration, prototyping, and cross-functional teams. For example, see *Sketching User Experiences* by Bill Buxton.

More on analysis and modeling:

- ❑ *Agile Modeling* by Scott Ambler emphasizes lightweight and collaborative approaches to modeling.
- ❑ *Applying UML and Patterns* demonstrates a variety of modeling techniques, including domain models, activity diagrams, and state-machine models.

This page intentionally left blank

Chapter

- Thinking About Design 282
- Behavior-Oriented Tips 289
- Technically Oriented Tips 317
 - Introduction to Interfaces and Interactions
Tips 323

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

DESIGN & ARCHITECTURE

There are 10 types of people: those who understand binary, and those who do not.
—anonymous

In landscape architecture there is an *evolutionary design* technique using *desire lines*.

Problem: Where to build, and how wide to build, outdoor pathways?

Solution: Wait for a year and observe the paths people naturally walk, and traffic volume. Create permanent paths along these desire lines, as wide as appropriate. Design is *pulled* from demand rather than speculatively pushed.



Although challenging to apply in product design, this is one source of inspiration in lean or agile design—a kind of *emergent design*.¹

There is probably a market for a great book on *Agile Large-Scale Design*; this is not it. This is not a treatise on technical design; it offers a few behavior-oriented tips related to design and large-scale development with agility, with a few noteworthy technically oriented tips—some analogous to desire lines. Some tips reflect lean software principles such as *decide at the last responsible moment*. Some reflect agile principles such as *the most efficient and effective method of conveying information is face-to-face conversation*. And many suggestions reinforce the ninth agile principle: *Continuous attention to technical excellence and good design enhances agility*.

1. No false dichotomies: This example is not meant to suggest avoiding technical excellence or thoughtful design; it suggests design and architecture that is gracefully adaptable in response to learning.

THINKING ABOUT DESIGN

Try...Think ‘gardening’ over ‘architecting’—Create a culture of living, growing design

See “Try...Clean up your neighborhood” on p. 346.

We considered calling this chapter simply *Design*, but decided on *Design & Architecture* because of the extant belief that the software code, design, and architecture are *separate* things, and therefore that ‘architecting’ and programming are separate.²

The word **architecture** has at least two broad implications in common parlance in software development:

- (*noun*) the large-scale static and dynamic themes and patterns
 - there is also *intended architecture* (speculated, wished for) versus *actual architecture*—which may *not* be wished for
- (*verb*) the creation and definition of the intended architecture, as in ‘architecting’ or, “When will you *do* the architecture?”
 - it is performed once near the start, often in documents
 - it overlaps with requirements analysis

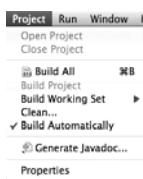
The term was borrowed from building architects. It turns out to be a weak analogy³ with interesting side effects for software development. Buildings are *hard* and so in that domain the act of architecting is only done once before construction—at least, these days—and then the building or architecture is more or less permanently fixed. Note also that the architects are different from the construction workers. But software is not a building, software is *soft*, and *programming is not a construction process*; “software architecture” is merely one imperfect analogy from a large list of metaphors that could be chosen.

-
2. They are separate things when one is creating a physical object such as a hardware device; we refer to software architecture.
 3. The term “software *architecture*” is not a ‘truth’; the name arose haphazardly by some people in a young field looking for analogies. Like all analogies (including ‘gardening’), it has strengths and weaknesses.

What other metaphors apply? In the oft cited paper “What is Software Design?” [Reeves92], the author observes

... the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code

I (Craig here) wrote a book on software analysis, design, modeling, patterns, and architecture [Larman04a]. I mention this not to suggest I’m any good (I have average development skills), but I’m probably not a ‘hacker’ (in the bad sense); I appreciate the art and value of modeling and ‘architecting.’ However, having also worked as a hands-on programmer since the 1970s, I recognize that *diagrams and documents are not the real design* but rather that the *source code is the real design*. To reiterate, “...the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code.”



The source code (in C, C++, ...) is the real blueprint. And near-unique to software, construction or *building* is almost free and instantaneous.⁴ Consequently, many do not see it for what it is: *Building* (construction) is the *compile and link* step. It is no coincidence that in development tools, the menu option to perform compile and link is labeled *Build*.

Scenario: In the early days of ProductX, suppose there were speculative but high-quality design documents for the large-scale elements, idioms, and interactions of the intended architecture, and suppose somehow the real design (the source code) well reflects these intentions. Seven years pass, all the original programmers are no longer programming, and 300 new developers have been hired who are poorly skilled and do not really know or care about the original large-scale design ideas. Imagine they have added 9.5 million lines of code—9.5 MLOC, suppose it is 95 percent of the total code—and it is a mess.

Where is the *real* architecture—good or bad, intentional or accidental? Is it in documents being maintained (or not) by an architecture group, or is it in the ten MLOC of C and C++ within tens of thou-

4. Three-dimensional (3D) printing, in which complex objects are built from a 3D printer, is similar in this respect.

sands of files? Obviously the latter—the source code is the real design and its sum reflects the true large-scale design or architecture. The architecture is what *is*, not what one wishes it to be. The ‘architecture’ in a software system is not necessarily any good or intentional.

First observation—*The sum of all the source code is the **true** design blueprint or software architecture.*

The software design/code improves or degrades day by day, with every line of code added or changed by the developers. The software architecture is not a static thing. Software is like a living thing, more like a plant or garden than a building, and the living design or architecture is *growing* better or worse day by day.

Second observation—*The real software architecture evolves (better or worse) every day of the product, as people do **programming**.*

The analogy to gardening, parks, and plants is salubrious [HT99]. For example, there is the noun and verb *landscape architecture*—it is normal and skillful to consider and ‘architect’ the *big picture* when planning a big garden or park. And yet people do not leave it at that. Because of the visible nature of a park, and because plants grow, it is crystal clear that the *actual* landscape architecture will quickly devolve into a jungle of weeds without constant gardening or pruning by hands-on master gardeners mindful of the park’s original or evolving vision. We have a friend who works as a landscape architect for golf courses. He sees with his own eyes the details of the real, living course while it is being created, walking around it and playing golf—in touch with the reality of what is.

This shift from the metaphor of architecting and building software to *growing* it like a plant has influenced many people reflecting on successful development. For example, Frederick Brooks, in his famous article, *No Silver Bullet*, shares his shift in understanding:

*The building metaphor has outlived its usefulness... If, as I believe, the conceptual structures we construct today are too complicated to be accurately specified in advance, and too complex to be built faultlessly, then we must take a radically different approach... The secret is that **it is grown, not built**... Harlan Mills proposed that any software system should be*

grown by incremental development... Nothing in the past decade has so radically changed my own practice, or its effectiveness... [Brooks87] (emphasis added)

Third observation—*The real living architecture needs to be grown every day through acts of programming by master programmers.*

Fourth observation—*A software architect who is not in touch with the evolving source code of the product is out of touch with reality.*

Fifth observation—*Every programmer is some kind of architect—whether wanted or not. Every act of programming is some kind of architectural act—good or bad, small or large, intended or not.*

What does this have to do with large-scale development and agility?

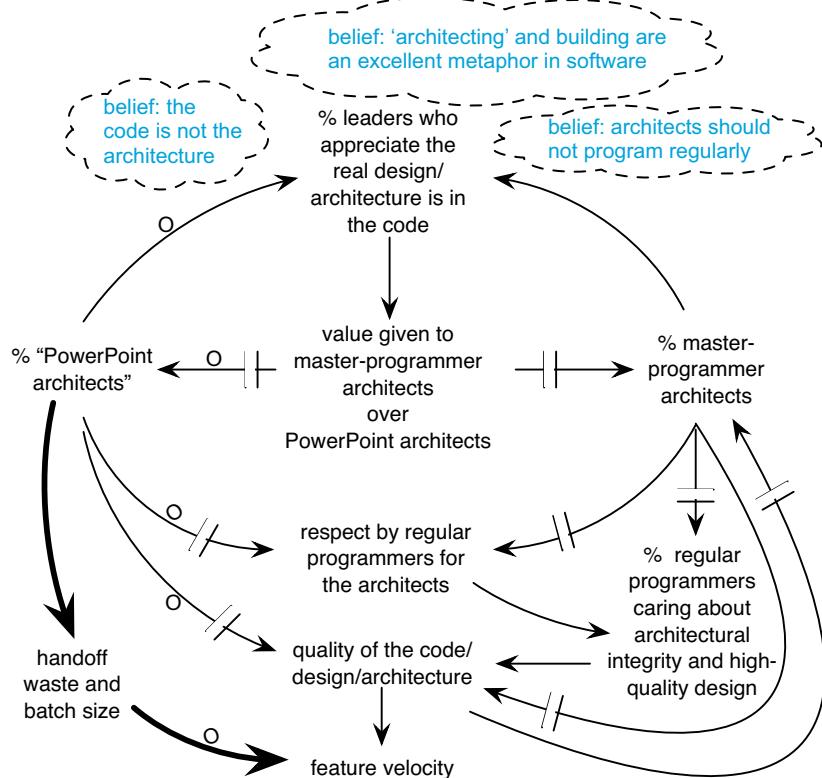
In a small product group with 20 people, people well understand the above, and there is rarely an institutionalized false dichotomy or division between architecting and programming. Also, if there is an official ‘architect,’ then this person is typically a master programmer, close to the code. But in a large product group with 600 people in a colossal enterprise, there is a common mental-model mistake—that design or architecting is definitely separate from the code and act of programming. Consequently, it is not uncommon to find an official architecture and/or systems-engineering group, an institutionalized ‘architecting’ step by them (before programming), and its members are not daily hands-on developers or (at least, no more) world-class code craftsman.

This architecture group (or systems engineering group) generally contains well-intentioned and bright people. But (there had to be a *but* here), in a traditional organization they slowly lose touch with the reality of the source code and become what are called *PowerPoint architects*, *ivory-tower architects*, or *architecture astronauts*—so high up and abstracted from the code (real system) that they are in outer space [Spolsky04].

The repercussions? In a large product group with (1) the mental model that the weak metaphor of *architecting* and building a software system like a building is believed to be a good metaphor; (2) the lack of realization that the true architecture is in the sum of the source code; and, (3) a cadre of architecture astronauts, all this leads

ironically to a degrading architecture over time. Why? Some of the dynamics in play are shown in the system dynamics model in Figure 8.1. Note the several positive feedback loops that can reinforce degradation or improvement over time.

Figure 8.1 causal loop diagram of some dynamics related to the ‘architecting’ metaphor



Also, what happens to the code—the real design—in a group with the following cultural value and message?... *There is the architecture group over there; you regular programmers are not architects.* The programmers naturally feel that the architecture is not their responsibility, and degradation of architectural integrity continues.

If the system dynamic increases the influence of PowerPoint architects, the outcome is that they have decreasing influence over time, since they are not in touch with the reality of the code. Eventually, they lose complete touch and end up writing documents for each other or for business stakeholders. The *real* accidental-architects (the programmers) basically ignore them.

We once had a discussion with a skilled programmer who wrote device drivers for network processors, part of a very large product. He was concerned because the ivory-tower architects—who were located two floors up, literally *in another tower*—had selected a new network processor that would require a complete rewrite of all the drivers—estimated to be at least nine months of programming work. And that did not even account for testing and resolution of unexpected behavior from a new processor family. Having written drivers for many processors, the developer was an expert on the subject, and he agreed that the new processor was better—at least on paper. However, he also knew that upgrading the existing processor to a newer model in the same family would achieve almost the same benefits, and with zero effort to change any drivers. He seriously doubted that the ivory-tower architects were aware of the effort and impact on the software development—none of them had talked with him; nor in fact did they spend time talking with any real hands-on programmers.

Architectural foundation?—“It is important to have the architectural foundation before you implement anything else, otherwise you can’t have an architectural foundation.” This false dichotomy *idea* stems from the building metaphor, as though a software system were made of *concrete* rather than *software*—as though major system elements could not be improved through learning cycles and refactoring. Coincidentally, while we were writing this section, we had a beer at a pub in Oxford, England, with Alistair Cockburn (an agile thought leader) who told us that he and his wife wanted a basement *added* to their existing house. The builders lifted the entire house, dug a basement beneath it, and put the house back on top. It’s amazing what ‘architectural’ foundational changes are possible if one thinks outside the box—and software is a lot softer than a house.

Certainly it is important to have great architecture. It is so important that every act of agile modeling and programming for the life of

the system should be treated as an architectural act. We all agree that good architecture is important; the question is, *what is a skillful way to achieve it?* Most of the tips in this chapter offer suggestions for how to create and maintain a great ‘foundation’ that is not based on the building metaphor or sequential life cycle. All of the following and more are detailed in the next subsections.

- Try...Architectural analysis before architectural design
- Try...Question all early design decisions as final
- Avoid...Conformance to bad or outdated architectural decisions
- Avoid...Architecture astronauts
- Avoid...“Don’t model” advice from extremists
- Try...Design workshops each iteration
- Try...Joint design workshops for broad design issues
- Try...A couple of days to a couple of weeks of design workshops before first iteration
- Try...Incrementally build ‘vertical’ architectural slices of customer-centric features
- Try...Do customer-centric features with major architectural impact first
- Avoid...Architects hand off to ‘coders’
- Try...Tiger team conquers then divides

No false dichotomy: Upfront modeling is fine, documents describing the intended architecture are fine, and so forth. But the architecture, and our learning about it, can *improve*. Speculative software architecture should be *made* concrete and not *of* concrete.

Agile architecture comes from the behavior of agile *architecting*—hands-on master-programmer architects, a culture of excellence in code, an emphasis on pair-programming coaching for high-quality code/design, agile modeling design workshops, test-driven development and refactoring, and other *hands-on-the-code* behaviors.

BEHAVIOR-ORIENTED TIPS

Try...Design workshops with agile modeling

A *requirements* workshop brings together customers and developers in face-to-face facilitated workshops. They are tremendously helpful, not only to better learn user needs but—key point—to create a common understanding among all participants.

See
“*Try...Requirements workshops*” on p. 240.

These same benefits apply to a *design workshop*. In contrast to a requirements workshop it does not include customers, but it does include all members of the feature team—the people with skill in programming, system engineering, architecture, testing, UI design, database design, and so forth.

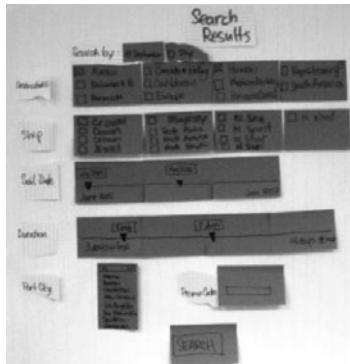
When?—Consider holding design workshops at the start of building each new item (for example, three design workshops for each of three items in an iteration), and just-in-time whenever else the team finds agile modeling at the walls useful.



Figure 8.2 design workshop—feature teams model in large ‘whiteboard’ spaces

Model what?—During a design workshop, feature teams focus on modeling related to their upcoming goals, or to the overall system architecture—or both. All kinds of design modeling occur: low-fidelity UI modeling with sticky notes or in prototyping tools, algorithm modeling with UML activity diagrams, object-oriented software design modeling usually sketched in UML-ish notation, and database modeling likewise.

Figure 8.3 agile modeling applies to UI design as well



This is not a requirements workshop; by the time your teams come together in design workshops, you should more or less understand the requirements under design. Naturally, there are always requirements clarifications or issues raised during a design workshop.



Vast ‘whiteboards’—A design workshop requires *massive* ‘whiteboard’ space. Standard whiteboards are possible but not usually sufficient—and in fact are often an impediment, because modeling is best done on **vast** open wall spaces *without borders*. You will

want to cover virtually all wall space with ‘whiteboard’ material, usually about two meters high.

We have noticed over the years as we facilitate agile design workshops that there is a *linear correlation* between their effectiveness and the amount of whiteboard space.

At office supply stores or sites you can buy “cling sheet” or “sticky sheet” whiteboard-like materials that either cling to the wall by static cling or by adhesive.⁵ You can also buy “whiteboard wallpaper”—an excellent solution for floor-to-ceiling ubiquitous whiteboards. One organization we coached bought cheap bathroom waterproofing plastic wall panels that worked great as whiteboards; they covered the entire room with them. Once these ‘whiteboard’ areas are formed, they can be left up permanently. Observe in Figure 8.2 how the cling-sheet material on the walls is set up.

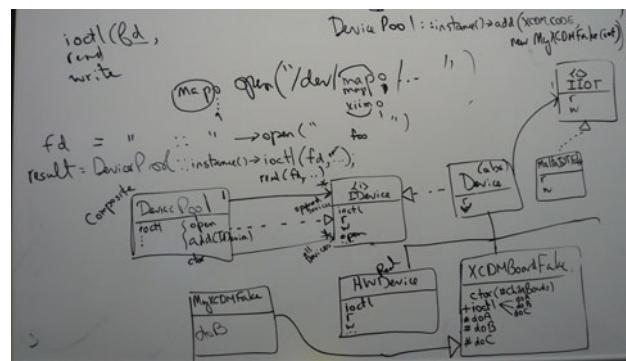


The best modeling tool?—I (Craig here) wrote *Applying UML and Patterns*. As a result, people who know this sometimes ask me what CASE or “model-driven development” (**MDD**) or “model-driven architecture” (**MDA**) UML tool I use. Or, if I’m facilitating a design workshop, they might ask what CASE/MDD/MDA tool to set up. They are usually amused when I answer, “The best modeling tool that I know of is a fresh black marker pen, a group of people, and a giant whiteboard area. *Sketching UML on the wall is great.*”

UML software tools are sometimes useful, and there are situations when we will recommend one. For example, they can be useful to automatically and quickly *reverse engineer* the code base into a set of diagrams that help one see the big picture. But for *forward engineering* or code generation, they can—given today’s technical limitations—inhibit some important goals, explored soon.⁶

-
- 5. For example, the brands *Write-On Cling Sheets* or *Magic Chart*.
 - 6. It is noteworthy that we know several people who used to—but no longer—work for UML CASE/MDD/MDA tool vendors, and none of them use those tools in their current development. It is also noteworthy that programmers at CASE/MDD/MDA tool vendor companies often do not use their own tool to develop their own tool!

Figure 8.4 useful, simple UML on an excellent UML tool—a wall



This collaborative sketching, simple-tool, and decades-old approach falls under the category of what has been **agile modeling** [Ambler02].

Leaving aside the many tips and techniques of agile modeling, why model in a workshop?

model to have a conversation

This is a reiteration of *The First Law of Diagramming* explored in the *Systems Thinking* chapter of the companion book:

The primary value in diagrams is in the discussion while diagramming—we model to have a conversation.

We encourage teams *not* to model together at the walls to *specify*, but to *have a conversation*—to explore and discuss together and come to a shared understanding about designs and requirements, to help develop a shared mental model, and learn together. No doubt some of the object-oriented UML models or UI prototypes on the walls will end up successfully realized in code, but that is a side benefit of taking the time to think, talk through, and sketch ideas together.

Models are not specifications—Any model created before code is just a guess (and a context for a conversation), not the real design, which only exists in the source code. In agile modeling it is rightly viewed that diagram sketches and text are *inspiration, not specification*. The best design documentation (for maintenance purposes) is created *after* code is complete, using the SAD workshop technique described in.

See “Try...Agile SAD with views & technical memos” on p. 310.

All models are ‘wrong,’ and that’s OK—People model to have a conversation, for inspiration and growing understanding, especially shared understanding. It is natural that models are ‘wrong’—that design evolves as people hit the reality of programming and learn.

Wiki photos—Teams often take photos of wall sketches and put them at their product wiki site.

Design workshops and architectural integrity—On a tiny six-person software project, it is possible to get by without structured group modeling workshops. As we scale to larger teams and projects, the value of group modeling to build shared understanding of design ideas is increasingly appreciated. **Architectural integrity** is a key issue in scaling systems; maintaining that integrity really boils down to the design ideas in the minds of programmers—are they converging or diverging? Design workshops help develop converging design ideas and architectural integrity.

Waste reduction, teaching—In lean thinking, there is a focus on improving through reducing the wastes, and lean product development focuses on outlearning the competition. Design workshops support these goals:

- Workshops reduce the wastes of *handoff* and *delay*. Rather than a technical designer or architect creating a design document and sending it to developers,⁷ rather than a person getting feedback on design ideas through indirect document review, in a design workshop these parties come together and communicate and give feedback directly and immediately. This also supports agile principle six—*The most efficient and effec-*

7. It may be useful to create design documents, but to reduce the waste of handoff a skillful means to discuss and understand its ideas is during a design workshop—at the walls.

tive method of conveying information to and within a development team is face-to-face conversation.

- They reduce the waste of information scatter, as people are in close conversation, discussing details together at a whiteboard.
- They reduce the waste of underutilized people, as people learn from each other and thus grow in capability.
- They increase knowledge, both in terms of teaching others and in terms of generating new ideas through the cross-pollination effect of a group of seven people creatively exploring together.
- In a lean organization, managers and seniors are also teachers. Design workshops provide an excellent forum for leaders to coach others in design skills and architectural themes.
- They encourage simple visual management.
- They encourage the lean principles of building consensus and cross-functional integration.

Figure 8.5 halls are excellent places to set up large whiteboard areas, and they intrigue others in the practice of agile modeling as they walk by a team actively engaged “at the walls”



Simple tools, flow, participation—Humans are not built visually and biomechanically to stare at tiny computer screens and move a mouse around. People are built for *cave art*. Try to have a collaborative, creative five-hour design workshop with seven people around a *computer display*. Death-by-meeting. Yet invite those same people to vast ‘whiteboard’ areas, give them marker pens, and good things will happen (especially if they have had some workshop and agile modeling coaching). These simple enjoyable tools—especially the vast

whiteboard space—encourage *creative flow* and *participation*. That's important.

Simple UML—Since humans grasp information well in graphical forms (“bubbles and arrows” rather than just text), we encourage people to become comfortable with some basics of a few UML notations, including activity, class, and communication diagrams. But detailed notation is quite unimportant—model to have a conversation, not to specify.

How long?—Two hours to two days. As with all events in Scrum, timebox the workshop beforehand so teams know the limit.

Multisite? Dispersed teams?—Some hints are offered in the *Multisite* and *Offshore* chapters.

Try...Just-in-Time (JIT) modeling; vary the abstraction level

In addition to larger and longer whole-team design workshops, consider this scenario: Someone (or a pair) is programming and becomes blocked. They need a different perspective. We often see such a pair grab a *small* piece of paper and sketch, but if they were working in a team room and the walls were covered with some kind of vast ‘whiteboard,’ this person, more effectively, could stand up, turn around, invite a colleague, and start sketching and discussing for a few minutes or a few hours. *JIT modeling*.

Notice that this allows people to vary their abstraction level frequently and easily—from code to models to code. A common false dichotomy is that the only time for high-level abstraction-thinking about the system is during a pre-programming phase. Not so. With the practice of agile modeling and a supportive environment, people can flip levels all the time.

Try...Design workshops each iteration

Plan for and hold *at least* one design workshop each iteration, at least near the start—and possibly more for each item undertaken in the iteration. Timeboxed in the range of two hours to two days. The focus will usually be for features of the iteration, though sometimes

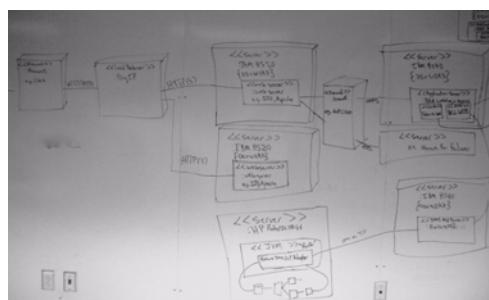
farther-horizon modeling makes sense. A team may hold a small design workshop before the first item goal, then another workshop four days later before the second goal, and so forth.

For very young systems, sometimes the design is so unclear in early iterations that the following is necessary: Suppose it is the last week of the iteration. After the Product Backlog Refinement workshop (it usually occurs mid-iteration and looks forward to future iterations), also hold a design workshop related to the likely goals of the next iteration, or beyond—if large-scale architectural issues need to be explored. This will clarify planning during the Sprint Planning on the first day of the next iteration.

**Try...A couple of days to a couple of weeks of design workshops
before first iteration**

Large products are rarely new products, but when it is an *initial* release cycle (there is no code yet), before the first iteration of the nascent product, hold design workshops with agile modeling for a few days or even a few weeks, depending on scale. This is not a ‘waterfall’ in sequential development; it is a middle way—the purpose is to *have a conversation while sketching* rather than to specify—and there will be ongoing design workshops each iteration to evolve the design according to the feedback of actually coding, integrating, and testing. Evolutionary design is a theme of agile methods—it is skillful to not be attached to original ideas that hold back improvement.

Figure 8.6 model of the physical architecture of a large system with a UML deployment diagram



Wide and shallow with deep dives—In the earliest workshops of a first-release large product, try agile modeling to explore widely across potential major structural elements (both physical and logical) and some of their communication pathways. Logical architectural modeling with UML **package diagrams** can help. Try physical architectural (or **deployment**) modeling with UML **deployment diagrams** to explore the compute nodes, processes deployable to them, and inter-process communication mechanisms (SOAP, MOM, ...). In addition—and opposite this “wide and shallow” advice—it is useful (because dealing with a concrete case clarifies design ideas) to do agile modeling for deep dives into some specific customer features that have non-trivial architectural impact; for example, a feature touching many elements or needing fault tolerance.

Try...Design workshops in the team rooms

It is useful if the walls of each team room are covered in vast ‘whiteboard’ material so that design workshops can be held there. When developers sit to program, they can easily look at the walls for inspiration, or get up for quick modeling conversations, and easily do JIT modeling.



Figure 8.7 the team is surrounded with whiteboards in a team room; people can see models on walls for inspiration while programming, and easily do JIT modeling

Try...Joint design workshops for broader design issues

See “Try...Plan infrastructure items by regular teams” on p. 168.



How to work on cross-team system-level design and architecture issues? How to work on cross-system “product line” design issues?

More broadly, suppose...

- several feature teams work on a common component or framework, since feature teams work cross-component and synchronize at the code level
- one team or product group takes on a *common* shared goal (such as a common feature or infrastructure) that will eventually be used by other groups
- teams are (suboptimally) organized as component teams rather than feature teams, and one customer feature spans several component teams
- representatives from many teams want to get together to explore and decide on system-level architectural design issues

In any of these cases, it is useful if the teams or team representatives—within one product or across products—hold a **joint design workshop** together. This is not a PowerPoint presentation while sitting around a table; this is people (from different teams) at the walls sketching together—agile modeling. They all work together on vast whiteboard spaces, or sub-groups may work on separate walls and visit each other’s work to learn and give feedback. Some teams may send a representative to the other team (wall) during the workshop.

Who attends? This is attended by regular feature team members, technical leaders involved in the hands-on programming—and not by PowerPoint or astronaut architects.

When? Consider a product-level joint design workshop (for system-level ‘architectural’ issues) at least once every few iterations.

After a joint design workshop, participants return to their home teams. Later, during the repeating *single-team* design workshops,

the returning people who attended the *joint* workshop share the decisions made at the joint level, and help the single team express these large-scale architectural decisions in their agile-modeling sketches on the wall—and then in the code through pair-programming coaching. So, there is a transmission of broad-design ideas and decisions from the joint design level to the team design level.

Note the emphasis on a culture of ongoing human infection and mentoring, rather than “documenting the architecture” and handing off The Architecture Document.

A joint design workshop is a *community of practice* activity—in this case, for a design or architecture CoP. So, who organizes regular joint design workshops? It may be a *design CoP facilitator*.

Another reason to have multiple teams in a joint design workshop is described next...

Try...Technical leaders teach at workshops

Problem: lack of general design skill and of specific knowledge (about the architecture, other components, ...). Education is a remedy. In lean, master-engineer managers are also teachers, coaching people in engineering. During design workshops, technical leaders, managers, and programmer-architects help their own ‘home’ team or help other teams. They may spend many hours with one team at the walls, educating to deepen people’s skills and to establish and maintain architectural integrity.



Try...Architects and system engineers are regular (feature) team members

Avoid...System engineers and architects outside of regular feature teams

The prior suggestions related to *joint design workshops* could give the impression that there is a separate architecture group or system engineering group—a misunderstanding. Teams in Scrum are cross-functional and do all the work necessary to deliver customer solutions—and that includes architecture and systems engineering. So, as a product group transitions to agile development, they dissolve the prior separate single-functions groups (such as an architecture group) and the members join regular Scrum feature teams, participating in the hands-on engineering—and, especially, mentoring during design workshops, joint design workshops, pair programming, and agile SAD workshops.

“Member of the team” does not mean a ‘fake’ team member—a person who receives work requests from one or more teams, does ‘their’ tasks separately, and gives ‘their’ completed work back.

Try...Serious attention to user interface (UI) skills and design

Try...UI designers in regular (feature) teams

This tip is not uniquely related to agile development or scaling, but we cannot help but share it, as we see poor UI, interaction and “user experience” design as a universal problem. The *interface* is primarily what people experience and value in (most) software-intensive systems. It is ironic that so much attention is devoted to non-UI architectural issues in a large product group, when the UI architecture—and there *is* a UI architecture, accidental or intended—is Job One from the user perspective.

Avoid...UI designers in a separate UI design group

Consequently, dissolve the separate UI or user experience design group⁸ and merge the experts into full-time membership within cross-functional Scrum teams, so that this key concern is addressed within the teams, and there is constant UI-design coaching from experts to others. If there are no existing UI design experts, invest in educating Scrum team members.

-
- 8. A separate UI group reflects sequential life cycle, promotes big batches of handoff, and inhibits learning by non-specialists.

To reiterate...“member of the team” does not mean a ‘fake’ team member who separately does work requests for various teams.

There is also a scaling issue: On large or multisite products with UIs being created by different teams, there is a risk of low UI integrity or consistency. The standard solutions are to hire usability engineers, educate developers, develop style guides, and so forth. Two practices frequently applied in the agile community, (1) design workshops and (2) communities of practice, can help.

See “Try...Communities of Practice” on p. 207.

Try...Architectural analysis before architectural design (repeat)

Some think of ‘architecting’ as primarily a design activity (such as deciding large-scale elements), but it includes *architectural analysis*, investigation that focuses on forces, requirements, and constraints that strongly influence the technical ‘architecture.’

There are simple tools to organize and guide architectural analysis, including architectural factor tables, quality scenarios, and Plan-guage [BCK98, Larman04a, Gilb05]. With early identification of architectural factors, you can find and prioritize those drivers that truly require early or upfront design decisions. For example, perhaps you decide that choice of programming language is a factor requiring an early decision.

Agile development emphasizes learning and ongoing evolution of the system design; therefore, architectural analysis is not done once, but repeatedly across the iterations—perhaps at the start of repeating *joint design workshops*.

Try...Question all early architectural decisions as final

So, you do some early architectural analysis and decide that the programming language should be chosen early, suppose C++. Encourage everyone to question and challenge all these assumptions and decisions, and to find ways to apply the lean thinking principle of *decide as late as possible* or *defer commitment*. For example, do fast prototyping in Ruby to first learn more. We know of one product that started with C++ for four iterations, and then switched to Java with relatively little effort.

Avoid...Conformance to outdated architectural decisions

All developers have had the experience, “Well, this is pretty awkward, but I’ll fit what I’m doing into the existing approach because it isn’t worth the effort to change things.” *Effort* includes technical effort and the political effort to convince the ivory tower of architects. On little systems, a culture of conformance over challenge-and-improve only creates moderate weakness because the technical debt is not so large...yet. In large systems—or systems that are destined to become large—this technical debt becomes a monstrous boat anchor that anchors the entire product group...forever. It is especially in the early years when the big and growing product is still ‘small’ that you want to encourage lots of challenge to the original architectural decisions and promote deep-change ideas (achieved with refactoring and continuous integration) before the boat anchor starts to drag your product under water.

Try...Hire and strive to retain master-programmer ‘architects’

Avoid...Architecture astronauts (PowerPoint architects)

In small organizations there is little money or time for “architecture astronauts” or “PowerPoint architects” or ivory-tower architects who draw and talk about systems at abstract levels, but cannot code them and are out of touch with the reality of the code. In large product groups, this type does appear. In the book that won the 2005 Jolt Productivity Award (for contribution to software development), the author comments:

These are the people I call Architecture Astronauts. It’s very hard to get them to write code or design programs, because they won’t stop thinking about the architecture... They tend to work for really big companies that can afford to have lots of unproductive people with really advanced degrees that don’t contribute to the bottom line. [Spolsky04]

In lean thinking, there is an emphasis on manager-teachers who are masters of the work and who mentor others, and on working as a hands-on engineer for years. Large product development following lean practices encourages a chief engineer with up-to-date “towering technical competence” as well as business vision. Architects who look down upon “only coding” as something they have evolved beyond have no place in a lean and agile organization.

As discussed in the “gardening over architecting” tip, several dysfunctions arise from the beliefs that the code is not the real design and that the technical leaders do not have to be in touch with the reality of the code.

Plus, always-evolving *programming-designing* practices and tools (test-driven development (TDD), refactoring, ...) should influence the thinking of the technical leadership. For example, really comprehending the subtlety and influence of TDD or refactoring takes long hands-on practice. Without that insight, an ‘architect’ is ignorant of certain forces, dynamics, or action tools in developing systems.

You want master-programmer architects, who are in touch with the code, and who are active developers and mentors—probably through pair programming and design workshops.

This tip does not imply that technical leaders only sit and program; naturally, they decide and communicate major design decisions (perhaps in joint design workshops) and stay in touch with the intersection of market forces and the architecture [Hohmann03].

As explored in the causal model in Figure 8.1, a “PowerPoint architect” is often physically and socially disconnected from the real work and the real workers—inconsistent with the lean Go See principle.

Avoid...“Don’t model” advice from extremists

There are several agile methods; of them all, only Extreme Programming (XP) had an extremely lightweight approach to design modeling before programming. And there are some extremists in the XP community who even discount *any* modeling before programming, although prohibiting modeling was not part of Kent Beck’s original XP message. The “no modeling” idea is a distortion; for example, Ron Jeffries, a key XP proponent who helped coach with Kent Beck on the first XP project, wrote the foreword to Scott Ambler’s *Agile Modeling* book, encouraging the practice for all software developers:

Well, it turns out that Scott recognized something that I did not [that agile modeling is useful]...read this book if you are a software developer who needs modeling skills as part of your development—that is, if you are a software developer. [Jeffries02]

*See “Try...Think
‘gardening’ over
‘architecting’—
Create a culture
of living, growing
design” on p. 282.*

In any event, the extreme advice is not part of lean thinking or Scrum, which is neutral/silent on the amount of modeling; a Scrum team could spend days modeling if they found it useful toward the goal of potentially shippable product each iteration.

On the other hand, agile principle 10 is *Simplicity—the art of maximizing the amount of work not done—is essential.* (This reflects avoiding the lean waste of *overprocessing*). In terms of design, this covers the common advice to avoid overengineering or overmodeling. At the same time, a skillful developer knows that some agile modeling is a powerful tool, remembering the advice of that great architect Tolkien’s Bilbo Baggins in *The Hobbit*:

It will not do to leave a live dragon out of your plans.

The “no design modeling before programming” message is odd advice from an extremist fringe promoting a false dichotomy—that the only two options are just programming or taking a big, upfront, ‘waterfall’-design-specification approach. Ignore it and ignore them.

Especially for large or multisite development, agile modeling in workshops—done by hands-on master programmers—is invaluable. Successful, robust big systems need some forethought regarding structure, elements, communication. For multisite projects there is a risk of low architectural integrity without time for people—across sites—to talk, model, and come to shared understanding of design. In this way, agile modeling supports the ninth agile principle:

9. Continuous attention to technical excellence and good design enhances agility.

Try...Prototypes in a *different* language

A throw-away prototype is an excellent way to learn more about a skillful architectural core, but if we had a dollar for every ‘prototype’ we have seen that mutated into the real system rather than being thrown away, we would be rich. Unfortunately, since the prototype was appropriately done with a quick-and-dirty attitude, there is then a foundation of dirty code/design. Valtech avoided this mistake when developing an oil-field economic modeling product by doing a prototype in Visual Basic, when they knew the client insisted on an

implementation in Java. This is an excellent way to resist the temptation of reinvigorating Frankenstein.

Try...Very early, develop a walking skeleton with tracer code

Old and wise advice is to develop a *walking skeleton* of a system—be it large or small—very early, to learn about an appropriate architecture by programming and testing vertical and horizontal (and every other direction) slices of the system [Cockburn04, Monson-Haefel09]. This is not component-oriented development or layer-oriented development; rather, it is cross-component, cross-layer ‘vertical’ development that evolves a suitable skeleton *in code*. Nor is it prototyping; this is production-quality development in which an architectural foundation is implemented. The creation is a learning process that can include short cycles of architectural analysis, design workshops with agile modeling, and programming and refactoring by master-programmer architects. This tip is related to many subsequent tips.

The programming part of this is essentially what Hunt and Thomas have called *tracer code* development [HT99].

Try...Incrementally build ‘vertical’ architectural slices of customer-centric features

I (Craig here) remember in 1995 at ObjectSpace we were developing a product for a customer. They wanted reporting and management of their business consultants and skills. For byzantine reasons beyond the scope of this story, we had to write our own object-relational (O-R) mapping subsystem in C++. So for the first three iterations (three weeks each, if I recall) the ‘talented’ developers developed the O-R component, focusing on creating that one subsystem first. Elegant *lazy-materializing proxies* with *templated smart pointers*, and other geeky qualities. Then the customer visited for a demo of progress. The team was proud.

The customer was angry.

They had no idea what the point was of our subsystem, and it seemed to them we had spent nine weeks of their money doing nothing.

ing they cared about. They wanted *reporting*, they wanted *consultant information management*. They wanted to pull the plug.

The moral of this story—that we learned the hard way—is a classic agile guideline: *Focus iteration goals on customer-centric features or activities, not on components or subsystems*. Yet as will be seen in the following tip, there is an important design qualification to this.

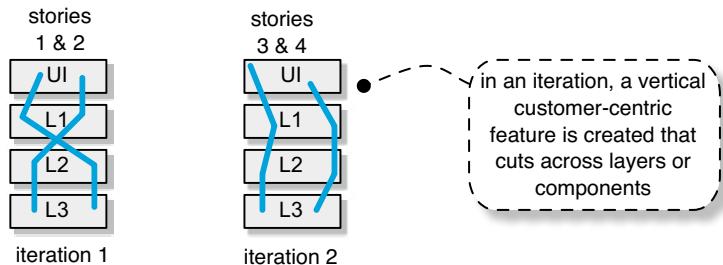
In Scrum, this is what is meant by doing a complete Product Backlog item within the iteration. In the context of XP, this has been called *story-based development*. In the Unified Process (UP), it is called *use-case driven development*.

Although there is no ‘vertical’ in software, given the way software structure diagrams are usually depicted, one could say to implement ‘vertically’ across layers and components (UI, database, ...) to fulfill the one-user story or scenario, evolve and discover the required architecture to support the user feature, and get feedback. Rather than fully developing ‘horizontal’ subsystems divorced from customer features, develop vertically across the layers and components to fulfill the feature, slowing building out horizontally the components as more customer-centric features are tackled.

These ‘vertical’ customer features are developed by the Scrum feature teams.

This can be summarized as *Incrementally build, iteration by iteration, architectural slices that tend to be vertical-cross-layer rather than horizontal-within-layer, driven by architecturally significant customer features*.

Figure 8.8 incrementally add architecturally significant customer-centric features each iteration, across layers or components



This last clause, *driven by architecturally significant customer features*, is discussed in the next tip...

Try...Do customer-centric features with major architectural impact first

This tip is similar to some of the previous tips, but with a stronger emphasis on the *risk-driven* prioritization.

We were coaching a small-ish (100 person) product group in Berlin some years ago. One of the operation and control user stories had a concurrency scaling goal of 80 simultaneous sessions (all with high responsiveness). *Do that early.*

In Boehm's *spiral invariants* model (of good architectural practices in large-scale development, [Boehm00b]) the fifth invariant implies that early iterations aim toward the milestone named life cycle architecture. By this milestone the core architecture elements (both hardware and software) should be programmed and in place—proven through early integration and serious testing of production code rather than speculation or mere prototyping (though prototyping in addition to this goal is also good). This is very sensible advice.

Therefore, choose to do, in early iterations, customer-centric goals (user stories, use cases, activities, ...) while at the same time choosing from a set of features that also have major architectural implications (for example, a feature with hard performance requirements or one that requires touching many components). Choose customer-centric features that by being implemented force people to discover and deal with major architectural issues early on. Not all customer features compel people to identify and resolve the major layers, components, communication themes, or performance issues. Ignore those features in early iterations and instead choose the hard ones.

This tip is an example of **risk-driven development** (a theme of the spiral invariants), in this case addressing two risks:

- business risk—of not aligning with what the client values
- technical risk—of not building a solid architecture

Both have to be addressed in early iterations. No false dichotomy.

Try...Architects clarify by programming spike solutions

In a gargantuan 20 MLOC product with a host of people titled ‘architect,’ it is *so* tempting and safe for these good people to think, “Well, that’s a pretty big and messy system now, and it’s been 23 weeks since I did any programming. It is so much simpler for me to write a document explaining what I want changed in the architecture. Why not? I know what’s going on.” Avoid that temptation; leave your comfort zone. Rather, encourage master-programmer architects to first refine and discover ideas through *programming a spike solution*—exploratory programming that drives a thin vertical spike through components [Beck99]. Follow that, perhaps, by leading a design workshop with agile modeling that conveys the insights to other developers or by documenting the discoveries in an agile documentation workshop.

Avoid...Architects hand off to ‘coders’

In large product development, this handoff is a common problem. Instead, move to a model of master-programmer architects, architects as pair programming mentors, architects as design workshop coaches, and so forth.

Try...Tiger team conquers then divides

For the initial release for a new product or a major rewrite of core architecture, try starting the work with a co-located “tiger team” of great programmer-architects in one team room. Do not start off with a giant group. Keep it to a small tiger team until it hurts; they first program and conquer the key architecturally-significant features.

Repeating a quote (from page 1) on the 1950s large SAGE development, a senior project manager was asked about lessons learned:

He was then asked, “If you had it to do all over again, what would you do differently?” His answer was to “find the ten best people and write the entire thing themselves.” [Horowitz74]

Then, assuming it is starting to hurt (more people are needed; the feature velocity is much too low), explore ways that the tiger team members can divide to help in the formation of multiple teams.

Perhaps half the tiger team members disperse to join new feature teams. This may mean returning home to Bangalore after four months in Boston. Maybe four or five new people join the now-shrunk first team.

Any roaming tigers will play a technical leadership role and educate new team members in new teams on the core ideas, through pair programming and during design workshops. See Figure 8.9.

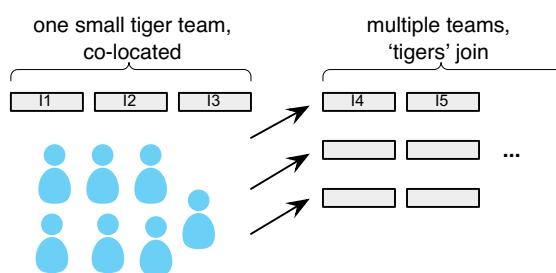


Figure 8.9 start
programming a new
product with one
tiger team

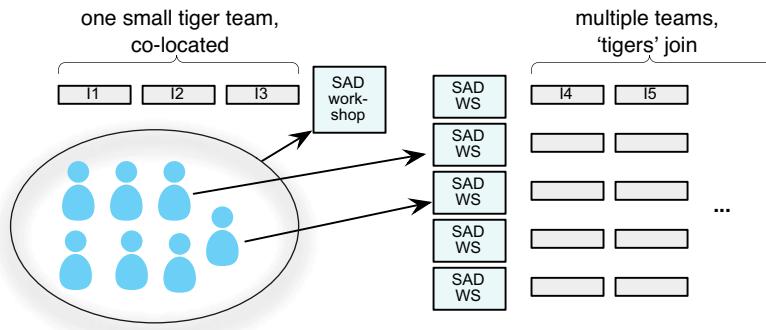
Elssamadisy and Elshamy [EE06] have cleverly coined this practice *Divide After You Conquer*.

We remember a horror story from a product group that did not apply this tip (we started coaching there during release-3): The first release was a disaster. What happened? It was a new product, implemented in C++. They took ‘experienced’ Powerpoint-architect ‘experts’ from a successful legacy product—who had never implemented a C++ or object-oriented (OO) system—to write architecture documents. Then 200 programmers started development from day one, distributed across two sites. Many had never worked with an OO language, so they were given a three-day course in C++. The product was two years late and during release-3 they were still fixing major quality issues and redesigning “the architecture.”

Try...SAD workshops at end of “tiger phase”

A system architecture documentation (SAD) workshop may be useful at the end of the tiger team phase, to provide a learning aid for the new teams that will soon join (Figure 8.10). It may be useful to start the teams with a second SAD workshop, for education. Note that these practices try to reduce the lean wastes of *handoff* and *information scatter*. See subsequent experiments for more on **SAD workshops** and creating an **agile SAD**.

Figure 8.10 a time to get SAD



Try...Agile SAD with views & technical memos

In a large system with 900 developers, inevitably multisite, documentation is helpful for some aspects of the design or architecture, though it can't replace “human infection” (see next tip). How to structure the SAD? What to say? How to record it? Try...

Try...Back up “human infection” with an agile SAD workshop

The lean waste of *document handoff* is a major invariant in software development—handoff just does not work well. Indeed, design documents rarely even get read by hands-on developers. Therefore, a theme of this chapter of tips has been to focus on educating developers through “human infection,” through careful and ongoing face-to-face coaching from manager-teachers who are up-to-date master-engineers, and from other technical leader-teachers. By ongoing—each iteration—participation in design workshops, *system architec-*

ture documentation (SAD) workshops, pair programming, and code reviews, these technical leaders and master teachers ‘infect’ their colleagues with ever-broader and deeper understanding of the system design.



Figure 8.11 an agile SAD workshop, sketching different architectural views; note the many whiteboards

Yet, this agile advice could be mutated into another false dichotomy: human infection *or* documentation. Especially in large systems, try both. Emphasize leaders-are-teachers, while at the same time backing this up with a SAD. Plus, by creating or evolving a SAD in **agile SAD workshops**, the event itself—*involving a medium-sized group, often with representatives from many teams*—becomes another opportunity for teaching and learning.

A SAD workshop is different than an agile *design* workshop. How?

- Design workshops are done *before* doing real design—the source code. The output of the workshop is conversation, learning, and speculation—sketches on walls. For large-scale system speculation, *joint design workshops* apply. Design workshops—for one item or the overall system—are highly creative.
- SAD workshops are done *after* the implementation (for example, shortly *after* a release every six months). A SAD workshop looks backwards at the finished system, and *describes* it. It is not creative, but it is informative—as the participants learn more about the *existing* architecture, and generate ideas for improvement to consider in future joint design workshops.

For the *content* of a SAD, consider the *N+1 view model* and *technical memos*; see the *Documenting Architecture* chapter in [Larman04a].

For *recording* the SAD, take digital photos of the N+1 view sketches on the whiteboards and store in them in a wiki. Also, type technical memos into wikipages.

Try...Technical leaders teach during code reviews

Code reviews are customarily characterized as an event for identifying defects and are seldom done (it seems) by senior product architects. But the event can be used for *education* rather than just defect discovery—especially to help improve design skills and to maintain architectural integrity. A key lean principle is “Go See” or “go see at the place of the source of the problem and fix it there.” Architects or technical leaders who attempt to establish and maintain architectural integrity only through creating presentations or documents will not succeed well. But master-programmer architects, who regularly spend time doing code reviews (the “real place”) with developers, have a chance to educate others in these goals in the most concrete way, while also keeping close to the true design—the code. This tip supports the lean focus of seniors-as-teachers.

Try...Experts participate in ongoing design workshops rather than late approval reviews

Avoid...Approval reviews by experts at the end of a step

What’s wrong with this picture?

1. Person or team creates a speculative design and documents it.
2. Send document to an expert (usually, an ‘architect’) for review and approval.
3. People wait for approval or amendments.

To start with, this increases the lean wastes of delay, handoff, and information scatter. There is also a lost opportunity for coaching and education: If the expert who received the document for review and approval is critical to ensure a good design, *they should be at the early agile design workshops* with the team, so that the *original* design is better, and so that they can *teach* the team to improve their speculative designs during original creation.

An external approver also forces an external process on the team; they are no longer fully in control of their work practices and improvement experiments.

At Nokia (where Bas used to work), they used to apply a traditional review/approval process: a document was inspected and approved (or not) after it was written. As common with handoff waste, there

was a delay until it was reviewed, and feedback was indirect. Plus, if corrections were required, the cycles repeated.

To improve, they introduced a workshop technique called *RaPiD7* [Kylmäkoski03]. The document was written and approved during *one workshop with all the relevant stakeholders there* so that there was no need for a separate inspection/approval cycle, and also to increase learning.

This experiment is not about stopping reviews or feedback; it is about changing the ways of work so that reviewing is a positive experience: value-adding, fast, and educational—rather than the traditional negative experience of delayed-approval processes.

Try...Design/architecture community of practice

Communities of practice (CoP) are an organizational mechanism to create virtual groups for related concerns. The technical leaders or programmer-architects who are responsible for knowing and teaching the architectural vision are members of feature teams. If these technical leaders are scattered onto various teams, they have a need to regularly get together for many reasons, and to have a shared information space. They can form a CoP and share a CoP wiki space.

See “*Try...Communities of Practice*” on p. 207.

Try...Show-and-tell during workshops

If related teams participate in a common design workshop, it is useful both for feedback and education for teams to visit other teams’ walls once or more, to hold “show and tell” sessions. This is also useful if one team (of seven people) decides to split into two sub-groups during the workshop and model different features in parallel. Group One is invited to the Group-Two wall to see and learn the design ideas, and help evolve them. And, vice versa.

Try...Component guardians for architectural integrity when shared code ownership

"Try...Transition from component to feature teams gradually" section on page 391

This tip was covered in the *Shared Responsibility for Design* section of the *Feature Teams* chapter in the companion; it examines several ways to support architectural integrity when there are feature teams and collective code ownership (rather than component teams).

Successfully moving from solo to shared code ownership supported by agile practices doesn't happen overnight. The practice of **component guardians** can help. Super-fragile components (for which there is concern⁹) have a component guardian whose role is to teach others about the component ensures that the changes in it are skillful, and help remove the fragility. She is *not the owner* of the component; changes are made by feature team members. A novice person (or team) to the component asks the component guardian to teach him and help make changes, probably in design workshops and through pair programming. The guardian can also code-review all changes using a ‘diff’ tool that automatically sends her e-mail of changes. This role is somewhat similar to the **committer** role in open source development, but with the key distinction of not blocking commits from others; blocking would create massive bottlenecks and delay.¹⁰ They are teachers and component-improvers, not ‘gates.’ Component guardians are another example of the lean practices of regular mentoring from seniors and of increasing learning.

Try...Component mailing lists

Another technique to support shared code ownership is a mailing list (or other channel) for each delicate component. People working often on a component discuss refactoring, structure, bugs, code reviews, announce training, and so forth. Of course, anyone can join

-
- 9. A typical reason for concern about delicate components is that the code is not clean, well refactored, and surrounded by many unit tests. The solution is to clean it up (“Stop and Fix”), after which a component guardian may not be necessary.
 - 10. But the roles are not identical. Guardians (or ‘stewards’) do more teaching and pair programming, and allow commits at any time. Committers also teach, but less so, and control the commit of code.

or leave a list according to need; any component guardians are long-term members.

Try...Internal open source with teachers—for tools too

Agile development encourages shared code ownership. And feature teams imply working on *all* necessary code for a feature. In this sense, agile development is similar to an internal open-source model of development, but with the difference of even more collective code ownership and no committer ‘gates’ that create delay; rather, component guardians—if needed—help without blocking. As an agile coach, “internal open source, with some guardian-teachers” can be a useful way to explain the idea of collective code ownership, because most people know that various open-source models can work well.

See “Try...Plan infrastructure items by regular teams” on p. 168.

Extend this to internal tools, not only shared components. Rather than “the team in Poland maintains our test tool,” experiment with an internal (or even public) shared code or open source model. Good developers master and evolve their tools; this model promotes that.

Try...Configurable design for customization

Several of our clients have dug themselves into a rather difficult hole by creating a separate branch for each customization of their product for different clients. Those who have experienced this—especially in very large systems—know all-too-well that this increasingly becomes a configuration, maintenance, and testing pain as the years and number of branches grow.

Avoid...Branches for customization

Rather than branches, try configurable designs (for example, with meta-data or some pluggable architecture) that activates/includes (or not) specific components or features.

See “Avoid...Branching” on p. 358.

Avoid...Create ‘designs’ and then send them for offshore implementation

*See
“Try...Experts
coach / review
rather than dic-
tate design” on
p. 474.*

We sometimes visit organizations that claim they no longer “do the waterfall” and yet have a requirements group, a design team, an implementation group, and a testing department—the waterfall expressed in their organizational structure, filled with the waste of handoff and silo mentality. Some groups starting to offshore work to India or China reintroduce and aggravate these problems by, for example, having a group in Europe do detailed UML diagrams of a speculative design that is then sent to a group of programmers in India to code. This is a familiar variation of waterfall mentality; avoid it. It is simply a mini-waterfall in short iteration cycles.

Also: See “Avoid...Architects hand off to ‘coders’” on p. 308.

Try...Architectural and design patterns

Detailed architectural design for large systems is beyond the scope of this chapter, which emphasizes process-oriented design tips. But there is a wealth of well-written robust solutions in the **design pattern** community to help create an agile architecture. Get the books, learn and apply them (see Recommended Readings).

As a theme, patterns provide a protection at some **variation point** in the architecture, through indirection, meta-data, interfaces and polymorphism, and more. These techniques reduce dependencies and enable more, and faster, concurrent development in large products with many teams. Creating more knowledge faster and delivering value quicker are key goals in lean thinking.

The ninth agile principle emphasizes good design: *Continuous attention to technical excellence and good design enhances agility.*

Try...Promote a shared pattern vocabulary

If technical leaders consistently develop and communicate (both in words and how they name software components) with well-known patterns, they help establish shared-design understanding and perhaps more architectural integrity. This occurs in part through creat-

ing a shared *vocabulary* among developers. Patterns have official published names, such as *Layers*, *MVC*, *MVP*, *Strategy*, *Broker*, *Service Locator*, and so forth. These proper names can be used consistently in documentation, speech, and code—for example, an interface named *RoutingStrategy*. Although the prime value of patterns is reusing good design ideas, they can also establish a common vocabulary for your system's design. When scaling to a system with 300 developers in many sites, that helps.

This tip seems obvious, but some technical leaders are not sensitive to the positive influence they could have as *teachers*. Creating *shared vocabulary* is a tool that skilled educators apply.

Try...Test on the old hardware as soon as possible

Usually, large embedded products have been around for some time and there is an existing hardware platform. A new hardware revision may be underway and will require unique integration testing, but it is not necessary to delay testing until the new hardware is ready. Integrate and test on existing platforms as soon as possible. If new software features being written depend on new hardware features, use data-driven configuration or stubs to disable or fake those elements when testing on older platforms.

TECHNICALLY ORIENTED TIPS

Over the time that we have worked with large products, usually embedded systems, we have built up a list of common tips that could have reduced some of the pain and suffering we see and our clients feel. This section lists a few of these tips. An entire book could easily be written on this subject...

Try...HTML-ize and hyperlink your entire source code, daily

With a small system one can navigate rapidly through all the source code simply loaded within your development tool. When there are 36,839 files and 15 MLOC, navigation is not easy. Use a free tool such as Doxygen (www.doxygen.org) to transform your source code

into a set of HTML pages, in which all source code elements (functions, ...) are hyperlinked. Doxygen (and similar tools) will also generate diagrams that reflect larger structures and groupings in your code base. Regenerate the pages daily. This is immensely useful for understanding and evolving a massive code base.

Try...Lots of stubs, plus dependency injection

Create **stubs**¹¹—or ‘fake’ code alternatives—for many things: classes, interfaces to other components, hardware, and so forth. Stubs are usually created with an alternative interface implementation or by subclassing the ‘real’ class in object-oriented designs, or with function pointers or alternate implementation files on a varying link path in C-based designs [Feathers04]; for example:

```
interface PrinterMotor {
    void start();
    ...
}

class CanonPrinterMotor implements PrinterMotor {
    ...
}

class PrinterMotorStub implements PrinterMotor {
    ...
}
```

If there is no interface (and even if there is), stubs can be created through subclassing and overriding relevant methods:

```
class CanonPrinterMotor {
    ...
}

class PrinterMotorStub extends CanonPrinterMotor {
    ...
}
```

Further, provide a “back door” in many classes that makes it easy to inject a dependency to an alternative stub rather than the real object; for example, with **constructor injection** [Fowler04].

11. Some incorrectly use the term **mock** when what is meant is a stub or, more broadly, a **test double**. Martin Fowler has addressed this in his online article *Mocks Aren't Stubs* [Fowler07]. In practice, stubs are far more common than mocks [Meszaros07].

```

class LaserPrinter {
    private PrinterMotor motor = new CanonPrinterMotor(); // default
    ...
    public Printer( PrinterMotor alternativeMotor ) {
        motor = alternativeMotor;
    }
}

```

The combination of many stubs with many back doors for **dependency injection** opens up tremendous advantages: increased concurrent engineering, early integration with stubs when the real components are not available, testing with stubs, stubs that provide fast and well-known demo data. In the context of large product development, massive use of stubs is a key technique to work in parallel and go faster, reducing the lean waste of waiting.

Avoid...Using stubs to delay integration

Wonderful! Now that everyone has stubs you can delay integrating all the code for months or even years. Don't even think about.

Try...Test-driven development for a better architecture

TDD can help improve the architecture of a system. How?

When we are coaching, a frequent request is help for dealing with our client's "inflexible architecture." This most often boils down to problems in high coupling between components—a common problem in legacy code written without TDD because the original developer did not try to test the component in isolation.

On the other hand, when a developer creates a new component (such as a class) with TDD, or refactors a legacy component to be unit-testable, they must break the dependencies of that component so that it is *testable in isolation*. That requires designing (or refactoring) for dependency injection and increased use of mechanisms for flexibility: interfaces, polymorphism, design patterns, dependency injection frameworks, function pointers, and more.

In this way, TDD encourages lower coupling and simple, flexible configuration—qualities of a good architecture.

Try...Dependency injection framework

Dependency injection and easy, flexible configuration are desirable qualities for an agile architecture; they make it easier to (1) test components, (2) quickly develop without waiting for completion of third-party components, and (3) evolve in response to change.

There are several frameworks for dependency injection and configuration, including Spring (for Java) and Spring.NET. Although less well known, frameworks also exist for C++.

Try...Use an OS abstraction layer

We work with two clients of similar large multi-MLOC embedded products. Client-A created a homegrown operating system (OS) and wrote the higher layers directly coupled to it. Client-B created an OS abstraction layer on top of their original OS (VxWorks)—a level of indirection for protection at that variation point. At some point, they both decided to move to a real-time Linux OS. Client-B finished the port in a couple of months; after some years, Client-A is still exploring. *Agility through low coupling.*

This tip is automatically satisfied if you are using Java or a similar platform. However, most of our embedded-product clients are using C and C++. In this case, try one of the existing open-source OS abstraction layers, such as Boost or the Apache Runtime Library.

Try...Create a low-level hardware abstraction layer (HAL) API

As an example in the Unix-like world, calling device drivers and thus controlling hardware is often realized through a low-level system call to the *ioctl* (“I/O control”) function:

```
int returnCode = ioctl( 12, 17, printerStruct );
```

As we are sure you can tell, someone is asking a printer to eject a page!

Some large product groups write their systems with many *ioctl* calls throughout their code (or equivalent), directly coupling to the low-level hardware control mechanism. This introduces a variety of

problems: obscurity, mixing levels of abstraction, old-fashioned error handling, and more.

Start to improve the design by introducing a thin HAL API layer on top of this lowest level, with well-named and stateless functions that express intent, and use modern exception handling. For example:

```
void ejectPage( printerStruct );
```

Or a similar low-level API class wrapper.

```
class PrinterAPI {
public:
    static void ejectPage( printerStruct ); ...
```

Try...Create a mid-level object-oriented HAL

Create a mid-level object-oriented HAL that calls the low-level HAL API, provides abstractions, may be stateful, exploits polymorphism, and easily allows object-oriented stubs and other dependencies to be injected. For example:

```
interface PrinterMotor {
    void start();
    void ejectPage();
}

class CanonPrinterMotor implements PrinterMotor {
    // public methods that call the low-level HAL API
    // private state
    ...
}
```

Try...Create simulation layers for hardware, etc.

Most of the large product groups we have served with are creating embedded systems: military radios, set-top digital TV boxes, network elements, printers, mobile phones, operating systems. A design tip that makes a significant improvement toward agility is to invest in creating a simulation layer of the hardware (or some part of it) or any software component that we need to integrate with but that is not available to us. Or, we want to simulate the hardware/software component because integrating with the real component slows us down, for example, having to download software onto a real printer every time we want to test something. Simulation layers—an expan-

sion of the concept of **stubs**—support the lean practice of concurrent engineering and reduce the waste of *waiting*.

Most of the embedded systems groups we have worked with have experimented with simulation layers or fakes in the past, but it is usually of the form, “Well, I think Jill built something three years ago, but she’s gone now. I’m not sure where the source code is.” Management, unaware of the many degrees of freedom that having useful simulation layers provides, are often unwilling to meaningfully fund the effort. It’s worth it.

A simulation layer does not have to be terribly complex. We have used and seen several approaches to lighten the effort:

- When there are existing hardware or software components to be simulated, create a *record-playback* solution that captures signals or output from the component. In the simulation layer, play these back as appropriate.
- Most hardware can be modeled with a finite state machine (FSM). Try open-source FSM tools that automatically generate state machine code from state tables.

A simulation layer can be realized through an alternative implementation of the low-level or mid-level object-oriented hardware abstraction layer discussed previously.

Voltaire noted, “Le mieux est l’ennemi du bien.” (The best is the enemy of the good). Some groups block themselves from building a simulation layer because they think in terms of a great or perfect simulation, or discussions devolve into “what about that special case...” Start simple, don’t delay.

Try...More FPGAs and fewer ASICs

In lean product development one tries to outlearn the competition. Themes include (1) trying to generate more and faster knowledge and feedback and (2) creating more alternative designs in parallel.

But some clients we work with focus their early hardware efforts only on ASIC development and along one design path. Slower development and lower feedback.

In contrast, FPGAs are an excellent alternative to quickly explore more alternatives, get earlier prototypes to software people, and deliver more quickly.

Although FPGAs can and should be used for specialized logic, it is now also possible to define more general ‘soft’ microcontrollers embedded within an FPGA; for example, using Xilinx PicoBlaze or alternatives. In addition to the chip-internal advantages—microcontrollers provide more efficient use of FPGA resources for some tasks—a microcontroller provides a higher-level abstraction for software that interacts with the FPGA. Software developers can program to a higher-level API provided by the FPGA, and this API may also remain (relatively) stable across new generations of chips.

Introduction to Interfaces and Interactions Tips

Defining and evolving interfaces between components and inter-component interaction are major issues in large-systems development. In fact, what Grady Booch¹² has called “designing at the seams” [Booch96] is arguably the dominant architectural issue in big applications. Note also that the pain of ‘integration’ in multisite or super-large products is a reflection of *interaction*. When you are working with a 15 MLOC behemoth composed of 234 major components, each containing on average 64 KLOC, it is the *interactions and interfaces* that tend to dominate day-to-day overarching architectural concerns, not the design of any one module—or even what modules are present.

Interfaces—In this section, the notion of ‘interface’ includes

- interface as used in Java or C# (local or remote)
- operation signature (function name and parameters)
- web service interface (for example, with WSDL)
- and the like

12. Software is a fast-changing field; thought leaders quickly become lost to a new generation. Do not miss studying the writings of Grady Booch, an OOD pioneer.

Large systems are usually old; lots of C code is common, and the ‘interface’ to another component may be simply a function signature, such as `debit(int, float)`. Another context for these tips is that in a 250-person product group, the client-programmer using a published API may be different than the service-programmer who implemented it years before.

Avoid...Big upfront interface design

An old—and unnecessary—strategy for the interface problem was “*Before programming, define and freeze the interfaces between major components. Then, use a change-control process when interfaces need to evolve.*” This is a decide-early *push model* of design; problems associated with it include

- delayed definition—owing to complexity and the many people involved
- lack of usage-based feedback
- incorrect interfaces (from lack of realistic feedback)
- slow change process
- extra conversion or adaptation code on both sides of an interface to deal with inevitable evolution when constrained by a frozen interface

There are workable alternatives to this unnecessary idea. The following tips offer lean thinking decide-as-late-as-possible alternatives.

Try...Start with some weakly-typed interfaces, then strengthen

Here, a **weakly typed interface** means to invoke operations of another component by using a simple *perform(Map)* method:

```
Map results = componentB.perform( request );
```

where *componentB* is some big foreign component and *request* is an instance of a *Map* of key-value pairs, perhaps of type *String*; for example,

```
Map request = {"opName" = "debit", "accountNum" = "1234",
               "amount" = "10.00");
```

The contents of the *request Map*, especially the values of the key-value pairs, may be more complex objects than simple *Strings*. The example is simplified for exposition.

The *perform* operation is implemented to analyze the *request Map*, and invoke the appropriate action based on the value of *opName*, for example, a *debit* action if the value is “debit”.

Note also that the return type is a *Map* object—an arbitrary collection of key-value pairs. With this, unanticipated return values (from none to anything) are possible.

This is in contrast to a **strongly typed interface** such as

```
interface Account {
    void debit(int accountNum, Money amount);
    void credit(int accountNum, Money amount);
}
```

With weakly typed interfaces, the evolving details of the requests or operations—the operation names and parameters—are encapsulated within the *request Map*, and results are likewise encapsulated in a *Map*. If the client-programmer sees the need to add another parameter or a new operation, she is not delayed (1) by the steps required to change a strongly typed interface, (2) by the coordination between her and other programmers, and (3) by the code itself.

For example, the programmer (in the role of a client to another component) discovers that a *currency* parameter is needed and changes the content of the *Map*:

```
Map operation = {"opName" = "debit", "accountNum" = "1234",
                 "amount" = "10.00", "currency" = "euro");
```

Of course, her changes will not immediately work in the service component—she still needs to change its code. But there are advantages:

- ❑ First and most important, before implementation the minimal ‘interface’ design effort is simply to add support for a *perform(Map)* operation on all components—fast, straightforward, flexible, supportive of change and learning, and no long, arduous upfront design effort to identify and freeze all interfaces.

- Changes do not break existing code; no new compilation errors,
- The programmer is not delayed in making a change,
- Others are not impacted or delayed by the programmer's change.

The discovery and improvement of operations through weakly typed interfaces is a simple, light process.

Middle way—Of course, you are not limited to—and we are not recommending—only the ultra-simple step of just adding a *perform* operation and ignoring further early interface modeling and design. It is perfectly appropriate to speculate likely operations (such as ‘debit’ and ‘credit’ and their parameters) and implement support for them through strongly typed interfaces and also through the *perform* interface. *Weakly typed interfaces simply give us another degree of freedom* to go faster and to increase agility.

Strengthen them—*Strongly* typed interfaces have advantages, including performance, clarity, compile-time type checking, refactoring, and automated code generation. So, people start with weakly typed interfaces when it seems useful. Then, after the operations of a component have stabilized through an evolutionary discovery process (that could take weeks or months), they strengthen them—replacing the flexible-but-obscure *perform* calls with strongly typed calls. The *perform(Map)* interface is always kept for future discovery steps, but stabilized operations are strengthened.

Conclusion—This tip is an analog to *desire lines* mentioned at the start of this chapter; you discover the paths in the ground through usage and then strengthen them. It illustrates the lean principle of *decide as late as possible* and supports learning and evolution.

Try...Simplify interface change coordination with feature teams

As explained in the *Feature Team* chapter in the companion book, a feature team is cross-component and changes all the code *across all components* necessary to complete a customer-centric feature. This reduces coordination problems related to interfaces because the same person or team works on both the calling and called side of the

interface. In contrast, separate component teams increase the complexity of interface coordination.

Avoid...Freezing interfaces

There are times when a published API truly needs to be frozen. But challenge these decisions, keep things as unfrozen as possible, and experiment with techniques to support evolution of interfaces. Some techniques are suggested here and others in the *Recommended Readings*.

Try...Wrap calls to remote components with proxies or adapters

Remote components—called via JMI, RPC, SOAP, message-oriented middleware (MOM), or a socket—are a *guaranteed* point at which people will want to inject a stub to allow testing in isolation, no longer talking to the remote element. Further, it is common that the remote communication mechanism (such as RPC versus MOM) will change.

Therefore, you want protection at this **variation point** in the architecture by always wrapping the calls to other remote components with objects and polymorphism, using the Proxy or Adapter design patterns [GHJV94].

Try...Start with indirect interaction between major components, then replace as needed

Large systems are typically composed of hundreds of major components, and these may be local or remote to each other. We see common problems related to interaction between *major* components (such as subsystems) in big systems:

- ❑ dependency on knowing what major component is the receiver of a message or operation call
- ❑ dependency on knowing the communication mechanism, such as a direct function call, RPC, SOAP over HTTP, and so forth
- ❑ complex and repetitive communication failure handing

- inability to use pluggable features/components because of high-coupling problems

The following tip may help...

The computer scientist David Wheeler was famously quoted as saying, “Any problem in computer science can be solved with another layer of indirection.”

A resolution to the above issues is to use an indirect communication mechanism between *major large* components (such as subsystems), in contrast to something direct such as a Java RMI or SOAP call. This “indirect interaction” is deeper than just adding an adapter or proxy between components; it means using some form of indirect messaging system.

There are several options for indirect messaging between major large components. One robust choice is message-oriented middleware (MOM), such as JMS and MSMQ. Rich with options, supportive of pluggable architectures, MOM is worth a close look. Home-grown or open source lighter-weight “message bus” MOM solutions are another option. Doing inter-component communication with MOM provides a degree of freedom that enables lower coupling and more pluggable architectures. MOM solutions also offer built-in communication fault-tolerance and recovery features.

Actually, there was a second sentence in Wheeler’s quote that is less known; here is the whole thing:

Any problem in computer science can be solved with another layer of indirection. But that usually creates another problem.

Sometimes, “another problem” is a *performance impact*.

A potential MOM disadvantage is a performance drop. In this case, as with the weakly typed interfaces tip, you can start with a MOM solution to discover the “desire lines” of communication while ignoring the performance degradation. Then, as communication pathways stabilize and you discover performance hot spots, you replace slower MOM interactions with faster mechanisms such as the Java RMI. This is another example of pull design. MOM remains the default mechanism unless it is not performant for a case.

If this tip is combined with the tip to always use proxy or adapter objects for remote-component communication, then when the backend mechanism is changed from MOM to RMI, the internal code is not affected—one simply needs to inject an alternative adapter.

CONCLUSION

Buildings are *hard* and static. Software is *soft* and dynamic. So, ‘architecture’ is far from an ideal metaphor for creating software; it can even promote the misunderstanding that there is some design other than the source code, and that the design is essentially frozen.

But the software design is continually evolving and *emerging* with every modification to the code by every programmer. The key question is: Will it emerge as a beautiful well-tended garden, or as a jungle of weeds?

The tips in this chapter encourage high-quality emergent design by a development culture of *gardening* and shorter and richer feedback cycles, rather than ‘architecting.’ And that requires great gardeners: master-programmer architects who actively *code the architecture* and who continually coach other programmers during pair programming and agile modeling design workshops.

For sustainable large-scale agile systems, it is vital for people to master design techniques for flexibility: design patterns, dependency injection, test-driven development, refactoring, and more. But without a culture of coaching-while-coding by technical leaders, these techniques will not be sticky or pervasive.

We suggest no false dichotomy between coding and modeling; the latter is valuable—especially in large-scale systems. In addition to a focus on code, *agile modeling design workshops* are a great, light-weight technique to quickly explore speculative designs and learn together. Perhaps the key ingredient is *massive ‘whiteboard’ spaces*, therefore, take over the walls!

RECOMMENDED READINGS

This section reiterates several texts recommended in the *Legacy Code* chapter; this is to be expected because agile design recognizes that the real software architecture is in the code.

- The site www.codingthearchitecture.com emphasizes the need for architects to be master hands-on active developers.
- Many of our clients have vast quantities of messy legacy code that is difficult to test in isolation and difficult to evolve. Michael Feather's *Working Effectively with Legacy Code* is an important antidote, covering the techniques that allow developers to start designing a more agile architecture within their existing code base.
- A key element of technical agility is design patterns. Consider these texts: *Design Patterns*, *Pattern-Oriented Software Architecture* (five volumes), *Applying UML and Patterns*, and *Pattern Languages of Program Design* (five volumes).
- Two books by Bob Martin encourage a more agile architecture: *Agile Development, Principles, Patterns and Practices* and *Clean Code: A Handbook of Agile Craftsmanship*.
- Two more useful quality-code-oriented books include *Code Complete* by Steve McConnell and *Implementation Patterns* by Kent Beck.
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce reinforces a culture of *growing* rather than specifying “the architecture.”
- *Domain-Driven Design* by Eric Evans encourages thoughtful iterative design, shared understanding, and a domain model that must be well-expressed in the code.
- The paper *Agile Product Development* [TR98] explores the business value of product development and design agility, and how development flexibility can be quantified.

This page intentionally left blank

Chapter

- How to Write New Legacy Code 334
- How to Avoid Writing New Legacy Code 335
- OK, I've Got Legacy, Now What? 343

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

LEGACY CODE

*I don't want to achieve immortality through my work...
I want to achieve it through not dying.*
—Woody Allen

If you work for a large product group, then by the time you are reading this chapter you are probably thinking, “This book contains some useful ideas, but we have five million lines of code in our homegrown programming language that we need to maintain. These ideas do not work in my environment.” Well, this chapter is for you.

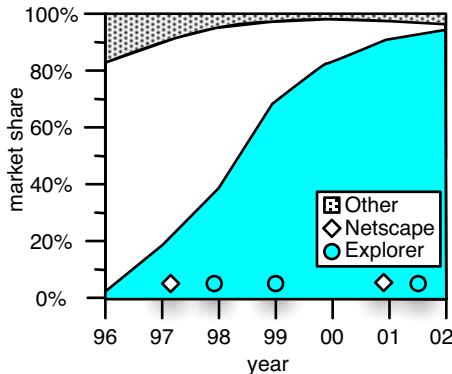
*Avoid...Legacy
code*

Existing well-structured reusable code is a valuable asset. However, this asset can turn into legacy code—poorly structured, inadequately documented code with lots of duplication and without automated tests. Legacy code constrains organizational agility and, as will be seen, leads to a serious competitive disadvantage. This chapter is about how to write that legacy code, and how to avoid it.

But before diving into the subject, it is worth appreciating how many jobs exist because of legacy code. We travel around the world and frequently work in developing countries. In these places, people have risen out of poverty because of the jobs created to maintain legacy code. In countries such as India and China, several cities exploded in size and wealth over the last decade because of the outsourcing industry, and much of this outsourcing relates to legacy code. It is worth appreciating this.

On the other hand, what would have happened if all this energy was put into creative, innovative new products? Besides which, legacy code has also destroyed companies...

Figure 9.1 browser market share and releases



One of the best examples is Netscape, which once *owned* the browser market. But in 1995, Microsoft realized the huge potential of the Internet and started what would later be known as “the browser wars” [CY00]. In 2000, Microsoft won the first battle of the browsers.

There are many reasons for this. One is that Netscape did not release a new browser for *three and a half years*. Why not? “*Because the browser was rewritten independently of the ‘legacy’ code that formed the basis of Netscape’s Communicator browsers*” [Festa00]. In 2007 AOL, (which bought Netscape in 1999) officially killed the Netscape browser [Netscape08].

This chapter solves all your legacy code problems... okay, maybe not. It will make your legacy code problem a little less painful and perhaps, one day, resolved.

How To WRITE NEW LEGACY CODE

Writing legacy code is easy—we can explain it in a few simple steps. Companies have generated piles of legacy code for decades. At Xerox we once heard a maxim, “There are many lessons taught, but few lessons learned.” This is particularly true for legacy code. How to prevent the lesson of legacy code being *taught* over and over again, but was never learned?

How long has it been taught? In 1967, in what is perhaps the first book on software project management, the author taught us:

Equally responsible for the initiation of project with predefined failure is management that insists upon having fixed commitments from programming personnel prior to the latter’s under-

standing what the commitments are for. Too frequently, management does not realize that in asking the staff for “the impossible”, the staff will feel the obligation to respond out of respect, fear or misguided loyalty. Saying “no” to the boss frequently requires courage, political and psychological wisdom, and business maturity that comes with much experience [Lecht67].

There are clear causes of legacy code:

- unrealistic deadlines with fixed content
- poor development skills

And of course, in these causes lie the keys to prevent legacy code...

HOW TO AVOID WRITING NEW LEGACY CODE

Avoid...Fixed content with unrealistic deadlines

“We promised this release to our key customer, and the *only acceptable commitment* from R&D is the first of February,” said an angry email sent by a director to the management of the product group we were coaching. We read it in disbelief and wondered about the *only acceptable commitment*. We decided to ignore the email—for now—and get back to *normal work*—coaching a developer in refactoring a legacy component that was hacked together last release to meet *that deadline*.

Many companies are stuck in a vicious cycle of *forced promises* and *unrealistic commitments*. In today’s time-to-market era, customers ‘force’ them to promise too much. “If you cannot deliver by the end of the year, we will buy from your competitor who will make that promise.” Sales people or executives could respond by being transparent and by working toward a mutual beneficial long-term relationship (customer collaboration), but instead they check whether the contractual penalty for being late is tolerable (contract negotiation) and reply, “Yes, no problem, we can do it!” After which the same cycle starts within the organization. The executive orders the head of R&D to “do it” and “make it happen” because “it is a customer prom-

See
“*Avoid...Product management negotiating a “release contract” (scope & date) with R&D*” on p. 106.

ise.” The promise travels through the organizational hierarchy to the developer, who cannot pass it on any further.

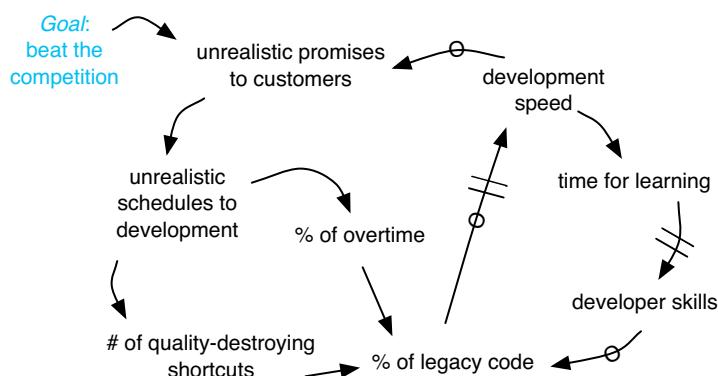
*see secret toolbox
in Systems
Thinking in the
companion book*

*see Lean Think-
ing in the com-
panion for
overburden—one
source of waste*

How does the developer react? Charles Lecht [Lecht67] already warned us over 40 years ago: The developer will “*feel the obligation to respond out of respect, fear or misguided loyalty*” and reluctantly *commit* to the deadline. The developer opens his *secret toolbox* and does everything possible to make the short-term deadline by using *tools* such as hardcoded, copy-paste-modify programming, skipping testing, working overtime, and other quality-destroying shortcuts [Schwaber07a]. Nobody notices the use of these ‘tools,’ and so the deadline is made. Management rewards developers for their hard work and applauds their “great teamwork” and “fighting spirit.”

These quality-destroying shortcuts result in bad legacy code, which slows down the development and the organization falls behind its competition. A predictable scenario unfolds. They need to reclaim the market and therefore make new promises, starting the vicious cycle all over again. The **technical debt**—the legacy code—makes development go slower. The **learning debt**—lack of renewal in developer skills—compounds this slowdown. Developers are so busy keeping unrealistic commitments that they have no time to keep up to date and refresh their skills (Figure 9.2).

Figure 9.2 dynamics of unrealistic deadline



Bob Martin, in *Clean Code*, argues that a software craftsman would not make such an unrealistic promise, and that the legacy code prob-

lem can be solved by educating developers to be more *professional* [Martin08].

Martin is partly right. But this view ignores the fact that a developer is part of a larger system that reinforces this behavior. Not only should software craftsmanship be enhanced, but also the *system* in which developers work.

In Europe, we once visited the director of a large (embedded system) product group and his management team. The director explained that the group had successfully met their last release goal, and so questioned the motivation to adopt large-scale Scrum. Just then, one of his managers spoke up and said, “Well, what really happened was that near the end of the last release we were far behind and so we did serious overtime work and pulled over one hundred people off another product and got them to help. That’s why we shipped on schedule. Now, we are seriously behind on the current release, because so much bad code was created in the last release that we are spending most of our time fixing defects reported from our customers, and having to work with a mess of a code base.”

Observe the relationship between these situations and the absence of lean thinking. For example, the *waste of wishful thinking* plays out in these scenarios. One of the three sources of waste in lean is *overburden*—it is easy to see how the heroic push near the end of release creates more waste in the future. And there is no *stop and fix* culture—quite the opposite, it is “carry on and don’t fix.”

see Lean Thinking in companion

Telling your customers “We do not know the content and we have no idea when it will be done” is commercial suicide. But this concern, which we frequently hear from executives, is a false dichotomy—either *make unrealistic commitments or do not make any commitments at all*.

Try... Transparency and customer collaboration

There is an alternative where both the customer and the client accept the reality of product development: It is not 95% predictable. You can accept this reality by being transparent toward your customers during development. How? For example, by...

see False Dichotomies chapter, companion

- reporting your development status to your key customer iteration by iteration; for example, with a Release Burndown chart and updated Product Backlog

- allowing key customers to give feedback on priorities and modified goals as they see how things are unfolding, and then adjusting the plan accordingly
- giving estimates with probability distributions or giving multiple estimates [DL03]
- other techniques that promote cooperating with customers frequently, based on realism and transparency

By such changes in how product companies relate to customers, the pressure to create bad legacy code is reduced.

see Systems Thinking in companion for a detailed look at the effects of adding people

A common quick-fix response by management to market pressure is to ‘order’ development to “add more resources,” since they are ‘cheap.’ A product group we worked with *was forced* to add hundreds of people within a one-year period. An exception? No, another example: The leader of a product group we worked with recently got ‘promoted’ to a new product. The new product had 900 people, 12 different sites, and 20 active branches. It was behind the competition, and the previous management tried to save it by adding more people—now it was even more behind.

This is another lesson that has been *taught* again and again. Perhaps the first large-scale project in the world was the Semi Automatic Ground Environment (SAGE) system that was developed during the 1950s. The project was in a hurry so...

*Within a year approximately **1000 people** were involved with the development of the SAGE system. People were recruited and trained from a variety of walks of life. Streetcar conductors, undertakers (with at least one year training in calculus), school teachers, curtain cleaners, and others were hastily assembled, trained in programming for some number of weeks, and assigned parts in a very complex organization ... The originally hoped for **capacities of the system were cut** considerably. The system was first delivered **over a year late** and **considerably more cost** than was originally expected.¹ [Schwartz74]*

1. emphasis added

Instead of a focus on cultivating great developers or hiring a few great people, there is a focus on hiring the maximum amount of *bodies* (or *heads*, as in *head count*) which in turn results in a rushed and inadequate new-hire education program. This quick fix leads to groups with low-average development skills, groups with a low aptitude for being great developers, and so ultimately to more and more bad legacy code.

*Avoid...Hiring
many weak
developers*

Poor Development Skills

The organizational dynamics of promises and commitment does not explain the whole legacy code story. Bob Martin is right—the industry definitely needs software craftsmanship.

It seems to us that the average skill level of software developers in large product groups is quite weak. Developers are frequently not familiar with basic good development techniques—simple practices such as information hiding and encapsulation, or good design principles [Martin02]. In embedded systems we sometimes hear developers exclaim, “Those are object-oriented techniques, but we are developing in C”, not realizing that some of these concepts were developed in non-object-oriented technologies (for example, [Parnas72]). We observed a trend:

The larger the product group, the smaller the knowledge of ‘modern’ development practices.

But these practices are essential for sustainable software development [Tate05]. Fortunately, development skills are not solely dependent on raw talent; they can be taught and improved by

- schools
- organizational support
- self-study

The leadership of a product group may believe they understand how these educational forums are working, but it may not be so...

Avoid...Believing universities teach development skills

Schools—Universities are not doing a good job in teaching basic skills to developers. There is a shocking gap between what is happening in industry and in universities. Many educators have never worked in industry and have not seen the long-term dynamics of development skills and legacy code. They also lack a Go See attitude. Some universities recently added agile development practices to their computer science curriculum. This is good. However, deep experience is required to really grasp agile practices such as test-driven development, and educators seldom have this experience.

As such, do not assume that university graduates have much skill in software development—especially in agile development.

Try...Increase organizational support for learning development skills

Organizational support—Most companies do a poor job at educating developers. We frequently hear, “Everybody who graduated from university can code,” thereby implying that educating basic development skills is unnecessary. Our coaching experiences suggest otherwise. Many developers in large product groups lack fundamental skills such as good design of software, efficiently working with editors, effectively using their programming language, or automating tasks by writing scripts. Organizations are failing to educate in these areas because many business leaders have reasonably but incorrectly assumed that people learned these skills at university—unaware that a computer science curriculum does not teach software development skills, and that most university professors do not know and cannot teach modern development practices.²

In contrast, lean organizations invest in educating their employees. One study shows that Japanese lean companies spend eight times as much effort educating new employees than their USA counterparts and twice as much as their European counterparts [WJR90].

see Lean Thinking in companion

Organizations also fail to recognize the need for continuous improvement. They not only need to provide education in basic skills, they need to create an environment in which employees are constantly challenged and learning. How? Managers acting as teachers, peers educating one another (for example, by pair programming) and internal or external dedicated coaches—all supporting an environment of learning and continuous improvement.

2. “Use your editor” is perhaps the most productivity increasing course you can give in many companies.

Self-study—Many developers do not keep themselves up to date. Quality guru Philip Crosby saw lack of knowledge caused by a shortage of learning as a main cause of bad quality.

*Try...Support
more self-study*

People subconsciously retard their own intellectual growth. They come to rely on cliches and habits. Once they reach the age of their own personal comfort with the world, they stop learning and their mind runs on idle for the rest of their days. They may progress organizationally, they may be ambitious and eager, and they may even work night and day. But they learn no more. [Crosby80]

In 1999, Dave Thomas and Andy Hunt published an excellent book, *The Pragmatic Programmer* [HT99], summarizing the attitude and behavior of a modern, professional developer. We encourage people to read this, and more broadly, to take responsibility for keeping up to date.

Avoid...Trivializing programming

“I’m an architect, writing code is something the low-level implementation people do.” We hear statements such as this from ivory-tower architects who consider programming to be beneath them. The organization for which this architect works has created a culture of trivializing programming. Such a culture de-emphasizes code, devalues writing clean code, and devalues learning about programming. In such a culture people want to rise in the social and organizational hierarchy—and that means move away from programming. Coding is just the early career phase that they have to go through. Such a culture is one that gives rise to legacy code.

Organizations trivialize programming by

- outsourcing the programming
- career paths
- salary differences

Outsourcing the programming—Especially in large product groups, we encounter businesses that do not consider writing code their “core business” and so have outsourced it. They write specifica-

tions, architectural documents, and design documents, and then send them to cheap-rate developers offshore to “do the implementation and testing.” A recipe for disaster. The source code is the place of real value—*gemba*. For more, see:

- See “Try...Think ‘gardening’ over ‘architecting’—Create a culture of living, growing design” on p. 282.
- See “Try...Architects and system engineers are regular (feature) team members” on p. 300.
- See “Avoid...Architecture astronauts (PowerPoint architects)” on p. 302.
- See “Avoid...Architects hand off to ‘coders’” on p. 308.
- See “Avoid...Create ‘designs’ and then send them for offshore implementation” on p. 316.

*see Organization
in the companion*

Career paths—Large organizations want to offer a future for their employees; predefined management or technical career paths are a typical solution. People who follow the management path shift away from technical work and become “professional managers.” Those who follow the technical path spend their time writing “architectural documents.” Whatever career path you follow, it won’t contain any programming.

Salary differences—Of all software-development-related jobs, the salary of programmers is, on average, among the lowest [Jones08]. Naturally but unfortunately, these salary differences do not promote becoming a better developer but instead promote stopping work as a developer. Is there an alternative? Pete McBreen promotes a model of software craftsmanship in which salary is directly linked to skill. Development skill is measured by a developer’s portfolio and peer references [McBreen01].

Try...Raise awareness of the negative impact of legacy code

More *legacy code* is more than a liability, it’s a *boat anchor*. It is hard to deliver value fast and adapt quickly when your massive 15 million lines of code is a steaming pile of... well, you know.

Some developers and many nontechnical people in product development do not grasp the negative impact of legacy code—in terms of cost servicing this technical debt and in terms of opportunities lost because of degradation of speed and ability to change.

We encourage technical leaders to proactively educate their business and technical community on this issue, and to explore the cost of legacy code.

OK, I'VE GOT LEGACY, NOW WHAT?

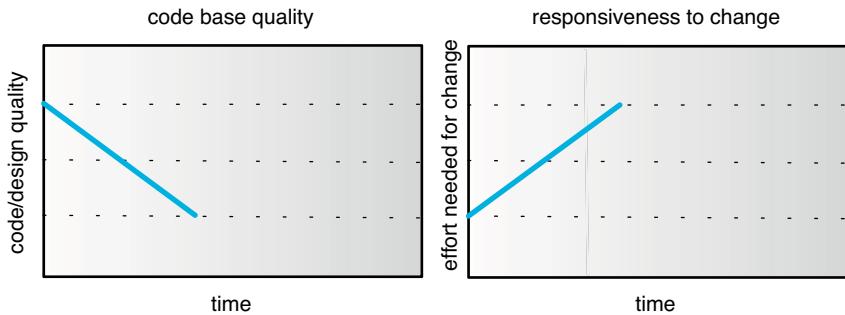
You probably recognize these causes of legacy code, but you already have piles of it. How to get rid of it? In *Working Effectively with Legacy Code* [Feathers04], Michael Feathers provides concrete code-level techniques for gradually improving your code. This chapter does not repeat these; we recommend Feathers's book. But we do cover some general strategies for dealing with legacy code.

Avoid... Rewriting legacy code

When confronted with legacy code, developers frequently suggest rewrite, redesign, or rearchitect—scrap the legacy and write it again. Next time it will be better... Resist that temptation. Why?

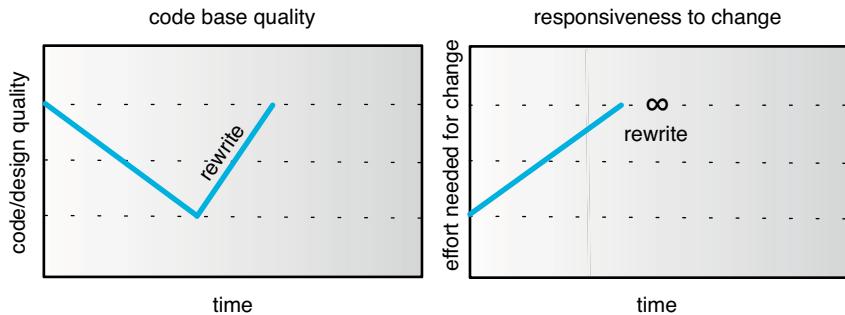
In a product group with a 30-year-old code base, a developer asked us if we could help refactor a 5000-line function. We thought he was exaggerating. But when we paired up and measured the function, we discovered it was slightly larger than 5000 lines of code (LOC). How are 5000 LOC functions created? Does a developer wake up and think, “Gosh, what a wonderful day today! Let's write a 5000 LOC function?” Probably not. When a developer writes new code, he usually *will* write it with decent quality. But over time the quality degrades. A function *becomes 5000 LOC*. Why does this happen? The customer requests a new requirement and this is hacked in because of poor development skills or unrealistic schedules. Code quality goes down and the effort needed to make changes goes up (see Figure 9.3).

Figure 9.3 code quality decreases over time



After some time it is too painful and takes too much effort to make changes to the code; developers start asking for a rewrite. At first the Product Owner refuses—a rewrite means high cost without new value. But as development speed drops, developers complain more, and eventually the Product Owner ‘agrees’ to the rewrite. During the rewrite, the ability to respond to changes—new requirements—is zero. But after the rewrite the code is of high quality, and consequently, new development is fast (see Figure 9.4).

Figure 9.4 rewrite increases code quality



After the rewrite is finished, what happens? Pressure to rush in new requirements leads to hacks in the freshly cleaned code, causing the quality to degrade again and the implementation effort to increase (Figure 9.5). After a while, developers demand another rewrite. In some large product groups, we have seen components being rewritten three times.

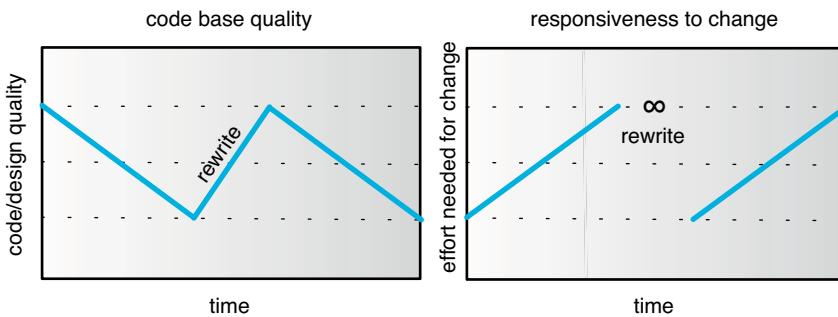


Figure 9.5 code base quality degrades again after the rewrite

Key insight:
The problem is not *having* legacy code,
it is *creating* legacy code.

The focus needs to be on preventing the creation of new legacy code instead of on the legacy code itself. It needs to be on *growing code healthfully* instead of allowing it to degrade over time. How? Improve the code every time a change is made. “*If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot*” [Martin08] (see Figure 9.6).

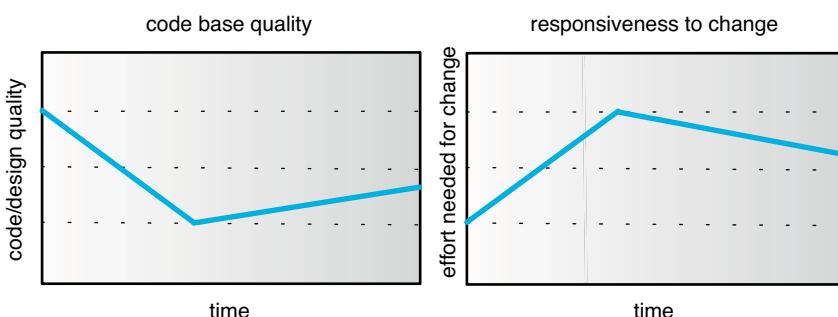
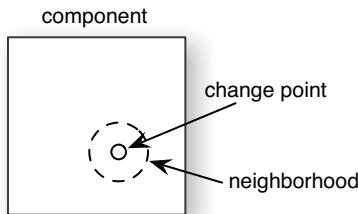


Figure 9.6 growing the code healthfully

Try...Clean up your neighborhood

Growing healthy code is a key strategy for eliminating legacy code. You can do so by cleaning up your neighborhood; by gradually fixing your “broken windows” [HT99]. Every time you make a change, look around your change point—the neighborhood—for code that can be improved—the broken windows—and add a couple of tests and refactor (see Figure 9.7). When starting this practice, every change is a little slower. But over time the code improves and the development speed increases because of the healthier code base.

Figure 9.7 clean up
your neighborhood



Having legacy code means having technical debt—and it costs to get out of debt. A rewrite strategy attempts to settle the debt all at once. On the other hand, cleaning up your neighborhood distributes the payments. It focuses the effort to the parts that change most—the most important ones. The legacy code that does not change does not get improved—and that is okay.

Try...Write both high-level and unit tests

see the Test chapter

We are frequently asked whether to start with unit tests or high-level tests. Another false dichotomy. They are both important! It is often easier to add high-level tests to a legacy code base, and they help in ensuring that existing functionality works. But unit tests run fast. Fast-running unit tests help when you are gradually refactoring legacy code. Therefore, write unit tests *and* high-level tests when cleaning up your neighborhood.

Try...Rewrite lethal legacy code

Sometimes it is impossible to gradually grow the code base healthfully. For example, suppose that part of the low-level code is written in PL/M and no one is willing to learn PL/M. Or, part of your code base is written in a home-grown language, whose compiler only runs on VAX/VMS. When gradual change is impossible³—the legacy is lethal—then it is necessary to ‘amputate’ that part of the code instead of letting it kill your product [Parnas94].

While replacing lethal code:

- cover it with test
- do not add functionality to the old code
- do not add functionality to the replacement code

CONCLUSION

There are billions of lines of legacy code in the world, and the total is increasing every day. This has created massive problems (for example, Y2K), and it will still create monumental ones in the future. But legacy code will not disappear unless the root causes are tackled:

- unrealistic deadlines with fixed content
- poor developer skills

These can be solved by educating people in the causes of legacy code and by improving developer education. However, the industry has failed to recognize these causes for decades. It is not likely to change in the next few years.

How to deal with existing legacy code? It is better to gradually improve the code than to replace it. This requires investing in development skills and applying modern practices such as test-driven development. The only way to grow better code is to develop excellent people. This is a theme in lean thinking.

3. It is rarely impossible to do a gradual change. Therefore, challenge each time someone says that a gradual change is not possible.

Making things is about making people. [Kato06]

RECOMMENDED READINGS

At the time we write this book, surprisingly little has been written about such a huge and costly problem as legacy code. Here are some references we found useful related to improving your code gradually:

- *Working Effectively with Legacy Code*, by Michael Feathers. Concrete advice on how to gradually improve your legacy system at code level.
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler. The classic work on improving existing code.
- *Refactoring Workbook*, by Bill Wake. A concrete guide for becoming better at refactoring code.
- *Refactoring to Patterns*, by Joshua Kerievsky. In this book, Joshua explains how to gradually refactor your code to standard, robust design patterns.
- *Refactoring in Large Software Projects*, by Stefan Roock and Martin Lippert. Large systems might need large refactorings. This book explains how to do these in as small steps as possible so that your systems stays stable.

The following material covers the organizational dynamics behind legacy code:

- *Enterprise Scrum*, by Ken Schwaber. Chapter 9 of *Enterprise Scrum* is one of the few descriptions explaining the relationship between customer promises and the creation of legacy code.
- Sustainable Software Development: An Agile Perspective, by Kevin Tate. This book does not cover many new techniques but provides an excellent overview of the practices for creating software in a sustainable way.

Software craftsman prevent creating legacy code and hence develop software at a sustainable rate. Some material on being a software craftsman:

- ❑ *The Pragmatic Programmer: From Journeyman to Master*, by Andrew Hunt and Dave Thomas. Classic book on modern software craftsmanship.
- ❑ *Software Craftsmanship*, by Pete McBreen dives in craftsmanship approach and compares it to the traditional software engineering perspective.
- ❑ *Agile Development, Principles, Patterns and Practices*, by Bob Martin. Also known as *Agile PPP*, links good code, modern practices, and eternal design principles to explain what it means to be a craftsman.
- ❑ *Clean Code: A Handbook of Agile Craftsmanship*, by Bob Martin. The subtitle says it all. *Clean Code* is the code-focused prequel to *Agile PPP*.

Chapter

- Try...Continuous integration 351
- Developer Practice 352
- Keep a Working System 353
- Small Changes 355
- Growing the System 355
- At Least Daily 356
- On the Mainline 358
- Supported by a CI System 359
- With Lots of Automated Tests 360
- Scaling a CI System 361
- Try...Speed up the build 361
- Try...Multi-stage CI systems 364
- Try...Visual management with CI 367
- Avoid...Large changes 369

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

CONTINUOUS INTEGRATION

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.

—Edsger Dijkstra

Continuous integration (CI) is essential for scaling lean and agile development:

Try...Continuous integration

Our conclusion is that there is no inherent reason why Continuous Integration and Automated Build processes won't scale to any size team. In fact... [they] become more essential than ever. [Magennis07]

With CI, developers gradually grow a stable system by working in small batches and short cycles—a lean theme. This enables teams to work on shared code and increases the visibility into the development and quality of the system.

There are misconceptions about CI; it seems a simple concept, but in practice it's not. To get one misconception out of the way: *Continuous integration is not automating the build and running tests.*

The classic paper on CI [Fowler06a] states:

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.

To expand:

Continuous Integration
<ul style="list-style-type: none"> <input type="checkbox"/> a developer practice... <input type="checkbox"/> to keep a working system <input type="checkbox"/> by small changes <input type="checkbox"/> growing the system <input type="checkbox"/> by integrating at least daily <input type="checkbox"/> on the mainline <input type="checkbox"/> supported by a CI system <input type="checkbox"/> with lots of automated tests

Each part is explained next.

DEVELOPER PRACTICE

Avoid...Believing CI is a tool

Discussions about CI all too often are about tools and automation. Though important, CI in essence is a developer practice. Owen Rogers, one of the original creators of CruiseControl.NET¹ writes:

Continuous integration is a practice—it is about what people do, not about what tools they use. As a project starts to scale, it is easy to be deceived into thinking that the team is practicing continuous integration just because all of the tools are set up and running. If developers do not have the discipline to integrate their changes on a regular basis or to maintain the integration environment in a good working order they are not practicing continuous integration. Full stop. [Rogers04]

Splitting changes into small increments, integrating them at least daily, and having the discipline to not break the build are all done by the individual developer. He needs the skill to work in small incre-

1. CruiseControl.NET is a CI system server for Microsoft .NET.

ments and keep his own copy of the system (or part of the system) working all the time.

Adopting CI requires a change in *human behavior*. We worked with several large products with an excellent automated build but where developers did not integrate their code frequently. Even worse, the message “thou shall not break the build” was strongly promoted, for example, by shaming the people who broke the build. The predictable result? Developers would delay their integrations out of fear of breaking the build. Despite their excellent always-green (always passing) automated build, they are doing the opposite of CI.

Test-driven development (TDD) with constant refactoring helps. When a developer is unit-test-driving his code, he ensures that his local copy is always working. All the tests pass all the time. In theory, he is able to integrate code every TDD cycle (about ten minutes²); in practice, he integrates after a couple of cycles.

see the Test chapter

CI on large products is hard precisely because it is a developer practice. If it were only about tools and automation, you could simply start a CI project or hire a company to “install CI.” But as a developer practice, CI requires a change to the daily habits of all developers. With many people, this is hard, takes time, and requires coaching.

With the right behavior, the developers will...

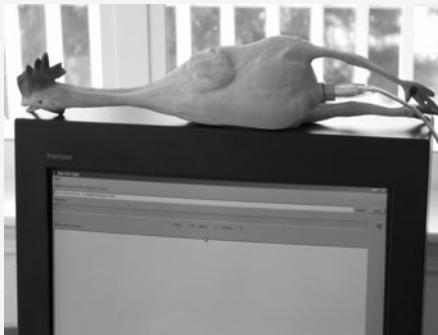
KEEP A WORKING SYSTEM

Similar to the lean concept of **jidoka**, CI means always having a stable system. When a test fails—run locally or on the CI system—the developer fixes it immediately and therefore always keeps a working and stable system.

Traditional sequential development has un-integrated work-in-progress (WIP). Nobody knows if these parts work together or if they

2. For Java, ten minutes is too long. For C++, about average. For C, probably too short. Ten minutes is an average language-independent TDD cycle.

Rubber Chicken



Agile developer James Shore, author of *the Art of Agile Development* [SW07], stresses CI as a developer practice. In his excellent article *Continuous Integration on a Dollar a Day* [Shore06], he explains how to do CI with an old development computer, a rubber chicken, a desk bell, and an automated build. The old development machine is used as the integration machine. The rubber chicken is the *integration token*—only the person who has the rubber chicken can integrate code. The

desk bell announces a successful integration. But the most important step in his description of CI is to get the developers in one room and let them agree with each other that “From now on, our code in revision control will *always* build successfully and pass its tests.”

The rubber chicken does not scale to large products. That said, the story is a vivid way of remembering that *CI is a developer practice*.

are free of defects. WIP makes it hard to predict when, or if, the system is deliverable. CI increases visibility by removing this WIP—always having everything integrated—and with this comes more control and predictability.

Note: CI and iterative and incremental development in Scrum have the same strategy. However, CI is a more fine-grained level than a Scrum iteration. Both reduce variability, uncertainty, and risk by working in small batches—iteratively.

This working system, evolved in small increments, is created by...

SMALL CHANGES

We once worked with a gateway product group in Finland who needed to make a change to their protocol stack. They insisted it could not be split into small changes. They made the large change and spent three months trying to get the system to work again. After that painful experience, they agreed to never make such large changes in one big batch.

Large changes to a stable system *will* destabilize and break it in *large ways*. The larger the change, the more time it takes to get the system back to a stable state.

Avoid large changes. Instead, break each change into small changes—the lean concept small batches. Each change integrates in the system easily.

By small changes you will be...

Avoid...Large batches of changes

GROWING THE SYSTEM

Growing versus building is an important mindset shift. Brooks, in his famous article, *No Silver Bullet*, reflects on his experience:

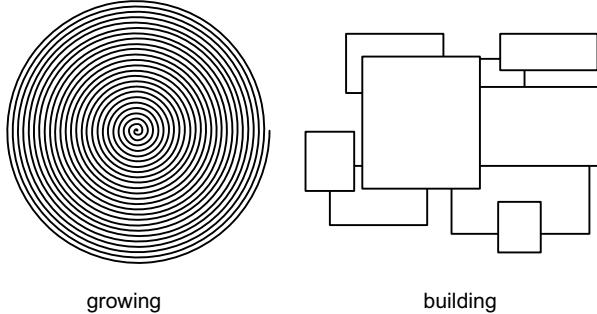
The building metaphor has outlived its usefulness... If, as I believe, the conceptual structures we construct today are too complicated to be accurately specified in advance, and too complex to be built faultlessly, then we must take a radically different approach... The secret is that it is grown, not built... Harlan Mills proposed that any software system should be grown by incremental development... Nothing in the past decade has so radically changed my own practice, or its effectiveness... The morale effects are startling. Enthusiasm jumps when there is a running system, even a simple one... One always has, at every stage in the process, a running system. I find that teams can grow much more complex entities in four months than they can build. [Brooks87]

See “Try...Think ‘gardening’ over ‘architecting’—Create a culture of living, growing design” on p. 282.

Building a system implies building separate components and, when they are finished, assembling them together. *Growing* a system

implies nurturing it and evolving it into a larger system (Figure 10.1).

Figure 10.1 growing versus building



Is this possible in big systems with legacy code? We get that question frequently. In almost every case, the answer is *yes*. If your developers or architects cannot do this or claim it is not possible, take that as a sign of lack of skill.

A developer continuously integrates his work while working on a task. He does not wait for the task or the whole feature to be complete and then “bolt it” on the system. Rather, whenever a small amount of work can be integrated without breaking the system, then he integrates it...

AT LEAST DAILY

How frequent is ‘continuous’? As frequently as possible! This is limited by

- the ability to split large changes
- speed of integration
- speed of feedback cycle

Ability to split large changes—splitting large changes into smaller ones, while keeping the old functionality working, is a skill that must be learned. The better developers are at this splitting, the more frequently they can integrate. TDD, in short ten-minute cycles, is an excellent technique for this.

See “*Try...Split Product Backlog items (such as stories)*” on p. 247.

Speed of integration—the more time it takes to integrate changes into the code repository, the less frequently developers will do so. Changes are batched for the sake of efficiency. Integration effort is impacted by the process overhead (the transaction cost), such as approvals and reviews needed before developers are allowed to integrate. Reduce this overhead or find creative ways to do things differently. For example, we worked with a 40-person product group where the check-in message had to mention the person who had reviewed the code. The result? Developers batched many changes to make the code reviews more ‘effective’ and thus delayed integration—a local optimization. Solution? Code reviews can instead be done on already integrated code—not delaying the integration.

Avoid...*Process preventing developers from checking in*

Speed of feedback cycle—a developer should only integrate changes that do not break existing tests. Ideally, he runs all the tests before integrating. For this to be possible, the tests must run very fast. If they are slow, the developer will delay the integration to “work more efficiently.” However, running all the tests quickly is hard for large systems. Therefore, developers only run a subset of tests before checking in, and a CI system runs the remaining tests. The CI system acts as a safety net by giving the developer feedback about the tests he did not run. What happens when the CI system is slow? First, there will be many changes during one cycle, increasing the chance the build breaks. Second, developers do not integrate their changes in the broken build; rather, they batch them. Finally, when the build is fixed, all developers integrate their batched changes, leading to a high chance of breaking the build again. Therefore, the safety-net-feedback cycle has to be fast. This decreases the chance the build will break and increases the ability to check in more frequently.

Rule of thumb for large products moving to agile and lean development: All developers integrate at least daily. Even though “*daily builds are for wimps*” [Jeffries04], *daily* is a first step for large products. Recall the “lake and rocks” metaphor used in lean thinking—

the big rock of just being able to build once a day *with everyone's updates* is hard enough on big old systems with 300 developers in four countries. Eventually, that big rock is removed, and shorter cycles are possible.

On large products, it will take time to learn to split changes, simplify the integration process, and set up a fast CI system running...

ON THE MAINLINE

Avoid... Branching

Developers integrate on the *mainline* or *trunk* [BA03]. Making changes on a separate branch means that the integration with the main branch is delayed.³ The current status is not visible, so you do not know if everything works together.

Branching during development breaks the purpose of CI and should be avoided. There are exceptions: First, customers might not want to upgrade their product to the latest release but still want patches. Thus, *release branches* are needed.⁴ Second, when scaling up a CI system, it can be useful to have *very short-lived* branches that are automatically integrated in the mainline—more about that later.

See “Try...Configurable design for customization” on p. 315.

What about branching for customization? Bad idea! Instead, manage these through a configurable design or parameterized build instead of using your Software Configuration Management (SCM) system. We once worked on a network-optimizing product being built by a co-located product group who insisted on branching for different configurations. Developers worked on these separate branches for over a year. Afterwards, it took them an *additional half-year*—and lots of fighting—to merge them in the trunk.

Successful mainline development is...

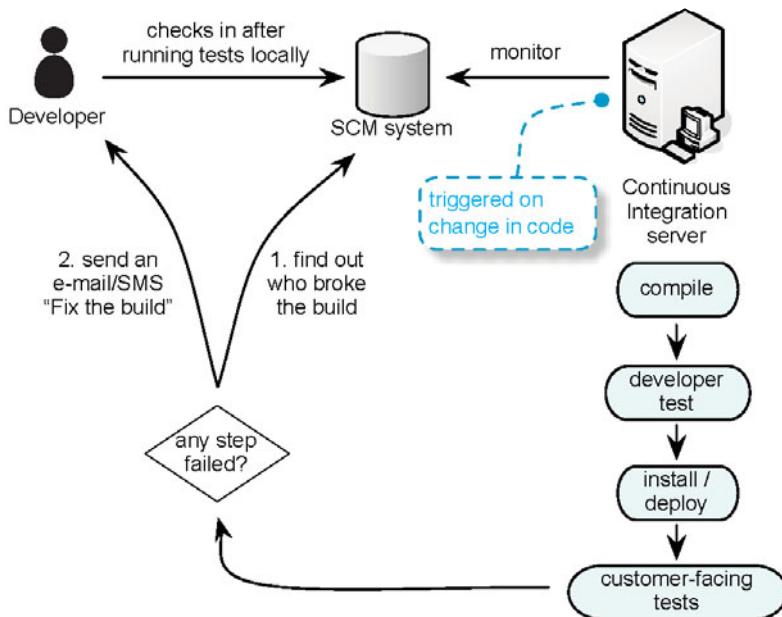
-
3. To be more precise, avoid branches that live longer than ‘hours’. Branching has gotten easier with modern distributed version control systems such as Git and Mercurial. Some *short* use of *short* branches can be useful...but it is a sharp-knife tool that can easily be misused [Fowler09].
 4. Tip: Create a release branch off the mainline *just before* release, not at the start of a release (a release line).

SUPPORTED BY A CI SYSTEM

Lean emphasizes minimizing inventory—one type of waste. Inventory acts as a buffer (another queue) in which mistakes are hidden. Mistakes become painfully visible when these buffers are removed—and that is a good thing. The same thing happens when all changes are integrated directly to the mainline. All developers update their local copy frequently; when someone checks in broken code, it is visible to everybody—they will be annoyed.

People make mistakes. That's OK. A lean stop-the-line safety net is needed to detect them early. Developers fix a mistake before it affects others. This safety net, an *andon*-like system, in Toyota terminology, is a CI system.

Figure 10.2 CI system



A CI system (Figure 10.2) listens to the SCM system. When a developer checks in code, the CI system checks out all the code and compiles it, runs some tests, installs it, and runs more tests. All this happens fast; Extreme Programming recommends within ten min-

Synchronous or Asynchronous Integration?

In the *Art of Agile Development*, James and Shane caution against the use of a CI system. They make the distinction between synchronous and asynchronous integration. Synchronous integration means a developer waits for his code to be integrated successfully. Asynchronous integration means the developer is supported by a CI system running tests while he moves on to work on something else.

Asynchronous integration seems more efficient but often leads to sloppiness and broken builds. On the other hand, synchronous integration doesn't work when the build takes too long, as in large products. When setting up a multi-stage CI system, you might consider using synchronous integration on the low level while using asynchronous integration for running higher-level tests.

utes. If a developer breaks the build, the CI system will query the SCM system and find out who made the change. It sends him e-mail saying: "You broke the build, fix it!" Fixing the broken build is the number one priority because it affects everybody.

For a small product, it is easy to have a fast, ten-minute build. For a large product with legacy code and many people, it is quite a challenge. Later, this chapter examines several techniques for scaling up a CI system...

WITH LOTS OF AUTOMATED TESTS

It is not very hard to have a CI system *compile* everything; it's not very useful either. You want to have as many tests as possible running in your CI system. The more automated tests, the better your safety net and the more confidence your system is working.

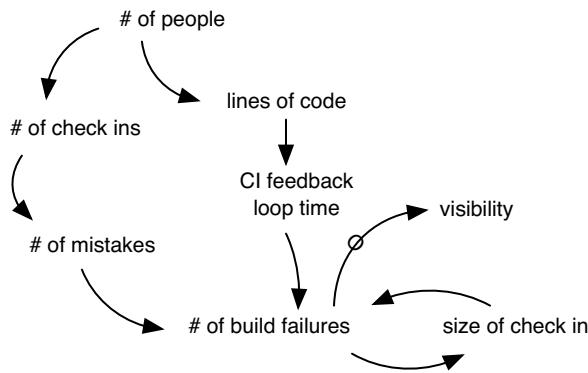
For new products, creating automated tests is not hard. However, many large products have legacy code without automated tests. Developers need to add automated tests—which is a lot of work. The chapter on legacy code covers this.

SCALING A CI SYSTEM

First, the build and test need to be fully automated. Many large products groups we worked with have manual steps in their build. See the recommended readings for texts that are useful for automating the build.

The obstacles for scaling a CI system relate to more people producing more code and tests. First, the probability of breaking the build increases with more people checking in code. Second, an increase in code size leads to a slower build and thus a slower CI feedback loop. These together can lead to continuous build failure (Figure 10.3).

Figure 10.3 the dynamics of broken builds



The solutions are simple:

- speed up the build
- implement a multi-stage CI system

TRY...SPEED UP THE BUILD

Job one: Speed up the build. If the whole product can be compiled and tested within *one second*, then scaling techniques are not needed. The one-second build is out of reach for large products, *for now*. Though every improvement brings us closer, the one-second

build helps. Giving general guidelines for speeding up builds is hard—it's often product specific. Some general solutions [Rasmussen04]:

- add hardware
- parallelize
- change tools
- build incrementally
- deploy incrementally
- manage dependencies
- refactor tests

Try...Add new hardware to speed up the build

Add hardware—the easiest way to speed up the build is to buy more hardware. Throw a couple of extra computers, extra memory, or a faster network connection at the problem and it goes away. Upgrading existing hardware takes investment and only minimum effort, making it the easiest and best choice. The build of one telecom product speeded up 50 percent by compilation on a RAM disk—and this only required a memory upgrade.

Try...Parallelize the build

Parallelize—related to adding hardware is to parallelize and distribute the build. It often requires redesigning the build scripts, changing tools, or even building new tools. Therefore, more effort is needed compared with just adding new hardware. A large telecom product speeded up their build by building every component on a separate computer.

Avoid...Using ClearCase

Change tools—upgrading tools to the latest version or replacing a slow tool with a fast one speeds up the build a lot. Simply by switching compilers we once made a 50 percent improvement in compile time. The most common problematic, slow tool we see used is IBM Rational ClearCase. Every time a product group switched from ClearCase to Subversion—a good free open-source SCM system—they... *First*, speeded up the build (our clients have seen 25%–50% improvement); *Second*, saved the company significant money by eliminating licenses; And *third*, improved the lives of the developers, since ClearCase is often the most hated development tool in the groups we work with. Some misinformed people incorrectly argue that Subversion is not suitable for large-product development. But we have seen it used successfully in product groups with four hundred people located at multiple sites around the globe. Ironically, the so-called large-scale features of ClearCase, such as multisite support, make real CI impossible because they force code ownership.

Build incrementally—you only need to compile changed components and run related tests. Easy in theory; hard in practice. Dependencies between components, changes in interfaces, or incompatible binaries are some of the things that make compiling only the change a difficult proposition. For the same reasons, finding all tests related to the changed component can be difficult. Incremental builds are rarely 100 percent reliable, and to prevent corruption of the incremental build, it's a good idea to also keep a clean daily build.

Deploy incrementally—on large embedded products, it can take a long time to deploy or install software; a radio network's telecom product we worked with took more than an hour to deploy. This is not unusual. Testing speeds up when the deployment is done incrementally—only the changed components are deployed. The changes need to be loaded, and this can be done by rebooting the system. However, starting a large system is time consuming, and therefore some systems are upgraded dynamically—an important feature in telecom and other industries where downtime is very expensive. Incremental deployment—especially dynamic upgrading—requires changes to the system, making this option difficult.

Manage dependencies—unmanaged dependencies are a common reason for slow builds. Examples: Header files including many other header files, or multiple link cycles to resolve cyclic link dependencies. For a multimedia product, we spent several hours on re-ordering link dependencies—cutting link time in half. Reducing dependencies speeds up your build and, as a side effect, improves the structure of your product.

Note a key insight: Improving the build improves the structure of your product. Why? Because bad structure becomes painfully visible when you try to shorten the cycle time of the build.

This is a lean insight discussed before: A powerful side effect of shorter cycle times is the need to dramatically improve the processes and the product to support short cycles and small batches.

*Avoid...Treating
test code differently than production code*

Refactor tests—unfortunately, many developers care less about test code than production code. The result? Badly structured test code and slow tests. We once spent only half a day refactoring tests—and speeded up the build by 60 percent! By profiling and refactoring tests you can frequently make these kinds of quick gains.

TRY...MULTI-STAGE CI SYSTEMS

A multi-stage CI system splits the build and executes it in different feedback cycles. At the lowest level, it has a very fast CI build containing unit tests and some functional tests. When this CI build succeeds, it triggers a higher-level build, containing slower system-level tests. Larger products have more stages.

A CI system is comparable to the “stop the line” culture at Toyota. When a defect is detected, Toyota stops the line, and the first priority is to fix the defect and its root cause. Isn’t a multi-stage CI system hiding defects and against this lean principle? No. A stop-the-line attitude is absolutely needed, but this does not mean that you should blindly stop all the work. Even Toyota does not do that [LM06a].

Toyota has developed a system that allows problems to be identified and elevated without necessarily stopping the line. When a problem is identified and the cord is pulled, the alarm sounds and a yellow light is turned on. The line will continue to move until the end of work zone—the “fixed position stop” point... the line will stop when the fixed position is reached and the andon will turn red.

A multi-stage CI system works the same way. You identify the problem early and act on it, but you do not want it to affect everyone. Only if the problem turns out to be really serious do you “stop the line.”

When building a multi-stage CI system, consider

- a developer build
- component or feature focus
- automatic or manual promotion
- event or time triggers
- the number of stages

A developer build—developers practicing CI need to verify their changes before checking in. Therefore, they must be able to work with a subset of the system, often a component, and be able to run unit tests for it. Take this into account when automating your build.

Component or feature focus—a traditional multi-stage CI system is structured around components. The lowest level builds one component, the next level a subsystem, and the highest level builds the whole product. With teams organized around components, the team takes care of “their” CI system [AKB04]. But where to include higher-level acceptance tests, and what about feature teams? An alternative is to structure your CI system around features. When someone checks in code, all the related feature-CI systems are triggered. Tests are now run in parallel, but the same component is compiled multiple times.

One distributed product group we worked with mixes the two approaches. On a lower level, the CI system is organized around components, and the output of this triggers multiple-feature CI systems running higher-level acceptance tests in parallel.

Try...A mix between feature and component CI systems

Automatic or manual promotion—letting all stages of a CI system listen to the mainline creates a mess. When a developer makes a mistake, all the stages fail. A higher-level CI system needs to be triggered by an announcement that the component can be used. Such announcement is called a *promotion* and is done by labeling (or tagging) the component. Promoting a component can be done automatically or manually [Poole08]. With automatic promotion the lower-level CI system promotes a component after it passed. Avoid manual promotion whereby the team decides when the component is “good enough” and promotes it.

Avoid...Manual promotion

Event or time triggers—every CI system is triggered by either an event or by time. The low-level CI systems are always triggered by an event—a code check-in. For higher-level CI systems, the trigger is either the promotion of a component or time. A trigger by promotion is faster, but for slow builds it is not worth the extra effort in configuration and maintenance. A higher-level daily build might be good enough [Vodde08]. For example, one distributed product group we worked with had low-level, code-triggered CI systems, a higher-level promotion-triggered CI system, and a daily build running tests that lasted over eight hours.

The number of stages—the size and ‘legacy-ness’ of the product determine how many levels of CI systems are needed. Common stages are

- ❑ *fast component-level*—a very fast low-level CI system for quick feedback. It runs unit tests, code coverage, static analysis, and complexity measurements.
- ❑ *slow component-level*—a slower low-level CI system. It runs integration or slow component-level tests.
- ❑ *product stability-level*—a very fast product-level CI system for quick feedback on the basic product stability. It runs fast functional tests (*smoke tests*).
- ❑ *feature-level*—a slower high-level CI system. It runs functional and acceptance tests.
- ❑ *system-level*—a slow high-level CI system. It runs system-level tests, which often take hours.
- ❑ *stability-performance-level*—a very slow high-level CI system. It continuously runs stability and performance tests, which often take days, if not weeks.

We have yet to see all stages in one product. Most products select stages most important for them and add more stages only when needed. An unnecessarily complex CI system is a waste.

Example Staged-CI System

Figure 10.4 shows an example staged-CI system. In this example, every component has a CI system executing unit tests, plus static analysis and code coverage metrics. A successful build promotes the component and triggers feature-level CI systems executing higher-level tests. A daily build executes system-level tests such as performance tests.

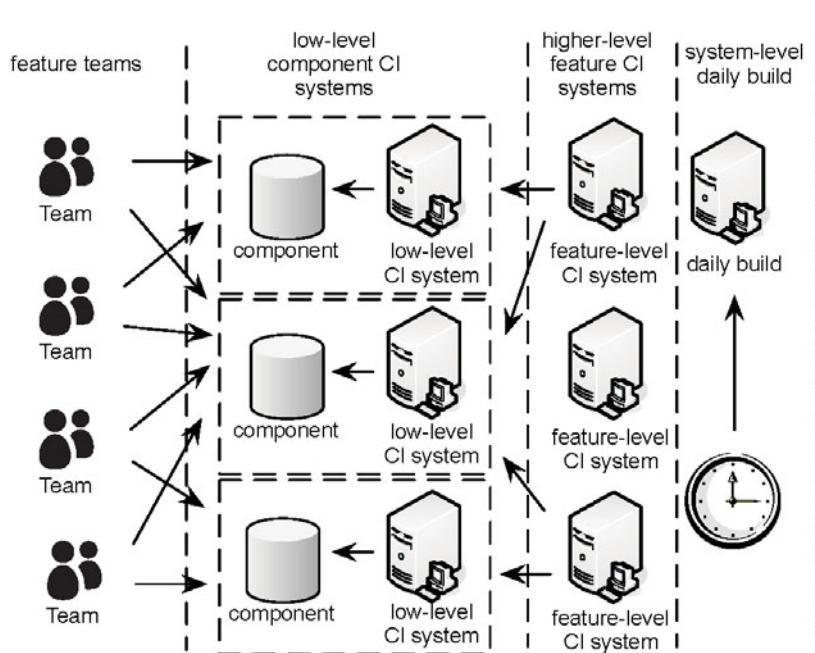


Figure 10.4 scaled CI system

TRY...VISUAL MANAGEMENT WITH CI

A CI system can effectively include visual management—a lean principle. When the build breaks, a visual signal indicates failure—an *andon* system (Toyota terminology). The intent is not for managers to punish the developer who broke the build; it is for developers to see the status of the build. What would they do with this information? Investigate what is going on or delay their integration when the build fails. If, after some time, the visual signal still indicates failure, more people may explore why it is not fixed.

A popular early visual tool to plug into a CI system was a lava lamp. A green bubbling lava lamp indicated a passing build. But when the build failed, a red lava lamp started bubbling.

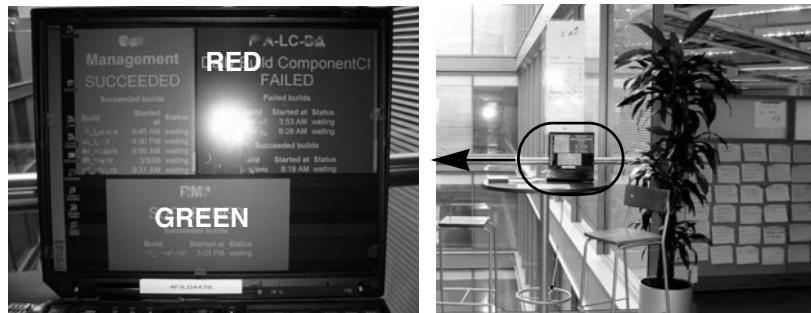
Try...Add red-green screens to your CI system



After lava lamps, people started attaching all kinds of displays to the build, such as Christmas lights, sirens, and moving skeletons that screamed when the build broke. Though less entertaining, a simple monitor showing a Web page of a large red or green color blob (a *red-green screen*) is more easily reproducible. Red-green screens seem to have become ubiquitous in large-scale CI. Some versions

include a yellow signal to indicate “the broken build is now being fixed.” The simple large color blob—visible from a distance—is the key element, but the display can also add text or chart data, such as build duration or test coverage. The information does not need to be limited to build information [Rogers08].

Figure 10.5 andon system for the build, in the hall by the coffee room



One warning related to visual management, well stated by Jeffrey Liker [LH08]:

Just because there is visual presentation does not mean there is visual management. It is relatively easy to set up nice display areas that are for show. The challenging part is making them “for go.” Many people who visit Toyota openly voice the difference of their approach. We often hear comments such as “Now I see, the things that Toyota displays are actually driving action on a daily basis.” This is indeed the difference, and Toyota would suggest that if it is not driving daily action, get rid of it.

Avoid...**LARGE CHANGES**

On large products it is even harder to split large changes into small ones. Developers sometimes want to restructure or re-architect their legacy system and are convinced that it must be done in one large change. But we have yet to see a large refactoring that could not be done gradually. Every time, after discussion with the developers, we found ways of splitting the must-be-done-at-once large refactoring [RL06].

Interface changes are a common problem in large systems. Many components use the interface and need to be changed—making it impossible to do gradually, right? Not so. In fact, an interface change in APIs is common, and there is a well-known solution:

1. Create a new interface.
2. Gradually move all clients to the new interface.
3. Remove the old interface.
4. Rename the new interface to the old interface.

Every step can be done independently and at different times. For public APIs, it is impossible to find out if there are still users of the old interface. This makes removing the old interface difficult. But most interfaces are not part of a published API, so do not forget to remove the obsolete one. We have seen many products with three or four file system interfaces or logging interfaces because the old ones were never removed.

*Avoid...Leaving
obsolete inter-
faces in your code*

CONCLUSION

How to start? Using CI requires

- changing developer behavior
- setting up a CI system

Changing developer behavior—Because large products have many people, this is the hardest task. Focus on TDD—a great way of splitting a large change into smaller ones. Using TDD coaches, who pair-program to teach, is an efficient way of learning TDD. But be

patient. TDD is a hard change for most developers and learning takes time.

Setting up a CI system—Most products we worked with set up a separate project for building a CI system. This works, though a better alternative is to add the work to the Product Backlog and let an existing feature team work on it. This creates more visibility and a sense of ownership—the developers are also users.

Avoid...‘Solving organizational problems with technical solutions’

Most problems implementing CI are organizational and not technical. In many products we worked with, CI became an organizational mess. It involved many traditional functions and roles: developers, managers, testers, test automation engineers, ScrumMasters, agile coaches, SCM administrators, and the IT department. The result was unclear responsibilities, blaming, and committees (aka “steering groups”) who were forever discussing without anybody doing real work. The result? No progress. If this happens, do not try to hide the organizational problems with technical solutions and do not give up because “*our product is too complex for CI*.”

Why not give up? Because every product group we have worked with who went through this “big rock removal” process toward CI has unequivocally found it *immensely useful*.

RECOMMENDED READINGS

Original texts related to CI:

- *Extreme Programming Explained*, by Kent Beck. The term CI was first coined in the Extreme Programming method.
- *Continuous Integration*, by Martin Fowler. Probably the best CI description available.

Recommendations related to automating builds:

- *Managing Projects with GNU Make*, by Robert Mecklenburg. When working with C/C++, you will probably use Make. This book provides a great overview of Make and also talks about Make in large-scale development.

- ❑ *Ant in Action*, by Steve Loughran and Erik Hatcher. When working with Java, you will probably use Ant. This book's focus is Ant, but it covers other topics. *Maven*—another popular build automation tool—is also covered.
- ❑ *Groovy in Action*, by Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skret. Groovy is a recent JVM-based dynamic programming language. It has some excellent build automation support.
- ❑ *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Apps*, by Mike Clark. A small book that covers lots of technology related to automating Java builds.
- ❑ *Continuous Integration: Improving Software Quality and Reducing Risk*, by Paul Duvall, Steve Matyas, and Andrew Glover. The focus of this book is on the automation of builds more than on the practice of CI.

Very little is published on CI in large development. Martin Fowler's original article discusses staging. Also:

- ❑ “Scaling Continuous Integration,” by Owen Rogers in *Extreme Programming and Agile Processes in Software Engineering 2004 Conference Proceedings*. Although a little dated, this is among the best of the available material (other than this chapter) related to scaling CI.

Chapter

- Thinking about Adoption & Improvement 374
- Early Days: Team & Management Changes 391
- Early Days: Breaking Barriers & Habits 394
- Early Days: Gatherings 397
- Coaching & Community 399
- Continuous Improvement 402

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

INSPECT & ADAPT

The taxpayers are sending congressmen on expensive trips abroad. It might be worth it—except they keep coming back.

—Will Rogers

We have worked closely in a few enterprise-wide lean or agile adoptions over the years, and have collected experiments. Some, covered later in the *Continuous Improvement* section, focus on scaling and multiteam coordination (such as a Joint Retrospective); many others focus on organizational design and culture. First, a story...

We were coaching in Europe and met with a manager who had been assigned the agile transformation responsibility; he wanted to show us his plan and ask for feedback. He presented a Gantt chart of his planned transformation: many stages of precise duration all in sequence, milestones, specific managers assigned to tasks along the way, cost estimates, and more. According to the plan, in twenty-seven months the group would have transformed to ‘agile.’ The detail was impressive—it was also the wrong approach.

Our colleague had confused *doing agile* and *being agile*. And he was applying command-and-control management thinking combined with predictive planning—in essence, traditional management ‘agile’ adoption. Fortunately, within a few minutes of chatting, the plan was jettisoned and his view shifted to serving the teams, using a backlog, and adaptive planning.

This misunderstanding to agile or lean adoption is common in corporations that (1) *mandate* a top-down ‘transformation,’ (2) think this is another *change project* with an end (“we have now finished chang-

ing to lean—you get the bonus”), or (3) have a centralized group responsible for pushing processes.

Adopt lean and agile principles the same way as applying them: With experiments, adaptation, self-organization, and a focus on the value-add work by applying Go See.

THINKING ABOUT ADOPTION & IMPROVEMENT

Avoid...Adoption with top-down management support

At a time when all of us are struggling to implement lean production and lean management, often with complex programs on an organization-wide basis, it is helpful to learn that the creators of lean had no grand plan and no company-wide program to install it. [SF09]

“Our agile adoption would be so much better if only we had management support.” We have heard that many times, but be careful what you wish for—you might get it! In one enterprise that got official “everyone do agile” management support after an informal adoption had been going on for several years, we hear the complaint, “I wish we never had management support; now people are doing things for the wrong reasons.”

Why? In some organizations the culture is

- management phones in their support but does not deeply learn lean thinking or agile principles¹
- management ‘drives’ change by setting targets and offering bonuses; in this case, the *agile adoption targets*
- management directs a centralized process group to “push out the new process”

1. At one of our clients a senior manager asked, “What is the *total cost of ownership* of adopting lean thinking?”

Then, what happens is a superficial cargo-cult agile and lean adoption, with widespread game playing, resentment, hidden resistance, and misunderstanding... another management fad that will pass away if ignored long enough. Perhaps there is a target: “50% of the teams have a ScrumMaster within the year,” and managers get a bonus if that is ‘true.’ Then, existing project or line managers are relabeled as ScrumMasters. Or, “Every product should have a Product Backlog.” The existing work breakdown structure of tasks is copied into a spreadsheet called the backlog. Nothing has really changed, and indeed things may be getting worse because of more disruption and gaming.

Avoid forcing—When coaching we encourage: *volunteering; do not force any agile or lean approach onto people; people should be left the choice to think and experiment.* Create a culture of coaching for those that want to experiment.

Focus—Strive to achieve skill and demonstrate excellence in the adopting groups, with concentrated long-term, high-quality support. The best, most *sticky* adoptions we have seen had this approach.

Try...Adoption with top-down management support

In contrast to the prior case, we have also seen groups with a high-quality management culture that cultivated genuine improvement.

We recall one client (at a bank) where the leadership team quickly dove deep into *reading many books* on agile principles, studied and *applied systems thinking*, all attended a *ScrumMaster training* with their team members, *talked with hands-on experts*, used *agile coaches*, and applied *Go See*. Quickly after starting, this group had made deep changes in organizational design and there was tangible improvement in the flow of value to users.

For ScrumMasters and other coaches the implication is: Only lobby for top-down support when you think the leadership team is seriously interested in learning and in organizational redesign.

Try...Individuals & interactions over processes & tools

One of our colleagues in an agile-coaching group observed, “This company has tried to use processes to compensate for a lack of competence of its employees.”

The first agile value, and the previous story about the effective agile adoption at a bank, reminds us of its veracity—*people, not processes, are the first-order effect* for success [Cockburn99].² A group cannot ‘process’ its way out of a deep hole dug by problems with the individuals in engineering and management—‘agile’ will solve nothing in that case.

So, focus on cultivating and hiring extraordinarily talented people.

But, no false dichotomy... as object-pioneer Grady Booch wrote:

People are more important than any process. ... Good people with good process will outperform good people with no process every time. [Booch96]

Try...Job and personal safety (not role safety)

It is difficult to get a man to understand something when his job depends on not understanding it.—Upton Sinclair

We were in Norway, dining on *lutefisk* with a colleague. He said, “My company has hired consultants for a *lean initiative*. They are identifying redundant employees and firing them.”

This is a perversion of lean thinking. Lean has nothing to do with terminating ‘redundant’ employees, nor with lean-by-consultants. The English name ‘lean’ was *not* chosen to imply removing the *fat* from an organization. Rather, it was chosen³ to contrast *mass* manu-

-
- 2. An inefficient *process* with large batches, queues, and handoff is itself a major force for failure, but it comes from *people* and their mindsets. Toyota says, “build people, then build products.”
 - 3. By John Krafcik while working on a graduate degree at MIT; Mr. Krafcik was the first American engineer hired by NUMMI, the Toyota-GM joint venture in California.

factoring with *lean* manufacturing—working in small batches and with less effort to produce goods.

Toyota strives to provide long-term job safety. This is part of the first pillar of lean thinking: Respect for People. And it is also intimately connected to the second pillar: Continuous Improvement. Who is going to strive for continuous improvement in the organization when the likely outcome is job termination? Yet, this does not imply role safety—which inhibits improving the system. For example, project-manager role disappears in Scrum; we have seen people then shift to hands-on engineering or product management.

Personal safety—In Los Angeles one December morning we waited in a room to meet with a team we had been invited (by higher-level managers) to coach for a few weeks. Soon they showed up. We welcomed them and asked, “What are the problems you’d most like to work on? Maybe we can help a little.” There was a long silence—people were uncomfortable to openly discuss problems. So, below the extreme case of job loss, there is the issue of personal safety and improvement. In the *Crystal Clear* agile method, it is identified as one of seven key properties set up by the best teams:

Personal Safety is important because with it, the team can discover and repair their weaknesses. Without it, they won’t speak up, and the weaknesses will continue to damage the team.
[Cockburn04]

A book we sometimes suggest to ScrumMasters (and others) is *The Five Dysfunctions of a Team* [Lencioni02]. The first two of these dysfunctions are *absence of trust* and *fear of conflict*. An *improving* Scrum team must resolve this. See the recommended readings for material that might help.

Offshoring is another context that we regularly see personal safety problems; a company terminates employees in higher-cost regions and shifts more work offshore. This impacts motivation and collaboration between people in different regions.

In a new large-scale Scrum adoption initiative, ScrumMasters and others need to be mindful of these dynamics: Is Scrum or lean development going to be viewed as a mechanism to ‘streamline’ and terminate overhead? And whereas in little companies active opposition

to the system is common, in large product companies there is often a sense of disempowerment and reduced personal safety to challenge the existing organization. Then, for instance, people complain that Scrum Retrospectives are ritualistic, useless, or dead. Or perhaps even worse, people develop a passive-aggressive attitude in response to this ‘streamlining,’ with subtle negative consequences.

It takes active ongoing encouragement from the leadership to keep kaizen mindset alive. As Toyota CEO Katsuaki Watanabe said:

The root of the Toyota Way is to be dissatisfied with the status quo; you have to ask constantly, “Why are we doing this?”
[SR07]

Try...Patience

Toyota has taken *decades* to cultivate a lean culture; similar patience is needed elsewhere. Further, Toyota rapidly expanded in the 1990s and then experienced more difficulty in spreading and sustaining a lean-thinking culture, especially in their satellite plants. It is easy to start losing that culture without ongoing constancy of purpose by lean-thinking manager-teachers [Womack09].



Daily stand-ups and visual management can be installed in days. But it takes years to develop an enterprise of people that know, teach, and apply agile and lean thinking. It is worth it—there lies the great leverage for sustained improvement. Hence the Toyota message, *build people, then build products.*

Avoid...Adopting “do agile/lean”

Be agile rather than do agile was the theme of the *Agile* chapter in the companion book. Agile is not a practice; it is a set of values and principles. Some of the clients we work with misunderstand this and

establish a large-scale transformation project that is measured in terms of *observed practices*, such as,

having a Product Backlog	doing a daily stand-up	working in time-boxed iterations
having information radiators on walls	doing planning poker	writing user stories

To be clear, we recommend trying these practices—indeed, the next suggestion emphasizes that *doing* concrete agile or lean practices is very important. But there is a big difference between a genuine jelled self-managing team that wants to hold a daily stand-up so that they can coordinate, and a group that has been told to have a Daily Scrum—especially if that is on someone’s checklist of “practices in place that prove we are doing agile.”

It is common to find groups where all these practices are *observed*, but where there is only superficial adoption or understanding and little or no *agility*.

Similarly, we recently visited a large outsourcing client in India that was “doing lean.” We asked what that meant. Answer: Using a software tool to measure their WIP levels, and trying to reduce it.

Avoid...Being agile/lean without agile/lean practices/tools

“We understand the Agile Manifesto and lean thinking, and focus on the big ideas—we understand that all practices are just context dependent. And the standard tools don’t work in our context, because we’re different. We have very lean analyst teams, component teams, and test teams, each focusing on their flow.”

In addition to seeing shallow practice adoption, we have seen the opposite: A claim to follow agile or lean thinking but no (or little) application of *any* concrete tools and practices. This is associated with relabeling existing ways of working as agile/lean, when in fact very little has changed or improved.

What happens if there is genuine effort to adopt *many* agile or lean practices or tools? For example, test-driven development, visual management of WIP (perhaps combined with a limited-WIP policy),

reduction in handoff, and more? This *doing* creates a *concrete* framework for learning and kaizen, and a force for deep transformation. Without that concreteness, it is easy to (1) miss subtle insights and context-dependent lessons, (2) miss discovering benefits of these tools, and (3) avoid really improving.

Walk before running

In *Agile Software Development*, Alistair Cockburn [Cockburn07] discussed the *shu, ha, ri* model of stages of skill development in Aikido and its applicability to practices-versus-principles in agile adoption. This parallels the *apprentice, journeyman, master* model. People need to walk before they can run—they cannot become masters without first spending time with tools, mastering them by the book, and experiencing different contexts.

The kaizen cycle starts with learning and applying a standard practice⁴ for similar reasons and because improvement should be against a baseline of insight gained by practice. And there is similar advice in Scrum...

Rule changes should only be entertained if and only if the ScrumMaster is convinced that the Team and everyone involved understands how Scrum works in enough depth that they will be skillful and mindful in changing the rules. [Schwaber04]

No false dichotomy—Principles without practices lead nowhere; practices without principles, theory, and context lead to misapplication and waste. Adopt principles and practices together: thinking tools and action tools are complementary.

Avoid...Agile/lean transformations or change projects

Framing the adoption of lean thinking or agile principles as a transformation or change project leads to the notion

4. Discussed in the *Lean Thinking* chapter of the companion book.

- it is a project, with an end
 - rather than lifelong continuous improvement based on experimentation and growing problem-solving skills
- it is something that people do
 - rather than a change in mindset, culture, and paradigm
- it is something to define and direct by managers

So, rather than framing this as “the agile change project,” experiment with framing it as...

Try...Agile/lean adoption forever

One of the pillars of lean thinking is continuous improvement; lean adoption is not a project with an end. Similarly, a group has never finished adopting Scrum; the framework implies inspect-and-adapt every iteration, without stop. Therefore, do not establish an agile change project; rather, build a permanent system for improvement.⁵ And rather than framing what managers do as managing “the agile change project,” experiment with framing what they do as...

Try...Impediments service rather than change management

Sometimes, phrases are influential. Consider the difference between *manage the agile transformation* and *impediments service*.

In the latter case, in the lifelong agile or lean journey (it is not a project), the team members and Product Owner create an *impediments backlog* of their impediments—policies, structures, environment, tools, and more. The role of managers—in the context of agile adoption—is to help the teams and Product Owner by never-ending *impediments service*—working to remove impediments forever.

This change in behavior—and phrasing—is a shift from top-down or command-and-control to bottom-up service.

5. There is an analogy here to the transition from project-mindset to *continuous product development* discussed in the *Organization* chapter in the companion book.

It leads to more Go See behavior by managers and the chance to serve as teachers rather than controllers or planners. For example, many team members will not even realize something is an *organizational design* impediment; lean-thinking manager-teachers have an opportunity to help them learn to see this.

Iterative and adaptive; pull from the backlog—This is also a shift from predictive to adaptive planning. In this model, agile adoption is based on (1) a prioritized impediments backlog, (2) short impediment-service cycles⁶ executed by managers, and (3) *adaptive iterative planning* based on a re-prioritized backlog each cycle. Who knows what will be done in the next impediments-service cycle?—As with Scrum, the impediments backlog is emergent and continually re-prioritized.

There is no predictive planning, schedule, milestones, targets, or Gantt chart with the “agile adoption schedule.” Rather, Scrum and agile adoption is *iterative and adaptive*, just as regular agile development.

Prioritization and impediments backlog owner?—An official backlog owner is probably not needed. Instead, experiment with this: Every team, when they add an impediment to the backlog, give it a priority. Then, prioritize based on (1) number of teams that raise the same impediment, and (2) average priority of the impediment.

Avoid ‘impediments’ added from quality and management areas—Some years ago, in China, we were coaching a Scrum-adopting product group that had an impediments backlog. All the original impediments came from hands-on workers. But after some time, *quality managers* and department managers started to add their own ‘impediments.’ These were not impediments of flow of value to customers, nor impediments from the value-worker viewpoint; rather, they were ‘impediments’ such as “not conforming to centralized pro-

-
- 6. As in Scrum-for-development, some management groups use *timeboxed* cycles to improve cadence, to address the Student Syndrome problem, and to motivate splitting large impediments into smaller ones—with smaller incremental solutions. But do not assume all the practices of Scrum (such as timeboxing) will successfully apply in non-development contexts, such as this.

cess practice <X>.” Avoid that; the important work is the value stream of the teams and Product Owner, and removing their impediments. All that said, …

Avoid ‘impediments’ added from hands-on workers—If you ask a typical existing team of testers or a component team, “What is the best team structure?” They will say, “Our current structure, of course!” It is common that people—arguably even more so in non-management positions—have *not* developed systems-thinking or lean-thinking skills, nor have they studied organizational design, team, or product-development research. Toyota (and management thought leaders) have emphasized the vital role of managers who have this kind of knowledge, educate others, and improve the system with insight. Suppose there was a recent shift to feature teams and early testing, and then ex-test-team members added an ‘impediment’ to the backlog: “the testers should be in a separate group, and avoid testing early so that it can be done efficiently at the end.” ScrumMasters and manager-teachers have a responsibility to debug these local-optimization thinking mistakes, and clarify problems that genuinely impede the flow of value. It is easy to fall into the trap of local suboptimization thinking—*watching the runner rather than following the baton*, forgetting gemba and Go See. We make this mistake too. Testing our ideas against people educated in systems thinking can help.

Managers add system impediments—Building on this last point, there are system weaknesses to the value stream (usually in policies and organizational structure) that team members are unlikely to grasp or consider candidates for change. Managers have a pivotal role in identifying and removing these. The *Organization* chapter in the companion book centered on these weaknesses.

Add impediments from the Product Owner and product management—The value stream is within the teams *and* in the work of the Product Owner and product management. Invite product management to impediments backlog workshops.

Accept the priority given by the hands-on workers—At one of our clients in Greece, we facilitated an initial impediments backlog creation workshop with team members. After all the voting, what was their number one impediment?—A slow network. For years that had been the dominant issue (it inhibited integration, for instance), but no one in management had done anything about it—the priority of

this and other impediments had never been *this* clear. Now, with a list of 50 prioritized impediments, the number one issue was unambiguous. To their credit, the management team—that had agreed to move to the model of impediments service—accepted its priority and within a few months, problem solved. This also built trust and cooperation because the teams saw managers genuinely helping solve their key problems.

Create the initial impediments backlog in a workshop—We have helped set up many impediment services for management teams, and have found the following approach useful to start it off:

1. Convene a workshop with hands-on people from teams, the Product Owner, and other product managers. In other words, focus on *gemba*—where the value work is. Start with *brain-writing* impediments on cards, in pairs.
- 
2. Next, form larger groups from four or five pairs. The groups discuss, merge, and refine the impediments into a new set of cards. Use the floor.
- 
3. Combine the refined cards from all groups into a central floor area. Do *affinity clustering* to group them. Remove duplicates. Then, do *dot voting* by all participants. Finally, lay out all the cards in (dot voted) priority order. Discuss and refine—final tweaking.
- 



4. Use visual management. Set up the backlog on a wall outside the office of a senior manager. (This photo shows a day-one backlog with no ‘service’ yet). For example, in Greece it was set up near the office of the head of the development center. During impediments-service Sprint Planning, or at other times, managers volunteer for an impediment, write their name on the card, and move it to the middle WIP column.

Rather than “manage agile transformation,” help agile-adopting teams and product management with *impediments service*.

Try...Human infection

Thinking and acting outside the box is possible but hard when everyone is inside it. Lean thinking, agile principles, self-organizing teams, test-driven development, feature teams, manager-teachers... these are mindset, culture, and behavior changes—and to be sticky or meaningful, these kinds of changes require *human infection* from experts through long-term face-to-face coaching.

In the most successful adoptions we have seen, the organization established internal coaches supplemented with external coaches (both were important), and emphasized lots of hands-on mentoring from these agents-of-change during the real work.

Avoid...Agile/lean adoption targets or rewards

Rewards work. An economist wrote in his blog a story to prove this: His son still wore diapers to bed each night. The economist told his son, “If you don’t wet your diapers tonight, tomorrow I’ll buy you the toy you want.” The next morning, the father went into his son’s room. His son had successfully fulfilled the goal and was looking forward to the reward. He had removed his diaper the previous night. The *bed* was all wet, but his *diaper* was dry.

The *Organization* chapter of the companion book summarized the hard facts that performance-based incentives lead to gaming, opacity, and a weakening of the system. We have seen their deleterious effect in promoting “fake agile” adoptions in several groups. Avoid that—and avoid “agile adoption” target setting. The quality guru W. Edward Deming, in his *14 points for management* [Deming82], summarized this in number...

11. *Eliminate management by objective. Eliminate management by numbers, numerical goals. Substitute leadership.*

Avoid...Competitive ‘improvement’

At some clients we have worked with, the introduction of kaizen gets mixed up with their prior management culture, such as competitive incentives. Then, teams or individuals are offered rewards if they improve more than others. This leads to a competitive rather than cooperative culture, in which parties are less willing to share or help others since they might ‘lose’ individually.

Avoid...Try...‘Easy’ agile or lean adoption

‘Easy’ agile adoption is an existing weak organizational design not meaningfully changed, and a thin veneer of practices painted on: managers relabeled as ScrumMasters, existing component/analyst/testing teams get their own “Product Backlog” and hold a daily stand-up meeting *every week*, and more. There is no significant improvement, and people do not take continuous improvement seriously—or worse, they think, “the agile adoption is finished.”

On the other hand, Scrum emphasizes the *art of the possible*. It may be that minor modifications are the current limits of change because of limits in mindset.⁷ These will not meaningfully enhance the value flow to customers, but perhaps (1) adding prioritized backlogs, (2) working in short timeboxes, (3) lowering WIP, (4) holding standups,

7. Sometimes, people have invested years in sequential life cycle processes and the existing team structures; they will not easily consider the possibility it was not ideal for flow of value.

and (5) reducing multi-tasking will help *fractionally*. It is a first step before deeper change and improvement. Then, we suggest...

If you're going through hell, keep going.—Winston Churchill

Try...Experiment rather than improve

The mandate to *improve* is a lofty goal, and can scare off people from experimenting. What if the *improvement*...doesn't? Kaneyoshi Kusunoki, a student of Taiichi Ohno and executive vice-president at Toyota, said about kaizen and management support:

A defining characteristic of the corporate culture at Toyota is that managers don't scold you for taking initiative, for taking a chance and screwing up. Rather, they'll scold you for not trying something new, for not taking a chance. Leaders aren't there to judge. They're there to encourage people. That's what I've always tried to do. Trial and error is what it's all about! [SF09]

Developing problem-solving skills through many experiments is central to lean thinking. The only bad experiment is the one not tried!

The real measure of success is the number of experiments that can be crowded into 24 hours.—Thomas Edison

In this light, the *Try...* and *Avoid...* ideas in this and the companion book are just experiments—and also because systems are too complex and variable to assume prescriptive advice will work.

Try...Encourage experiments; offer coaching

The mandate “adopt agile development” is daunting and large. The mandate “do continuous integration” reflects command-and-control, forcing practices. An alternative to both these approaches is to foster the kaizen mindset encouraged in lean thinking: People are encouraged to experiment and are supported with coaching and education. For example, a ScrumMaster can explore with teams the problems associated with delayed integration, describe continuous integration as an alternative, and arrange coaching if the teams want to try it.

Avoid...Forcing adoption of practices

Avoid...Adopting <X> because “agile didn’t work here”

Survey decades of management and product-development trends, and some patterns emerge. Possibly the dominant one is

1. difficulties exist due to system weaknesses in organizational design, poor engineering skill, and ineffective management
2. try new ‘thing’ to address a problem (insert: MDD, PMI certification, Kanban, CMMI, Scrum, SOA, agile, next-generation lean, ...)
3. do not address the systemic issues; try ‘thing’ superficially
4. after two years, abandon ‘thing’ because “it doesn’t work here”
5. go to step 2

We see this in some groups trying Scrum. Scrum is a simple framework that acts as a mirror: Rather than fixing problems, it increases visibility of systemic weaknesses, inviting inspect-and-adapt with experiments. In some groups, rather than fixing the system, it is easier to try the next thing... “Let’s call in new consultants specializing in Scrum failure, and then adopt...next-generation lean.”

Avoid...IBM/Accenture/... agile adoption

This is not about IBM or Accenture per se; it is about

- the misconception that agile is a process or practice
- shifting responsibility for agile/lean success to an external consulting group

From this stems the notion it can be bought and installed—and there are companies happy to take your money claiming so. Plus, it is related to the misunderstandings summarized in the *False Dichotomies* chapter of the companion book: *agile means iterative development, Scrum means daily stand-ups*, and so forth.

Avoid...Adopting agile with “agile management” tools

“We’re starting to do agile. What tool should we buy for agile project management?” This is a question we hear often; our suggestion is always the same—and we mean this even for the very large-scale cases: “Avoid any special agile tools until several years after starting the adoption. Keep it simple. Use the wall or, in the most complex solution, a simple spreadsheet and wiki.” Why?

Problems from system weakness cannot be solved with processes or tools. Worse, attempting to *quick fix* systemic problems with tools creates an illusion of control or change but no real improvement... Executive: “What is the agile transformation progress?” Agile-change manager: “We have installed <AgileToolX> and three of the projects are using it. Come take a look at the burndown charts...”

Avoid the lure of “tools to do agile management” for at least several years after starting to adopt agile or lean development, so that *people’s focus* can be where it belongs: on the *system*. By removing all crutches and quick-fix illusory solutions, people may—just possibly—be prompted to squarely face the important but hard issues: competent individuals, interactions, organizational design, the illusion of command-and-control, and so on.

If you automate a mess, you will get an automated mess.—anon

We are not suggesting agile-management tools are poor—or good. This is about focusing on important things first and avoiding the dysfunctions that accompany management-reporting tools.

After <N> years? Prefer free tools so that the cost of experimenting is low and there are fewer barriers to discarding tools. We have heard the following several times: “We can’t stop using tool (or process) X because we have invested so much in it.”

We have seen thousand-person multisite development groups successfully apply large-scale Scrum with some Excel spreadsheets for their Product Backlog and Release Burndown chart. Indeed, they are almost certainly better off for doing so; it keeps their attention more on fixing the system.

Also, there is a more subtle, pernicious danger with agile-management tools. These are the fifth and eleventh agile principles:

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

11. The best architectures, requirements, and designs emerge from self-organizing teams.

A theme in Scrum (and other agile methods) is self-managing teams, as covered in the *Teams* chapter in the companion book. And the fifth principle emphasizes *trust* and *support*, which is quite different from monitoring people's work. So what?

The agile-management tools we have seen emphasize tracking and displaying individual and team tasks and Sprint Backlogs to managers—almost the antithesis of these principles. In Scrum, the team's tasks (the Sprint Backlog) are created by the team to help them self-manage, not to report their status to others. As the well-known team researcher, Richard Hackman, explains, “In self-managing teams, the responsibility of tracking the progress is delegated towards the team” [Hackman02]. Since the team is self-managing, they are not to be tracked or monitored; such tools are a slippery slope that may reinforce a traditional command-and-control culture rather than a culture of self-management.

We know a coach who works for an ‘agile’ tool vendor. He told us that they had been joking about adding a “*real Scrum*” button to their tool. This button would turn off all the non-Scrum and unnecessary features that were requested by their traditional-management clients...and there would be almost nothing left in the tool.

There is a well-known case of a company where project managers inspected daily the Sprint Burndown charts of teams, and “solved the problem” when the charts did not go down. Ken Schwaber—the Scrum co-creator—was visiting and noticed that all the burndown charts had almost no deviation between the burndown and ideal lines. Eventually he discovered that a team kept two charts: a fake one for the managers so that they would stop interfering, and a real one to support self-management.

Computerized management-reporting tools can also take people away from gemba and the practice of Go See. Lean thinking emphasizes—to understand what is really happening—go with your feet and see with your eyes at the real place of work, help solve problems there, and build relationships with the workers there.

Finally, these tools are optimized for *reporting*—not for success, improvement, or a better flow of value. What *meaningful* problem do they solve?

EARLY DAYS: TEAM & MANAGEMENT CHANGES

Try...Transition from component to feature teams gradually

Over the years that we have been involved in the transition to feature teams from component teams (in large groups involving hundreds of people), we have seen several strategies—and not always smooth. In *Feature Teams* in the companion book we shared two:

- big-bang reorganization
- gradual expansion of component teams' responsibility

see the Feature Teams Primer chapter

The first strategy can work better than one might expect, but not many organizations want to take that plunge because the change is big and they consider it risky. Plus, it *is* a challenge in a 20-year-old multisite product group with 100 long-established component teams. The second strategy does not work that well, because it creates both the drawbacks of feature and component teams.

"Try...Component guardians for architectural integrity when shared code ownership" section on page 314

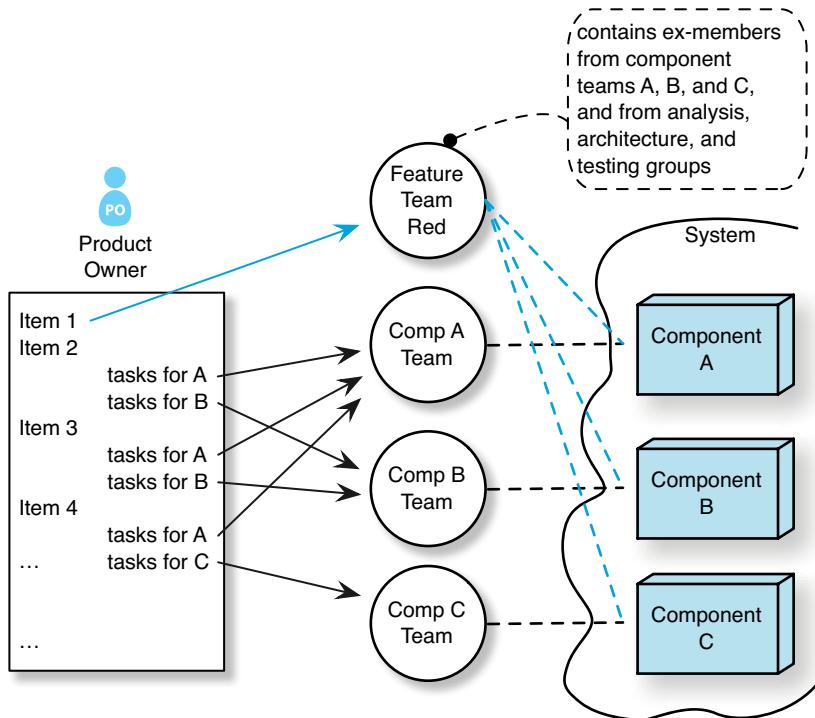
Another strategy we have experimented with (not described in the companion book) is the gradual introduction of feature teams, *applied only to the most important new customer features*.

"Try...Component mailing lists" section on page 314

For instance, take the most important new feature, *item-1*. Form one new cross-component and cross-functional feature team, *Team Red* (Figure 11.1), by extracting only a few members out of existing component and single-function teams (such as analysis and testing). The old teams remain, slightly smaller, and Team Red is born: starting life by working on item-1. In this way, new high-value work benefits

from the speed and simplicity of feature teams, while change impact is softened.

Figure 11.1 a gradual transition from component to feature teams, focusing on the most important features



stable teams: see the Feature Teams and Teams chapters in the companion book

Note!—Team Red is not a temporary project group formed only for the purpose of feature-M. We are not suggesting the traditional practice of resource management with resource pools for short-term work groups. Rather, Team Red is a new *stable team* that will stay together for years; feature-M is but the first of many features they will eventually do.

Disadvantages—This approach also has drawbacks. The first, broadly, is conflicts caused by having two ‘competing’ organizations in place at the same time...

- feature teams change code that component teams own
- the analysis and architecture groups lose ‘control’ over deciding how to implement a feature, and the test group over the testing

The second drawback is that this approach is slow—not a major problem for big product groups that are around for a long time!

Avoid...Waiting for the organization chart

Official agreement on changes to the organization chart for a reorganization to cross-component and cross-functional teams can take a *long* time—especially in long-established large groups. In the groups we work with, the successful strategy is to not wait for that, but to immediately and informally create new cross-functional Scrum teams by dispersing the old teams. The existing line managers (say, a test manager) then have people ‘reporting’ to them from multiple teams. Usually, after some months, the organization chart catches up.

What about the prior line managers, such as the test-group line manager? They may become line managers of several new cross-functional cross-component Scrum teams.⁸

Avoid...In-line ‘ScrumMaster’ line- or project managers

Before adopting agile development, most groups had project managers or line managers. In some, during early days of agile adoption, rather than supporting the emergence of self-managing teams (the 11th agile principle) with a real ScrumMaster, the managers relabel themselves ScrumMasters of their in-line teams—often to meet a top-down target to do Scrum. Avoid that, since a ScrumMaster is not the team’s line or project manager and has no authority over the team they serve; there would be a conflict between having authority and no authority.

see “*Avoid...Fake ScrumMasters*” in the companion

“*Avoid...Scrum-Master coordinates*” section on page 197

On the other hand, some line managers can serve as excellent real ScrumMasters—they may have the right skills and servant-oriented character, they may have some influence in the organization, and this role increases their focus on improving the system. How to

Try...Line manager as Scrum-Master of out-of-line team

8. This assumes that the new teams report to a line manager, which is not required by law nor in a self-managing organization; see the recommended readings in the *Organization* chapter of the companion book for companies that do not organize in a hierarchy.

resolve? In some groups at Xerox, for example, a line manager of team-A *offers* to serve as a ScrumMaster for *out-of-line* team-B; team-B decides on the offer. The two points are (1) it is an out-of-line team, and (2) ScrumMasters are chosen by the team, not imposed.

EARLY DAYS: BREAKING BARRIERS & HABITS

Try...Break the walls—team areas with whiteboards

ScrumMasters remove barriers for teams. At Valtech India, when we saw the cube farm on the left, we arranged to gut the interior of the building, and create team areas with plenty of whiteboards.



Try...Two-week iterations to break waterfall habits

Although Scrum allows iterations of up to four weeks, this is seldom advised or practiced. The *Scrum Guide* suggests:

Tip: When a Team begins Scrum, two-week Sprints allow it to learn without wallowing in uncertainty. [Schwaber09a]

When we started coaching large-scale groups in Scrum, we assumed that four-week iterations would be useful to gradually “lower the waters in the lake.” What we discovered, however, was that four weeks is *just* long enough to maintain old habits: sequential life cycle practices, the existing single-function teams, and handoff between groups. Consequently, there was no strong force for out-of-the-box thinking or transformation to a profoundly different organizational design with concurrent engineering, continuous integration, feature teams, and so on.

But, two-week iterations—with the goal of getting items really done according to *done*—do not readily allow for old habits. Things have got to change—dramatically.

A similar suggestion, for other good reasons, is found in the first book on scaling agile development:

*Although you may have heard otherwise, the larger the team is, the more important **short** cycles are. The reason is simple—if a large team takes a completely wrong course from the entirety of its three-month development cycle, the cost of correcting the course will be enormous. And even if the team took the correct course, it wouldn't benefit from the frequent feedback that is possible with short development cycles. [Eckstein04]*

Try...One flip chart for tasks of one Product Backlog item

Figure 11.2 shows a common-style Sprint Backlog, with one row of task cards for each Product Backlog item, and three columns: *to do*, *underway* (meaning, WIP), and *complete* (meaning, done).



Figure 11.2 Sprint Backlog—rows for each item, columns for *to do*, *underway* (meaning, WIP), and *complete*.

In the early days of a big-group adoption, a coach will notice—by looking at this display and in the behavior of the team—two symptoms of old habits:

- Many tasks cards at the same time are in the *underway* column—there is high WIP.
- Key point—task cards for *multiple backlog items* are in the WIP column *because people are thinking “I only do my special tasks.”*

For example, “I am an interaction designer. I have finished *my* interaction design tasks for item-1. Therefore, no more tasks for me in item-1, so I will start on *my* interaction design tasks for item-2.”

Team members have *primary specialities*, and will do tasks in those areas, but when those are finished, the idea is for team members to take on other tasks of *the current item in progress*, in less familiar areas—perhaps in an area of *secondary speciality*. This both reduces WIP and increases multi-area learning.

A visual management technique to encourage this is illustrated in Figure 11.3. Now, the Sprint Backlog is spread across a set of flip chart posters. Each Product Backlog item has task cards on a separate poster—and each poster has the three common columns: *to do*, *WIP*, *done*. Now—key point—the team displays only one or two posters on the wall at a time;⁹ the other posters (items) are out of sight. Then, the *whole team* focuses on getting one item at a time *done*, increasing learning and reducing WIP.

9. Two items may be in progress either because each is so unusually small that the entire team cannot realistically work on one item together or because something is blocked.

Early Days: Gatherings

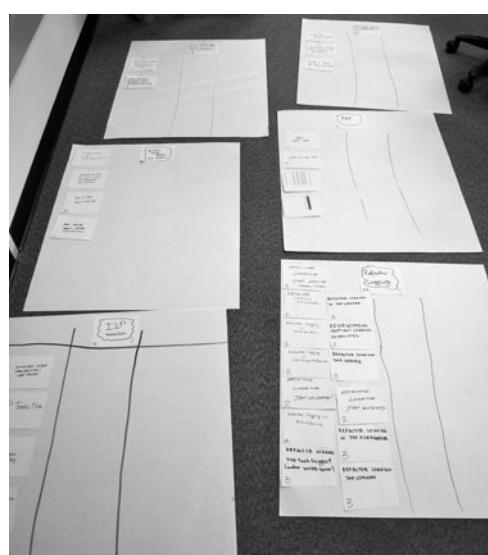


Figure 11.3 one flip chart for each item

EARLY DAYS: GATHERINGS

Try...Repeating large-audience introductions

When there are tens of thousands of people in a company, it is useful to convey a consistent introductory message to everyone. One technique is written material, but that is low-impact—few read it, and the nuance of “bringing Scrum to life” is lost.

Frequent one-day large-audience seminar introductions (say, 200+ people at a time) make a bigger impact—due to immediacy, Q&A, and especially the many ‘discussions’ that take place during coffee and lunch breaks. These seminars *break the ice* and *add some steam*.

Try...Open-Space Technology for early-days adoption



From India to Hungary to the USA, we have seen the positive impact of using Open Space Technology (OST) [Owen97] during the early days of large-scale Scrum adoption within groups. We usually serve as facilitator, starting by announcing the theme of “agile adoption at companyX,” explaining the time-space board, and briefly sharing the OST principles and laws.

OST is a meeting technique that encourages emergence and self-organization; it is highly complementary to agile principles and Scrum, and we encourage groups to experiment with it in multiple contexts: early days, Scrum-of-Scrum meetings, and more.

Figure 11.4 OST
early-days agile
adoption events:
Budapest and
Bangalore



Try...Big gatherings to share stories & experiments



During the first few years of Scrum adoption at one of our clients, we helped organize an annual internal Scrum Gathering in which

hundreds of people from around the globe came together to share stories and tips, listen to expert speakers, and so forth. This sustained and added momentum to the adoption.

COACHING & COMMUNITY

Try...Central coaching group

In some of the enterprise-wide adoptions that we have seen, an internal agile or lean coaching group was established, consisting of hands-on agile experts who go and work directly with teams. Try that.

Form a cross-functional coaching group to learn the diversity of perspectives and issues and to build support for change in more diverse areas. For example, include product management, software development, hardware development, field service, sales, manufacturing, marketing, and more. That said, in the early days of adoption, the focus is typically within R&D and product management, so the original scope of coaches is usually limited to these areas.

Avoid...Central coaching group with formal authority

Caution—Avoid a group that has formal authority to mandate practices, policies, and processes. Rather, create a group that focuses on coaching people interested in adopting agile or lean development.

Try...Concentrate the coaching on a few products

Genuine learning and change of behavior within a product group takes a lot of coaching and time. Plus, misunderstandings are easily created without sufficient coaching. We have seen product groups flounder because they received only a smattering of occasional education. It is better to concentrate the attention of the internal coaching group—supplemented with external coaches—on a few products. Only move on to new groups after solid mastery in old groups.

Try...External agile coaches

Good external agile or lean coaches are worthwhile because they bring fresh perspectives and ideas, sometimes have more credibility than internal coaches (even if not justified) and can therefore make a quicker change-impact, and they can “speak the unspeakable.” Also, ...

Try...Pair external agile coaches with internal ones

When external coaches visit, pair them with internal coaches. There are several advantages, including

- *learning from each other*—for example, the external coach will learn things about the enterprise—policies, politics, and so forth—that would otherwise be difficult or slow to grasp
- *increased learning in the broader coaching network*—the two coaches connect each other to broader networks (internal and external) which share and learn from one another

Avoid...Advisors/consultants who are not hands-on coaches

Big companies often have a centralized process or improvement group. The people working in this area sometimes drift away from doing hands-on development and become *PowerPoint process consultants*. Avoid people like that in an agile or lean adoption initiative. Similarly, watch out for consultants or coaches who may not have read the foreword to the four agile values:

*We are uncovering better ways of developing software by **doing it** and **helping others do it**. (emphasis added)*

Some ‘agile’ consultants do not directly develop software with the teams—coaching agility and lean thinking at *gemba*. Rather than *doing it* with hands-on developers and practicing Go See, they sit in rooms presenting or reviewing process diagrams that may have little to do with what is really happening, or they write emails speculating about problems and their solutions. Managers and consultants may be pleased with the *agile PowerPoint process*, but the reality on the ground is different.

Instead, develop a cadre of internal and external agile/lean coaches who apply Go See and who are masters of the real value work (programming, testing, ...). These coaches and consultants spend most time with engineers while coaching, and only occasionally leave *gemba* to meet with senior management—bringing their insight of what is *really* happening at *gemba*.

Try...Structured intensive curriculum for all teams

For example: At one of our clients the focus is on lean development plus agile engineering practices. In collaboration with management, we set up (and coached) the following curriculum for development people (organized by team). There are intervals of several weeks to several months between each step:

1. Short warm-up e-learning (web-based) courses that focus on basic concepts and terminology related to lean thinking.
2. Lean development-1 (LD-1): Five days in classroom with class projects, with an emphasis on hands-on doing.
3. LD-2: Five days in a structured workshop with teams, applying the skills from LD-1 to their real products, and learning some new skills. A coach mentors. The workshop is in a separate location from their normal work environment.
4. LD-3: For five days, a coach visits the team at their normal work area, reinforcing LD-1 and LD-2 skills in the context of their day-to-day work, doing pair work, and facilitating workshops (such as Sprint Planning).
5. LD-4: Same as LD-3.

Thousands of people are involved in this multiyear coaching endeavor, and the leadership's commitment to in-depth meaningful lean and agile coaching is an illustration of the foundation of the Toyota Way: manager-teachers who have long-term constancy of purpose with lean thinking.

Avoid...Internal agile/lean cookbooks

"Let's write an internal agile cookbook so that all the people can better adopt agile development in our company." It sounds like a good idea: more efficient, more harmonized, ... But we have seen—through Go See with the teams—the subtler dynamics at play...

- ❑ It reduces critical thinking—people assume that if something is written in a corporate-sanctioned guide, then it is good.

- ❑ It reduces challenging the status quo—people assume that what is written in corporate guides should be accepted or followed, rather than challenged.
- ❑ It reduces learning, especially *good agile/lean learning*—high-quality agile, lean, and Scrum teachings have been written in *books* by founding thought leaders; but rather than study these *original sources* for good learning, people assume that secondary corporate guides contain reliable insight.
- ❑ (Related to prior point) it increases misrepresentation—in the interest of ‘harmonization,’ internal process writers *revise* these systems... “let’s remove *self-organizing teams* from our agile description—people won’t like that.”
- ❑ It reinforces the corporate illusion that system problems can be solved with processes and process documentation.
- ❑ If there is an internal group that only writes documentation, and the people in this group do not do hands-on agile coaching, then (1) what is written is undesirable because it is not based on experience, and (2) it perpetuates more overhead work away from gembas.

A group at Toyota described their early documentation effort, and what Taiichi Ohno thought of that:

So we went to work on preparing a systematic description of our [Toyota] production methodology. ... Ohno, of course, hated that kind of deskwork. If he saw people poring over written work like that, he’d tell them to get out onto the plant floor. So the team couldn’t do its work within his sight... [SF09]

CONTINUOUS IMPROVEMENT

This section has two categories:

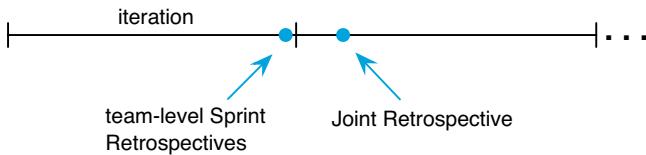
- ❑ multiteam coordination, such as a Joint Retrospective
- ❑ other general experiments

Multiteam Coordination Experiments...

Try...Joint Sprint Retrospectives

An iteration ends with an individual team Sprint Retrospective, where the focus is team-level improvement actions. In large-scale Scrum there is the bigger system to inspect and adapt. For this, experiment with **Joint Retrospectives** each iteration.

When?—Since the iteration ends with a team retrospective, most of our clients hold this early in the first week of the subsequent iteration—when the issues of the previous iteration and recent team-level retrospectives are still fresh in mind.



Who?—In general, one or two representatives from each team. Since ScrumMasters are closely involved in understanding and helping improve the system, they are good candidates. However, avoid ScrumMaster-only meetings; this gives the wrong impression that ScrumMasters are solely responsible for improvement (rather than other team members too), and it increases bias during the workshop.

Scope of teams?—This depends on the scale: If there is only one small 10- or 20-team group at one site, one Joint Retrospective with representatives from all teams suffices. If it is larger and there are *requirement areas*, then each area is a good scope for a retrospective. Because many issues are site specific, a site-level retrospective is also useful: one in Curitiba, one in Chengdu, and so on. Finally, for larger groups, experiment with a top-level Joint Retrospective (above the site and requirement areas); in this case, it is most often a multisite retrospective.



Where?—Use a *big room*, with lots of whiteboards since there may be dozens of people in a Joint Retrospective. See the *Multisite* chapter for tips in that case.

"Try...Requirements workshops for Product Backlog refinement" section on page 243

"Try...Coordination working agreements" section on page 212

How?—As with any retrospective, *variety* of workshop activities over time is a guiding principle. Broad suggestions:

- Try Open Space Technology [Owen97], World Café [BI05], and Future Search [WJ00] for Joint Retrospectives.
- Apply the *diverge-merge* pattern—useful in any large workshop.

What?—Too often, a retrospective focuses only on problems. Experiment with sharing what is going *well* for a site or team, that others may try. This is the *yokoten*—spread practices laterally—approach used at Toyota. A joint retrospective is also a time to review and change existing *coordination working agreements*.

Try...Joint Retrospective big improvements in Product Backlog

Major (expensive) improvement ideas are added to the Product Backlog so that they are visible to—and prioritized by—the Product Owner. This is even more important when there are intermediate Joint Retrospectives below the overall product level. For example, suppose there are 20 teams in Curitiba (Brazil) and 20 teams in Chengdu (China). Each sub-group holds its own site-level retrospective and identifies the same major improvement goal. These need to flow into a common list, the backlog, to prevent duplication and so that the Product Owner sees cross-site problems.

And who takes on this work? An existing feature team.

Note—This relates to other suggestions in this and the companion book. If the improvement goal involves common software, this leads to a feature team working on shared infrastructure (see *Feature Teams* in the companion). If it involves creating common test-auto-

mation testware, this leads to a feature team doing test automation (see the *Test* chapter).

Try...Cross-team working agreements

External-to-team **working agreements** usually define how teams agree to work together; for instance, holding a *joint design workshop*. They may or may not be product-wide; a subset of teams that work together frequently can have their own agreement. They are defined or evolved in Joint Retrospectives.

"Try...Coordination working agreements" section on page 212

Try...Joint Sprint Reviews

A Joint Retrospective is vital to inspect and adapt the system-level ways of working. Similarly, a **Joint Review** is pivotal to focus on inspect-and-adapt for the overall product. At one of our large-group clients, the last day of the iteration runs as follows:

1. Product-level Joint Review—The overall Product Owner (PO) and supporting PO representatives are in meeting rooms around the world, all linked together with video conferencing and shared desktop technology. There are also representatives from various teams.¹⁰ What is presented? A subset of items that are of special or overall interest to the entire product group. What is discussed? Issues relevant to the overall product.
2. Single-team Sprint Review or multiteam Joint Reviews—When a supporting PO representative is served by only one team, a standard Sprint Review occurs. When the PO representative is served by several teams or the Area PO is involved, we have seen clients either (1) stagger the Sprint Reviews so that the PO representative or Area PO meets separately with each, and (2) a Joint Review with several teams together.
3. Single-team Sprint Retrospectives.

10. With the exception of Joint Retrospectives, we discourage Scrum-Masters from acting as representatives, to avoid giving the wrong impression that they are the team representative or manager.

A review bazaar—A Sprint Review involves conversation, not only a demonstration of the product; nevertheless, showing the running system is important. One technique applicable to a Joint Sprint Review is a *bazaar* [Schatz05], analogous to a *science fair*: A large room has multiple areas, each staffed by team representatives, where the features developed by a team are shown and discussed. Members of the Product Owner Team and Scrum teams visit areas of interest.

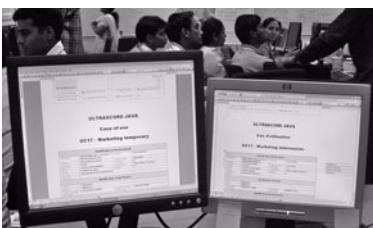
Avoid...Try...Individual team-level Sprint Review

If an individual team has its own separate Sprint Review, there is a danger—one that we have seen in action—that the team focuses on ‘their’ result instead of the overall product created by all teams together. This leads to a loss of systems focus and an increase in local sub-optimization. Avoid that. However, a Joint Review does not review all items developed during the iteration (since there are so many), and the team that developed a feature might need detailed feedback from their Product Owner. If separate reviews are held, people need to watch out for a loss of product-level focus.

Other Experiments...

Try...Spend money on improving, instead of “adding capacity”

Very large product groups become large because their default response to delivery-speed problems is to hire more people. Avoid that, and in contrast, apply the lean-thinking strategy of removing waste to improve the flow of value—reducing handoffs, WIP, and so forth. *Note that the approach is more subtractive than additive.* Often, this waste removal does not even incur additional capital investment or operating expense.



And yet, spending more money (“increasing cost”) can contribute to improving—without using it to hire more people. For example, when I (Craig here) started working at Valtech India, I noticed that people had only one small monitor. Research suggests improvements if

people have more than one [Atwood08], so we bought a second monitor for everyone.

Other common—and valuable—examples include hiring expert coaches who mentor people, and classroom education with great teachers.

Try...Lower the waters in the lake



One metaphor for continual improvement—sometimes used in lean thinking—is the *lake and rocks*.¹¹

How to work toward flow of value to customers and continually improve? Do this by gradually lowering the waters in the lake. The water level symbolizes the amount of *inventory*, *WIP*, *batch size*, *handoff*, or *cycle time*.¹² That is, gradually decrease their size. As they grow smaller—as the water level lowers—new rocks hidden below the surface of the water are revealed. These represent the weaknesses and impediments in the system.

For example, perhaps a group first moves from a long two-year sequential life cycle to a four-week timeboxed iterative cycle. Some outstanding weaknesses in the system—the biggest rocks—will become painfully obvious; for instance, lack of automated tests and efficient integration. The group works on these big visible rocks; eventually they shrink in size. Then, as discussed in the “Try...Two-

11. This metaphor was also presented in *Queuing Theory and Lean Thinking* in the companion book.

12. These are interrelated; for example, a big batch means more WIP.

week iterations to break waterfall habits” section on page 394, the cycle time is lowered to two weeks to confront deeper problems.

Especially in large traditional groups there is a *massive* pile of rocks. The scale of improvement work can seem overwhelming. The strategy behind this metaphor makes the work tractable, while also signifying that kaizen is never finished.

Avoid...Rotating the ScrumMaster role quickly

It takes study and practice to become an effective ScrumMaster—at the very least a year. And a ScrumMaster ought to focus on organizational change—and that requires long-term constancy of purpose.

If the role is rotated quickly within a team, that necessary period of practice is missing and the organizational-improvement focus is missing or diminished. Therefore, do not rotate the position quickly.

On the other hand, a learning self-managing team should not be forever reliant on one person for this skill, and different team members should eventually have the opportunity or challenge to grow as ScrumMaster. Rotate the role—very slowly.

Try...Reduce harm of policies that cannot yet be removed

“We know that performance appraisals and performance-based incentives weaken the system, but we can’t do anything about them—they’re mandated by HR.” We hear variations of this from some people who then want to give up trying to improve the system. But Scrum encourages *the art of the possible*. With creativity, the harm from various policies can often be reduced. And possibly sometime in the future, eliminated.

For example, Bas used to work in an organization that mandated performance reviews, targets, and bonuses. When he met with people that reported to him, instead of focusing on performance in their ‘normal’ work, they set targets related to learning, such as reading books and giving presentations. During the next review, they talked about the learning and how it applied at work. One person told Bas

that nobody believed it when he told friends that he got a bonus for reading books.

Similarly, if performance-based rewards are mandated, perhaps they can be shifted to team-based goals so that there is a reduction in competition and an increase in cooperation.

CONCLUSION

Gandhi (at least as reported by his grandson Arun) once said, “We need to be the change we wish to see in the world.” This is equally applicable to the world of work—an agile adoption needs agile adoptees. Scrum and lean development cannot be successfully adopted with command-and-control management, predictive planning, or process recipes or “best practices” coming from ivory towers.

Even when those involved in an agile adoption have a conducive mindset, a repeating problem we have seen is a lack of Go See behavior, and therefore, a lack of insight into the real problems and useful solutions. How many product leaders or process engineers spend time regularly sitting with developers while doing the real hands-on work? Without that experience, initiatives have little useful impact; they can also focus in the wrong area—on management-level ‘improvements’ rather than at *gemba*.

Scrum, lean, agile development: these are never finished being adopted. *Agile is not a change project*. Rather, continuous improvement is a pillar of lean thinking, coupled to the idea that the people best suited to create improvement experiments are the workers.

Naturally, hands-on workers at *gemba* also have limitations. All people—including us—get stuck in inside-the-box behaviors and beliefs that inhibit challenging the status quo. So, in a lean enterprise, manager-teachers who deeply understand lean thinking, who have constancy of purpose, and who inspire kaizen mindset in others are a key positive force to promote and sustain a culture of agility.

But meaningful change and improvement cannot rely on manager-teachers; it relies on...us.

RECOMMENDED READINGS

- *The Birth of Lean*, edited by Shimokawa and Fujimoto, offers a glimpse into the evolution and adoption of lean production and thinking at Toyota. For example: “*At a time when all of us are struggling to implement lean production and lean management, often with complex programs on an organization-wide basis, it is helpful to learn that the creators of lean had no grand plan and no company-wide program to install it.*”
- *Fearless Change: Patterns for Introducing New Ideas* by Mary Lynn Manns and Linda Rising comes from authors with experience in change initiatives and knowledge of agile development; they emphasize a bottom-up approach to change.
- The site www.solonline.org, from the *Society for Organizational Learning*, contains many learning resources and recommended readings related to organizational improvement.
- Taiichi Ohno, in his *Workplace Management*, conveys a sense of the importance—for creating a lean culture—of leaders who truly grasp lean thinking, and relentlessly coach others in this.
- There are several good (and more bad) books on team building; some are of the better ones are recommended in the *Teams* chapter of the companion book. Two mentioned in this chapter include *The Five Dysfunctions of a Team* and *Overcoming the Five Dysfunctions of a Team* by Patrick Lencioni.
- *Teamwork Is an Individual Skill: Getting Your Work Done When Sharing Responsibility* by Chris Avery emphasizes taking personal responsibility for creating an effective team, and shares tips for how to do so.
- *The Fifth Discipline: The Art & Practice of The Learning Organization* by Peter Senge, is a classic in systems thinking, learning, and the qualities needed by effective leaders for sustainable, high-impact organizational improvement.
- *Agile Retrospectives: Making Good Teams Great* by Esther Derby and Diana Larsen covers core retrospective skills. And *Project Retrospectives* by Norm Kerth explores how to do retrospectives with larger groups.

Continuous Improvement

- ❑ *Agile Coaching* by Rachel Davies and Liz Sedley captures many practical tips for ScrumMasters and other agile coaches, from two experienced coaches.

Chapter

- Thinking about Multisite 414
- Team Structure and Sites 417
- Interaction & Coordination 423
- Multisite Culture & Norms 437
- Tools 438

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

MULTISITE

Folk who don't know why America is the Land of Promise should be here during an election campaign.
—Milton Berle

In multisite work, all the usual Scrum events and practices apply.¹ Large-scale distributed development has been done with Scrum for years; we have seen and coached it at many clients. Others have also reported success; see the *Recommended Readings*.

The following terms are used:

multisite development—One product group at two or more sites, also known as **distributed development**.

dispersed team—One ‘team’ (for example, of seven people) with members in different ‘sites’; also known as a **distributed team**, although other meanings are also ascribed to this latter term. *Different sites* exist along a continuum from “different rooms or cubicles” to “different planets” and so likewise the dispersion of a group exists along a continuum.

co-located team—A team together in a team room. Working in the same city will not qualify as ‘co-located.’

Erran Carmel, a long-term researcher in multisite development, does a good job of laying out the big-picture problems in *Global Software Teams* [Carmel99]. He identified five *centrifugal* forces that

1. Suggestions related to multisite agile development are occasionally explored in other chapters, especially *Offshore*.

pull people, teams, and a product group apart and inhibit performance in distributed development:

- geographic dispersion
- coordination breakdown
- loss of communication richness
- loss of teamness
- cultural differences

Carmel also discusses some practices that ameliorate the centrifugal forces of multisite development, including:

- collaboration technologies
- common process framework—
shared concepts and vocabulary;
a shared iterative life cycle is
most effective
- building trust, communication,
and personal bridges

The following experiments (and *Offshore* chapter suggestions) address the centrifugal forces and reiterate the successful solutions summarized above—and more.

THINKING ABOUT MULTISITE

Try...Fewer sites

We know of one product group that is spread across 13 cities. Ouch! Challenge the status quo assumption that many sites are needed. Since multisite development brings with it a host of complications, reduce rather than increase the number of sites.

Sometimes, there is a silo between the group that decides *site strategy* and the R&D division adopting large-scale Scrum; the former may not be aware of the implications of agile development on team structure and life cycle—and hence, on site strategy. Sometimes, two products are merged or another product company is acquired. In all these ways, the number of sites increases. As an agile coach, actively communicate with stakeholders so they are clearly aware of the negative impact of an explosion of sites...and blowing up R&D.

Try...Think ‘multisite’ even when close

Do not assume that people need to start reckoning with multisite agile development only when groups are far apart in two cities.

The impact of distance does not start after hundreds of kilometers. Multisite issues rear up as soon as teams are a short distance apart—and short is *short*... We have seen people at opposite ends of the same floor ‘review’ a design idea via email rather than *moving* together to a whiteboard, because of the inertia related to distance.

Most of our work has involved introducing Scrum in multisite product development spread over at least one continent, and more often, several. Certainly the impact is clear across a continent, but what is clear to us—when working at a single campus—is that ‘multisite’ issues start to be felt as soon as teams are in two separate buildings or zones within a building, especially if walking between them takes any meaningful time.

Chair inertia is a strong force.

Avoid...Believing in multisite *Daily Scrum* magic or that multisite forces are inconsequential

Beware any report that suggests successful large-scale multisite agile development does not take *special* attention. Some may claim that distribution is no longer a significant factor in large-group development thanks to improved telepresence and communication technologies, especially when combined with the potential benefits of agile practices. While it is true that technologies mitigate some problems, still, the most comprehensive research on the subject concludes that distance will *always* have some negative impacts [OO00]. And a claim of easy “large group” multisite success may be for a small 50-person group rather than a 500-person group. The COCOMO data on productivity and systems development show that multisite has a non-trivial impact [Boehm00a]. Another study indicated that multisite development takes at least twice as long as co-located development [HM03].

Anecdote: The product development expert Don Reinertsen told us (and wrote) that he has informally polled thousands of development

people over the last decade and *not once* has he found a hands-on group that, having had both the contrasting experience of co-located versus distributed development, would choose the latter again [Smith07]—and this result even in the Internet Age.

Even with advances in virtual presence, time zone differences are a hard constraint—there is the lost potential benefits of *synchronous* communication to reduce the wastes of delay, misunderstanding, and information scatter, and to help form real relationships... *it is never a good idea to marry based on email.*

The upshot is this: Do not believe that multisite-specific issues are inconsequential in large-scale development or *magically resolved by holding a distributed Daily Scrum*—that will not solve the big problems. Centrifugal multisite forces are strong and require *concerted long-term attention*. To quote:

...we have no record of companies which would indicate that it is easier to do R&D on a global scale than in a geographically centralized approach. Globalization of R&D is typically accepted more with resignation than with pleasure [DM89].

Avoid...Thinking ‘distributed’ must mean ‘dispersed’

There is a misunderstanding about “distributed agile development.” It is related to the *Agile means small* misconception: It is the incorrect belief that “distributed agile” must mean that one team is dispersed across multiple sites—one member in Dallas, two members in London, and so forth. That understanding would indeed be true if the total product group were *seven* people. But the context of large-scale Scrum is more likely a 200-person product group with 25 teams—or at least a 14-person group with two teams. In this context, there are a set of co-located feature teams at site-A, a set of teams at site-B, and so forth. No one team needs to be dispersed across sites, though dispersed teams are an option.

Avoid...Thinking distributed pair programming is required

This misunderstanding arises from the *agile means XP* and the *distributed agile means dispersed team* misconceptions. Pair program-

ming is only an XP practice; it is not required in Scrum. And with no requirement for dispersed teams in multisite development, there is no requirement for distributed pair work—including distributed pair programming.

That said, distributed pair programming is useful when a dispersed team cannot be avoided, or when some expert knowledge needs to be shared. For example, a component guardian that is not part of a colocated team and that is located at a different site may want to pair-program with a member of the team. If so, this is trivially easy with a shared desktop tool and Skype (for example²).

"Try...Component guardians for architectural integrity when shared code ownership" section on page 314

Try...One iteration (Sprint) for the product, not for the site

An iteration or Sprint in Scrum is for the entire product, with the perfection challenge of creating a potentially shippable product increment at the end of each two- or four-week timebox. There is no separate and independent iteration unique to each site. If there are 200 people in Chengdu, 100 in Hyderabad, and 200 in Dallas for one product, the implication is that at 17:00 on Friday Aug 1 (Dallas time, for example) the *one* iteration for all 500 people ends, and the product could be shipped. Likewise on Friday Aug 29. This of course demands worldwide continuous integration, massive test automation, feature teams, and one repository.

TEAM STRUCTURE AND SITES

Avoid...Sites organized by components or functions

The weaknesses associated with component teams were explored in the *Feature Teams* chapter of the companion book. These are compounded when sites are organized by components (subsystem, module, ...) and component teams, because of degraded communication and other impediments. Similarly—and as a corollary to avoiding sequential life cycle development—avoid single-function groups

-
- 2. Avoid *commercial* tools, because they inhibit widespread long-term use, especially in a multisite context with new or “low cost” sites.

(*testing group in Bangalore, architecture group in Atlanta, ...*) distributed in different cities.

Rather...

Try...Allocate a whole feature to a co-located feature team

Key point: *The dominant problems in multisite development arise precisely because of the organizational design choices of poor life cycle, task allocation, and team structure.*

Key management decisions (life cycle, task allocation, and team structure) will make things much worse or better in multisite development. For example, a sequential life cycle imposes a site/team/task strategy that reflects the phases: requirements in Munich, design in Boston, programming in Hanoi, testing in São Paulo. This is of course *fundamentally inconsistent* with agile principles and practices, is associated with dramatic inventory and handoff waste from a lean perspective, is correlated with long delays and disastrous average cycle time as shown in queueing theory, and inhibits valuable feedback from an information theory perspective. So that model will not be discussed further.

On the other hand, suppose management believes they are doing timeboxed iterative development. One possible team structure and task allocation strategy is *component teams* and tasks organized by components. The many weaknesses of this model were examined in the *Feature Teams* chapter of the companion book. These are further aggravated in distributed development, because a customer feature—the value-goal of lean and agile development—does not map to one component, and so completing a feature involves delay and coordination among many component teams that may now be half-way around the world. Further, as was analyzed in the *Feature Teams* chapter, component teams enforce the existence of sequential life cycle development, with separate upstream requirements and design groups, and separate downstream component/programming and test groups. Back to the ‘waterfall’ or V-model. All of the problems of sequential development exist but are now compounded by the centrifugal forces of distribution. For example, with component

teams there will be a separate testing group and they might not even be on the same *continent* as the component teams.

The above apparent iterative life cycle model is simply a degeneration into a series of slow mini-waterfalls because of the task allocation and team structure decisions—a far cry from the implication of parallelism and speed in Scrum and other agile methods.

Co-located feature teams eliminate these problems and are one key to multisite large-scale Scrum. When a complete customer-centric feature is given to one cross-functional, cross-component, co-located, self-organizing feature team (that is, a normal Scrum Team)—and they do the planning, analysis, design, programming, and testing for the feature—there is no need for management coordination with other groups and no need for project management overhead. That being so, the major multisite problems,

- geographic dispersion
- coordination breakdown
- loss of communication richness
- loss of teamness
- cultural differences

are eliminated or ameliorated. One product group may have 10 feature teams in Singapore, 15 feature teams in Kuala Lumpur, and 20 feature teams in Athens, but since each feature team is self-contained and does not require much or any coordination with any other team to complete a feature—except at the level of code, as discussed in the *Feature Team* chapter—multisite development becomes dramatically simpler.

Avoid...Dispersed groups or ‘teams’

Tom DeMarco was asked if a real *jelled* team can exist if people are not together and seeing each other very frequently:

No. Teams have to be together. Remoteness makes the good [community aspects] impossible. Too often we name a group of people a team. They don't acquire teamhood. They have it ascribed to them. [DeMarco95] (emphasis added)

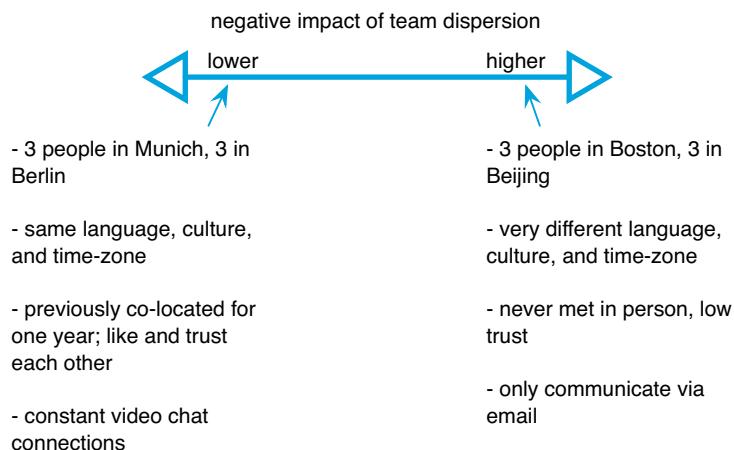
This hits the nail on the head. Very often a *group* of people is artificially labeled a *team* but they have none of the qualities of a *real*

see Teams in the companion book

team. On the other hand, a Scrum feature team has a common goal to build one Product Backlog item. It is not seven people working on their own, but seven people closely coordinating and collaborating with a shared vision and goal. This is awkward and difficult to achieve with a dispersed group; it is hard to say if one could ever call a dispersed group working in different sites and time zones—perhaps speaking different languages—a real jelled team. Therefore, avoid dispersed ‘teams.’

Naturally, the impact of dispersion varies; see Figure 12.1.

Figure 12.1 varying impact of team dispersion



And yet, there can be motivations for a dispersed group...

Try...A dispersed feature ‘team’ only if it really hurts

“Try...Cross-pollination” section on page 432

All things being equal, prefer co-located feature teams. But not all things are equal. For example, there can be a variety of *us-them* problems when a new site is added to an existing product group. Or, knowledge and competency problems: People at the new site do not know much about the product. Plan A is *cross-pollination*. Plan B may be to create some dispersed feature groups or pseudo-teams, intentionally accepting the productivity impact, coordination problems, and loss of teamness to address other weaknesses.

If you must have a dispersed group, encourage teamness by emphasizing *synchronous over asynchronous* communication tools, such as lots of video sessions (including a video-session multisite Daily Scrum), instant messaging, shared desktop pair programming, and so forth.³

Synchronous communication implies that a dispersed team should have significantly overlapping work days, such as no more than three or four time zones apart. It is improbable (impossible?) to form a real *team* that does not participate regularly in synchronous communication.

Try...Gradual transition to co-located Scrum feature teams

Some clients followed a ‘waterfall’ site strategy, thus their starting point for adopting Scrum is...software architecture in Dublin, testing in Budapest, and so on, with 500 people in six cities. Naturally, the move is *gradual* to co-located Scrum feature teams at each site.

What are some transition patterns our clients use? First, the overarching strategy is to focus the team-structure changes gradually around the highest-priority Product Backlog items, where the investment in change has the largest and most immediate benefit. For more, start by reading the “Try...Transition from component to feature teams gradually” section on page 391.

So, new feature teams are slowly created around top-priority items. Here are some experiments for creating these new teams (when there are multiple sites) and for evolving the role of existing teams:

- Using the approach described on p. 391, a new Scrum feature team, Team Blue, is formed of *programmers* from several same-site—say, Bangalore—component teams, choosing people from old component teams such that the new Team Blue has knowledge of at least some of the components involved in the first high-priority feature, feature-X, they will develop. The programmers also take on the responsibility for automated feature
-
3. In fact, we broadly encourage a little more synchronous communication in large groups (rather than email), especially using simple video tools such as Skype. And Intel has tried “email-free Fridays.”

testing, using *acceptance TDD*. Therefore, the existing testing group in Budapest is no longer needed for testing feature-X. Team Blue learns to work in unfamiliar components, perhaps via distributed pair programming with experts from another site (such as a *component guardian*) and/or with code reviews from those experts. Team Blue also takes on the analysis and design work for feature-X, collaborating with the Product Owner to clarify the feature.

- If there is a separate software architecture group in one city, there is a chance that among the group are people that—as in the case of Team Blue above—can form a new feature team and start doing hands-on programming and testing.
- A variation of the Team Blue approach above is to create a *dispersed* team that includes the core feature team in Bangalore and a few test experts from the Budapest testing site. These test experts may participate in Budapest-based requirement workshops, using video technology, and help *distill* the example requirements into automated tests, as part of acceptance TDD.
- Meanwhile, at the Budapest testing site, other test experts increasingly focus on writing fully-automated tests (using the same technology used for acceptance TDD); these tests are included in the multisite continuous integration system.

Try...Temporary co-location of a new dispersed team

If, unfortunately, a dispersed team is needed for some reason,⁴ experiment with first co-locating all the new team members at the same site for several iterations, to build genuine relationships.

Try...Learn at existing sites, rather than add ‘expert’ sites

Suppose that developing some new features involves expert knowledge that John has, and John is in Houston—a location not (yet) among the sites being used for development. Rather than assume it necessary to add the Houston site (and John and his team), consider

4. The context we see most often is when a large multisite group had sites organized by function, such as testing in China, and so on.

if one of the existing teams (Team Green) in the development can learn what is necessary. Perhaps John can...

- fly in and work with Team Green for some time, coaching them
- participate in video sessions to talk or workshop, to help the team learn and create
- review requirements, design, and code created by the team
- do distributed pair programming with Team Green people

Try...Prefer co-location of feature teams and Area Product Owner of one requirement area...but do not restrict this

Just as it is awkward to create one dispersed feature team, it is awkward to disperse the teams of one requirement area. Prefer co-locating related feature teams at a common site. It is also useful for the Area Product Owner to co-locate with the teams, or at least to visit frequently—especially to attend the Scrum events.

On the other hand, this suggestion could degenerate into the sub-optimization (a very literal *local* optimization) of artificially freezing the requirement areas and their teams by site. Do not do that. The number of feature teams may slowly grow or shrink in a requirement area as it increases or decreases in market importance. A requirement area is a customer perspective into the major feature sets of a product; it has nothing to do with a product group's sites or architecture. Consequently, do not restrict a requirement area to one site; simply prefer it.

see “*Introduction to Requirement Areas*” section on page 555 and the *Requirement Areas* chapter of the companion

INTERACTION & COORDINATION

Try...Treat all sites as equal partners

The maturation stages of multisite development involve moving from *Stage II* in which the original headquarter site centrally defines and coordinates the work of other sites, to *Stage III* (globally integrated) in which all sites have essentially become equal partners [Carmel99]. Motivation, trust, and the quality of interaction

increases when ‘child’ sites are treated like equal partners. Signs of this are the physical relocation of the Product Owner or Area Product Owner to the site, and the site participating in driving the vision and architecture of the product.

Try...Continuous integration in “one repository” across sites

The scope of continuous integration (CI) is the product, not the site. Fifty feature teams in four countries need to share one repository and be *continually* integrating *all* code. For example, one radio-network product group (several hundred people) adopting Scrum that we coached has sites in China and Europe. They all share a common Subversion (Svn—the free open-source de facto standard worldwide) repository hosted in Europe that is accessed over a fat and fast pipe. A write or update from anywhere is *fast*.

Although slow Internet connections are becoming a problem of the past in most countries, there are still occasional sites where we see a problem. One 150+ multisite group we coached with a center in Bangalore had this constraint. How to use Subversion (Svn) in this case? Svn is based on a central (single) repository that could be slow to access if hosted far from Bangalore. Their solution was the free open-source Svn slave replication system *Pushmi* [Kao08]. With Pushmi installed, normal Svn writes are transparently pushed immediately to all replicated slave repositories (worldwide), and normal Svn checkouts or updates are quickly obtained from the local slave.

Locking—Do not use a blocking or locking version control system—this is bad enough in co-located development with seven people, and disastrous for multisite CI.

Git and other distributed revision control systems—The heading for this experiment punctuates “one repository” in quotes to signify that there are *distributed* systems that can create a similar *consolidated view*. This family of tools—including Git, Mercurial, and Bazaar—are also called *decentralized version control* systems, and are being successfully used in large multisite developments, including Linux.

ClearCase—As mentioned in the *Continuous Integration* chapter, the most frequent tool tip we observe and hear related to CI and multisite development is to not use IBM Rational ClearCase. The multisite groups that we have worked with (or asked) who tried to do multisite CI with ClearCase found it a challenge. Those that switched to Svn—the usual choice when switching—or Svn with Pushmi report that it is easier and faster to do multisite CI with Svn than with ClearCase, and it saves the group significant money.

“Avoid...Using ClearCase” section on page 362

The tool also slows down a build for multisite CI. One massive product we worked with (with many tens of millions lines of code) reduced their compile time immediately by over 25 percent simply by replacing ClearCase with Svn, because of the slow performance of this 1980s-based tool.⁵

Try...Seeing is believing—ubiquitous cheap video technology and video culture

This revisits the tip made in the *Offshore* chapter, but here focuses on general multisite video communication and Scrum events rather than on the forces particular to offshore development.

“Try...Seeing is believing—video sessions” section on page 451

The sixth agile principle boils down to... *face-to-face conversation*, and the first agile value is *Individuals and interactions over processes and tools*. To build meaningful relationships and improve interaction richness in multisite development, people need to regularly *see* each other.

The cheapest approach is high-quality webcams for the video channel, with a free tool such as Skype. For the audio channel, Skype Audio may suffice, but it is imperative that the audio be *perfect*, without delay or distortion. If it is not perfect, use a telephone call and speakerphone for the audio channel to complement the video.

5. ClearCase is a mid-1980s-architecture commercial tool still sold for multisite development, but it was designed for an old pre-Internet network model by Atria Software from 1989-1991, derived from the Apollo computer system the developers had previously made, called DSEE. It was designed for a 1980s manual and delayed-integration model that makes the agile practice of CI more difficult—and costly—than modern open-source alternatives.

Cheap, simple, and ubiquitous (including in the team room) are important qualities for sustainable videoconferencing with a team. Using an expensive system in a distant room is not a very sticky practice. It is also useful to have a *large-screen projector* in the team room so that group meetings are possible (Figure 12.2).

What about free *three-way* video conferencing? Some alternatives:

- Apple's iChat supports three-way video.
- Use Skype for one two-way video call, and Google Talk for another two-way video call. Then use Skype's "share desktop" feature to include the Google Talk session.⁶

In addition, individual webcams at workstations are useful, including in private break-out or meeting rooms away from the team room, so people have lots of options.

In other words, *ubiquitous cheap video technology* and *video culture*. Try replacing audio multisite communication with audio-video communication. These days, with webcams built into most computers, high bandwidth networking, and free video tools, there is no excuse to remain in the older-generation audio culture rather than an audio-video culture.

All that said about cheap tools, you can still pay for more quality. For example, one of our clients has Tandberg video systems for multisite communication.⁷ Because of expense, these are in shared meeting rooms rather than replicated in dedicated team rooms. This inhibits a video culture, but the fidelity of video and audio is high. A group in Los Angeles can control the direction and zoom of the camera in New York, and vice versa. We have used these to hold multisite joint Sprint Retrospectives and joint design workshops for product groups of over 200 people adopting Scrum for embedded systems development in North America. The systems are on wheels and can be rolled in front of a whiteboard. The fidelity is high; the impression is almost that the other group is in the same room. We can easily see their whiteboard and the people. The picture in Figure 12.3 illus-

6. Other simpler solutions will surely emerge.

7. Coincidentally, some Tandberg product groups are also adopting agile and lean development.

trates the controllable camera, but does not clearly convey that the display is very large and that it rolls on wheels. This latter feature is excellent for joint design workshops at whiteboards.



Figure 12.2 team room with projector and web camera for multisite video sessions, Valtech India



Figure 12.3 joint Sprint Retrospective (multisite) with high-end video technology, USA

In large-scale Scrum, within the constraints of time zone limits, these video technologies can be used for joint Initial Product Backlog Refinement, Sprint Planning Part One, Sprint Review, the joint Sprint Retrospective, joint per-iteration Product Backlog Refinement, and joint design workshops.

Try...Include diverge—converge cycles in large video meetings

Small multisite video meetings, such as two groups of five people at two whiteboards (at two sites), can be held ‘synchronously’ in the sense of all ten people participating as one group. On the other hand, there are opportunities in large-scale Scrum for *large* multisite video meetings. For example, consider a joint Sprint Retrospective with representatives from 30 teams from three cities; perhaps a total of 30 people in the workshop. It is not easy—with current technical limitations—to have a high-value, three-way ‘synchronous’ workshop with all 30 people collaborating as one big group at all times.

Therefore—and this is also true for *co-located* large workshops, since the issue is not multiple sites but largeness of group—let each site spend time alone on an activity, such as 5 Whys root-cause analysis or causal loop modeling. This is a *diverging* period.

Then, have all sites do *show-and-tell*, in which they present their results (such as a whiteboard diagram or flip chart paper) over the video channel to the other sites. This may be followed by whole-group activities (all 30 people from three sites), such as open discussion or more structured whole-group exercises. This is a *converging* period.

Include both converging and diverging periods in large video meetings, and do cycles of these, so that feedback from the converging periods informs the diverging periods.

Try...Start early multisite video meetings informally

We have been involved in many multisite meetings just starting to increase communication by introducing a video tool. Initially, people feel uncomfortable with this new way of interacting and may respond to this by rushing into “business.” It feels stiff. Explore ways to lighten things up, be informal, and form social relationships by starting with non-business chats. Talk about the weather... or better yet, point your webcam out the window and *show* them.

Try...Multisite planning poker (estimation poker)

We have coached and seen planning or estimation poker [Grenning02] used effectively for multihundred-person product groups, both for products with sites in Europe and Asia, and for products with sites around the USA plus in Europe. First, as a theme, consider avoiding specialized web-based ‘agile’ planning tools, as these tend to focus people’s attention on the tool rather than on each other, and because physical tangible tools such as cards tend to energize and engage people better. If you try a specialized software tool for multisite estimation, beware these weaknesses.

Two simple multisite techniques have worked...

With both techniques, the Product Owner has a wiki page of the items under estimation. When (verbal) questions and answers occur for any item, the Product Owner answers verbally and also enters the answer on the wiki page for all sites to quickly read.

Technique: Webcams and normal (physical) planning-poker cards—This technique works well when there are only two or three sites. People use big cards with big thick numbers on them, to increase the chance of their being seen over a webcam. Each site has a webcam so that people can see each group, which is arranged in front of the camera. Note that a webcam may not provide the visual acuity to read the physical planning-poker cards at another site. If so, then each site has a facilitator that asks the local group to show their cards when required. Each facilitator verbally reports the scores to everyone in the multisite meeting if it is not easy to see the remote cards. A shared Google Spreadsheet can show everyone the growing estimate results (see Figure 12.4).

The above is a simple and interactive technique—and therefore worth trying.

Technique: Shared spreadsheet—This technique is useful when there are people distributed at *many* sites involved in the estimation, because five simultaneous webcam sessions with groups of seven people at each of the five sites does not work so well with today’s technologies.

Google Spreadsheet allows dozens of people in different sites to update a common spreadsheet concurrently. We have worked with multisite product groups that hold multisite planning-poker sessions with this and similar tools. Steps:

1. At all computers at all sites people are connected to a shared Google Spreadsheet and they are talking on a common audio channel, such as a conference call.
2. There is one moderator.
3. Each person is assigned a different spreadsheet cell for their poker vote; for example, you might get ‘C20’ as your cell.
4. When it is time to “show your card” in planning poker for item-X, the moderator says over the audio channel, “Type in your number.”
5. All numbers are typed into respective cells simultaneously, and can appear essentially instantaneously on all browsers viewing the spreadsheet.
6. The moderator (and everyone else) can see if there is convergence on a common estimate, or if another round is necessary.
7. Assuming there is convergence, she fills in a growing spreadsheet table (at the top of the spreadsheet that all participants can see) with the story point for item-X; see Figure 12.4.

Figure 12.4 Google Spreadsheet that shows the growing estimation results; it is shared with all participants

	A	B	C	D	E	F	G	H
1	1	2	3	5	8	13	20	BIG
2								
3	item-4	item-1	item-2	item-3	item-6	item-7	item-10	item-15
4	item-12	item-5	item-11		item-8		item-17	
5		item-9	item-14		item-13			
6					item-16			

Try...Multisite Open Space to replace Scrum of Scrums

We coached at a client with about 30 teams spread across two sites (and time zones) with some overlapping work hours; both sites had many meeting rooms with either video support or speaker phones.

They tried a multisite Scrum of Scrums (SoS) meeting with one room in each site, and with speaker phones. They were dissatisfied.

We suggested mini-Open-Space meetings instead. The approach:

1. We used a shared Google Spreadsheet (which both sites could update) for the space-time board. At both sites, the spreadsheet was displayed via a projector.
2. *Space*—Different corners of the two big rooms, and also some adjacent small rooms. Some areas had speaker phones or video technology; these were the *multisite spaces*, which were paired across the two sites.
3. *Time*—Twenty-minute session periods. A total of one hour.
4. The session started with the same environment as the prior multisite SoS: two rooms and a speakerphone.
5. Five minutes were used to fill the *space-time* board with burning issues. People called out their issues and their space-time choice, and a typist at each site updated the spreadsheet.
6. A few minutes were spent moving the location of sessions that needed to be at multisite spaces, based on people's interest.
7. One hour for sessions, applying “the law of two feet.”

“Try...Open Space” section on page 204

Try...Experiment with multisite Scrum meeting formats and technologies

We considered offering prescriptive advice—based on our past experiment—about how to hold multisite Scrum Reviews, Retrospectives, and so forth. There *are* some hints and concrete suggestions, but we especially decided to suggest...keep experimenting.

Keep it simple, avoid fancy tools, and focus on tangible aids such as physical poker cards.

Anything prescriptive will either not apply or become outdated. There are many context-dependent forces and a high variability. Technologies improve every year, opening the way to new experiments. It is possible within 30 years that excellent virtual presence technology will exist so that it seems as though a group that is actually scattered around the world seems to be together in one room. Or maybe that will happen next year... Daily Scrum with avatars.

Try...Cross-pollination



Many words have been written about this practice over the years, and the suggestions can be organized into several fine-grained tips such as *ambassador*, *seeding visits*, and so on (see *Recommended Readings*). But it all boils down to something basic: If you are a multisite product group and not visiting each other from the start—especially at the start—then you are

in trouble. Send lots of people *here*. Send lots of people *there*. Cross-pollinate. Not just management, but many hands-on developers as well. Encourage visitors to visit for at least a few months, and have them work very closely with the local group—group workshops, pair work, pair programming, and so forth, rather than individual work. Repeat forever.

"Introduction to Requirement Areas" section on page 555

In terms of large-scale Scrum, it is important for the Product Owner and Area Product Owners to cross-pollinate and participate in Scrum events at other sites.

The *Offshore* chapter emphasizes that visitors should not be *intermediaries* to the other site—bottlenecks and filters. Rather, encourage visitors to be *matchmakers* that help local people form direct relationships with people at the other sites... “Oh yeah, you should have a video chat with Yang in Shanghai about that; he knows a lot. Here’s his Skype address. By the way, he likes to start work late in the morning because he coaches an amateur swim team.”

"Try...Matchmakers rather than intermediaries" section on page 453

At some point, *them* becomes *us*. This is both good and bad. It is probably a good time for the visitor to return home. It is important for visitors to absorb your ways of working and mental models and to establish a shared vision, but you also want to absorb the perspectives and contacts the visitors bring from their planet. Once they start to lose a fresh and active connection with their home site, their value in cross-pollination diminishes.

Try...Welcoming committees and buddies

I (Craig here) remember the first day that I arrived at Valtech Bangalore and walked into the reception area. There was a sign that said, “Welcome, Craig.” The company does that for every arriving visitor; a small gesture but appreciated. Imagine how it feels—when you are a member of a multinational, multisite product group—to travel 5000 kilometers to a foreign land for three months, show up at the new office on the first day, and no one knows or has prepared for your arrival? It is kind of depressing and does not build a feeling of connection and trust.

At NSN, some sites organize a weekly “site news” email that summarizes all known visitors for the week: arrival dates, a brief “who is this person,” and explanation of why they are visiting.

Consider “welcoming committees” (probably a committee of one) at each site that are well aware of who is coming and when. Prepare for their smooth arrival—what team, what chair, network access, and so on. Greet the visitor on the first day. Ask someone on the team to serve as a ‘buddy’ responsible for the first week to help the visitor connect and figure things out.

Try...Multisite communities of practice (CoP), including a communications CoP

This tip reiterates the general CoP suggestion, an organizational practice of broad and general value. In the context of multisite CoPs, asynchronous communication tools play a heightened role, such as CoP wikis, mailing lists, blogs, web feeds, and so forth.

“Try...Communities of Practice” section on page 207

Also in the context of multisite agile development, a formally recognized *communications CoP* and some active communications CoP coordinators can help. This was cited as particularly useful in the large-scale agile product development discussed in [Eckstein04].

Try...Retrospectives at several levels

What is the most important group for a Scrum Team? Themselves, of course. In multisite development the following is the typical pre-

cedence order of groups that people care about improving: (1) our team, (2) our requirement area, (3) our site, and (4) our product.

In large-scale Scrum, there will be joint Sprint Retrospectives. Start with a team-level retrospective. When the need is felt for higher-level joint retrospectives to improve the *system*, try holding one simply at the level of the requirement area that the feature teams belong to. Beyond that, consider a site-level retrospective. This is the dominant physical and cultural domain that people work in and want to improve... “*We need more rooms with whiteboards.*”

In short, do not only hold (1) single team retrospectives and (2) full-blown multisite product-level retrospectives—there are intermediate levels that are also useful.

Avoid...ScrumMaster representing the team

“Try...Team is responsible for coordination” section on page 194

A ScrumMaster is not a project manager, team leader, or team representative, although these are common misconceptions. A healthy self-organizing Team, not a ScrumMaster, should be responsible for managing their external communications.

A confused ScrumMaster who inappropriately acts as the representative to other teams can be bypassed in a single-site group; the team members can sooner or later take charge and walk over to the other teams they need to interact with, bypassing the *meddling* ScrumMaster. There are many informal effective lines of communication and relationship within a single site.

“Avoid...Scrum-Master coordinates” section on page 197

But the dynamics of correcting this mistake are more awkward in multisite development because the informal effective lines of communication and relationship are more difficult to form and maintain. For example, the ScrumMaster may attend some multisite meeting or visit another site, and in these ways, the other sites may view the ScrumMaster as the team representative and start favoring communication with her. Another site’s team cannot walk over to the team with the meddling ScrumMaster (and vice versa). The ScrumMaster creates an impediment to the team managing their own external relationships that is harder to resolve because of the multisite context.

Try...ScrumMasters acting as and encouraging matchmakers

This reiterates a tip from the *Offshore* chapter: Encourage some people to act as connectors or matchmakers rather than as intermediaries or representatives. This is an appropriate role for the ScrumMasters to play, and to know and encourage in others... “Jose on our team is very interested in that. Let me give you his Skype address so you can connect with him.”

“Try...Matchmakers rather than intermediaries” section on page 450

Try...Improve multisite design with *Design* chapter tips

The foundation of simplified multisite design is to prefer feature teams over component teams, continuous integration across all sites, and a common repository. In addition, *many* tips in the *Design & Architecture* chapter can help in multisite design work. Here is a small sample:

- See “Try...Design workshops with agile modeling” on p. 289.
- See “Try...Tiger team conquers then divides” on p. 308.
- See “Try...Technical leaders teach at workshops” on p. 299. This includes roaming to other sites.
- See “Try...Design/architecture community of practice” on p. 313.

Try...Basic practices for multisite meetings

Some of our instincts for how to speak and listen politely need to be modified in a multisite meeting. Tips...

1. **Prepare the environment**—Don’t keep people waiting by setting up the video technology or conference call after everyone arrives.
2. **Say your name every time**—“This is Peter. I think you said...” Mandatory for audio-only multisite meetings. Also useful in video-session meetings in case each speaker is not visible or in case not all participants are known to each other.
3. **Speak up, and interrupt the speaker as soon as you can’t hear**—Keep the microphone close when you are speaking. When listening, if the speaker becomes inaudible, tell her immediately.

4. **Speak in small batches of words in short cycle time**—Do not speak for a long time. Speak a small subset of what you want to say, and then...
5. **Ask if they heard and understood**—After saying a small amount, check if the other sites heard and understand.
6. **Practice and teach active listening, especially paraphrasing**—A foundation of any good meeting is *listening to understand*. You may not agree, but you should understand. *Active listening* includes both mental and verbal habits to better understand. **Mental habits** include *willful concentration or attention on the speaker* (and willful neglect of internal mental activity) so that the message is simply heard. **Verbal habits** include *checking or paraphrasing the message*. In multisite meetings this is doubly valuable because of transmission problems. “This is Raj. Let me check my understanding. I think you said <X>. Is that correct?” This checking is directly useful, and has the indirect benefit of cultivating an attitude of *mutual inquiry* in addition to the more common *competitive advocacy*. Both elements are useful in work meetings [APS85].
7. **Move the camera**—Do this if the speaker isn’t clearly visible.
8. **Ask for repetition, again and again**—If, for whatever reason, the statement was not clear, ask for repetition. Don’t be shy about asking repeatedly.
9. **Ask for another speaker to make the statement if there is an accent problem**—Sometimes in multisite meetings the reason for difficulty in understanding is related to the accent of the speaker. Ask a colleague of the speaker to make the statement.
10. **Provide running commentary to other sites when something local and rapid happens**—Sometimes the people in one room start to have a local conversation rapidly. The other sites cannot keep up with this local rapid discussion. Someone at the local site can act as a commentator and share with the other sites (in real time) what is going on.

MULTISITE CULTURE & NORMS

Try...Vigilance for shared agile vocabulary and concepts

“How long does the build take?” If they think you mean, “When will the compile and link be finished?” rather than, “How long to run all the tests?” there is a problem.

“Do you have a ScrumMaster?” If they think you mean “A Scrum-Master is the manager who went to a course and now calls himself a ScrumMaster” rather than “Do you have a *ScrumMaster*?” there is a problem.

“Does the team hold a Daily Scrum?” If they think you mean “Do we stand up each day and report our status to the fake-ScrumMaster-Manager who went on a course?” rather than “Does the team hold a *Daily Scrum*?” there is a problem.

Creating some baseline common culture through shared vocabulary and concepts is important in multisite development. It helps to require sending many of the internal thought leaders, all Scrum-Masters, and the leadership team (including the Product Owner Team) to common process-oriented courses and to require reading common process-oriented books. And to require this of new members as they join. A core set of common education and readings is the foundation for a common vocabulary. In Scrum, another mechanism to help establish these goals is a set of well-educated true Scrum-Masters who know Scrum by-the-book and beyond-the-book.

Try...Cultural education

What are the implications when they tell you that “next week is Chinese New Year”? Why is that team not frank and open? Good communication and relationships in multisite development involves people understanding this topic. There are a few relevant tips in the *Offshore* chapter.

*“Agile Culture”
section on
page 468*

There are also more studies than you might imagine that *quantify* various average tendencies that are relevant to understanding culture in the work place, such as *degree of individualism* and *degree of*

risk taking. It is useful to study these to get a broad picture of the culture at the site you are interacting with. *Global Software Teams* summarizes some of this data [Carmel99]. The cultural studies of Hofstede and of Hall are foundations of this area, and worth knowing in a work context [HH05, Hall76]. Some of the data in [HH05] comes from extensive studies at IBM sites in many nations, which provides a relevant ‘normalizing’ context for readers interested in the intersection of engineering work and culture. Here are some examples of what one can learn, that may be relevant to better understanding that new site you are starting in Malaysia:

- the amount of “power distance” people feel between managers and subordinates, for 74 countries
- the amount of individualism, for 74 countries
- masculinity versus individualism, for 74 countries

Try...Vigilance about a common coding style

Common coding style is a basic Extreme Programming practice, important when just seven people are working on shared code. In large-scale Scrum with feature teams, we want all teams to be able to work on all code without *friction*. When you are 60 teams in five cities working on 20 million lines of shared code, it is endless friction when every function in every file is laid out differently, and the only common naming convention is that people *use 26 letters*.

TOOLS

Try...Multisite tool that records audio or video

For some multisite product developments Valtech (including Valtech India) used a tool that allowed audio/video multisite sessions *and* recorded these for playback later on. The playback feature turned out to be surprisingly useful for reference and clarification. Unfortunately, the tool was otherwise rather awful (for example, poor video and audio) and so it was not a sticky practice, but everyone remembers, “That playback feature was *really* useful.” We predict this feature will improve and be replicated over time. Worth trying.

Try...Tablets for shared sketching

This is a tip we have heard of but not seen for ourselves. The *Design* chapter shared the value of design workshops and agile modeling. Video technology in front of two whiteboards is one way to hold a multisite design workshop with agile modeling. Another is to use tablet computers that allow one to sketch free hand on a large virtual space that is shared over a network with other people in other locations, combined with video technology for face-to-face conversation.

"Try...Design workshops with agile modeling section on page 289

Avoid...Commercial 'agile' tools for multisite collaboration

We have noticed, time and again, that software tools deaden the energy of collaboration in a co-located or multisite workshop, such as an estimation session or other Scrum event. And the fancier and more specialized the 'agile' tool, the more it seems to depress the spirit of real interaction between people. The focus is attracted toward the tool rather than toward the interactions of people.

It is common that agile coaches—us included—encourage simple and physical (tangible) collaboration tools in a workshop, such as cards and whiteboards. As described in the planning-poker example, this is easy and effective for multisite workshops as well. You can go a long way with ordinary webcams. When collaboration software tools are needed, keep it *very* simple and use free tools; for example, Google Spreadsheet.⁸

Beware the advertisements and sales pitches from companies trying to take your money by telling you that you need their agile tool to succeed with multisite collaboration—another silver bullet in which the tool vendor gets your gold. The multisite challenge is not a tool; it is people and mindset.

8. Due to intellectual property control, some organizations require all tools and data be hosted within the enterprise. There are a growing number of free and commercial simple web-based spreadsheet tools that can be locally installed if Google Spreadsheet is not an option.

Avoid...Commercial development tools; use free tools

This reiterates a tip in the *Offshore* chapter. Commercial tools can slow down multisite development in several ways. They can create a delay or impediment in opening a new site or in getting a newcomer equipped because of actions required to obtain more licenses. Sometimes the tool needs to be connected to the corporate network to access a license server—preventing off-network development work. The delayed-purchase problem is especially true at “low cost” sites where there are more budget constraints. The fifth agile principle emphasizes giving people the environment they need. For-fee commercial tools are an impediment. Fortunately, there is a vast selection of high-quality, free open-source tools for development, version control, testing, and more. Prefer these at all sites. This is also an easy way establish common tools across sites—in addition to saving the group money.

Try...Wikis as your share point; employ a WikiGardener

“*Try...Wiki for all requirements*” section on page 462

Wikis are discussed further in the *Offshore* chapter. Focus all content in a shared wiki when doing multisite product development. Move away from old-generation document-centric models (for example, Microsoft Word, and document management tools such as SharePoint) and toward a “Web 2.0” model where all content is in something like wiki pages (or a Google Wave), and WikiWords are exploited for hypertext. This reduces the lean wastes of delay and information scatter. Note that using a wiki does not mean inserting links to Word documents; it means to stop using documents, and rather, put the content in wiki pages.

By the way, some resist wikis because, “You can’t create a nice document.” Not true. Several wikis allow automated generation of nicely formatted PDF files from a set of wiki pages; useful.

At the risk of ethnic stereotyping, find a German WikiGardener to keep it more-or-less organized! Seriously though, it *is* useful in a large group with a large wiki to have a formal WikiGardener role that spends at least a little time each week keeping the inherently amorphous and flexible wiki semi-structured, creates template wiki pages, participates in a wiki community of practice, and so forth. Because a wiki is meant to encourage emergent bottom-up struc-

ture, this requires a light touch in which the WikiGardener acts as a gardener who shapes the living wiki and prunes away the mess.

Avoid...ClearCase for multisite continuous integration

See “Try...Continuous integration in “one repository” across sites” on p. 424.

CONCLUSION

The dominant problem in multisite development is...multisite development—degraded communication and so forth. These knock-on problems are greatly aggravated precisely because of the organizational design choices of poor life cycle, site strategy, task allocation, and team structure. If a group uses serial development, with product management and analysis in Boston, system engineering in Taipei and Tel Aviv, programming by component teams in seven different countries, customer documentation in Prague, and testing in Lisbon and Bangalore⁹...then multisite development is going to be painful, slow, and explosive.

The most powerful solution is not to answer the question, “How can we do agile development in this case?” The essential solution is to change the organizational design and site strategy—an example of the difference between *point kaizen* and *system kaizen* in lean thinking. Changes include

- fewer sites
- at each site, a small group of extraordinary people who work in feature teams, rather than large groups of mediocre talent
- co-located feature teams rather than dispersed teams or single-function teams
- fast and continuous integration across all sites

9. This example may seem extreme; but in fact it is typical of how several of our clients organized their work.

Also, *seeing is believing*—sight is important to form relationships and for rich communication. Make video technology ubiquitous, but avoid expensive systems in dedicated video rooms. Rather, focus on cheap, widespread, in-the-team-rooms solutions such as Skype video with regular computer projectors. And experiment with video technology for multisite active workshops (such as joint design workshops) using diverge-merge cycles.

Multisite development often leads to some sites treated as second-class citizens. Watch out for that—and watch out for the friction that arises by the choice of tools that are not equally and easily available at all sites for many years to come; free open-source tools tend to reduce that friction.

RECOMMENDED READINGS

- Erran Carmel's books, *Global Software Teams* and *Offshoring Information Technology*, are two of the better high-level books that explore multisite development.
- Jutta Eckstein's *Agile Software Development with Distributed Teams* is written by a consultant and coach with hands-on experience in both agile and multisite development.
- Keith Braithwaite and Tim Joyce summarize key principles and practices in their paper *XP Expanded: Distributed Extreme Programming*. Although written in the context of Extreme Programming, it applies to all agile development approaches.

This page intentionally left blank

Chapter

- [Expectations 446](#)
- [Interactions 450](#)
- [Requirements 458](#)
- [Test 463](#)
- [Teams 466](#)
- [Agile Culture 468](#)
- [Partnership 469](#)
- [Choosing an Agile Outsourcer 475](#)

- [Appraisals, Certifications, and CMMI 480](#)
- [Contracts 494](#)
- [Tools 495](#)

Book

- | | | |
|---------------------|------------------------|-----|
| 1 | Introduction | 1 |
| 2 | Large-Scale Scrum | 9 |
| Action Tools | | |
| 3 | Test | 23 |
| 4 | Product Management | 99 |
| 5 | Planning | 155 |
| 6 | Coordination | 189 |
| 7 | Requirements & PBIs | 215 |
| 8 | Design & Architecture | 281 |
| 9 | Legacy Code | 333 |
| 10 | Continuous Integration | 351 |
| 11 | Inspect & Adapt | 373 |
| 12 | Multisite | 413 |
| 13 | Offshore | 445 |
| 14 | Contracts | 499 |

Miscellany

- | | | |
|----|----------------------|-----|
| 15 | Feature Team Primer | 549 |
| | Recommended Readings | 559 |
| | Bibliography | 565 |
| | List of Experiments | 580 |
| | Index | 589 |

OFFSHORE

An intellectual is a person who has discovered something more interesting than sex.
—Aldous Huxley

Many tips relevant to agile offshore development with Scrum are covered in other chapters—*Multisite*, and so forth. In offshore, all the usual Scrum events and practices apply.

This chapter focuses on the intersection of agility with typical ‘offshore’ issues:

- different culture and language
- knowledge/requirements transfer
- short-term projects rather than long-term product development
- a heightened sense of *us-them*
- fixed price, fixed scope
- skill differentials
- super-/subordinate relationships
- CMMI

Most of the experiments focus on **offshore outsourcing** companies doing new short-term projects, where *unfamiliarity* (of the domain and client) is a dominant problem; for example, a 6-month project for a travel website, then a 12-month project for a retail chain point-of-sale system, ad infinitum.

A smaller number of suggestions focus on **offshore insourcing** (and *outsourcing*) companies doing *long-term familiar* products and working with a familiar onshore partner, such as a dedicated development center in China working on a telecommunications product.

Valtech¹ has an offshore outsourcing center in Bangalore that applies agile principles and Scrum to projects for clients in Europe

1. Where Craig served as chief scientist, working at their India site to help create “agile offshore” development.

and the USA. Another example of real agile development in India is at ThoughtWorks. Nokia Siemens Networks² (NSN) has major offshore development centers applying large-scale Scrum in China, among other locations.

Over the years, we have spent a lot of time introducing Scrum, agile principles and practices in these countries. We have also worked in Brazil and Eastern Europe at offshore or “low cost” centers applying agile methods, both for multisite product development and single-site outsourced (usually fixed-price) projects. Craig has been visiting or living in India since 1978 and speaks some Hindi, and Bas has lived in China for many years and speaks Mandarin.

The forces to reckon with vary by region. For example, Bangalore averages 20 percent or higher annual attrition in IT companies. Since agile and lean thinking focuses on values and principles, and values are part of a culture built up through conversation and learning to work together, it is a challenge to sustain an agile culture in a company located in the hot IT cities of India.

Figure 13.1 Scrum team in their room, Valtech India



EXPECTATIONS

Try...Educate that agile offshore is not just short iterations

The “agile offshore” literature—from misinformed journalists, analysts, and managers new to the subject—abounds with the miscon-

-
- 2. Where Bas served as NSN-wide lead coach for large-scale agile adoption. He also lived in China, working in large multisite development adopting Scrum.

ception that ‘agile’ primarily means to deliver in short two- or four-week timeboxed iterations. The “agile is anything that is not the waterfall” and “agile is a practice” misunderstandings are broad misconceptions worldwide, especially pronounced in offshore work.



Figure 13.2 visual management in team room, Valtech India

In our experiences in India and elsewhere, the lack of engaged stakeholders (customers, managers, ...) who understood and applied the agile values was the key weakness, and on the other hand, successfully shifting a group to these insights and new behaviors was the most positive win for all concerned.

Therefore, educate the onshore customer and management that agile is a set of values and principles (rather than a specific practice such as timeboxed iterations) and that these values imply a close and ongoing collaboration with feedback loops between the real customer and real developers.

Step one to succeeding with real agile offshore development is for the leadership spearheading this initiative to clearly and consistently communicate this message, and emphasize that ‘agile’ involves a change of mindset and behavior among all stakeholders, including the customer and management team. This tip is true for all agile transformations, but even more so in the offshore world.

Figure 13.3 agile team in common room during Daily Stand-up,
ThoughtWorks India



Figure 13.4 visual management — product goals,
ThoughtWorks India



Try...Agile guide for sales people and prospects

Sales people are so *positive*. That's great, but a common side effect is that the new client is promised the moon and the stars, and none of the behavior-changing implications of adopting Scrum are communicated to them—such as participating actively as Product Owner each iteration. Sales people are also not renowned as process geeks who spend their weekends reading books about Scrum and lean thinking. To avoid nasty surprises, better prepare a clear, short guide that both the sales people and prospective client need to read... *Partnering with Your Agile Offshore Team*.

Try...Kickoff agile workshop to educate customers

How to educate the customer (and management) in agile values and in Scrum? A common situation with offshore outsourced projects is that traditional-minded clients believe they can hand over the speci-

ficiations, wait until the end, and receive a system. Plus, they usually have old and ineffective project management ideas for tracking progress, such as conformance to a speculative Gantt chart (“Microsoft Project chart”) of sequential tasks with time estimates. And, they will be unused to the idea of *directly* talking with the hands-on developers (in India, for example) rather than talking with an intermediate representative such as a project manager.

Therefore, try starting a project with a new customer with a one-day “agile offshore” workshop. For example:

1. Share the essential ideas of agile development—not practices such as timeboxed deliver, but the four values and twelve principles and how they may creatively be expressed in the daily work. Also, introduce at least the categories of waste (waiting, handoff, information scatter, ...) from lean thinking and explore how they may be reduced.
2. Introduce Scrum (and its key values of transparency and empirical process control), explaining the events that the customer needs to participate in the Sprint Planning Meeting and Sprint Review, the Product Backlog, the Scrum rules, and so forth.
3. Clarify especially for the customer the Product Owner role—responsibilities and actions.
4. Present a “what’s different?” list.
5. Simulate, for a few hours, a hands-on Scrum release cycle that goes through a couple of mini-iterations (for example, two iterations of two ‘days’ each 10 minutes long) in which all Scrum events, rules, activities, and artifacts are created, and in which the customer plays a proper Product Owner role.
 - We once coached a Scrum kickoff for a product group through a Scrum simulation. Using their previous release features, we created a Product Backlog in a short workshop. Then, an imaginary Sprint Planning. The next day, we imagined the first iteration was finished and held a Sprint Retrospective. The amazing thing is that they already knew the problems they were likely to have. The action list created in the ‘imaginary’ Retrospective was actually used.

The workshop is best held physically at the offshore site so that the customers start to develop “Go See” mindset and form direct relationships with the programmers and testers, but onshore may be the only option until the client develops more trust and engagement. At Valtech, they encourage clients to come visit their teams in Bangalore as early as possible.

In addition to more obvious participants, include

- the sales person and account representatives who sold the project and handle the ongoing commercial aspects
 - this has a double benefit, as sales people understandably often have vague understanding of agile details, and benefit from deeper and regular education
- some programmers and testers from the teams that will serve the customer
 - emphasize *removing* project representatives and instead increase direct contact of the true customer (rather than onshore project managers) with the real developers and testers (rather than offshore project managers)

INTERACTIONS

Try... Remove barriers between offshore team and onshore client

This tip generalizes several more specific tips in this section. On the first offshore project we coached, on our arrival, there was an onshore project manager who acted as an intermediary between the hands-on workers and the real clients. This was the root cause of several dysfunctions. It is critical to remove such barriers.

Try... Matchmakers rather than intermediaries

Avoid... Single point of contact

Traditional offshore development uses intermediate representatives (sometimes, a *single point of contact*) between the team and real customer, increasing the waste of handoff, among other weaknesses. On

the other hand, consider a *matchmaker* whose job is to encourage people to meet and, perhaps, fall in love.

When possible, this is what Valtech tries to do on their offshore agile engagements. They, as with many outsourcers, usually have one or more onshore consultants at the customer site. And as usual, there is a Valtech customer-engagement leader at the Bangalore center. But rather than these people acting as *intermediaries* to the team, their job—in part—is to encourage the customer and the team to meet regularly face-to-face, by applying the *Video sessions* and *Customer visits* team tips.

Their goal is for the clients and whole team in India to feel that they can have frequent, easy, face-to-face meetings, without the need for intermediate representatives.



Figure 13.5 Daily Scrum at Sprint Backlog (with visual management), China

Try...Seeing is believing—video sessions

We once coached an offshore project (in Bangalore) and attended a conference-call telephone meeting. The call included, in Bangalore, the Valtech engagement manager and the technical lead. In France were some onshore Valtech people and client representatives.

The project was experiencing the usual assortment of complications, surprises, and variation. What was immediately clear in the speakerphone-based discussion was the lack of human rapport, and a subtle *us-them* quality in which it felt as though *them* was boss and *us*

was servant. Part of what was missing was the sixth agile principle: *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*

So, we arranged for both sides to get web cameras. We set up a computer projector in the client room in France where the onshore Valtech people sat, and a projector in the team development room in India. Note: The visualization was in the Indian team's common project room, not a special meeting room.

Then, for the next meeting, we got most Indian team members in the room to sit together facing their camera and the large-screen projection of the (free) Skype Video session of the other group in France, and vice versa.

Tip: It is critical that the audio channel be perfect (no delay, ...). If Skype Audio is not perfect, use a telephone call and speakerphone for the audio channel to complement the video.

Now, both groups could see each other, face-to-face. The client was able—for the first time—to relate to the Indian developers and testers as real people rather than as disembodied voices on a phone, and vice versa. And the client was not relating to just one or two team representatives, but directly seeing and talking with a group of ten people in Bangalore on a large screen projecting a Skype Video session.

Figure 13.6 team room with projector and web camera for offshore video sessions, Valtech India



The difference in rapport was noticeable. And that is no small thing; the first agile value is *Individuals and interactions over processes and tools*. The quality of human relationships in development is so important. Humans need human relationships. Plus, since the entire team was present, there was no more handoff waste and mis-communication between team members and the client. As well, face-to-face video sessions increased the bandwidth of full non-verbal communication between the groups.

In Scrum, these video sessions can be used for Sprint Planning Meeting, Sprint Review, Sprint Retrospective, and offshore requirements workshops.

Cheap, simple, frequent, and located in the team room are important qualities for sustainable videoconferencing with the team. Using an expensive system in a distant room is not a sticky practice.

Reality check—A management report regarding status, morale, and so forth does not reveal the truth. Simply seeing and talking with the real workers might not, either. However, it is step one in forming a relationship that might help.

Meaningful confirmation—A development team wants to do a good job, and hear that their real customer (not a development manager) appreciates the results. During regular video sessions—which may include the Sprint Review at the end of each iteration—the real customer has a chance to say, face-to-face, directly to the team, “That was a great job, I appreciate your work and results.” Development teams value that kind of feedback from a meaningful client.

ESL benefit—English was *everybody’s* second language. Because everyone could see each other’s mouths while listening, it was easier to disambiguate what was being said. Tip: Do not be shy to frequently ask people to repeat things in an ESL situation.

Room-cam—On one project at Valtech India where *us-them* and trust was an issue with the onshore European client, Valtech installed a webcam in the team room that showed the entire room, 24x7. Onshore at the client site, Valtech set up a permanent monitor that showed this image, refreshing every 10 seconds. The client could see a group of 20 people working full-time for them.

Time zone issues—Europe to India time zone differences are small enough to support video sessions without complication. And fortunately, in Bangalore, IT working hours are until 19:00 or later. So for east-coast USA with a 9.5 hour difference to Bangalore, common video time is easy. For west-coast USA, it is not too large a stretch for the India team to occasionally stay another hour later or for the onshore USA group to get together a little early.

This video session tip is reiterated in the *multisite* chapter, but bears repeating here because of its use to deal with the strong offshore forces of us-them feelings, unfamiliar people, different languages, and super-/subordinate power relationships.

Try...Remote Sprint Review

The Sprint Review will of course be done with the full Scrum Team in Bangalore and the Product Owner in Boston. Valtech India experiments with different practices to make this remote event effective:

- video session
- remote shared desktop of the running demo
- onshore Valtech representatives also present with client

Try...Seeing is believing—client visits team

To reiterate, humans need human relationships. At Valtech, they encourage the client (ambassadors) to visit the Bangalore development center and to meet with their team, at least during kickoff, so that during the first iteration the client can spend part or all of the iteration in the project room, helping with knowledge transfer and building relationships with the team members. When the client returns home and has video sessions with the team, there is better rapport and ability to collaborate. Plan for several visits: At the start, halfway through, and so forth.

Try...Team members visit client

This is a traditional, and excellent, offshore practice. Team members better understand the client business and build relationships. However, it suffers the disadvantages of being expensive for many members to travel or if only one member travels, that one person potentially falling into the unhealthy role of intermediary bottleneck for the team. For long-term major projects, Valtech usually sends one or two team members from India to visit the client.

Try...Rotating ambassadors

Ambassadors who travel to other sites are good, in both directions—as long as they encourage matchmaking over acting as intermediaries [Eckstein10]. But avoid long stays, rotate the ambassadors occasionally. People may get unhappy if away from home too long, and perhaps more importantly, *different perspectives* are important.

Offshore, there will always be a person called the engagement manager or “project manager,” who spends most of their time offshore with the team. Tip: Include this person among the ambassadors.

Try...Translator on team

In India, usually enough people in the Scrum teams speak enough English to talk (in video sessions) with English clients. But some of Valtech’s clients are in France and other countries where a common language is a problem between the onshore and offshore teams. And in China, *any* foreign language is problematic.

For projects where this may be a problem, Valtech usually hires a translator to join the team, sitting in the team room.³ Likewise, onshore at the client site, there is usually a translator. During video session meetings, everyone still sees the others face-to-face, but if there is trouble, the translators fill in the gaps. At other times, the translator helps with translating documents from the onshore client (for example, French to English) and in the spirit of multi-skilled workers, may learn new skills such as testing.

3. One can find good translators for many languages in Bangalore.

Figure 13.7 Scrum Release Planning Workshop, agile estimation, Brazil



Try...Offshore team speaks English

We speak some English, French, Dutch, Hindi, and Mandarin—all lovely languages, but English is the de facto standard for international communication, so it is important if most or all offshore Scrum team members speak decent English, so that in the likely case that the onshore clients speak English (at least as a second language), they can directly talk during video sessions. To help this, for example, NSN China offers English education for all employees.

Warning: Avoid the quick fix of channeling communication through the one and only good English speaker.

Try...Clients participate in a Sprint Retrospective

A common offshore outsourcing problem is that the true issues are hidden from the onshore clients. So, if they are visiting offshore, it is helpful to invite them to a Sprint Retrospective or to hold a multisite video session retrospective. First, the team has to be comfortable with this idea, perhaps having done a mini-retrospective by themselves first. For this to succeed, a good facilitator (often a Scrum-Master) is needed to set the tone on learning rather than blaming, and to create a sense of personal safety so that people can be frank.



Figure 13.8 Sprint Retrospective, Valtech India

Try...Offshore group first does several iterations onshore

Among the most successful Valtech India projects are those in which a subset (due to travel costs) of the offshore team executed the first iteration (or more) onshore in close collaboration with client stakeholders and subject matter experts, forming the core of a larger offshore group. A less successful—but still useful—variation is that a Valtech onshore team (such as people from Valtech Germany) executed the first iteration onshore, and then travelled to India to form the core of the offshore team.

Try...Proactively find and educate an onshore Product Owner

New clients are not used to actively participating in and driving development as real Product Owners. They have in mind the traditional view that the offshore site will decide priorities, and so on. Also, they will usually provide their own project manager (often from their IT department) to look after the ten-month project. Further, this project manager may not have a sense of what the real client *users* want from the system.

If possible, use your onshore consultants at the client to quickly find a better candidate for Product Owner, someone closer to the real users, probably not someone in the client IT department. That person needs to be part of the kickoff workshop and perhaps separately educated, for example, being sent to a Scrum Product Owner course.

A less desirable alternative is for the client's project manager to play the role of Product Owner. Failing commitment for even that, an onshore representative of the offshore company can act as a Proxy Product Owner.

Avoid...Believing 'yes'; ask open questions

Agile offshore development involves plenty of ongoing communication between parties, so be warned: In India and China, if you ask, "Can you do X?" or "Do you understand?", 'yes' means maybe, no, yes, "I heard that," or "I don't know." Also, watch out for "*We can do that.*" Instead, ask open questions in a way that tests the understanding or plan. Also, try reverse questions.

REQUIREMENTS

Try...Offshore requirement workshops each iteration

Situation: Thirty developers and testers in Bangalore. Suddenly, they are asked to develop a customer rewards system for a mobile telephony provider in Germany. It will probably take around six months. No problem, except... what is a customer rewards system for a mobile telephony provider? And the client is not going to be in the team room to explain the domain; they are 3000 kilometers away. This is the *knowledge transfer problem* (often just called 'KT' in India). All outsourcers have it, whether based in Germany or in Bangalore. But it is exacerbated when you are very far away, and the time zone, language, and culture is different, and you are an offshore outsourcing company doing an endless series of novel six- or twelve-month *projects* rather than a dedicated offshore insourcing center working on the same *product* for years.

A classic offshore response is to send some people to the client in Germany to write up the requirements and become, relatively speaking, subject matter experts (SMEs). Some may stay at the client and some may return to India. Very reasonable. But the traditional model usually stops at that and these SMEs become bottlenecks and points of weakness. Furthermore, as new written detailed requirements are sent from Germany to India each itera-

tion, the only serious readers of the requirements are the SMEs. In other words, our project has a very low *truck number*⁴ and high handoff waste.

In addition to having SMEs, try an *offshore requirements workshop* each iteration. The following steps have worked:

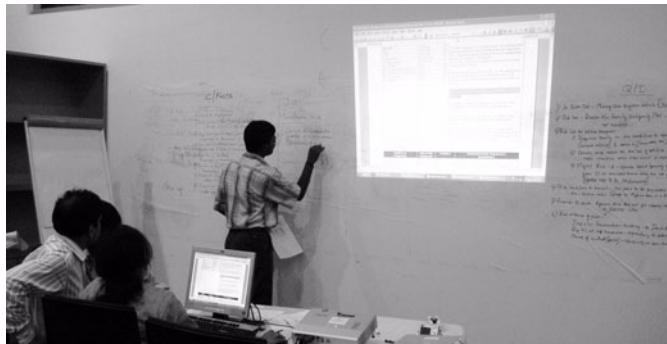
1. About one week before the end of iteration N, the onshore SMEs provide updated detailed requirements for iteration N+1. Desirable if recorded in a set of Wiki pages.
2. Key practice: Shortly after receiving the requirements, the entire offshore team in India holds a 4–8 hour *offshore requirements workshop* to learn and critique the detailed written requirements created by the onshore group in Germany. They use a projector to display the material on the wall for the team to see and read together, and surround themselves with flip charts and whiteboards to write down questions and clarifications as they consume this material—as a team. Lots of conversation happens. Eventually, they start to write questions and issues back into the Wiki pages for the German team to read. See Figure 13.9.
3. Very shortly after the offshore requirements workshop, onshore and offshore teams have a group video session to go through the questions and issues raised by the offshore team. Conversation happens; the Wiki is updated with answers.

In this way, there is a reduction in handoff waste and deeper knowledge transfer to the entire team each iteration.

This offshore requirements workshop may be part of ongoing Scrum Product Backlog refinement workshops each iteration.

4. *Truck number*: How many people on your project have to get hit by a truck before you are in trouble?

Figure 13.9 off-shore requirements workshop by Scrum team, Valtech India



Try...Offshore domain and vision workshop

Offshore outsourcing teams have recently been asked to deliver a bank regulatory monitoring system for a government in Africa. It will probably take over a year. Now, the team bumps into the *knowledge transfer problem*. A couple of team members know this domain—which is why the company won the contract—but 90 percent of the people do not. Before starting the first iteration, hold a workshop (probably for several days) that first introduces the domain to the entire team. It helps to establish common concepts and vocabulary by agile modeling at whiteboards (actively involving all participants) to create an object-oriented domain model and a set of UML activity diagrams (see Figure 13.10).

This “domain workshop” is followed by a “product vision workshop” so that the entire team has a sense of the vision, big picture, and major features of the upcoming product.

In contrast to single-specialization and handoff, the goal is to encourage the Scrum goal of “whole team working on whole features.” For that, the teams need the big picture.



Figure 13.10 joint design workshop, during a diverge cycle, Valtech India

Try...Requirements documentation adaptively ‘simple’

A common misconception is that detailed written requirements are inappropriate in all agile methods.⁵ Yet, in Scrum, people can create elaborate, detailed requirements if the Scrum teams find them useful. And some written detail *is* useful when the teams are in Shanghai and the requirements donors are in London.

What is the correct level of detail to prepare onshore and send to the offshore team? Relax—you don’t have to make a final decision. In Scrum, *inspect and adapt* of the practices occurs each iteration, during the Scrum Retrospective.

Start simple. Can you succeed with only the short *user story* format, combined with video session conversation with onshore requirements donors? If that does not work, increase the detail. And perhaps later on, it will be possible to simplify again—there is no final answer in empirical process control with Scrum.

However, beware the local optimization mistake of increasing written documentation simply to avoid face-to-face video sessions.

Try...Frequent onshore UI prototypes

A great user interface (UI) is key to user satisfaction. As part of providing more requirements details to the offshore team, do UI prototypes (paper mock-ups, digital mock-ups, ...) early and often (each iteration) onshore with the customer and onshore UI designers, to maximize the bandwidth of conversation and feedback. If onshore

5. Usually due to the old “agile means XP” misunderstanding.

UI designers are not an option, do real-time screen-sharing sessions between offshore designers and onshore clients, to creatively and collaboratively evolve UIs together. Small batches, short cycles, and feedback are *critical* for the UI.

Try...Semi-detailed requirements documentation for iteration

Techniques for writing more detailed requirement documentation are explored in the *Requirements* chapter. In all cases, this elaboration is done onshore directly with the requirement donors. An onshore consultant is also involved.

Try...Detailed requirements with A-TDD

"Try...Acceptance test-driven development section on page 42"

This tip is appropriate to all kinds of development, but is reiterated here because of the special value of getting Acceptance Test-Driven Development (**A-TDD**) in place *from the very first iteration* when doing short unfamiliar outsourcing projects with a new client far away. It rapidly exposes misunderstandings and drives down ambiguity in requirements.

These requirements-as-tests, along with any natural language requirements, are reviewed each iteration at the *offshore requirements workshop*.

Try...Wiki for all requirements

Most Valtech agile offshore projects swear by using a wiki (they use Confluence Wiki) for all requirements (and other project information), rather than an old-generation document-centric tool (for example, Microsoft Word). Wikis reduce the lean wastes of waiting (to see, modify, and share) and information scatter (with the magic of WikiWord hypertext). Further, tools such as Confluence Wiki support threaded discussions attached to each wiki page. Detailed requirements can be written up in London and immediately seen and commented on in Bangalore, by multiple teams. The wiki-centric model consolidates all discussion and clarification of a requirement in an easy, fast, “Web 2.0”-centric tool that encourages participation.⁶

As a corollary, do *not* use email for any project discussions. That's a recipe for communication disasters. Use the centralized, persistent, and shared communication tool for all project discussions—the wiki.

This tip is repeated elsewhere, but reiterated here because of its importance in offshore development.

TEST

Try...A-TDD for UAT

This suggestion is appropriate to all kinds of development, but is reiterated here because of the special value to putting in place A-TDD for user acceptance testing (**UAT**) *from the very first iteration* for short, unfamiliar outsourcing projects with a new client far away.

Try...Manual (if you must) UAT each iteration

The Definition of Done ideally includes UAT, but that would only be possible if fully automated A-TDD was used for UAT, with no manual testing. That would be great, but fully automated A-TDD for UAT is not always possible; clients unfamiliar with the idea might *want to do manual UAT*.

In traditional (sequential) offshore outsourcing, manual UAT is done once, after all implementation is finished, followed by a high-stress rework and negotiation phase on what to do with all the problems. In agile offshore development, if manual UAT is necessary, *it is best done each iteration*. If the client is new to Scrum, this will be a challenging recommendation, requiring careful education. It helps to share stories or quotes from other clients.

If you are really lucky, you can convince the client to do manual UAT after each user story (small feature) *during the iteration*, reducing the batch size even further. Good luck.

6. Other free “Web 2.0”-centric options include Google Wave.

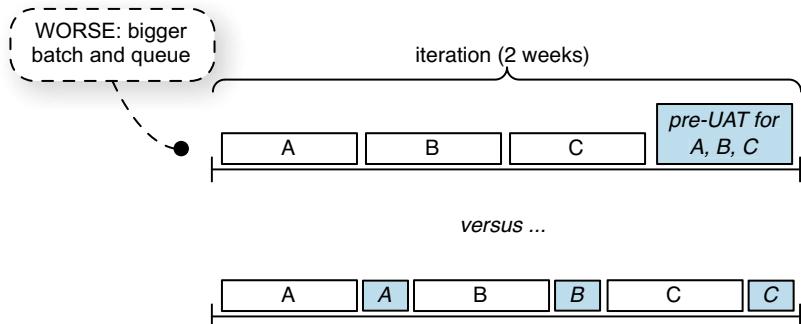
What happens if failures are discovered by the client during manual iteration-UAT? *Typical scenario:*

1. Manual UAT for iteration N starts in iteration N+1.
2. UAT results are typically known half way in iteration N+1.
3. The Scrum teams plan for slack to handle iteration-N UAT small defects within iteration N+1.
4. On the other hand, big bugs are put on the backlog and handled in iteration N+2.

Try...Manual pre-UAT after each feature

Assume there is (unfortunately) manual UAT and/or old-fashioned ‘automated’ test scripts for testing through the GUI⁷ that the client will do each iteration. Fact: Offshore outsourcers will do an offshore execution of these manual UAT tests before delivering the iteration to the onshore client for real UAT. This is **pre-UAT**. Advice: Do *not* create a queue of features to test and then do pre-UAT near the end of the iteration. Avoid a mini-waterfall. Rather, do this pre-UAT in smaller batches, keeping the queues smaller, after each feature is implemented. See Figure 13.11.

Figure 13.11 pre-UAT after each feature



-
7. Acceptance TDD is done with table-based keyword-driven tools that do not require testing through a GUI.

Try...Iterative requirements onshore to offshore

In Scrum, the Product Backlog items offered to the team during the Sprint Planning Meeting must *already* be well analyzed before the meeting starts. So, before the end of iteration N, some group needs to prepare the detailed natural language requirements and (ideally) automated acceptance tests for iteration N+1. They must be ready perhaps one week before the end of iteration N, so that they can be handed off (unfortunately—the waste of handoff) to the offshore team for their *offshore requirements workshop*. See Figure 13.12.

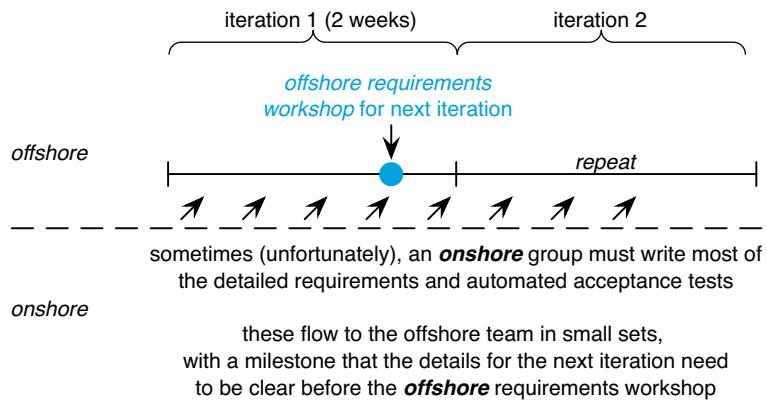
"Try...Offshore requirement workshops each iteration" section on page 458

Who does this detailed requirements preparation before the offshore requirements workshop? It is usually led by people onshore—typically, consultants co-located with the Product Owner and other users—because it involves long (for example, five days of work) and close discussion and review with the client. Try to include a few offshore team members in some video sessions during this period, to reduce the pain of handoff from onshore to offshore.

If the *offshore* team must be deeply involved in the requirements analysis for the next iteration (because there are no consultants from the offshore company visiting the client), then the offshore Scrum team will have to reduce the scope of their implementation work and allocate significant time to play the role of the onshore people shown in Figure 13.12. There are several options for how they work:

- ❑ The *entire* offshore team holds relatively long offshore requirements workshops (in preparation for the next iteration) and holds video sessions with the Product Owner and users.
- ❑ A *few* members of the offshore team do this work while the remaining members do the usual work of implementing features; then, during the offshore requirements workshop, the *whole* offshore team digests the wiki pages and the tests created by the two or three offshore team members.

Figure 13.12 iterative requirements onshore to offshore



TEAMS

Try...Stable offshore Scrum teams

Long-lived stable teams are a proven desirable goal. The cross-functional Scrum team (feature team) of 7 ± 2 people is ideally stable, going through the slow social process towards improved jelling.

stable teams: see the Feature Teams and Teams chapters in the companion

So what? At India and China outsourcers, it is especially common to observe a very mechanistic mindset about people and teams, and little awareness of research into effective team structures. One frequently hears the term ‘resource’ rather than ‘person,’ and old-fashioned resource-pool and temporary project group or virtual team thinking is widespread. Single-function teams (developers, testers, ...) are common. People are shuffled into new groups frequently. In short, poor ideas.

It does not have to be that way. Virtually every outsourcing project requires at least one team of seven people. So, an agile offshore outsourcing organization gradually develops more stable Scrum teams who stay together for a couple of years. This is easy if the project lasts a few years; on the other hand, if the projects are short, then management needs to recognize the value of trying to keep the teams together across several short projects.

Trade-off between learning, stability, and delivery—Stable learning teams are desirable, but there is the “art of the possible” to consider. Suppose

- The outsourcer has initiated a four-month project for upgrading an air travel website written in JavaServer Faces with a Hibernate backend.
- No existing stable team knows most of this.
- The learning curve is non-trivial.

It could be faster—in the short term—to form a new group based on competency if different experts in these areas exist scattered around the company.

On the other hand, this is a local optimization that avoids long-term improvement of existing stable teams. If possible, *stop and fix*: prefer to find a stable team that already knows the most, that can be *supplemented* with other temporary experts who act as coaches.

Try...Simple titles map to special titles

In developing economies such as India and China, moving *up* is understandably critical to people. On the other hand, Scrum teams de-emphasize titles and hierarchy, encouraging multi-skilled workers. It is not very impressive to tell your mother-in-law, or next prospective employer, that your official job title is... *team member*. So, one approach considered at Valtech is to have both an *internal* and *external title*. For example, internally, *engineer* levels 1 to 20, or *coach* levels 1 to 10. Simple, flexible. When a person leaves, something more special may be provided, such as *senior architect* or *senior project manager*.

Try...Encouraging the teams to say ‘no’

‘Yes’ means nothing and anything in some places. One reason is culture—deference to seniority, and so on. Another is a willingness to be flexible in an asymmetrical power relationship where *us* has the money and *them* does not. One of the wastes in lean is *wishful thinking*, and it is fascinating (to us) to examine all the ways saying ‘yes’

without fail generate problems. Fact: A Scrum Team (not a manager) has the authority to decide how much to take on each iteration, and to descope the goals if necessary. So, it is important for the team to feel they have the *personal safety* to say ‘no’ and break out of wishful thinking and over-commitment behaviors. This supports the eleventh agile principle of self-organizing teams, plus empowerment.

As a ScrumMaster or client, it is helpful to reinforce to the offshore teams—hesitant to appear inflexible—that they *can* say no. One of the Scrum rules is that work cannot be pushed onto a team; the Product Owner offers items for the iteration, and the team pulls as many as they decide they can do at a sustainable pace with good quality. The team needs to know this, and not be made to feel bad when they say ‘no.’ We know of one client who went out of their way to regularly thank the team when they said ‘no’—as this client had suffered the effects of wishful thinking all too often. See also the impact of over-commitment on the creation of bad legacy code in the “How to Write New Legacy Code” section on page 334.

Try...A ScrumMaster intent on self-organizing teams

Self-organizing teams, the eleventh agile principle, is a big change for most organizations worldwide. It is even more challenging in countries where deference to seniority is strong. One antidote is to ensure you find ScrumMasters who are *crystal clear* on the importance of self-organizing teams and working actively for them.

AGILE CULTURE

Try...Long-term agile coaching group if high attrition

Every enterprise agile-transformation initiative benefits from a dedicated central group of expert coaches who go sit with teams and coach them. This is doubly important in offshore cities with high attrition due to a robust job market. Lean and agile principles only work if embedded in the culture and mindset—this ideally comes from stable employees (a goal in Toyota) and conversation. That is hard to achieve in cities such as Bangalore or Shanghai. Compen-

sate with a *permanent* central agile-coaching group that continually re-infects teams with agile and lean viruses.

Try...Outside-the-site agile coaches

Every organization benefits by bringing in outside agile coaching experts to act as *viral agents*. This is doubly true for offshore organizations steeped in traditional command-and-control management and process culture. If the offshore organization is multinational, start by looking for in-company coaches from other nations.

Try...Buddy system if high attrition

When USA Navy members join a new ship, they are assigned a *Sea-Daddy*—an old-timer who, over several months, shows the new people around and helps them understand the *culture* of the ship. The USA military recognizes the criticality of cohesive culture and the need to quickly socialize people to the local culture. This buddy system is a cornerstone practice that can also apply to offshore sites with high attrition.

PARTNERSHIP

Avoid...Onshore management, offshore development

We sometimes work in countries where a vision widely promoted is, *organize so that the management is done by expensive onshore European managers, and all the “messy work” of actual software development is done offshore*. Taylorism at its worst. Besides that, do people not know it is possible—and inevitable—to *outsource the management*? Fast-forward thirty years into the future to a ‘developed’ country with lots of overhead where few people can do the concrete customer-valued work (such as hands-on design and development) and talented competitive people in “developing nations” are doing the management and development of new products. Who will win that competitive battle?

Of course, more broadly, “onshore management, offshore development” breaks just about every practice and goal in lean and agile development if one wishes to *sustainably deliver value faster and faster with high quality and morale*.

We once worked with a multisite product group trying to adopt Scrum where a manager decided that the ScrumMasters for the offshore teams (in Asia) should be *onshore* (in Europe). The ScrumMasters asked us what they should do...

Try...Offshoring features, not disciplines or components

If you want to be agile, do not outsource a discipline, such as testing. That approach is for a sequential life cycle with single-discipline teams and handoff—fundamentally inconsistent with agile and lean thinking.

Rather, outsource a set of complete customer-centric features to offshore *feature teams* that do most of the work (design, implement, test) to complete the features. As mentioned in the “Try...Iterative requirements onshore to offshore” section on page 465, a common exception is an onshore group that helps do detailed requirements analysis in close proximity to the real customer.

Similarly, do not outsource a component or subsystem. If one sends offshore a subsystem of a large product, then all the delays and problems examined in *Feature Teams* (in the companion) and in *Continuous Integration* will be even more painfully felt than usual, due to the physical and social remoteness of a separate outsourcing company.

Try...Treating the offshore organization as internal partners

An important Toyota (lean) principle is to help partners become lean and to essentially ignore organizational boundaries. Toyota coaches work inside partner sites to help partners apply lean thinking. Plus, when there is a problem, Toyota coaches visit the partner and help find the root cause, rather than blaming the supplier [ISV09]. The attitude is partners are more ‘inside’ and less ‘outside.’ Also, Toyota wants long-term sustainable partnerships in which all parties thrive

and support each other instead of looking for temporary lowest-price suppliers.

Rather than superficially evaluating partners based on management reports and measurements, Toyota people practice *Go See*—master engineers spend time inside the partner site and grasp with direct insight the nature of their existing or potential partner.

A story: We once coached at CompanyX that *offshore insourced* and *offshore outsourced* some of their product development. Both insourcer and outsourcer worked on the same product and were located in the same city. A joint inter-company “spread knowledge” group (the lean *yokoten* practice) was formed between insourcer and outsourcer and met weekly to share useful practices applicable in both companies. It successfully led to improvement at both partners. There was complete transparency and cooperation on a very concrete level, and the boundaries between the companies were not inhibiting the real work of making the product. Healthy situation. Then, one day, a “partnering manager” from CompanyX came to visit the insourcer site to assess the cooperation with outsourcer. He asked for the contracts, reviewed them, and asked us why we did not establish “quality criteria” for the outsourcer to conform to. We suspect he had recently been to a PMI or CMMI course and was now putting his ‘education’ into practice. The insourcer team was puzzled. Why would they establish quality criteria? They had worked together with the outsourcer in a transparent and very detailed way to make sure that what was delivered was the best possible and that both parties were improving and learning together. How was establishing conforming “quality criteria” going to improve anything? If the outsourcer or insourcer knew how to do any better, then they would have done so already and shared with the other group. It made no sense. This is typical of traditional management education that promotes reporting and conforming, rather than the Toyota Way values of *Go See*, transparency, and coaching with partners.

Offshore development with lean thinking applies these principles, especially for multisite product development that includes an offshore site (insourced or outsourced). For example, if you are evaluating an offshore partner, spend time sitting with random developers at their site and look at their code while they are working on it—get direct insight into the work and workers.

Tip: An early point for removing boundaries is to ensure that the offshore teams *share the same code repository* as the onshore (or other site) teams. For example, NSN has multisite development with centers in Europe and China; all sites share the same Subversion code repository. Removing technical barriers reduces basic *us-them* issues.

Try...Dispersed feature team if *us-them* is a problem

A **dispersed feature team** is one whose members are in several locations. A co-located team is correlated with higher productivity than a dispersed team, is simpler to coordinate (especially with time zone variance), and more likely to jell as a real team. Although there have been anecdotal case studies and speculation that dispersed teams can be productive, arguably the most thorough research on this subject concludes that a dispersed team will never be as potentially productive as a co-located team [OO00].

So, all things being equal, prefer co-located feature teams. But not all things are equal. For example, in offshore insourcing where a product group originally had only one site in Europe and then adds a second center in China, there can be a variety of *us-them* problems. Creating dispersed feature teams—intentionally accepting the productivity impact to address other system problems—is one mechanism to address this.

If you must have a dispersed team, encourage a “real team” by emphasizing *synchronous over asynchronous* communication tools, such as Skype Video and instant messaging. *Seeing is believing.*

Avoid...Unbalanced onshore favoritism or bias

A sure way to increase *us-them* problems is to consistently favor the convenience or cultural bias of *onshore* stakeholders. For example:

- Multisite meetings are always held at times convenient for onshore but inconvenient for offshore.
- Offshore holidays are ignored.

These situations can be exacerbated in Scrum because of the short iterations, timeboxing, and regular short meetings. When planning the timing of future iterations, ask, “What are the local holidays?”

Avoid...“four-year programmer” partners

At one time, Valtech considered buying an outsourcing company in India that had a “CMM level-4 certification” (their words). Valtech first contracted with the prospective company to develop a system. After it was done, they looked in detail at the source code. *Garbage!* Clearly done by people who were not good programmers. Leaving aside the observation that ‘certification’ was virtually meaningless, the experience opened eyes to another offshoring problem...

“Avoid...Believing CMMI appraisal or certification means much in creative R&D work” section on page 489

Worldwide, there is a problem with the quality of programmers. They do not usually learn much useful about good programming/design at university, because computer science professors—though brilliant and gifted in their specialties—know little of the *craftsmanship* of great code for real-world product development, and they certainly don’t spend time pair-programming and coaching students in any meaningful way. The professional-programming skill of professors in India and China is arguably even worse.

So, in these countries especially, the average person first joins an outsourcing company with very low programming skill. It gets worse: The **four-year programmer problem**. After about that duration, a person expects to stop being a programmer and become a manager. Motivation is understandable—more money and status.

So there is a pool—on average—of programmers of low skill who leave the value work after *just starting* to achieve a modicum of skill and productivity.

This is very different from the lean product development principle of *long-term great engineers*. In Toyota, an engineer remains a working engineer for a significant time.

If you are a client looking for a partner that can demonstrate lean principles in development, investigate the average term and skill of the programmers. By the way, it is not useful to do this by asking the company—you will not learn the truth. If you really want to know,

the lean *Go See* principle is needed: Visit the site and randomly sample the people around the building, sitting with programmers and looking in depth at their code.

If you are an offshore organization wanting to become truly agile and lean, an improvement in culture and incentive is needed so that a person feels valued, and is paid appropriately, to want to remain working as a hands-on software engineer for many years.

Try...Experts coach/review rather than dictate design

See “Avoid...Architects hand off to ‘coders’” on p. 308.

See “Avoid...Create ‘designs’ and then send them for offshore implementation” on p. 316.

Programmers in India and China are typically young, especially ill-prepared by their universities for great code craftsmanship, and suffer the *four-year programmer* problem.⁸ One quick-fix and unskillful response to this problem is that onshore people (for example, in the USA or Europe) or very experienced offshore ‘designers’ will do detailed design diagrams and specifications (such as UML diagrams) and then dictate that the offshore ‘coders’ implement them. More sequential life cycle mindset, more big batch transfers, more handoff waste, and more design specification that tends to ironically lead to more bad code. Naturally, this is de-motivating for offshore developers who want to grow, demonstrates the lean waste of *underutilizing people*, and is inconsistent with the lean culture of coaching others to improve.

Instead, try the experienced onshore or offshore ‘designers’ in a different role: coach and reviewer. Let the offshore developers take their own *small* steps in creatively coding/designing something. Do more pair-programming by juniors sitting with seniors, pair-programming by networked shared desktops and Skype Audio or Video with onshore experts, and more code review in short cycles. Move from a design-dictating role to a design-educating role.

8. This problem exists, in varying degrees, in all countries.



Figure 13.13 start
of a design
workshop, Valtech
India

CHOOSING AN AGILE OUTSOURCER

Avoid...Outsourcers saying “Leave it to us, we *do agile* for you”

The large Indian and multinational outsourcers will quickly and easily adopt genuine agile development, lean thinking, and Scrum because they will *gladly* embrace

- giving up management command-and-control
- encouraging and supporting people to serve as hands-on master programmers for 15+ years
- close engagement between the hands-on developers and customer, without intermediaries
- simple, free tools
- managers and architects who are pair-programming coaches
- real transparency—of delays, what's going wrong each day, ...
- the candor to say “we don't know”
- reducing the overhead management roles they charge to clients
- stopping sequential life cycle practices and mindset
- empirical process control
- replacing *cube farms* with team rooms and visual management

Oh... sorry, that was another universe!

As ‘agile’ goes through the predictable fad phase, these outsourcers will likely mutate it into something they can redefine, control, and sell to appear up to date. “IBM Agile,” “Accenture Agile,” “Infosys Agile” or such like—better than mere regular Scrum, able to scale

with special IBM knowledge, requiring special managers, a private recipe or cookbook requiring expert consultants and customizing... for a fee. And of course, for-fee commercial tools from the vendor. Such companies will demonstrate the “agile means iterative,” “secret sauce illusion,” and “*do agile*” mistakes rather than *being agile*, transparent, and adaptive with empirical process control.

For example, one “agile offshore” book written by a manager at an Indian outsourcing company is rife with un-agile practices and misunderstandings, even mutating the eleventh agile principle, *the best architectures, requirements, and designs emerge from self-organizing teams*. Apparently, self-organizing teams are not acceptable at his company, because the author decided to *rewrite* the principle for his audience: *Realize that the best requirements, architecture, design, development and testing come only from an organized and motivated team* [Venkatesh08]. Well, there’s hardly any difference between a self-organizing team and an organized team!

Avoid...Outsourcers with top-heavy management

In India and China, becoming a manager is an important goal—your mother-in-law expects it and it pays better. Combined with the *four-year programmer problem*, traditional outsourcing companies are top-heavy with management. Naturally these people, typically with the best of intentions, are expected to *manage*—plan, measure, direct, report, set targets, reward, and so forth.

In a top-heavy outsourcing organization, this creates a tension that inhibits the adoption of real Scrum and agile principles because so many people are actively participating in and reinforcing traditional management ‘control.’ This effectively prevents the goal of self-organizing teams, empirical process control, and so forth. This large population acts—unwittingly perhaps—as an *organizational antibody* to agility. In such offshore outsourcing companies one typically sees **fake agile**—‘agile’ is trivially misrepresented as working in time-boxed iterations, while all the other traditional structures remain—project and program managers directing teams, sequential mindset, single-function teams, handoff, command-and-control, and so on.

Such companies are unlikely to have the structural ability to be truly agile or lean—the cards are stacked against them.



Figure 13.14 Daily Scrum, Valtech India

Avoid...“four-year programmer” outsourcers

This reiterates a general tip applicable to all kinds of partners: Beware *outsourcers* with weak programmers. *Go See* the code and the programmers-while-programming with your own eyes, and do not pay attention to *reports* about the quality of people.

Avoid...Outsourcers whose environment does not “walk the agile talk”

OK, it is a mixed-up metaphor, but the point is that lean principles and agile methods encourage a co-located team in a team room without communication barriers between members. The leadership has to make a non-trivial investment in a supportive physical environment, one sign of their long-term, meaningful support for a serious agile transformation. If you visit a potential “agile outsourcer” and people are working in a traditional cube-farm-from-Dilbert-Hell, it’s one sign of *fake agile*.⁹

Years ago, when we first visited the just-acquired Valtech Bangalore site, it had a typical Indian cube-farm layout. With the full support of the CEO, we basically ripped out the interior of the building and created team room environments for about 500 people (see

9. Beware also “cargo cult agile adoption” in which the organization has the superficial trappings in place, such as a team room and a daily stand-up, but there is no culture of lean and agile principles.

Figure 13.15 and Figure 13.16). Because offshore outsourcing involves many quickly shifting project sizes, we created a flexible-wall solution out of hundreds of tall rolling whiteboards, which also serve as tools for creativity and visual management surrounding each team. Some team rooms have fixed walls, but most are flexible.

Figure 13.15 site when first acquired; note lack of interaction, Valtech India



Figure 13.16 team rooms after change, Valtech India



Avoid...Outsourcers with analysis, coding, or testing ‘factories’

The domain of manufacturing factories is *profoundly* different than product development—a domain of learning, *inherently* high variability, and creative endeavor. So, any outsourcing company—and there are several in India and China—that promotes the factory metaphor for software development is on the wrong track, and clearly does not understand feedback loops. And in the case of a *testing factory* it is promoting the lean wastes of handoff, test-at-the-end, high WIP, and over-processing with manual testing that should instead have been automated with acceptance TDD. This reinforces big batches of work on long queues, and single-specialist teams transferring partially done work to other teams.

That is inconsistent with lean or agile development, and the cross-functional Scrum team. Such outsourcers are far from being good candidates for understanding and promoting real agile development.

Avoid...Try...Large outsourcers

Avoid—With rare exceptions, there seems to be an inverse relationship between the size of the company and the quality of the engineers. Also, in big organizations the management is often far away from the real work, out of touch with how to do it skillfully, and does not coach others—in contrast to the Toyota Way and Toyota as a big company. Small technology-focused companies tend to be better. Agile offshore needs the ninth agile principle: *Continuous attention to technical excellence and good design enhances agility*. And lean principles emphasize the need for “towering technical excellence.”

Try—On the other hand, big offshore outsourcers are federations of many smaller business units, organized around domains such banking, insurance, and so on. It *may* be possible to find really high-quality programmers in one of these smaller units, but don’t count on it.

And don’t believe it by claims; rather...

Try...Interview outsourcer-programmers by programming

“Are you a good programmer?” “Oh, yes!” “Great! You’re hired.”

Not good enough. The only way to *really* evaluate a programmer’s programming skill is to *look at their code*; talking, questions, project histories, and resumes are not enough. And in the case of countries that especially suffer from the “four-year programmer problem” this direct visit to *gemba*—the source code itself—is doubly important.

Therefore, take the time, when selecting a team offered by an outsourcer, to give a programming challenge for each candidate. Then observe the quality of the code they create—while they are creating it. Either sit with them while they are programming the solution, or if they are in a remote location, use a shared desktop technology, combined with a Skype video live session.

Of course, this test applies to candidate ‘architects’ as well.

Try...The great programmers forever

Once you have gone to the trouble of finding great programmers (as discussed in the prior suggestion) ask to keep these same people with you...forever.

Try...Improve together with your outsourcer

Toyota emphasizes building partners with stable relationships, trust, and coaching in lean thinking. Once you have chosen an outsourcing partner, encourage them to participate in a lean and agile adoption with you. For example, (1) create (at the outsourcer site) offshore team rooms (rather than cubes) with support for visual management, (2) install Skype or similar video technology in the onshore and offshore team rooms, (3) remove intermediary outsourcer project managers, and (4) educate the offshore teams in Scrum—for example, through courses, books, and coaching.

APPRAISALS, CERTIFICATIONS, AND CMMI

CMMI is not necessarily related to offshore development; it is also adopted by companies hoping for USA Department of Defense (DoD) contracts, among others. However, offshore development is a very common context in which one is likely to see CMMI because of its widespread adoption among India offshore outsourcers in the 1990s. Hence, the topic is covered in this *Offshore* chapter, although the tips are relevant to any CMMI adoption.

This issue deserves careful analysis, and so is examined in some depth. To start, it is useful to know there are misconceptions.

CMMI Misconceptions

CMMI is based on a sequential life cycle and requires heavy documentation—Not true. The CMMI process improvement framework is life cycle neutral; there is nothing in it that requires the use of a sequential (for example, ‘waterfall’ or V-model) model.

The CMMI does not mandate heavy documentation—it is relatively free of prescriptive advice on *how* to perform a practice or process.

Why did that misconception arise? First, the original CMM—rooted in 1970s traditional development experience at IBM and in the USA defense industry (at MITRE and other companies)—did have a subtle bias toward sequential models, coming largely from people with a background in serial phase-based development. Second, many of the early (and current) CMM/CMMI consultants and appraisers had a background in sequential processes and documentation-driven development, since the group has been weighted toward people from the USA defense industry and from traditional organizations.

In the evolution from CMM to CMMI, the SEI has been careful to be life cycle neutral and not prescriptive in how a particular practice should be done. But some leaders in the CMM/CMMI movement have complained that it has been distorted by misinformed consultants and appraisers. For example, Mark Paultk, the lead author of the Software CMM, wrote about distortion of both agile methods and CMM:

We have had the same problem [distortion and lack of understanding] in the [CMM] software process world, and I must admit to some amusement at watching the agile methodologists struggle with the abuses of their methods. just as I have struggled with those who abuse the Software CMM. [Paultk05]

This distortion seems to exist in all new movements because it attracts novice consultants, including in lean development, Scrum, and Extreme Programming.

CMMI and agile methods or lean thinking are fundamentally incompatible—Not true. CMMI emphasizes process improvement, not a specific method. Mark Paultk, a key CMM leader, has written in favor of Extreme Programming and other agile practices [Paultk01]. And there are now experience reports on applying Scrum and other agile practices in an organization that is required to apply CMMI (usually because of USA defense contract requirements).

Craig has led Valtech India's (which was a SW-CMM adopter) adoption of large-scale Scrum and agile and lean principles, and helped

agile adoption at several other CMM and CMMI-adopting companies in the USA and Asia.

However, all that said, there *are* non-trivial differences in the paradigm of process improvement in CMMI versus agile or lean models. It seems rare that a group adopting agile methods will voluntarily also adopt the CMMI framework, unless required to participate in the DoD or offshore contracting world.

Barry Boehm gave one of the keynote speeches at the *Extreme Programming and Agile Processes 2006* conference in Finland. He posed the question, “What does a traditional auditor or appraiser look for?” His conclusion [Boehm06]:

Traditional Auditor

- *processes & tools* over individuals & interactions
- *comprehensive documentation* over working software
- *contract negotiation* over customer collaboration
- *following a plan* over responding to change

Agile Manifesto

- *individuals & interactions* over processes & tools
- *working software* over comprehensive documentation
- *customer collaboration* over contract negotiation
- *responding to change* over following a plan

The left-hand column typifies themes in traditional process improvement advice and audits. It *is* possible to integrate CMMI with Scrum and agility, but without careful attention, it will be inconsistent with agile values.

CMMI prescribes how to do a practice or process—Not true. The CMMI is essentially silent on how to do any particular practice. This misconception is demonstrated by CMMI consultants who confuse means with ends and lead companies to incorrectly believe that “doing CMMI” implies writing documents, developing in serial phases, and observing other specific practices.

CMMI promotes a staged approach with maturity levels—Not true. It *is* true that the original CMM was based on a broad-based staged approach of moving up the ‘maturity’ levels. However, the CMMI introduced the *continuous representation* that moves away from broad organizational stages and maturity levels, and instead

focuses on improving only a few processes of interest to the organization, each of which may improve at different rates.

A good certification implies good code—Not true, and this may be the most practically important issue, because the key output you are paying for is *the code*. You may recall the Valtech story (“Avoid...“four-year programmer” outsourcers” section on page 477) of evaluating a “CMM level-4 certified company” that created *extraordinarily bad source code*. This is an often-repeated pattern—the illusion of good-quality software at companies making CMM and CMMI claims. See [Cone08] for more stories of poor code from highly certified companies.

The SEI certifies company appraisals and levels—Not true. Contrary to popular myth, *the SEI does not certify the CMMI maturity levels given to an organization*. Quotes from the SEI [SEI08]:

Does the SEI monitor or certify appraisals? No.

How does my organization receive CMMI certification?

The SEI does not certify the results of any appraisal nor is there an official accreditation body for CMMI. True certification of appraisal results would involve the ongoing monitoring of organizations’ capabilities, a shelf life for appraisal results, and other administrative elements. The SEI does not have a defined requirement for periodic follow-up after appraisals, nor does it accept legal responsibility for the performance of appraised organizations.

CMMI lead appraisers go through an SEI education and certification process. But there is no SEI guarantee regarding the validity or veracity of an assessment.

CMM/CMMI appraisals are reliable and the ‘number’ is meaningful—Not true. Here’s the bottom line:

Do not believe that an appraisal, rating, or certification in *any* process improvement model—*including Scrum, agile methods* and ISO certification—means much of anything, other than the ability to *somewhat* pass an appraisal at least once.

“Bursting the CMM Hype” in *CIO Magazine* [Koch04] explores the myth of CMM and CMMI ‘certifications’—mythical because fraudulent. “*When you talk about something simple like a number and lots of money is involved, someone’s going to cheat,*” said Watts Humphrey, a CMM founder. The article examines the problem of false claims, bribery, assessing a tiny group and then suggesting company-wide assessment, and other dysfunctions in the CMM and CMMI industry.

Note also that CMM and SW-CMM were retired by the SEI on Dec. 31, 2007, and any claimed assessment in that model is no longer recognized. Yet one may still find companies (usually offshore outsourcers) that promote themselves ‘certified’ despite outdated assessments.

Pay special attention to the common case in which one small *department* was assessed, but the offshore outsourcer generically advertises “company X was assessed at maturity level 5.”

How meaningful is a numeric CMMI capability/maturity level ‘score’?... We suggest that something as complex and subtle as the process capability of a development organization can not be meaningfully summarized by something as superficial as a number between 0 and 5. Even a quality as “one dimensional” as the financial performance of a company—contrasted with the rich and subtle multi-dimensional complexity of analyzing process capability— involves a slew of numbers: earnings per share, EBITA, operating income, and dozens more.

How many give the appraisal, and how is it done?... Is it from a team of 100 up-to-date master engineers who all spend many months within the company sitting at the place of value work with hundreds or thousands of developers, inspecting their code and pair-programming for thousands of total hours so that they can see with their own eyes what is really going on, through in-depth long-term direct experience?

Or is it a few appraisers who visit for a few weeks (after the organization has spent weeks in careful trial runs and preparation to pass the inspection), interview only 50 people for a couple of hours each, and look at documents? What kind of insight into the true nature of

a development organization and of the code/design quality (the essential output) does that provide?

Contrast this with the Toyota practice where master *engineers* spend a long time *working closely inside* their potential and existing partner sites to see with their own eyes what is really going on.

These two alternative approaches to evaluation reflect an absence or presence of the lean *Go See* behavior—go see at the *real* place of value work. In software development, the most common real place of work is *at the source code*.

Does a good assessment indicate effectiveness or reality? Examples...

- We were once invited to a company assessed at CMM “maturity level 5” to kick off an agile adoption initiative. While admitting they had not derived enough benefit from CMM (which is why we had been invited), nevertheless, the managers who had invested so much in CMM explained how they applied it. However, during a break, a developer and first-level line manager who were listening to the presentation came and said privately, *“Don’t believe a word of that! If we had to follow our CMM rules we’d never get anything useful done. We ignore it and fake the reports so that senior management is happy.”*
- A client with CMMI “high maturity levels” has poured many millions of dollars and long effort into achieving that goal. They sent us a proposal (for coaching) because—after years of trying—they have finally decided to move away from CMMI to agile development (*their next silver bullet?*). Why? They now say that their CMMI-based processes and improvement strategies do not work well, and have created, to quote, “high management costs, long delivery period, and low enthusiasm and creativity of project members.”
 - Attempts to adopt Scrum or agile development can also be ineffective and superficial if people do not focus on improving the underlying *system*. However, that said, we have not heard of even a *bad* agile adoption that was characterized *creating* “high management costs, long delivery period, and low enthusiasm and creativity of project members.” (Though it is perhaps possible).

What is measured?... An appraisal may focus on *adherence* to repeatedly following defined practices rather than on the question of their inherent skillfulness. For example, defining that a group follows a sequential life cycle with large batch transfers, and then repeatedly doing so, may not be skillful but may pass an appraisal. This issue reflects the common anti-pattern *measure what is easy rather than measure what is useful*. Such appraisals lead to a focus on conformance and overhead rather than innovation, removing waste, and delivering value quickly to the client.

What is reported?... If a group spends three million euros on a CMMI program, what will they report to their sponsoring group? Data reported about process improvements often reflects what people get rewarded for and what attracts money [Austin96]. If people are rewarded for a successful ‘agile’ adoption, the ‘data’ will reflect success. Likewise with CMMI and other initiatives.

Conclusion... Consider the quotes below from several offshore outsourcing company websites. The following claims will be attractive to people following the superficial advice to find an offshore outsourcer with favorable CMMI ‘certification’ or a high number, but in fact it is something of a mirage.

[Accenture India] *CMMI Level 5 certified for IT Application Development* [Accenture08]

[IBM India] ... *certified at CMMI Level 5* [IBM08]

[Wipro India] *Wipro is the World's first CMMI Level 5 certified software services company* [Wipro08]

[HP India] ... *centers in Asia are CMMI Level 5 certified.* [HP08]

All large non-SEI studies show that process improvement with CMMI significantly helps productivity—Not true. The COCOMO II model [Boehm00a] is based on large and longitudinal studies of effort and productivity¹⁰ in systems development—including many groups applying CMM and CMMI for process improvement. The COCOMO II research data shows that “process maturity” (based on a key CMMI goal) is among the *least* important factors for productivity. What is the top factor, far more influential than CMMI-

10. Note that the definition of *productivity* in creative knowledge work is problematic—the definitions are questionable or debatable. This large can of worms will remain unopened—for now.

oriented process maturity? *The capability of the people on the development teams*—reflecting the Toyota lean perspective and the agile value of “people over process.”¹¹

Many roots of CMM and CMMI are in the USA defense industry; it is sponsored by the DoD. And the defense industry was its only meaningful area of adoption before interest from Indian offshore outsourcers. In addition to the COCOMO II data, other large software ‘productivity’ studies show that the *defense industry*—with many years of CMM and CMMI application—has the worst productivity of any sector [Jones08]. *And the CMMI leaders (authors, SEI managers, appraisers, ...) continue to be dominantly people from the defense industry.*

Leaving aside statistics, do you believe that the defense industry is a great place to look for value generation and efficiency?

CMM/CMMI is strongly associated with successful improvement—Not true. For example, in a report in *IEEE Transactions on Engineering Management* the researchers summarize:

While some organizations have achieved various levels of success with the CMM, the vast majority have failed... The most recent report from [SEI] puts the rate of failure at around 70%. [NN03]

There are “Scrum plus CMMI is important for success” stories—Maybe. One oft-cited story of success with Scrum and CMMI is the story of Systematic, a company that works in the defense industry (among others) [SJ07]. The paper is meant to suggest the combination is important and desirable. A key point is that the article focus is on the CMMI *generic* practices:

-
- 11. Interestingly, according to the COCOMO research, the second-most important productivity factor is “low product complexity.” Note that sequential development with big batches *increases* product complexity (harming productivity), but that Scrum encourages small batches that support low product complexity.

For the purposes of our discussion, we will look at the 12 generic practices associated with maturity levels 2 and 3 in the CMMI and how they might help an organization use Agile Methods.

What are these CMMI generic practices? They include *train people, provide resources, manage configurations, correct root causes of problems, identify and involve relevant stakeholders*, plus some others.

We are not convinced that CMMI is needed to do well in these essential *basic* areas. In essence, the article makes the point that having organizational focus and improvement in training people, providing resources, and so on, is important. It is not a necessary conclusion that CMMI is an important ingredient in a Scrum-adopting organization to achieve skill in these generic areas.

Critically, the paper does *not* explore the key controversial elements of CMMI, such as the Specific Goals, Specific Practices, and the appraisal and rating framework—that are central to the contentious questions of value and waste in this model.

One final quote from the paper is noteworthy:

Scrum now reduces every category of work (defects, rework, total work required, and process overhead) by almost 50% compared to our previous CMMI Level 5 implementation...

In other words, after spending major efforts moving from one CMMI level to the next, finally the group reached level 5. Given CMMI claims, it is reasonable to then expect a high degree of efficiency—that most waste and inefficiency was driven out from the operations.

But when this group adopted Scrum, it cut “every category of work” *by half*. What does that tell you about the efficiency in the CMMI Level 5 implementation?

Because of these misconceptions, the first tip in this section is...

Avoid...Believing CMMI appraisal or certification means much in creative R&D work

And that goes for Scrum certification, ISO certification, and all others as well. Certification or appraisal does not imply good things will happen—especially in the variable and creative work of systems research and development.

There are practical concerns about choosing an offshore outsourcer based on a CMMI appraisal, as was covered in the previous *Misconceptions* section:

- Perhaps most important, a good appraisal does not imply good code, and code is essentially what you are paying for.
- Appraisals are not reliable, and a ‘number’ is not meaningful.
- CMMI process improvement is not a key factor for productivity.
- CMMI is not strongly correlated with successful improvement.

And other concerns have been raised...

Concerns with the CMMI Framework

Not needed by successful development companies—ThoughtWorks India—arguably one of the most talented offshore outsourcing development group in Asia, successfully producing difficult systems with solid code—has no CMMI program. Also true of Google, Nokia, Adobe, Apple, Microsoft,¹² Activision, Oracle, eBay, Xerox, MySpace, Nintendo, Yahoo!, ... and almost every other really successful software-intensive company in the world. If the approach and goals of CMMI are critical to successful and sustainable quality development and improvement, why is it virtually ignored by almost every great product-development company of the world? On the other hand, many of these companies have adopted agile development, usually based on Scrum.

12. Ironically (but predictably), Microsoft offers the “MSF for CMMI” process guidance (for customers buying their tools) but does not follow a CMMI program themselves.

In “The Immaturity of CMM” in *American Programmer* [Bach94] the author questions assumptions of the model. On its lack of use at leading software-intensive product companies, he writes:

At worst, the CMM is a whitewash that obscures the true dynamics of software engineering, suppresses alternative models. If an organization follows it for its own sake, rather than simply as a requirement mandated by a particular government contract, it may very well lead to the collapse of that company's competitive potential. For these reasons, the CMM is unpopular among many of the highly competitive and innovative companies producing commercial shrink-wrap software.

Commenting on “silver bullet” mentality with any process model (a mistake also repeated in the adoption of agile methods), Bach writes:

Still, it has become a lot clearer to me why the CMM philosophy is so much more popular than it deserves to be. It gives hope, and an illusion of control, to management. Faced with the depressing reality that software development success is contingent upon so many subtle and dynamic factors and judgments, the CMM provides a step-by-step plan to do something unsubtle and create something solid. The sad part is that this step-by-step plan usually becomes a substitute for genuine education in engineering management, and genuine process improvement.

Manufacturing and mechanistic mindset mistake—The USA DoD has been driven by good intentions in their promotion of the CMMI—to make sure that taxpayers' money is spent wisely when the DoD acquires systems. Unfortunately, the solutions devised are not a clean fit to the creative, highly variable, innovative and learning-oriented work of systems research and development; it is as though there is a belief that the model for *growing* software can be similar to the model for acquiring buildings or manufacturing hardware—the mistake of thinking *development* is not development.

CMMI emphasizes repeatable deterministic processes, for a domain (creative product development) that is inherently non-repetitive and stochastic—domains where empirical process control is appropriate.

The analysis in “Competing Values in Software Process Improvement: An Assumption Analysis of CMM from an Organizational Culture Perspective” was published (in the *IEEE Transactions on Engineering Management*) by researchers not involved with agile methods or otherwise biased to one model. It reiterates the point that CMM is but one viewpoint from a subset of process people with a shared value system—rather than the universally accepted model—and that this group emphasizes a mechanistic view:

CMM espouses an organizational culture form in which people and processes are treated mechanistically like a machine, for which the operation and performance can be quantified, measured, and controlled. [NN03]

The analysis posits that the CMMI and SEI value system is that *process is the most important factor*.

For example, on the opening page of *CMMI*, “...three critical dimensions... people, procedures and methods, and tools and equipment. But what holds everything together? It is the processes used in your organization” [CKS07].

And in a popular SEI text *Software Process Improvement* the author (from IBM) writes, “An interesting view is to look at organizations as a set of permanent processes, while management and employees are transient phenomena that pass through these processes to serve and enable them, and then leave to be replaced by others” [Zahran98].

This mechanistic view extolling *process over people* is the polar opposite of the first agile value: *Individuals and interactions over processes and tools*. Toyota says the critical dimension is primarily their *people* and *kaizen* culture. Within the company they say, “Build people, then products.”

...experienced leaders within Toyota kept telling me that these tools and techniques were not the key to [Toyota Way]. Rather the power behind [it] is a company’s management commitment to continuously invest in its people and promote a culture of continuous improvement. [Liker04]

No formal theoretical basis—CMMI has no formal theoretical basis and is not widely accepted by all process experts.¹³ It is based

on the values and conclusions of only one subset of people—many with a background in military software or working at IBM with traditional development in the 1970s and 1980s. On the basis of some subset of “expert opinion,” *any* alternative model from another subset of process experts with different views is equally justifiable.

“Taylorist’ assumptions and unsound promotion of “best practices”—To quote from the SEI on the benefits of CMMI:

*The CMMI Product Suite is at the forefront of process improvement because it provides the latest **best practices** for product and service development and maintenance. [SEI09] (emphasis added)*

One assumption in the CMMI model is that there are the “best practices.” This is related to Taylorism, the model of Frederick Taylor and “Scientific Management” in which “the one best way” is identified by managers or process experts, and then workers are trained to follow this one best way [Kanigel97].

“No Best Practices—and no Fractal Practices” section on page 4

The assumption of “one best way” or “best practice” is arguable in creative work with inherent variability—even more so in young fields with changing technologies. A related Taylorist assumption is emphasis on following procedures over individual innovative thinking and experimentation in processes; this is a corollary of “best practice”—if people have been shown the best way, then why change or challenge the status quo?

We guess that no CMMI educator promotes overt Taylorism, and CMMI books do not. Indeed, the goal of process improvement is part of the model. However, having visited many CMM and CMMI-adopting organizations over the years, *it seems there is something in the model or the way it is applied that creates a stultifying effect*. When working in such groups we hear, “No, we can’t do that, it’s not allowed.” Or, “We have to do it this way, it’s part of the process.”

We also see these responses in groups adopting Scrum or lean principles—they have missed the central idea of empirical process con-

13. Nor are agile methods, although as explored in the companion book, queuing theory, control theory, social psychology, and information theory have connections to supporting agile and lean practices.

trol and kaizen. The key, as emphasized by Toyota, is a culture of lean-thinking manager-teachers who encourage kaizen mindset.

Conclusion

Some problems suggested above apply equally to any process improvement framework, including Scrum, because *people trump process* [CH01] and people will *game the system* to get rewards. The first agile value reminds us what is going on: *Individuals and interactions over processes and tools*.

Toyota people understand this. What leads to meaningful improvement and superb execution is not rules and appraisals, but a workforce of great long-term engineers encouraged to be systems thinkers, free to change to remove waste and increase value throughput, guided by experienced-in-the-real-work managers who are also systems thinkers. Hence the internal Toyota motto:



Bottom line advice: If a process improvement approach, including CMMI, is useful *for the hands-on value workers doing the real work*—which has nothing to do with what appears in a report—keep doing and improving it. Otherwise, don't.

Avoid...Believing 'agile'—or any—certification means much

Just as CMMI certification means little in terms of great product development, value generation, or great code, likewise with any agile—or other kind of—certification because *process is only a second-order effect* and *Individuals and interactions over processes and tools*. The following is worth repeating: Data reported about process improvements often reflects what people are rewarded for and what attracts money. If managers are rewarded for an ‘agile’ adoption, the data will reflect success. This measurement dysfunction is deep; the antidote is *Go see with your own eyes at the real place of work*.

Both of us are Certified Scrum Trainers who sometimes teach a “Certified ScrumMaster” (CSM) course. We think education is valuable, especially if the teacher is a master of the subject and a skilled educator. The most successful large-scale Scrum product group transitions we have coached are those in which all of the management team joined us in a CSM. But ‘certified’ is not going to make any difference in the ability to transform mindset or culture; it is merely something HR departments want to hear. If someone hopes that sending people to a two- or three-day course will create ‘agile,’ that is just quick-fix wishful thinking. How can you certify that someone is a systems thinker with meaningful insight, cares to foster self-organizing teams and empirical process control, and has kaizen and a Go See attitude? In Toyota this is not done by certification but is encouraged by a stable culture of trusted manager-teachers who mentor others in these qualities.

Avoid...Toxic CMMI consultants and appraisers

The CMMI is life cycle and practice neutral—it does not say to use a sequential life cycle. Beware an industry filled with *toxic CMMI consultants* who promote sequential development, big batch transfers, single-function teams, cargo-cult ritualistic conformance to check lists and process recipes, and heavy document-driven practices under the mistaken belief that such are required in the CMMI. Avoid those people, and seek out the rare few consultants and appraisers with deeper insight and agile mindset.

If you must adopt the CMMI model to win a government contract, explore *simple practices*. For example, an appraisal in CMMI must provide evidence that certain things happen. A traditional response is textual documentation. Try alternatives, such as using a digital time-stamped photo or digital recording of a meeting.

CONTRACTS

Try...Alternative contract models

See the *Contracts* chapter for alternatives to the traditional fixed-price, fixed-scope contract common in offshore outsourcing. In addi-

tion, a clear understanding of the Scrum Product Backlog, pull versus push, and adaptive iterative planning in Scrum is important for working with this issue.

The alternative contract models have an impact on how one plans an iteration and a release. Again, the *Contracts* chapter is the focal point for the related topics; it also points to relevant material in other chapters, such as *Planning* and *Product Management*.

Try...Fixed price and fixed scope with agility

If you must do fixed-price and fixed-scope projects, look into the *Contracts*, *Planning*, and *Product Management* chapters for tips on this.

TOOLS

Avoid...Commercial tools

Some offshore projects are initiated by onshore decision makers in which commercial (paid) software products are needed for development or version control, servers, testing tools, and so forth. Onshore, the costs of these licenses is not considered an issue. But in “low cost” sites, these licenses are an impediment because they are relatively expensive. If the offshore site does buy a set of licenses, it is reluctant to buy more later because of the additional expense or because it needs to ask for approval from an onshore group.

The fifth agile principle emphasizes giving people the environment they need. Lean emphasizes eliminating delay. For-fee commercial tools are an impediment. Fortunately, there is now a vast selection of high-quality, free, open-source tools for development, version control, testing, and more. Prefer these both onshore and offshore.

“Avoid...Product management negotiating a “release contract” (scope & date) with R&D” section on page 106

CONCLUSION

Those who have tried it—and seen in-depth with their own eyes what is really going on—know that offshore development, especially outsourcing, can be frustrating and wasteful. As so often, it boils down to the first agile value: *individuals and interactions over processes and tools*; plus, setting aside the easy-sounding chimera that the solution can be ordered like a meal in a restaurant (in this case, a restaurant *very far away*), rather than collaborating with the offshore team to create a new recipe. Because agile offshore development does not just imply that the offshore teams adopt Scrum, but also a change in the relationship between those teams and the onshore customers.

Part of that change is forming a direct and more personal connection between the onshore customers and offshore Scrum teams. Remove intermediaries such as project managers and other “single points of contact,” and frequently use video technology so that the true customers and real team members can relate to each other.

Bridging the communication gap is hard enough with co-located customers and development teams—and made worse with offshoring. This makes the practice of acceptance TDD even more valuable when doing offshore outsourcing with onshore customers. Also, bridging that gap takes lots of time and travel; bring offshore team members onshore for some time, and spend time offshore with them.

The lean principle of Go See is *vital* when choosing an offshore outsourcer or team. Do not take on faith a potential outsourcers’ claims regarding the skill or agility of the people. Rather, (1) spend time pair programming and doing other pair work with potential team members, (2) look with your own eyes at the quality of the code they are creating, (3) see if the offshore teams have team rooms with visual management, and (4) observe if they are directed by a project manager or are truly self-managing. Similarly, do not assume that any certification or assessment—including CMMI, ISO, or Scrum—has any value or reality. Rather, Go See *the code and the team*.

That direct observation is what Toyota people do; they spend time inside of a potential partner site to see the reality at *gemba*. Then, once they find a good partner, both parties enter a long-term stable

relationship based on trust, transparency, and mutual support to learn and improve together.

RECOMMENDED READINGS

There is not much material to recommend on agile offshore development; some is awful, and some is essentially traditional offshore development combined with iterations—with no fundamental change in the relationship between the team and customer—rebranded as ‘agile.’

- ❑ All the recommendations in the *Multisite* chapter are relevant, such as *Offshoring Information Technology* and *Agile Software Development with Distributed Teams*.
- ❑ Two of the largest agile-offshore outsourcers in India are Valtech and ThoughtWorks. In Martin Fowler’s online article *Using an Agile Software Process with Offshore Development* (at martinfowler.com) he describes lessons learned at ThoughtWorks, which parallel those at Valtech.

Chapter

- Part 1: Thinking about Contracts 500
- Part 2: Common Topics of Agile Contracts 518
- Part 3: Contract Models 531

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

CONTRACTS

with Tom Arbogast

History will be kind to me, for I intend to write it.
—Winston Churchill

Companies have been successfully writing and using “agile contracts” for many years. For example, at Valtech (where Craig worked), they apply Scrum in the outsourced projects they take on—both in their Bangalore development center and elsewhere—and write contracts that support this. Other agile outsourcers, such as ThoughtWorks, have done likewise.

This chapter is written with two audiences in mind: non-lawyers and (contract) lawyers. We encourage sharing this chapter with legal professionals since some of the material is written for them¹—because most of the work in creating contracts that support agile values and practices is not in the language of the contract, but in *educating legal professionals* about these values. This involves understanding and appreciating traditional legal concerns, addressing those, and helping lawyers grasp the implications of agility and systems thinking. So the early suggestions focus on *understanding*. Later topics focus on a few concrete agile-contract suggestions.

Our co-author of this chapter, Tom Arbogast², is a lawyer with many years of experience in IT projects and contracts, combined with in-depth knowledge of agile principles, systems thinking, and lean thinking. He has worked *three sides of the fence*: (1) as a contract lawyer for *customers* of outsourced IT services, (2) as a legal profes-

-
1. This chapter summarizes core agile concepts already familiar to the expert agile reader, assuming legal professionals new to the subject are an important audience.
 2. thomas.arbogast@gmail.com

sional for IT outsourcers ('suppliers'), and (3) in business development for suppliers (sales agreements influence contract content).

Caution...

For every complex problem, there is a solution that is simple, neat and wrong.—H.L. Mencken

Do not assume that contract negotiations are much less complex or vigorous for legal professionals who grasp the implications of agile principles. It is important to recognize that contracting is an inherently complicated process, even more so in a domain of high complexity and uncertainty such as software development. And lawyers, by training and duty, must continue to pay close attention to the frameworks necessary to deal with a breakdown of trust and collaboration between parties.

PART 1: THINKING ABOUT CONTRACTS

Try...Share these key insights with contract lawyers

The following points are central; they need to be clearly explored with legal professionals:

- The structural and legal aspects of agile-project contracts are the same as for contracts of more traditional development styles. The key difference is the approach to and understanding of *operational process* and *delivery* and how this is captured in or intersects with contracts.
- An understanding of agile and lean principles and systems thinking is necessary for contract lawyers. Why? Because applying these thinking tools leads to less risk and exposure, and that needs to be expressed in the contract. An agile approach enables rapid incrementally deployable deliverables and collaborative decision-making between the parties, and so relieves pressure on liability, warranty, and similar issues.
- Contracts reflect people's hopes and, especially, fears. Successful projects are not ultimately born from contracts, but from

relationships based on collaboration, transparency, and trust. ‘Successful’ contracts contain mechanisms that support the building of collaboration, transparency, and trust. As trust builds between a customer and supplier, the commercial and contract model should ‘relax’ to support increasing “customer collaboration over contract negotiation.”

Overriding fundamental insight

Everyone’s number one priority is to deliver a successful project (in other words, to realize the business case), and each member of the organization, including legal professionals, must strive to reduce local optimizations, silo mentality, and wastes.³ Other (legal) concerns are important, but subordinate to the goal of project success. This is a shift in mindset because many lawyers see their discrete *function* as the priority—that is, to deliver a ‘successful’ *contract*.

Try...Lawyers study agile, iterative, & systems-thinking concepts

A lawyer writing a contract for an agile project (most commonly, done with Scrum) needs to grasp the key ideas before she can articulate an agile contract. We suggest that legal professionals study introductory material in these subjects. For example:

- in the book *Agile & Iterative Development* [Larman03], chapter two, *Iterative & Evolutionary*, and chapter three, *Agile*
- *The Scrum Primer* (www.scrumprimer.com)
- the section on *Continuous Improvement* in *The Lean Primer* (www.leanprimer.com)
- articles on systems thinking; www.thinking.net has both articles and many links
- in the companion book *Scaling Lean & Agile Development* [LV08] chapter two, *Systems Thinking*

3. Wastes: 1. Overproduction of features; 2. Waiting and delay; 3. Handoff; 4. Relearning; 5. Partially done work; 6. Task switching; 7. Defects; 8. Underutilizing people; 9. Knowledge loss and scatter; 10. Wishful thinking.

- this chapter

Try...Appreciate a traditional lawyer's point of view

Legal professionals are wired differently. This rewiring starts from the moment the student enters law school. The concepts of *Professional Responsibility* and *Advocacy* become ingrained into a lawyer's way of thinking. Legal professionals are trained to act, under legal **duty**, to advance their client's interests and protect them against all pitfalls, seen or unseen. How do you define a client's interests? A *client* would probably say simply the successful delivery of the project. A *legal professional* will say she is successful if she protects her client to the greatest degree possible against exposure and risk, while at the same time advancing the end goal of the contract/project.

One has only to look so far as statutory definitions of a lawyer's duty to see how a lawyer perceives her role:

(5) A lawyer should endeavour by all fair and honourable means to obtain for a client the benefit of any and every remedy and defence which is authorized by law. The lawyer must, however, steadfastly bear in mind that this great trust is to be performed within and not without the bounds of the law.⁴

So lawyers view their role as being there to *protect* clients from things they may not even know about. A lawyer is ostensibly trained to be *distrustful*—not necessarily of other people—but of unrealistic expectations and outcomes (the waste of wishful thinking), particularly at the start of a project.

It is important to appreciate this dynamic in the context of a contract negotiation. When a lawyer states that part of her role is to address—contractually—the point where trust deteriorates, it does not imply that the lawyer does not trust the other party. Rather, it means that she does not necessarily trust the *expectations* of the anticipated outcome and is mandated to deal with most anticipated outcomes—good and bad.

4. The Law Society of British Columbia; Rules of Professional Conduct.

The third value of the Agile Manifesto is *customer collaboration over contract negotiation*. Naturally, when first reading this, a contract lawyer will take note, react, and perhaps think, “*That’s nice, but I am here to ensure that my client is properly protected. She can think anything she wants, but I bet she wouldn’t say she values collaboration over contract negotiation when everything goes south and a lawsuit is filed.*” It is the lawyer’s *duty* to consider the ‘unthinkable’ in contractual relationships and provide a framework—expressed in the language of the contract—for dealing with unpleasant outcomes. Lawyers are educated in, and all-too-experienced in, dealing with what happens when relationships deteriorate and trust fractures.

Stare Decisis⁵

Lawyers are creatures of habit. This comes from how the law has developed.

The life of the law has not been logic; it has been experience.—
Oliver Wendell Holmes

It is often said that law is behind the curve and not ahead of it. This is because of the very nature of how law develops. Cases are brought before courts and analyzed in the context of prevailing legal principles. This idea applies to all areas of law, including accepted contracting principles.

Once an issue has been reviewed and analyzed *ad nauseam*, including in legal academic circles, it will then be accepted into common practice. This process could take a decade or more.

Lawyers therefore look to past models that are tried and true, dusting off old precedents and looking to accepted law as a guide. Anything that is perceived as new or a sea change (for example, agile methods) is seen with skepticism and distrust. And this dusting off applies to contract models—it is easier to reuse an existing model than to create something new.

5. *Stare decisis* implies that precedent rules and will not be altered until an alternative is accepted by the courts; it applies principally to countries with a *common law* system.

Traditional project assumptions: Impact on contracts

What do lawyers assume is the nature of software projects? First, it is common that they view it as similar to a *construction* project—relatively predictable—rather than the highly uncertain and variable research and development that it usually is. Second, that in the project (1) there is a long delay before something can be delivered that is well done, with (2) late and weak feedback, (3) long payment cycles, and (4) great problems for the customer if the project is stopped at any arbitrary point in time. *These assumptions are invalidated in agile development.*

Naturally, these assumptions are expressed in the language of the contract, and in the time and attention lawyers give to concepts such as risk and liability for delay, termination, indemnification, acceptance testing, payment criteria, and warranty, amongst others.

Try...Debug common misunderstandings when lawyers are introduced to the third agile value

As mentioned, legal professionals new to agile values will react to first reading *customer collaboration over contract negotiation*. It is useful for non-lawyers to be aware of likely reactions and help address misunderstandings through discussion. And lawyers can correct these misunderstandings by studying these:

False dichotomies—The first and perhaps most common misunderstanding is to misinterpret the agile values in terms of a false dichotomy; that is, “customer collaboration is good and contract negotiation is bad” rather than, to quote the Agile Manifesto, *...while there is value in the items on the right, we value the items on the left more.* Legal professionals need to appreciate that this value does not mean that the contract is subrogated to the collaborative effort, but rather that collaboration is dominant for successful delivery of a project.

Not only should this collaboration be expressed in the behavior of the parties during project development, it can and should be expressed in the contract language—and behavior of lawyers. The contract can define a framework to encourage collaborative prac-

tices, and in this way the legal professionals can support their clients' goals of agility, and, most importantly, enhance project success.

Non-legal ‘contracts’—Another common misunderstanding is assuming that the third value is solely for *legal* contracts. But “contract negotiation” does not exclusively refer to business or legal contracts. It is meant to include the broader notion of agreements or specifications between parties in product development, and whether the emphasis is on *nailing down* these agreements or on ongoing collaboration, learning, and evolution. For instance, a traditional approach includes an early detailed specification of requirements and then “signing off” on these, which are then passed on to development teams for realization—a ‘contract’ of requirements.

“*Avoid...Product management negotiating a “release contract” (scope & date) with R&D*” section on page 106

Legal professionals may exacerbate or ameliorate, by the language of the legal contract, an unhealthy focus on these non-legal ‘contracts’ during project execution. For example, if they draft a contract that contains a clause requiring the definition of and the signing-off on the specification of all or most requirements before starting implementation, there is a lack of agility in the project and an undesirable emphasis on (non-legal) ‘contract’ negotiation.

Try...Lawyers study problems arising from silo mentality and lack of systems thinking

Figure 14.1 (from the International Association for Contract and Commercial Management, IACCM) depicts the top ten (of thirty) contractual terms corporate lawyers were concerned with from 2002 to 2007. It is difficult to imagine that delivery personnel are concerned on a day-to-day basis with most of the issues listed. And it is striking that *a description of the object of the contract is not mentioned*.⁶ That is an astounding observation. The very thing the contract is ultimately about, the expectation of a deliverable (for example, software that will accelerate bills to be processed), is not in the top ten issues.

6. *Delivery/Acceptance* is referenced in item-9. However, this references the concept of delivery meeting a specified acceptance regime, and is not concerned with the underlying object of the delivery.

Figure 14.1 top ten
(of thirty)
contractual
concerns of
corporate lawyers

	Top 30 Terms in 2007	▲▼	2006	2005	2004	2003	2002
1	Limitation of Liability	-	1	1	1	1	1
2	Indemnification	-	2	2	4	10	3
3	Price / Charge / Price Changes	▲	4	6	3	5	7
4	Intellectual Property	▼	3	3	5	3	2
5	Termination (cause / convenience)	-	5	7	7	7	5
6	Warranty	-	6	5	2	2	6
7	Service Levels	▲	11	10	13	-	-
8	Payment	▲	9	4	6	4	11
9	Delivery / Acceptance	▼	8	9	8	12	13
10	Confidential Information / Data Protection	▼	7	8	10	14	15

Consider this scenario: A lawyer at a large company is asked to “measure the success” of contracts the legal department has entered into. The lawyer answers, “We entered into over six billion dollars worth of obligations over the past year encompassing over 400 different contracts, and we have only been sued, or had to sue, on two of those contracts. This is consistent with our year-to-year performance, amounting in my estimation to a 99%+ success rate.”

In the traditional lawyer’s world this is their definition of success, a ‘best’ or ‘optimal’ situation. But of course it is only *locally optimal*. The lawyer did not address if the business case behind the project was realized, if the consumers of the new software were delighted, if the project was delivered, or if too much had been paid over the life of any particular contract.

How do lawyers measure success with respect to a contract negotiation? There is a traditional saying regarding contract hardball that, “you know you have a good contract when both parties are unhappy” because neither party got what it wanted. An agile mindset argues the opposite for both parties, and that a “win-win” approach is really what is mutually optimal.

But regardless of how one measures the ‘goodness’ of a contract, one thing will be constant, and it goes to the heart of a lawyer’s fear in drafting a contract: If something goes wrong, the client will look to the contract (and therefore the lawyer) to ensure that the issue is covered in the client’s favor. This fear, as well as expectation from the client, leads a lawyer to locally optimize strictly from the client’s point of view with respect to *legal problem scenarios*.

This then comes down to the concept of local optimization, or the tendency of actors within a complex system to do the ‘best’ thing in the confines of their own duties and roles, without understanding the larger impact of their choices and actions or ignoring higher-level goals of the system.

The lawyer’s response to the query about contract success was cogent in a local context but does not appreciate the larger systems issues. And why is this? There is...

- a wide gulf between the legal and delivery groups
- endemic silo mentality among contract lawyers
- a lack of systems thinking and resulting local optimizations
- measurement and incentives based on legal concerns

On this last point: Measurement and incentives not only inject dysfunction and locally optimizing behavior into project delivery, they do likewise in contract writing. If professionals in a legal department are rewarded on the basis of legal outcomes, there may be fewer legal issues—but not greater project success.

Form versus function

We have all been in buildings that, whilst beautiful and aesthetically pleasing from a distance, are dysfunctional and confusing internally. This is the difference between form and function. Any legal professional will tell you that, when a contract is finished and dusted, with the ink just drying in the signature boxes, the gleaming end product—which could be a meter-thick stack of document and appendices—is a beauty to behold.

But of course the real test of a contract is in the execution stage of the project, when the people on the ground are working together. During this stage, any need to refer to the contract is arguably a sign of failure—not only of collaboration but also of the legal professionals’ ability to foster a framework for collaboration and success.

That said, if reference to the contract is needed, this is where it takes on a life of its own, much like a building. *It is thus critical to*

envision what the contract will be like in everyday use. This view goes hand-in-hand with a systems-thinking approach.

All this then begs the question: How much time is spent negotiating different areas of the contract? Are legal professionals locally optimizing the concerns and language of the contract (reflecting their silo mentality) on necessary but secondary issues, and as a consequence actually increasing the risk project failure?

Many lawyers spend an inordinate amount of time and concern on ‘legalistic’ areas of a contract (for example, spending bone-numbing hours on areas such as force majeure and liability). These areas are certainly important to consider, but how important are they in the larger picture of ensuring the success of the underlying focus of the contract—the project?

There is an amusing story [Parkinson57] told by the British civil servant, C. Northcote Parkinson, illustrating his *Law of Triviality*: Time spent on any item of an agenda is inversely proportional to the cost of the item. He shares the story of a government steering committee with two items on the agenda: 1) the choice of technology for a nuclear power plant, and 2) the choice of coffee for the meetings. The government mandarins, overwhelmed by the technical complexities and science, quickly pass the technology recommendation of the advising engineer, but *everybody* has an opinion on the coffee—and wants to discuss it at length.

A similar dynamic plays out amongst lawyers writing project contracts: There is an inverse relationship between time spent on the terms that are being negotiated and what is being dealt with on a day-to-day level during execution of the project.

But there is good news with respect to negotiating issues: An agile and iterative approach can—by design—decrease risk. Therefore, pressure on negotiating “big issue” terms (such as liability) is alleviated because agile methods imply early and frequent incremental delivery of *done* slices of the system. The early feedback and delivery of a working system every two weeks (for example) fundamentally changes the dynamics behind negotiating some terms, whose excruciating negotiation in traditional ‘waterfall’ projects is driven by the assumption (and fear) of a long delay before delivery.

One can understand how extreme pressure comes to bear on articulating terms, when viewed in the light of a big “all or nothing” delivery model. Because of the small, iterative nature of deliverables in an agile approach and the ability to stop the project at any two-week boundary (since each incrementally small slice of the system is done and potentially deployable or ‘shippable’), there should be less pressure on concepts such as liability multiples and indemnity.

In *The Fifth Discipline*, Peter Senge states that *systems thinking* and a *learning organization* are ultimately aimed at building “...organizations where people continually expand their capacity to create results they truly desire, where new and expansive patterns of thinking are nurtured, where collective aspiration is set free, and where people are continually learning how to learn together.” In this context, it is critical for legal professionals to acknowledge that the project contract is secondary, though admittedly necessary, to expanding that capacity. So it is critical for the Legal department to acknowledge that the contracts they craft can all too often degrade project success and degrade organizational learning because of a lack of systems thinking, a silo mentality, and local optimization on secondary issues—and this point holds true also for Finance and Human Resources, amongst other departments.

Try...Lawyers study the impact of potentially deployable two-week increments on assumptions and contracts

The Lexus LS versus the Lexus IS



Traditional non-agile projects envision an end product that is akin to buying a top-end Lexus LS. The final delivered solution has all the fine features, nicely polished. And—consistent with the car metaphor—lawyers probably envision an implementation

approach in which one first builds the chassis, then drops in the engine, then the body and electronics, then the interior and paint. So you do not get to see how the final product all fits together until the very end, when all the components are assembled.

The corresponding pressure that this puts on contractual mechanisms designed to protect exposure is enormous. For a customer, it means that there is a delayed, complex, end-user acceptance regime that must occur after *final* delivery. And it means the customer cannot ascertain the quality of the ‘car’ until it is *finally* delivered. In software projects, this means that a customer will want to have maximum protection for the overall scope of the project—usually a liability multiple of the overall cost of the project. This means that a supplier cannot fully be comfortable with the deliverable until the end of the project, and may not therefore be able to recognize total order value until the final deliverable.

An agile project addresses both sets of concerns. It aims to build *not* partial components of a project iteratively, but rather to build a deployable working model of value to the customer that can be accepted and used at each two-week iteration. This is a critical point that legal professionals new to agile concepts do not always grasp; they may misinterpret agile development as incrementally delivering *undeployable* components rather than the agile model of delivering a *useful deployable* system after each short iteration, with gradually more functionality.



After the first iteration, the deployable solution or model may be characterized as a Lexus IS—a simpler entry-class vehicle. As each iterative solution is delivered, the level of the working model goes up in functionality and stature. In a sense, it is like a trade-in of the previous model every two weeks.⁷

The implications for this approach are critical for concepts such as liability and exposure. The customer has something of value that she has paid for and accepted. The supplier can be confident that it has delivered something from which it can recognize revenue and value. If there were, *heaven forbid*, a breakdown in the relationship and the project went to *hell*, each party will be nearly whole in terms of its relative exposure.⁸ The customer will not be left having paid

-
- 7. This analogy is imperfect: Unlike trading in cars, software—and contracts—can *evolve* into something grander each refinement cycle.
 - 8. This simplified analogy does not address the issue of expectation costs, consequential damages, lost profits, and other damages.

for a partial project that is now nothing more than shelfware, and the supplier will not be left with having expended effort on something that it will not get paid for. Granted, the ultimate expectations for either party may not have been met, and the partial system may not have enough functionality to usefully deploy,⁹ but from a pure business and exposure perspective, the relative concerns of the parties are not nearly as extenuated as they may be in a traditional ‘waterfall’ project scenario. *It is vital for contract lawyers to appreciate the implications of this point in how they contemplate, negotiate, and draft project contracts!*

Try...Lawyers study how agility reduces risk and exposure

There are three general areas to be concerned with when drafting a contract:¹⁰

- risk and exposure (liability)
- flexibility to allow for change
- clarity regarding obligations, deliverables, and expectations

An agile-project contract may articulate the same limitations of liability (and related terms) as a traditional-project contract, but the agile contract will better support avoiding the very problems that a lawyer is worried about. That is, a contractual approach that embraces agile methods will actually *decrease* risk and *advance* a client’s relative interests. A contract that does not address agile methods may actually do the opposite of what is intended and *increase* risk and *inhibit* a client’s interests.

An example of this is in the area of requirements gathering and testing of the software that is developed to meet those requirements. In a sequential-development project, the lawyer will enforce (via the contract language) the client’s wishes to articulate every possible case and attendant testing to meet the anticipated requirement.

-
- 9. The well-done quality of the partial system makes it easier for another development group to pick it up and continue.
 - 10. There are clearly many different aspects of a contract, but they can generally be subsumed into these three categories.

An agile approach contemplates that requirements will be articulated in an iterative and evolutionary manner so that time and money is not wasted in developing software for requirements that are not ultimately needed. It also recognizes that money may be better spent for requirements that were *not* recognized at the beginning. Requirements identified and developed in a sequential-development project may never be used, because they were ill-conceived or lacked effective engagement with real users. And after delivery of a system that “conforms to the contract,” requirements still need to be added to meet the true needs. From a contractual perspective, this means that a contract based on a sequential approach will actually increase the risk that the client pays more and gets less than she expects, and that the reverse will occur when the agile approach is understood and addressed in a contract.

This point cannot be overstated, both from a legal and financial perspective. In a sequential-development project, a client could pay much beyond the anticipated cost to get what she initially expected. The attendant contract will not protect against this scenario but will actually promote it by incorrectly assuming that it is quite possible to define and deliver a large set of requirements without ongoing feedback and evolution of understanding.

Contracts that promote or mandate sequential life cycle development *increase* project risk.

For a legal practitioner, the implication is that agile principles can protect a client from things they may not know. This dovetails with the earlier recitation of the perceived duty of a lawyer to her client. Hence, once a lawyer knows about agile principles, she will be neglectful if she does not protect her client’s interests by continuing to allow (by continuing to write traditional contracts) that client to pay for what she doesn’t need and then allowing that client to pay extra to realize what she truly needed.

This means that an agile approach

- reduces risk because it limits both the scope of the deliverable and extent of the payment

- allows for inevitable change
- focuses negotiations on the neglected area of delivery

As an initial imperative, hands-on people from the business area (for the new system) and supplier development-team members must be closely involved and collaborating throughout the life of a project. Legal professionals are encouraged to look for signs that the parties have this intention in mind and to encourage it during contract negotiation and drafting.

This ongoing collaboration of customer and supplier does not mean that a lawyer has vast opportunity for further billables (or if one is internal, that a boatload of more work is now necessary) due to the increase in interaction and *joint discovery*. Rather, it means that contractual constructs must be created to allow for continual customer participation, assessment, and evolution. If the right model is created, a lawyer may have minimal further involvement—at least, unless conflict arises—because the right framework is in place to facilitate the cooperation inherent in an agile approach.

Try...Heighten lawyer sensitivity to software project complexity by analogies to legal work

If you are a coach or manager interested in increasing the appreciation among legal professionals as to the inherently complex, variable, discovery-oriented nature of software projects, try sharing the following thought-experiment with them:

“I want a fully complete project *contract* for my new project: A new enterprise-wide financial management system that will probably involve around 200 development people in six countries involving four outsourcing service providers never used before, and that takes between two and four years to complete. To the exact hour, how long will it take you to negotiate and write the contract with the four providers? To the exact word count, how many words will be in the contract? What will be the exact cost?”

Discuss the parallels between that scenario and software development, and what are realistic versus unrealistic, and effective versus ineffective ways to deal with uncertainty, discovery, and variability.

A lawyer will most certainly say that in this case it is impossible to ascertain, to any degree of certainty, what the end contract will look like, because of the evolutionary nature of contract drafting and negotiation. The lawyer may be able to give a ballpark figure in round numbers as to how much time, generally, it would take to negotiate a complete the contract, say, 200 hours, but would never commit to anything concrete in terms of actual total hours. Yet, ironically, lawyers and business leaders (in a waterfall mindset) expect that IT people will be able to ascertain, via requirements analysis and articulation, what a software project—something of far greater complexity, size, and variability than a “contract project”—will look like and how much it will cost, to a high degree of certainty.

Avoid...Incentives and penalties

It is common for those involved in contracts (legal professionals, sales people, procurement agents, ...) to spend considerable time inventing, negotiating, and writing incentive and penalty clauses in contracts. There is an unquestioned assumption and belief that incentives (related to performance or target dates) or bonuses are beneficial. However, this is inconsistent with evidence-based management research [PS06, Austin96, Kohn93, Herzberg87], and there is ample evidence incentives lead to increased gaming, a reduction in transparency and quality, and other dysfunctions. Research was summarized in the *Organization* chapter of the companion book.

Penalties (negative incentives) lead to the same problems.¹¹

Incentives and penalties also foster a competitive us-them relationship between customers and suppliers, rather than cooperation.

Alternatives?

- simplicity—no performance-based incentives or penalties
- if the customer is extremely dissatisfied with performance, terminate the engagement at the end of iteration
- shared pain/gain models

11. We are not referring to major penalties for gross negligence, but to penalties connected to performance variation.

Try...Share the pain/gain

Some upcoming sections, such as the “Try...Target-cost contracts” section on page 540, present models that share the pain or gain. This can foster collaboration and improving together. For example, in a target-cost model, if the actual cost is lower than the target, the customer pays less and the supplier profit margin is higher.

Avoid...“Quality Management Plan” and “Deliverables List”

Traditional *contracted* outsourced work involves low levels of transparency and trust, and a long delay until some software is *done*. One (of many) classic contract-responses to this is to mandate a conventional “quality management plan” or “deliverables list” that defines a long checklist of documentation to provide, rather than a focus on delivering real value: *working software*. One negotiator shared with us the following story:

An agilist involved in contract negotiation needs to be skeptical about extra documentation, and argue for measuring the cost of producing it—but of course be willing to discuss why an organization may require something. My experience [with deliverables lists] has varied, including in the worst case a very long negotiation with a company that had a huge internal chasm between the agile-friendly business people and traditional internal IT. The internal IT group had been “thrown a bone” by the business person who was the primary negotiator with us, by IT gaining agreement that a “Quality Management Plan” would be agreed to. The IT representative tried to reinstitute waterfall thinking and documentation, and traditional command-and-control, through various drafts of the ‘quality’ plan. Fortunately, we finally succeeded in effectively eliminating both the quality plan and the authoritarian non-value-producing “quality manager” role that the IT representative was trying to build for himself.

What obviates the (assumed) need for a deliverables list, if anything? In Scrum, it is the *Definition of Done* that the supplier teams and client-side business-area Product Owner—rather than an IT manager or legal professional—define and evolve each iteration. Contract lawyers need to understand the Definition of Done because it changes how agile contacts are framed, and how projects are done.

See “Try...Product-level Definition of Done” on p. 170.

Try...Collaborate early and often with lawyers

Collaboration over negotiation and more and earlier feedback loops apply not only to the customer and supplier in an outsourced contracted agile development project—they also apply to engagement with legal professionals.

The system dynamics model in Figure 14.2 illustrates, in broad terms, possible outcomes of increased support for flexibility and collaboration in contracts. But especially relevant to this section, Figure 14.2 also illustrates the impact of more and earlier collaboration of lawyers in business ventures and projects. This closer engagement is part of a larger theme explored in the *Teams* chapter of the companion book: *cross-functional teams*. People often assume—wrongly—that the boundary of a cross-functional team is the people within the R&D or IT department. Not so. For example:

Cross-functional means that team membership includes all the key functions involved in the project, usually Engineering, Marketing, and Manufacturing, at a minimum. [Smith07]

Beyond “at a minimum” is the inclusion of Legal.

The following is a case I (Tom here) saw where the impact of delayed lawyer engagement—and silo mindset—was costly: Business leaders identified marketing and cost-saving opportunities by creating a new web-based billing system. The business case hinged on developing it cheaply—which they *believed* could be accomplished by off-shore outsourcing. Eventually, after a proposal and bidding process, a finalist was chosen and the parties started negotiations. This was and is usually the stage where legal professionals get involved to craft the contract terms and conditions.

Unbeknownst to the business leaders, many jurisdictions have recently tightened the rules regarding export of *personal data* across national boundaries. This only became apparent with the late engagement of some lawyers. Offshoring was infeasible due to these rules. The business case thesis—and project—was invalidated.

Fortunately, the mistake was caught before it was too late. But it would have been easy—given the limited and siloed engagement of the lawyers—to miss this issue completely, leading (as shown in

Figure 14.2) to increased company exposure. And even though it was caught before project commencement, the initial work consumed significant business resources. In addition to the waste of this abandoned work, the subsequent reworking of a new business case and a new cycle of proposals and bidding reduced the time and energy that business people had to explore other business opportunities.

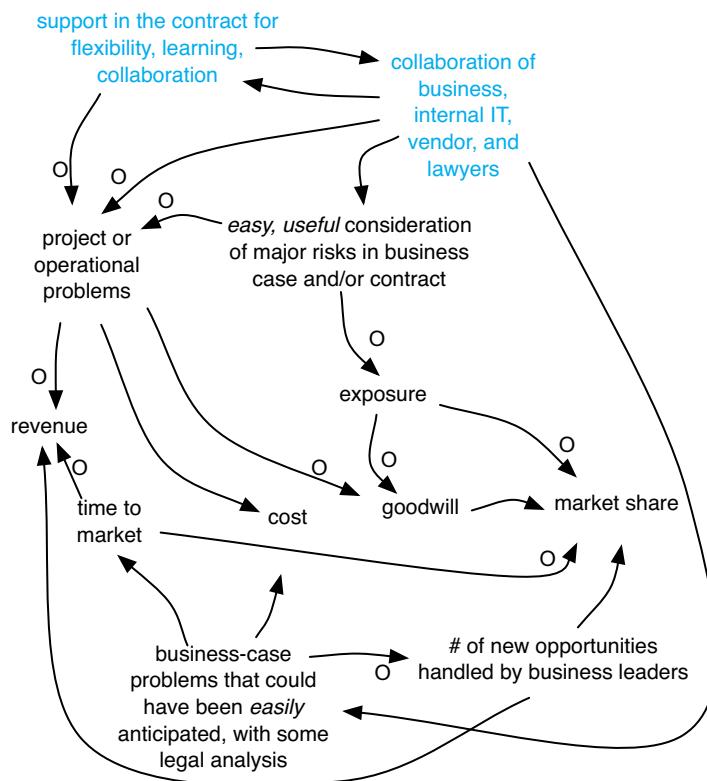


Figure 14.2 system dynamics of degree of contract flexibility and early lawyer collaboration

In addition to the value of cross-functional teams that include legal professionals, this case illustrates a point that IT people do not necessarily appreciate: A contract lawyer has a duty to consider *two* kinds of risks:

- internal project risks

- these are reduced with agile development, and so it behooves legal professionals to support this contractually
- risks from knock-on effects (such as data export violation)
- these are reduced by cross-functional teams that include legal professionals, and early, regular collaboration

PART 2: COMMON TOPICS OF AGILE CONTRACTS

Why No Specific Contract-Language Examples?

When drafting this chapter, we first considered including example clauses from agile-project contracts that have been created at Valtech, ThoughtWorks, and other parties. There are many corporate examples, in addition to variants such as the DSDM and Norwegian PS-2000 contract templates.

However, the feedback from lawyers who reviewed chapter drafts, and the opinion from our co-author, Tom, were consistent: *Copy-paste* is a real and present danger among lawyers and sales people, who—instead of grasping the underlying domain-specific principles (such as agile or lean principles) embodied in contract language—simply copy-paste clauses to draft new contracts. The legal professionals involved in this chapter had a clear message: Focus on principles that help educate both IT people and lawyers about the intersection of agile and lean development and contracts; sample clauses obscure what is important.

Topics Overview

The major *topics* of agile-project contracts (such as acceptance and termination) are the same as for traditional-project contracts. However, the *content* of these topics in the contract—and legal professional’s mindset behind it—contains elements that support collaboration, learning, and evolution.¹²

12. The special case of fixed-price, fixed-scope contracts is covered later.

Agility implies “responding to change over following a plan” and “customer collaboration over contract negotiation,” how does this impact the following contract topics?

- | | |
|--------------------------------------------|---------------------------------------------------|
| <input type="checkbox"/> delivery cycle | <input type="checkbox"/> project scope |
| <input type="checkbox"/> change control | <input type="checkbox"/> termination |
| <input type="checkbox"/> acceptance | <input type="checkbox"/> deliverables |
| <input type="checkbox"/> timing of payment | <input type="checkbox"/> pricing |
| <input type="checkbox"/> warranty | <input type="checkbox"/> limitations of liability |

Delivery Cycle

For legal professionals new to agile development, it is imperative to understand the new delivery cycle. The cycle, from the start, is simply this:

- At the end of each two-week (or up to four-week) timeboxed iteration, deliver a deployable system with useful features.
 - it may have insufficient functionality to be of interest to deploy, but each cycle it is closer to *interesting deployment*

Incremental delivery is not a novel concept in contracts; many identify intermediate milestones, either fixed by date or by goals with associated acceptance criteria or a statement of work. The noteworthy differences for legal professionals to grasp regarding delivery cycle and milestones in agile development include

- *doneness and deployability*—each iteration delivery is *done*, programmed, tested, and so on, and is in theory deployable
- *duration*—smaller, usually two weeks
- *timeboxing*—fixed time but variable scope

Project Scope

Agile contracts do not define an exact and unchanging project scope, although there are variations¹³ in the degree of scope specificity and

change, ranging from low to high. These variations are usually related to the pricing scheme, as will be seen.

Near one end of the spectrum are **target-cost contracts**, in which the overall project scope and details are identified at the start as best as possible (in order to establish the original target cost), but with mechanisms for change throughout. At the other end are **progressive contracts**, in which no (necessary) scope is defined beyond one iteration.

Summary of vision—In the contract

There are contract examples in which the *scope, vision, and business motivation* of the project or product is utterly inscrutable. Avoid that because it suggests that the contract framers are not involved in the project. Rather, they may be demonstrating legalistic, locally optimizing silo mentality—a weakness discussed earlier. Plus, a contract without a project overview is less comprehensible.

Therefore, invite the legal professionals to creatively write from their own understanding—not to copy-paste—a Moore-style vision statement [Moore91]. To achieve this, they will need to participate in project visioning (for example, during a workshop) and other project-engaged activities. Also, include a summary of the general contract, price, and payment model. Place both of these in the contract preamble. For example:

For Accounting and Marketing—Who want to consolidate bills, reduce billing costs, and do targeted marketing with bills—Our new product KillBill is a new billing system—That provides web-based billing presentation and payment, and customized marketing. Unlike our existing billing system—Our new product is web-based and has 80% lower operating costs.

Contract Model: This is a target-cost model. The basis is an expected delivery price of \$YYY. Supplier will deliver and be paid for, on an incremental basis of two-week iterations.

-
13. Specific contract models, including target-cost contracts, are discussed in a later section.

Change Management

The issue of change is largely *inherently* addressed within the overall philosophy of an agile approach because of a re-prioritizable backlog and adaptive iterative planning; no special (traditional) change-management process, board, or request mechanism is needed. Indeed, it is critical for legal professionals to expunge old change-management language from contracts because such language may violate the essence of agility.

This does not mean that all kinds of change management are dispensed with in contractual form. Pertinent concepts in agile-project contracts fall under two categories:

- change in relationships between parties
 - For example, when a party is being acquired by another entity, a fundamental change in corporate direction may occur. Then, existing change-management language commonly used in contracts is likely still applicable. However, keep in mind that, because of the nature of iterative deliverables and concurrent payment inherent in an agile approach, there will be less pressure on relative expectations and the impact a major change will have on them.
- change in project scope
 - This area requires the most care in contracting, to prevent subverting the point of agile development: to make change easy and frequent in the collaboration between customer and vendor. Avoid mandating change-management boards, change requests, or special change processes.
 - But, as with project scope, there are variations in change-management flexibility, ranging from high flexibility without penalty when using flexible-scope progressive contracts, to medium flexibility with shared gain/pain when using target-cost models.

Also, see *Termination...*

Termination

The concept of termination is linked with change control in that an agile project should be amenable to changing course, to the point of actually *stopping further effort at the end of any iteration*. In contrast to conventional project thinking, legal professionals need to understand that *early termination should be viewed as a positive, desirable event in an agile project*, because early termination need not mean failure—it can mean that success was achieved quickly.

Arguably the ideal termination model in an agile contract is to allow the customer to stop, without penalty, at the end of any iteration.

Naturally, if the vendor has dedicated 100 people for an anticipated two years, and termination is unexpectedly much earlier, they likely have an expensive problem on their hands. Thus, agile-termination-clause variations include a sliding scale of penalty to the customer that reduce over time (and iterations).

Termination can be one of the most difficult areas to negotiate in any contract. The key mitigating differences in an agile approach is that (1) the customer has a working system each iteration, and (2) both parties will have clear and up-to-date views on the state of the deliverable. These are crucial points for legal professionals to grasp.

Acceptance

“Is it done?”—“What to do if not done?”—“We have now decided to change our minds and reject the iteration delivery from three iterations ago. Do you mind?”

These are vital questions in outsourced project work, and ambiguity around such issues is a possible source of conflict—and of litigation. Clarity (in so far as practically feasible) regarding *doneness, acceptance*, and *correction* both in the minds of the parties and the contract language should be a leading concern for legal professionals. They can help considerably in defusing the explosives in this minefield with careful attention to *negotiating* a contractual framework for acceptance that encourages *collaboration*.

Acceptance still exists, but is much simplified because of iterative delivery and acceptance, and because acceptance is incremental and adaptively agreed upon for each iteration. Further, agile practices usually include highly automated acceptance testing so that little or no manual (human) effort is required for validation.

Acceptance builds upon itself such that the final acceptance is the culmination of a number of acceptances that have occurred throughout the life cycle of the project, ideally most being repeatedly verified with automated acceptance tests.

In terms of contract work, this means that acceptance definition and negotiation does not have to be a massive comprehensive exercise; only the *framework for acceptance* must be contractually clear.

Broadly, for each iteration, acceptance is based on conformance to a prior agreed-on acceptance-test set, and in the case of Scrum, in conformance with the “definition of done.”

Another element of acceptance in agile development—worth considering in the contract framework—is to include candidate users of the new system in the definition of acceptance and acceptance testing. Legal professionals concerned with a successful project should ask, “Are the right people—the hands-on users—involved in acceptance, and at each iteration are they collaborating with the supplier?”

“Try...Acceptance test-driven development” section on page 42

“Try...Product-level Definition of Done” section on page 170

Sample clauses

In contrast to the chapter’s general avoidance of sample clauses, we decided an example in this case helps clarify the suggestion:

- a) *Customer and Supplier define acceptance of the Deliverable as follows:*
 - i. *Deliverable passes all new automated and manual acceptance tests that were defined before the most recent iteration.*
 - ii. *Deliverable passes all prior automated and manual acceptance tests, verifying that no regression has occurred.*
 - iii. *Deliverable conforms to the “definition of done” that was defined before the iteration.*

- b) *Acceptance tests are incrementally defined together by Customer and Supplier members (“Acceptance Group”), including candidate users of the Deliverable, each iteration. The Acceptance Group reviews acceptance at the end of each iteration, starting at Sprint Review.*
- c) *Customer will have a period of half the business days of one iteration (“Evaluation Period”, “Half Iteration”) after provision to it of the final Deliverable to verify that the Deliverable or part thereof is not deficient.*
- d) *If Customer notifies Supplier in writing prior to the expiration of the relevant Evaluation Period that the Deliverable or part thereof is deficient in any material respect (a “Non-conformity”), Supplier will correct such Non-conformity as soon as reasonably practical but no longer than the length of one iteration, whereupon Customer will receive an additional Half Iteration period (“Verification Period”) commencing upon its receipt of the corrected Deliverables or part thereof to verify that the specific Non-conformity has been corrected.*
- e) *Customer will provide Supplier with such assistance as may reasonably be required to verify the existence of and correct a reported Non-conformity.*

Limitation of Liability

Negotiation of liability clauses is perhaps the most difficult area in any contract negotiation, and an agile approach does not change that. However, it may help. For instance, it can attenuate liability because there is a usable deliverable at the end of each iteration.

For example, a defect in an iterative deliverable may have a lesser impact in operation because the negative consequence is discovered sooner. This does not mean there are no knock-on effects that never have to be addressed through the liability paradigm, but the consequences could be less.

Consider a case that I (Tom here) came across: In a traditional sequential life cycle project for a new billing system, it was discovered, after the “one big delivery at the end,” that duplicate and erro-

neous charges were sent to *many* customers. The fallout and extra costs were considerable: the company had to cut new bills, offer rebates, and repair its image with customers—plus paying to correct the underlying problems. There was then an ensuing fight with the external supplier as to who should pay for the damages- liability.

In an agile approach, the same problematic bills could be sent. But it is also possible that those bills would be sent early to a much smaller subset of customers, using an early release of the system with just-sufficient functionality to field-test this critical feature.

This would reduce cost, exposure, and damage to goodwill. It might also be cheaper to fix because the system would be smaller with fewer entanglements between its software components.

Hence, liability may be attenuated with agile development.

Warranty

Similar to liability, the concerns related to warranty are attenuated in an incremental approach; the risk profile associated with the final warranty is considerably less because of the confidence and acceptance in the deliverable itself, due to incremental acceptance. This is especially enhanced if automated acceptance testing is employed.

As with liability, warranty should be tied to each incremental working deliverable (at the end of each iteration), though there is still an overall warranty to the final product.

Deliverables

Traditional project contracts often include a detailed, prescriptive list of what should be delivered (many documents, ...), and how acceptance of these artifacts is accomplished. These details are sometimes embodied in a statement-of-work or “quality plan” appendix. Avoid such specificity and rigidity—avoid including a detailed deliverables list in the contract. Why?

- It leads to an increase in waste activities rather than a focus on working software, and there is a presumption—possibly untrue—of knowing what artifacts are valuable.

- There is a focus on negotiating and conforming to “quality plans” rather than cooperating to create useful software.
- It reinforces (the illusory) command-control predictive-planning mindset rather than learning and responding to change.
- It reinforces the (untrue) belief that a fully defined system can be predictably ordered and delivered as though it were a meal in a restaurant rather than creative discovery work.

All that said, we have seen custom software for which the *source code* was never provided by the supplier—somebody forgot. So, there are cases in which the customer does not at first understand what is critical. But in such cases, discovery of valuable deliverables can be more simply achieved through frequent incremental delivery and deployment, rather than through a contract deliverables list.

*“Try...Back up
“human infection” with an
agile SAD workshop” section on
page 310*

On occasion, technical documentation to support maintenance is valuable—usually as a learning aid for people new to the system—and its delivery is often specified in a traditional project contract. Yet, maintenance of a recently deployed system is frequently outsourced to the same people that created the system and so have less need for such documentation. Therefore, it could be wasteful to require it as an early deliverable. If, at some future time, there is a *demonstrated* need for documentation for the customer (for instance, if the customer takes over the maintenance work), then it can be created by the supplier, perhaps in a joint agile-documentation workshop with the customer.

Timing of Payment

Perhaps the most popular system is to pay each iteration, once there is final acceptance of the deliverable. In the simple case, such as with basic progressive contracts, payment is 100% of the agreed iteration price. More complex payment schemes are usually tied to more complex overall project pricing schemes. For example, in the various “shared pain/gain” systems such as target-cost contracts, in addition to iteration payments there will be a final deferred payment at project end. Or, at each iteration there may be an X% holdback that accumulates and may be paid at various intermediate milestones.

Pricing

Time and materials

Variations of time and materials (T&M) make for good agile-project pricing models: simple, straightforward. Recommended. Note that T&M applies to both fixed- and flexible-scope contracts.

One traditional concern with T&M, common on sequential development projects, is that customers are locked into a seemingly endless cycle of payments before they see useful results. Another classic concern is whether customers are getting good value for their money. These concerns are ameliorated in an agile approach with a usable system each iteration—progress measured in terms of usable software features, high transparency, and termination that can occur at the end of any iteration.

T&M requires trust and transparency between the parties. That takes sincere effort and time to develop. On several projects, Valtech India has started with variations of fixed-price contracts, and after building trust, has been able to move to variations of T&M models with their clients.

Several variations of T&M limit the customer's exposure and/or balance the pain/gain. For example:

- capped (“not to exceed”) T&M per iteration
- capped T&M per project or release
- capped T&M per iteration, with adjustment—For the next iteration, the price is capped, but if all original iteration goals are delivered and accepted, at a T&M cost below the cap, there is an adjustment payment to the supplier, such as one half of the savings below the cap. A similar shared pain/gain pricing scheme is used in the project-level *target-cost* model.

Fixed price per iteration (per unit of time)

This model has the virtue of simplicity and predictability, and is not uncommon among agile outsourcers. There are two key cases:

- requirements defined and agreed-on before the iteration
- highly flexible or no predefined requirements

In the first case, the issues are identical to fixed-price per large project...The supplier has to clarify the work and have sufficient confidence in the estimates in order not to lose money. The small scope of an iteration makes this much more likely than for a large project. The key issue (or cost) for customers is that the supplier adds a contingency fee to the rate because of the risk associated with variability in research and development work.

In the second case, the key issue is customer trust in the supplier. Transparency, frequent delivery, and easy termination help.

Fixed price per unit of work

Several agile outsourcers offer a fixed price per unit of work (UoW) model. In contrast to traditional development, where a UoW might mean a document or other incomplete solution, an agile UoW reflects the seventh agile principle: *Working software is the primary measure of progress*. That is, the UoW is related to running, tested software features.

These models go by various names and with various systems to estimate a UoW: “price per story point,” “price per function point,” “price per feature point,” and so on.

Points of size versus value—In the schemes we have seen, the ‘point’ is always related to an estimate of size or effort—and so, related to *cost*. Although vendors may claim the price model is *value*-related by using modern agile-sounding terminology such as “price per point,” it is not accurate to say that a “story point” or “feature point” is a value measure—in the idiom of “bang for buck” a *point* reflects *buck*, not *bang*. However, there does exist *in theory* the ability to define points in terms of *business value impact*—where this is measured using a system such as Tom Gilb’s *impact estimation tables* (and he has proposed such a value-impact price model [Gilb05])—but we do not know any application of this approach. For more on this topic, see also the “Try...Prioritize with multiple weighted factors” section on page 141.

We have seen agile outsourcers use one of two schemes to determine the fixed price per point: (1) an average based on several previous projects, and (2) a customized amount. In the latter case, the customer pays the supplier-average point value for a few iterations (or pays T&M, or ...) during which time detailed costs are tracked. Then the supplier and customer agree to a custom fixed price per point, based on this cost plus some profit margin.

This model is congruent with agile and lean themes of being delivery- and value-oriented: Assuming that an agile UoW is loosely related to value to customers (which is not always true), they pay proportional to value received. However, since in the schemes we have seen that a point is related to size or effort rather than true value impact for the customer, this *point* is somewhat lost.

A key issue to attend to in this model—and one that needs consideration in the contract—is a clear and common (for customer and supplier) framework for defining a *point*. For example, only function points are relatively unambiguously defined—and can be identified and verified by certified function-point analysts. In contrast, story points (also known as relative effort points) have no *independent* meaning.

Pay-per-use models

XPLabs (an agile-development company in Italy) promotes pay-per-use contracts with their customers. Every ‘use’ (usually, a transaction) of a custom-built or pre-built system that is deployed for a client is automatically tracked by XPLabs. The customer is regularly invoiced based on frequency of use—a simple payment model. The approach tends to align the interests of the customer and supplier, and both parties win if the system is increasingly used.

If it is a pre-built solution, the model is especially attractive to customers: they have no maintenance or update costs, and only pay extra for custom enhancements. Each new customer deployment is based on the same, simple contract model.

If it is a custom-built solution for only one customer, the contract model for the development work may be any of the other approaches

discussed here, such as T&M, perhaps at a below-average rate to adjust for the future anticipated pay-per-use revenue.

Hybrid shared pain/gain models

There are numerous hybrid pricing schemes in business, well known and not repeated here (see the recommended readings). One hybrid shared pain/gain model applicable to agile development was proposed by Bob Martin [Martin04]:

Discounted fixed price per unit of work, plus discounted T&M—For example, assume the following project scenario:

project estimate	average velocity (140 people, 2-week iterations)	original person-day estimate	payment if \$500 per person-day
100,000 points	4,000 points	35,000	\$17,500,000

In this model, a lower person-day rate is offered, with a complementary per-unit-of-work rate. For instance, assume a standard person-day rate of \$500. The supplier offers a discounted price per person-day of \$150, and a discounted *price per point* of \$122.50.¹⁴ Then:

actual person-days	actual customer payment	change in estimate-to-actual effort	change in estimate-to-actual payment	effective person-day rate
30,000	\$16,750,000	-14%	-4%	\$558
35,000	\$17,500,000	0	0	\$500
40,000	\$18,250,000	+14%	+4%	\$456

Observations:

14. The price \$122.50 is derived: Given a person-day rate of \$150, 35,000 person days, and 100,000 points, it is the price per point needed to reach a total payment of \$17,500,000.

- If actual effort equals original estimate (35,000 person-days, 100,000 points), the customer payment is equal to a simple T&M scheme at \$500 per person-day.
- If actual effort varies, customer payment varies less severely.
- As with target-cost and some other adjustment schemes, the customer and supplier are sharing the pain or gain if the project takes more or less effort than original estimate.

Fixed price per project and target-cost pricing

Both these pricing schemes are covered in the next section. Their impact extends beyond pricing, to overall contract or project model.

PART 3: CONTRACT MODELS

Humans have been writing contracts since the dawn of time, encapsulating their hopes and fears. There are myriad models and variations on those—see the recommended readings for a broader treatment. This section focuses on common models and their variations that customers and suppliers in agile projects will frequently see or consider.

Avoid...Fixed-price, fixed-scope (FPFS) contracts

Fixed-price, fixed-scope—and worse, with fixed duration—contracts and projects tend toward lose-lose situations for both the customer and supplier; customers often do not get what they really need, and suppliers can easily lose money. And in an effort to deliver something within the constraints of price and scope, suppliers will often degrade the quality of their work—reduced code quality, less testing, and so forth. All this leads to an increase in future costs for customers, who will eventually have to pay for the sins of the past, as follow-on change requests¹⁵ to get what they truly need and as increased maintenance costs for software of low quality and high “technical debt.”

Fixed-price bids have added a large risk contingency (a percentage of estimated cost as high as 50%) to the overall price—this premium is usually hidden in the effort estimate.¹⁶ This leads to a reduction in transparency and increased gaming during project execution because the supplier wants this premium as profit rather than consumed budget. Plus, since the early requirement specification that is signed off is almost never what is actually needed (due to myriad factors in this inherently complex domain), the supplier generates further revenue—in India, outsourcers call this ‘rent’—through a series of follow-on change requests, each for an additional cost beyond the original fixed price.

Fixed-price projects have been promoted under the guise of various local optimizations (other than project success); we encourage legal professionals to watch out for these:

- As a customer, most important is to know the cost, for financial reporting or budgeting.¹⁷
- As a supplier, most important is to book the total order value.
- As a sales person, most important is to book the total order value, to get full commission.
- As managers, most important is to avoid using time on the project. We want to order something, go back to other work without ‘interruption,’ and then get it delivered at the end.

But there are companies, that for one reason or another, still try. In that case, the most frequent question we get is, “How do you do fixed-price, fixed-scope projects with an agile method?” First, it is possible; Valtech India and other agile outsourcing suppliers have done so (because of market demand)—though this is their least-favorite model.

-
15. Traditional offshore outsourcers in India enjoy this change-request model because they make so much profit from it; they know very well that their FPFS contracts do not deliver what the customer really needs, and they look forward to the ‘rent’ (as they call it in India) they obtain from evolving the system to meet the true needs.
 16. For this and other reasons, FPFS contracts are also called “latest date, most cost” contracts.
 17. See the *Beyond Budgeting* section of the *Organization* chapter in the companion book, for an alternative to traditional budget processes.

There are often two misunderstandings behind the question of FPFS and agile methods:

- ❑ The first misunderstanding is that the overall project release requirements are not known or estimated before the first iteration when an agile method is used. Not true. Rather, in Scrum, before iteration-1, there may be initial release planning (“initial Product Backlog creation”) in which all identifiable release requirements are clarified and estimated.
- ❑ The second misunderstanding is that requirements must change with agile methods. Not true. Rather, all agile methods provide the opportunity and mechanism to support learning and change, but do not require it. Scrum can be used with a fixed-content Release Backlog—and still provide benefits, thanks to better and more frequent feedback about ways of working, technologies, test results, and smaller batches.

With Scrum or any other approach, there are keys to avoiding ruin when taking on an FPFS project:

- ❑ Apply the best possible due diligence in terms of large, detailed upfront requirements analysis, thorough acceptance test definitions, skillful effort estimation of all requirements, all done with experienced people.
- ❑ Do not allow any changes in requirements or scope, or only allow new requirements to displace existing requirements if they are of equal effort.
- ❑ Increase the margin of the contract price, to reflect the significant risk inherent in FPFS software development—a domain that is fraught with discovery, variability, and nasty surprises.
- ❑ Employ experienced domain experts with towering technical excellence.

Note that a lean culture of long-term, hands-on great engineers and manager-teachers who are experts in the work and coach their teams provides the environment to build the experienced people necessary to reduce risk on FPFS projects.

Payment timing

Payment timing for FPFS is usually per iteration, with a final lump sum on completion (if total payments were not previously exhausted). The per-iteration amount is a fixed percentage of the overall price, either based on an estimate of total number of iterations or if the project is also fixed, on the duration of the predefined number of iterations.

Flexibility in FPFS projects with Scrum

There are several areas of low-risk increased flexibility when executing a FPFS project with Scrum; the ability to...

- displace existing requirements with new ones of equal effort
 - this *replaceability* option is important to highlight
- change the order of implementation of the fixed requirements
- improve the “definition of done” each iteration

Schwaber¹⁸ also suggests two other contract provisions:

- Customer may request additional releases at any time, priced with T&M.
- Customer may terminate early if satisfied early, for a payment to supplier of 20% of remaining unbilled value.

Legal professionals need to be aware that this flexibility can and should be expressed in clauses of an FPFS contract.

Should we do FPFS projects with a sequential, traditional approach?

A question that is sometimes asked is, “If we have to do an FPFS project, should we use an agile method or a sequential life cycle (waterfall, ...) and traditional approach?”

18. Ken Schwaber is co-creator of the Scrum agile method.

There is evidence that sequential life cycle development is correlated with higher cost, slower delivery, lower productivity, more defects, or higher failure rates, compared with iterative, incremental, or agile methods [MacCormack01, Reifer02, DBT05, MJ05, CSSD05, AV07, PRL07].

Consequently, the *last* thing you want to do with an FPFS project is make matters even worse by applying a traditional sequential development approach.

Quite the opposite: If you execute an FPFS project with Scrum, you will have less waste, less queues, less WIP, and you will gain early realistic feedback about the true nature of the project. Based on that early feedback, you can *adjust early* rather than late. Especially in FPFS projects, you want to know *how bad things are as fast as possible*; agile methods enhance that feedback.

There is a more subtle advantage to using an agile approach on an FPFS project: It may evolve into a collaboration-oriented flexible project. Many customer stakeholders understand that the FPFS model may not solve their problems, but it was imposed on them—perhaps by the Legal or Finance arms. Once customers start interacting directly with the agile supplier on the ostensibly fixed project and see rapid delivery every two weeks of a well-done solution, and realize their ability to change the iteration-order of implementation of their fixed requirements (and to replace requirements), and trust and collaboration builds with the supplier, ‘fixed’ can become ‘flexible.’ The customer relaxes, sees the advantages of “customer collaboration over contract negotiation,” and agrees to pay less attention to the original definition and more attention to evolutionary development to meet their real needs.

Finally, after an initial FPFS contract has been completed with Scrum, the customer may be willing to use one of the alternative

Contract Evolution on a Large Agile Project

with Greg Hutchings

An example of multi-phase variable-model contract evolution was a three-year 15,000 person-day project with Valtech (India) and a retail customer. There were four contract phases, created (not pre-planned) in response to learning and adaptation: (1) FPFS per project, (2) progressive, T&M per iteration, (3) progressive, fixed price per unit of work, and (4) progressive, capped T&M per iteration.

(1) *FPFS per project*—The customer was only familiar with sequential life cycle projects, and new to Valtech—trust was low. Therefore, they expected (and got) a traditional FPFS contract based on a sequential life cycle, with an agreed deadline that was not contractually binding. Despite substantial effort to validate specifications and estimates, much—as usual—was discovered to be unknown and misunderstood. After the first year, costs exceeded budget and delivery was months beyond the *wish*. But, some trust had developed by Valtech’s effort to be transparent and responsive, and so the customer agreed to replace the FPFS contract.

(2) *Progressive, T&M with bonus/penalty*—During the first year, the customer was gradually exposed (by Valtech) to agile development principles, and so was open to a new kind of development and a new contract: a progressive variable-scope contract based on Scrum, priced with T&M per iteration, adjusted (at customer request) with bonus/penalty clauses for (1) quality of iteration deliverables, and (2) velocity. Not surprisingly ([Austin96]) these adjustments subtly affected developer’s behavior (a reduction in transparency, more gaming) to “avoid penalties.” Progress (now with Scrum) was *much* faster, and confidence and trust improved with a release each iteration and close customer collaboration. (*continued...*)

agile contract models for a later project. Several agile outsourcers have experienced these positive patterns with their customers.

Try...Variable-price variable-scope progressive contracts

In their purest form, progressive contracts¹⁹ imply completely flexible scope that is adaptively defined each subsequent iteration. They are a good candidate for agile projects [Poppendieck05]. They are master service agreements or umbrella-framework contracts that

19. Also known as, or a variant of, *open-ended variable scope*, *open-ended incremental*, or *indefinite delivery indefinite quantity (IDIQ)*.

(continued...)

(3) Progressive, fixed price per unit of work—In the third contract phase, pricing changed to fixed price per use-case point (UCP), because the client felt it could provide better value for money. Further, the bonus/penalty element was removed. UCP was chosen because the customer was very familiar with use cases, and wanted a unit of work estimate related to that. Payment varied each iteration, based on the number of UCPs delivered. Also, by this time, the customer was comfortable with iterative, collaborative acceptance-test definition to drive iterations and acceptance, and this, combined with better broad-theme integration testing, smoothed and improved acceptance each iteration. This contract form was considered by both Valtech and customer the most successful. However, the unit of work estimation method (UCP) required more upfront requirements analysis than otherwise necessary—a rolling wave of detailed use-case specification occurred usually about two iterations before implementation.

(4) (Support phase) Progressive, capped T&M—The final contract form and phase was established after product deployment, for support and minor enhancement. The customer support budget was fixed (per annum), and the scope of work variable. Therefore, capped (per month) T&M was acceptable—simple to administer, and a high level of trust had been established.

define the overarching relationship and pricing scheme per iteration, but do not define scope. Progressive contracts do not define a total fixed project price—although one variation has a project cap.

Customer exposure is controlled because termination can occur at the end of any iteration—with a working system. If both parties are happy with the relationship, progressive-contract projects can continue indefinitely.

Usually (but not required) before each subsequent iteration, the customer and supplier define the goals of the upcoming next, perhaps with acceptance tests. Sometimes—recurrent at Valtech—the goals for iteration N are clarified during iteration N-2.

Pricing per iteration runs the gamut of variations: fixed-price per iteration, T&M per iteration, and so on.

Variations—At Valtech, a *capped-price variable-scope progressive* contract is common; there is an overall project price cap. Pricing per

iteration is any variation, such as T&M. Also frequent is a *capped-price variable-scope progressive contract with a non-binding Release Backlog*, in which the parties create—before the contract is written—a backlog of release goals. This backlog is included as an appendix to the contract. However, it is agreed that *nothing* in the original backlog is binding. (This is classic Scrum.) Why create the non-binding Release Backlog before the contract is written? It is used to estimate the overall release cap and to provide a starting-point common vision. It is also common to create this non-binding Release Backlog in a prior, separate, *contracted* phase.

"Try...Multi-phase variable-model frameworks" section on page 543

Progressive contracts are a common model with agile outsourcers and the long-term customers with whom they have built a relationship of trust. A frequent pattern (not a recommendation) is

1. early contracts that are variations of fixed price and fixed scope
2. later, a shift to progressive contracts with simple T&M or capped T&M per iteration

Try...Increase flexibility in project and contract variables

A variable-scope, variable-price, variable-date, pure-progressive contract is flexible. Any variable (scope, price, date) can vary in flexibility, depending on the level of trust and collaboration between customer and supplier—or otherwise constrained, such as by government regulation. Contract variations that agile outsourcers have created include the following:

Capped-Price,²⁰ Variable-Scope—Discussed in the previous section.

Capped-Price, Partial-Fixed-Scope—A relatively small set of requirements are fixed—leaving room for learning and adaptation.

Fixed-Price, Variable-Scope—The *optional scope* contract [BC99] is a variation of this model and also fixes the end date.

20. In this section, *price* refers to overall project price.

Bounding risk in flexible contracts: The multi-phase model

The multi-phase model is described in the “Try...Multi-phase variable-model frameworks” section on page 543. Briefly, it implements a longer project from a series of shorter contracts.

If trust is low, customers can bound their risk (and fear) by using a series of short-duration, flexible contracts. For example, one *year-long*, fixed-price, fixed-date, variable-scope contract may be viewed with trepidation. But a series of *two-month*, fixed-price, fixed-date, variable-scope contracts—with the ability to terminate at the end any cycle—is more palatable.

In addition, after a few contract cycles, trust can build. At that point, the customer may shift to a simple progressive contract with T&M pricing per iteration.

Models That Share or Adaptively Shift the Pain/Gain or Risks

There are potential risks and rewards in a project—for both parties. And these may shift over time. For example, FPFS projects appear to shift much of the risk to the supplier, although this is an illusion for reasons previously identified.

Some frameworks, explored next, have been explicitly crafted to share these risks and rewards and to shift risks to the appropriate party that can do something about them.²¹

In the best case, these frameworks engender an increased alignment of motivations for the parties since they both have “skin in the game.” And they may improve fundamental fairness and relationship building. This philosophy is at the heart of the concept of a win-win approach, and it will create the trust and relationships that will foster further business.

21. That usually means placing requirement-related risks ('what') in the hands of the customer, and placing implementation and technical-related risks ('how') in the hands of the supplier.

But contracts do not themselves create trust and alignment. In the worst case, these kinds of contracts are abused as part of a blame-game, to shift pain to the other party and only individually gain.

These frameworks include

- target-cost
- multi-phase variable-model
- profit sharing

Try...Target-cost contracts

Target-cost contracts can help align motivations of both parties. They are used in Toyota with their suppliers, reflecting the pillar of *respect for people* in lean thinking, in which Toyota tries to build stable long-term relationships with suppliers, based on trust and mutual support.

This model assumes an initial release planning step in which overall project scope is identified. This is part of *stage one* to establish the **target cost**:

1. In collaboration between customer and supplier, identify, analyze, and estimate all possible project requirements.
2. In collaboration, estimate the cost of change or scope increase during the project. This is important; target-cost contracts must *realistically* account for overall effort and cost as best as possible.
3. From these two elements, establish the **target cost**.
4. Calculate **target profit**, based on *target cost* (for example, 15% of target cost).
5. Share all details and results with customer (this is important).

During *stage one*, keys to the success of this model are (1) a best-effort, no-wishful-thinking target cost generated with skillful due diligence, and (2) (supplier) open-book costing, so that the customer transparently sees all details leading to the calculation of target cost.

Stage two in a target-cost contract is project execution—for example, with Scrum. As will soon be appreciated, a vital practice for success is tracking all *actual costs* as they are incurred (for example, developer time spent, meetings, hardware), and *transparently sharing* all cost information with the customer in near-real time.

The key aspect of target-cost contracts is a shared pain/gain formula for an adjustment related to the difference between actual and target cost. There are several variations in the formula.

In the simplest case, an example:

```
Adjustment = (ActualCost - TargetCost) * CustomerShareOfCostDiff
CustomerPayment = TargetCost + TargetProfit + Adjustment
```

As will be seen, *Adjustment* may be positive or negative.

Assume the agreement is that 60% share of any cost difference is to the customer, and 40% share is to the supplier. Then:

target cost	target profit	target customer payment	actual supplier cost	adjustment	actual customer payment	actual supplier profit
1,000,000	150,000	1,150,000	1,100,000	+60,000	1,210,000	110,000
1,000,000	150,000	1,150,000	900,000	-60,000	1,090,000	190,000

If costs are higher than estimated, both the supplier and customer share the pain: The supplier's profit is lower and the customer assumes some of the burden of the cost. If there is a cost savings, both parties share the gain: The supplier's profit is higher and the customer pays less than the original target payment.

An implication of this is that both parties may—no guarantee—proactively promote ways to reduce waste during the project.

Payment scheme variations

Contract drafters have birthed myriad variations, including

- capped (ceiling) customer payment
- reduced supplier rate if target cost exceeded

Adjustable target cost and target profit

Another essential element of target-cost contracts, supportive of agile and iterative values, is the ability to adjust the target cost (and profit) by ongoing negotiation between parties. Keys for success in adjusting target cost include

- high transparency and near-real-time, open-book project accounting by the supplier so that the customer sees the true state of expenses within the supplier
- a spirit of working together by both the customer and supplier to continuously improve
 - this is something Toyota works hard at [ISV09]
- early agreement between the parties, expressed in the contract, on the guidelines for target-cost adjustment
- a *moderate* adjustment cycle, to avoid being overly reactive or using excessive overhead on adjustment; for example, once per iteration may be too frequent
- acceptance test-driven development, to reduce ambiguity

These practices reduce but do not eliminate contention during ongoing re-negotiations, because adjustment issues are often inherently fuzzy—variations of “Is that a defect or a feature?”

One group reported²² that the following pre-agreed classification scheme for adjustments reduced contention:

Type	Description	May Adjust Target Cost?
fix	Changes to an implemented requirement, due to the supplier not doing what should have been ‘reasonably’ understood or done.	No
clarification	Changes to a ‘correctly’ implemented requirement, due to customer learning based on feedback.	No
enhancement	New feature.	Yes

We do not recommend this or any other scheme; it could help or lead to long and useless negotiation rather than cooperation.

Try...Multi-phase variable-model frameworks

Projects have a changing uncertainty or risk profile over time, ideally improving—for both parties. This can be reflected in *multi-phase* frameworks that reflect these shifting profiles for the customer and supplier. Any one phase can use any model: FPFS, progressive, target-cost, and so on.

One Valtech multi-phase variable-model example reflects the common Scrum pattern: (1) initial Product Backlog creation (2) adaptive iterative development. It was for a large B2B solution involving stakeholders in 23 countries:

1. *Phase 1—Fixed-price, fixed-duration, variable-scope.* Essentially, this was initial Product Backlog creation. The output of this phase was a Product Backlog—more precisely, a Release Backlog—and various business-analysis (market analysis, vision, ...) documents, based on workshops and team analysis. Although a specific list of documents to deliver was identified in the contract, the scope of analysis or content varied.
 - with a Release Backlog, a cost estimate was possible, so...

22. In [EMH05], a story about a target-cost contract for an agile project.

2. *Phase 2—Progressive contract, T&M per iteration, release cap, non-binding Release Backlog, fixed duration.* Phase two was essentially a classic Scrum project: nothing in the original backlog was binding, but it served to estimate and bound release project cost, provide an initial overview, and kickstart what to do in the first iteration.

Why bother with these multi-phase models instead of simple progressive contracts? Typically, the motivator is (1) lack of trust, (2) a regulatory constraint, (3) a belief by a party they can make or save more money or better reduce their risk, (4) a need to define the vision, high-level requirements, or cost of a later phase (and the effort of this work itself is large), or (5) ‘optimizing’ on a secondary goal (other than project success) such as better cost predictability.

Another example [Larman03]:

1. *Phase one—Fixed-price partial-fixed-scope for three iterations (fixed duration).* The “mandatory” fixed scope is low (for example, 20% of available effort), leaving plenty of room for variability and discovery when uncertainties are high. Risks to the customer and supplier are both bounded, and both parties learn a great deal about the nature of the continuing project for phase two.
2. *Phase two—FPFS, variable duration.* Risks to the supplier in taking on an FPFS phase are reduced because of increased knowledge and prior reduction of some sources of variability during phase one.

CONCLUSION

“How can we possibly do agile development when contracts are involved?” This is a question we have often been asked. But the key issues are not with the contract, they are with the contract writers and the clients they serve—reflecting the belief that success revolves more around contract negotiation and less around customer collaboration, or that *project success* is not the *goal* of contract work. That said, to reiterate a systems-thinking aphorism, “there is no blame,” implying that the behavior of people within a system is shaped by it—in this case by encouragement of departmental silos,

local targets (and rewards) leading to local optimizations, and the subtle message, “lawyers don’t need to learn about operational details or new approaches to R&D, that’s someone else’s job.”

Also, when this question is asked, ‘contract’ is often used as a synonym for “fixed-price fixed-scope” contract, which of course is not at all necessary—there are a wide variety of contract models, including variable-scope progressive contracts, and others.

Legal professionals have a duty to consider the ramifications of a breakdown of trust and collaboration—and other problems—when framing a contract. Just as contract lawyers need to learn more about lean and agile principles, other parties need to learn more about the necessary, valid concerns of lawyers. And just as product work improves with cross-functional *development* teams, further improvement is possible by including legal professionals in even broader cross-functional teams.

RECOMMENDED READINGS

There are several resources, most on the web, related to agile development and contracts; however, some of it is speculative rather than experience-based—keep that in mind when reading. Suggestions:

- Mary and Tom Poppendieck, thought leaders in lean software development, have organized several contract workshops over the years, and collected and share “lean and agile contract” papers and presentations at their website www.poppendieck.com. Following a theme similar to this chapter, the Poppendieck’s own material on agile contracts emphasizes the underlying issues of trust, collaboration, and transparency related to contracts.
- Some people new to the subject assume that contracts that encourage flexibility, collaboration, and alignment of interests (‘agile’ contracts) are a novel concept, but in fact much has been written and promoted in this area over the years, including within the USA government (for example, see *Administration of Government Contracts*). There are dozens, if not hundreds, of books and websites that discuss a variety of contract models.

- There are several ‘public’ contract models that we have reviewed, explicitly supporting iterative, evolutionary, or agile development. However, Valtech and ThoughtWorks—and other agile outsourcers that we know of—write their own contracts rather than use these models. We discourage “copy-paste” contract writing, but these are worth study for ideas.
 - The DSDM consortium (DSDM is an agile method) offers a sample contract, available to members at www.dsdm.org. Note that the contract is occasionally revised.
 - The Norwegian PS 2000 Contract was created for iterative and evolutionary development, by an alliance between industry and government, available at www.dataforening.no. To quote, “[*The*] Contract is designed to be used when it is particularly difficult or unserviceable to draw up a detailed specification prior to tendering, the idea being to leave open for the developer to find the best way to attain the objectives and needs of the customer.”

Miscellany

Chapter

- [Introduction to Feature Teams 549](#)
 - [Choose Component Teams or Feature Teams? 554](#)
 - [Transitioning to Feature Teams 555](#)
 - [Introduction to Requirement Areas 555](#)

Book

1	Introduction	1
2	Large-Scale Scrum	9
Action Tools		
3	Test	23
4	Product Management	99
5	Planning	155
6	Coordination	189
7	Requirements & PBIs	215
8	Design & Architecture	281
9	Legacy Code	333
10	Continuous Integration	351
11	Inspect & Adapt	373
12	Multisite	413
13	Offshore	445
14	Contracts	499

Miscellany

15	Feature Team Primer	549
	Recommended Readings	559
	Bibliography	565
	List of Experiments	580
	Index	589

FEATURE TEAM PRIMER

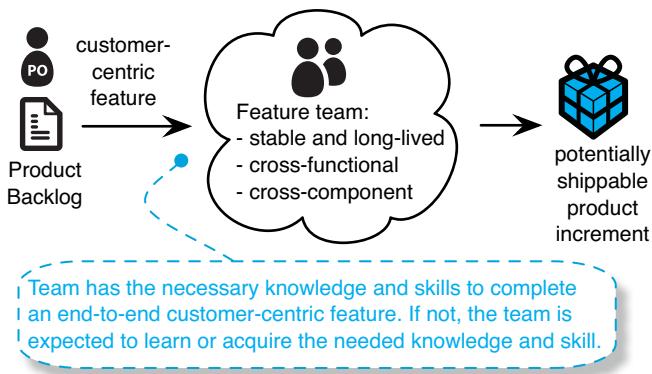
Speech is conveniently located midway between thought and action, where it often substitutes for both.
—John Andrew Holmes

Feature teams and **Requirement Areas** are key elements of scaling. They are analyzed in depth in the Feature Team and Requirement Area chapters of the companion book. This chapter summarizes a few key ideas.

INTRODUCTION TO FEATURE TEAMS

A **feature team**, shown in Figure 15.1, is a long-lived,¹ cross-functional, cross-component team that completes many end-to-end customer features—one by one.

Figure 15.1 feature team



1. Feature teams stay together for years, implementing many features.

The characteristics of a feature team are listed below::

Feature Team
<ul style="list-style-type: none"> ❑ long-lived—the team stays together so that they can ‘jell’ for higher performance; they take on new features over time ❑ cross-functional and cross-component ❑ ideally, co-located ❑ work on a complete customer-centric feature, across all components and disciplines (analysis, programming, testing, ...) ❑ composed of generalizing specialists ❑ in Scrum, typically 7 ± 2 people

Applying modern engineering practices—especially continuous integration—is essential when adopting feature teams. Continuous integration facilitates shared code ownership, which is a necessity when multiple teams work at the same time on the same components.

A common misunderstanding: every member of a feature team needs to know the whole system. Not so, because

- ❑ The team as a whole—not each individual member—requires the skills to implement the entire customer-centric feature. These include component knowledge and functional skills such as test, interaction design, or programming. But within the team, people still specialize... preferably in multiple areas.
- ❑ Features are not randomly distributed over the feature teams. The current knowledge and skills of a team are factored into the decision of which team works on which features.

Within a feature team organization, when specialization becomes a constraint...learning happens.

A feature team organization exploits speed benefits from specialization, as long as requirements map to the skills of the teams.

But when requirements do not map to the skills of the teams, learning is 'forced,' breaking the overspecialization constraint.

Feature teams balance specialization and flexibility.

Table 15.1 and Figure 15.2 show the differences between feature teams and more traditional component teams.

Table 15.1 feature teams vs. component teams

feature team	component team
optimized for delivering the maximum customer value ^a	optimized for delivering the maximum number of lines of code
focus on high-value features and system productivity (value throughput)	focus on increased individual productivity by implementing 'easy' lower-value features
responsible for complete customer-centric feature	responsible for only part of a customer-centric feature
'modern' way of organizing teams ^b — avoids Conway's law	traditional way of organizing teams — follows Conway's law ^c
leads to customer focus, visibility, and smaller organizations	leads to 'invented' work and a forever-growing organization
minimizes dependencies between teams to increase flexibility	dependencies between teams leads to additional planning ^d
focus on multiple specializations	focus on single specialization
shared product code ownership	individual/team code ownership
shared team responsibilities	clear individual responsibilities
supports iterative development	results in 'waterfall' development

feature team	component team
exploits flexibility; continuous and broad learning	exploits existing expertise; lower level of learning new skills
requires skilled engineering practices—effects are broadly visible	works with sloppy engineering practices—effects are localized
provides a motivation to make code easy to maintain and test	contrary to belief, often leads to low-quality code in component
<i>seemingly</i> difficult to implement	<i>seemingly</i> easy to implement

- a. The different optimization often makes the feature team *feel* slower—from the local view.
- b. Relatively ‘modern’ as feature teams have a long history in large-scale development, for example, Microsoft and Ericsson.
- c. [Conway68] *observed* this undesirable structure, he did not *recommended* it—in fact, quite the opposite.
- d. This additional planning is visible in more “release planning meetings” or “release trains” and more management overhead.

Table 15.2 summarizes the differences between feature teams and conventional *project or feature groups*.

Table 15.2 feature team vs. project group

feature team	feature group or project project
stable team that stays together for years and works on many features	temporary group of people created for one feature or project
shared team responsibility for all the work	individual responsibility for ‘their’ part based on specialization
self-managing team	controlled by a project manager
results in a simple single-line organization	results in a matrix organization with resource pools
team members are dedicated—100% allocated—to the team	members are part-time on many projects because of specialization

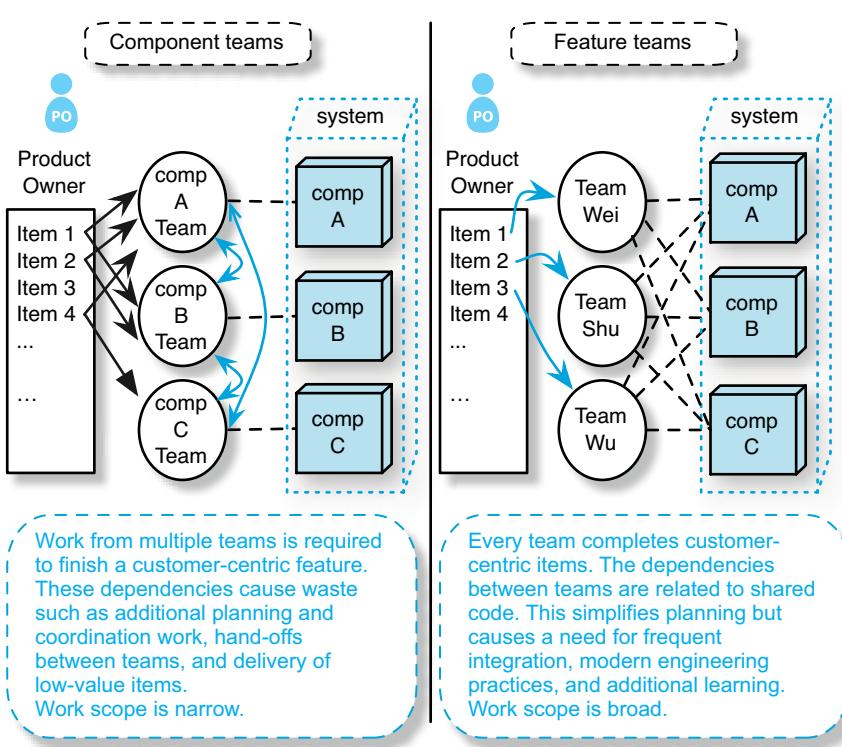
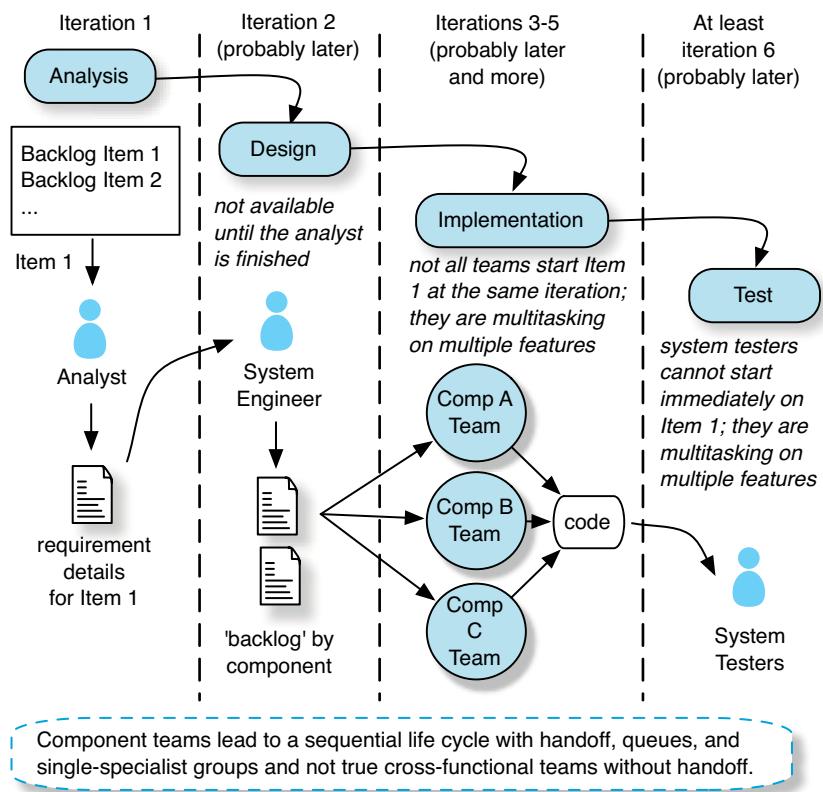


Figure 15.2 feature vs. component teams

Most drawbacks of component teams are explored in the companion book, Figure 15.3 summarizes some of these.

What is sometimes not seen is that a component team structure reinforces sequential development (a ‘waterfall’ or V-model), with many queues with varying-sized work packages, high levels of WIP, many handoffs, and increased multitasking and partial allocation.

Figure 15.3 some drawbacks of component teams



Choose Component Teams or Feature Teams?

A pure feature team organization is ideal from the value-delivery and organizational-flexibility perspective. Value and flexibility, however, are not the only criterion for organizational design, and many organizations therefore end up with a hybrid—especially during a transition from component to feature teams. Caution: hybrid models have the drawbacks from both worlds and can be...painful.

A frequently expressed reason in favor of a hybrid organization is the need to build infrastructure, construct reusable components, or clean up code—work traditionally done within component teams.

But these activities can also be done in a pure feature team organization—without establishing permanent component teams. How? By adding infrastructure, reusable components, or cleanup work to the Product Backlog and giving it to an existing feature team—as if it were a customer-centric feature. The feature team temporarily—for as long as the Product Owner wishes—does such work and then returns to building customer-centric features.

Transitioning to Feature Teams

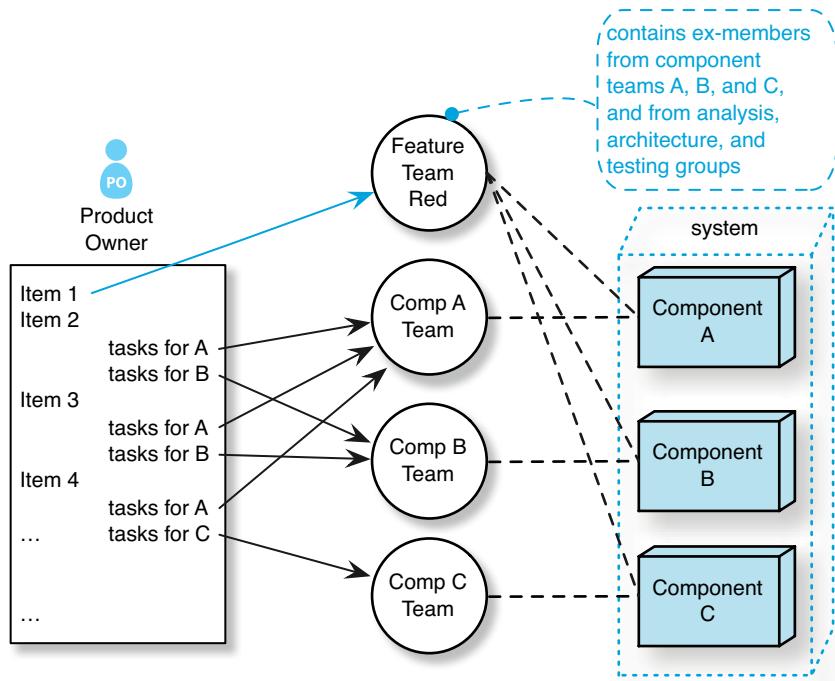
Different organizations require different transition strategies when changing from component to feature teams. We have experience with many strategies that worked...and failed in a different context. A safe—but slow—transitioning strategy is to establish one feature team within the existing component team organization. After this team performs well, a second feature team is formed. This continues gradually at the speed the organization is comfortable with. This is shown in Figure 15.4.

INTRODUCTION TO REQUIREMENT AREAS

Feature teams scale nicely, but when their number goes above ten teams—about a hundred people—additional structure is needed. Requirement areas provide this structure and complement the concepts behind feature teams. A **requirement area** is a categorization of the requirements leading to different views of the Product Backlog.

The Product Owner (PO) groups every Product Backlog item under exactly one requirement category—its requirements area. After this, he generates different views on the overall Product Backlog—called an **Area Backlog**. The Area Backlogs are prioritized by an **Area Product Owner** who specializes in part of the product—from a customer perspective. Each Requirement Area has several feature teams working from the Area Backlog, as shown in Figure 15.5.

Figure 15.4 gradual transitioning from feature to component teams



Requirement areas are scaled-up feature teams. Scaling up by structuring teams according to the product's architecture is called **development areas**. Table 15.3 summarizes the differences.

Table 15.3 requirement areas vs. development areas

Requirement Area	Development Area
organized around customer-centric requirements	organized around product's architecture
no subsystem code ownership	code ownership per subsystem
temporary in nature; should change over the lifetime of the product, but not at every iteration	tends to be more fixed over the lifetime of the product

Requirement Area	Development Area
focus on the customer, using customer language	focus on the architecture, using technology language

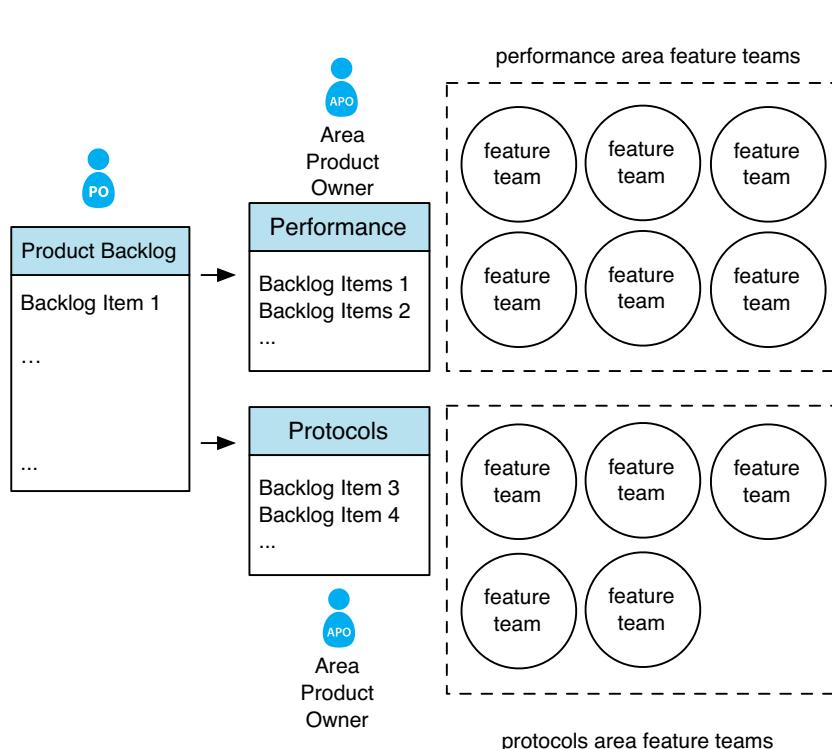


Figure 15.5 requirement areas

Finally, an Area Product Owner is different than a *supporting* Product Owner—someone that works with one or two teams to help a busy *overall* Product Owner. An Area Product Owner has different responsibilities and focus, and works with (probably) at least *four* teams, not just with one. This avoids local optimization toward the activities of one team.

CONCLUSION

Feature teams are stable teams that are given complete customer-centric features. These teams resolve local optimizations and extra coordination overhead caused by component team organizations. However, feature teams are not without challenges themselves.

Requirement areas scale the feature team concept by creating customer-centric views on the overall Product Backlog and thus creating a structure that allows feature teams to be scaled up to any size.

RECOMMENDED READINGS

- The *Feature Teams* chapter in the companion book.

RECOMMENDED READINGS

Large-Scale Scrum

- The companion book, *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, focuses on foundations supporting the practices in this book.

Test

- *Agile Testing*, by Lisa Crispin and Janet Gregory. A great overview of the role of testing in agile development. It covers the challenges organizations face when adopting agile development and also describes the concrete role of testing during the iteration.
- *Lessons Learned in Software Testing*, by Cem Kaner, James Bach, and Bret Pettichord. This book describes the lessons learned from decades of experience in testing and also introduces the context-driven school of thinking in software testing.
- *Agile Testing Directions*, by Brian Marick. A series of blog posts wherein Brian Marick introduces the agile testing quadrants.
- *A Practitioner's Guide To Software Test Design*, by Lee Copeland. An easy-to-read catalog of test design techniques.
- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen. A thorough coverage of different test design techniques. Starts off with mathematics for testing.
- *Bridging the Communication Gap*, by Gojko Adzic. At this moment, Gojko's book is the only book purely on the subject of A-TDD (which he calls agile acceptance testing). It has a strong focus on requirements clarifications and workshops.
- *Acceptance Test Driven Development: An Overview*, by Elisabeth Hendrickson. A blog post and related paper providing an overview of A-TDD by giving a detailed example of using Robot Framework.
- *Fit for Developing Software*, by Rick Mugridge and Ward Cunningham. This book has a strong focus on improving the communication of requirements by means of Fit tables.
- *Robot Framework User Guide*. Does not cover A-TDD but does provide an excellent introduction to the Robot Framework tool.
- *Test-Driven .NET Development with FitNesse*, by Gojko Adzic. This book has less emphasis on A-TDD and more on FitNesse. But it does a good job in describing the tool.
- *Exploratory Testing Explained*, by James Bach. An article available on the web; it is the classic reference related to this subject. Definitely worth reading.
- *Exploratory Testing in an Agile Context*, by Elisabeth Hendrickson. A freely available mini-book related to the role of exploratory testing in agile development. Easy to read.
- *Test-Driven*, by Lasse Koskela. A well-written thorough book on the subject. It uses Java and also covers A-TDD.
- *Test-Driven Development*, by Kent Beck. A classic and one of the first books on the subject. It uses Java.
- *Test-Driven Development in Microsoft .NET*, by James Newkirk and Alexei Vorontsov. A good introduction to TDD in .NET.
- *xUnit Test Patterns*, by Gerard Meszaros. More than you ever wanted to know about xUnit.

- *Test-Driven Development in C: Modern C Programming for Embedded, Mobile, Open Source and You*, by James Grenning. Does this work for embedded software? Yes. James discusses how to use TDD when developing embedded software. Not yet published.
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce reinforces the value of evolving design based on feedback from tests.

Product Management

- The *Product Development and Management Association* (www.pdma.org) offers online and printed learning resources (such as *The PDMA Handbook of New Product Development*), and an online list of classic P-M literature. Some of the material assumes the traditional Contract Game or sequential life cycle development, but much is still worth investigation.
- *Innovation Games* by Luke Hohmann emphasizes simple, creative, and collaborative techniques—applicable in workshops—for customer-focused product definition.
- *Agile Product Management with Scrum* by Roman Pichler explores envisioning a product in the context of Scrum development, the role of the Product Owner, and more.
- The text *Product Strategy and Management* is written by researchers with long in-depth study into product management and development. It contains many solid suggestions, and a vast number of references to the major papers and researchers in this field. This book is an excellent window into the breadth and depth of P-M-related research.
- *Product Management* by Lehmann and Winer is a solid introduction to market analysis, product strategy, pricing, distribution channels, and more.

Planning

- For envisioning and vision workshops, two books already recommended in the *Product Management* chapter are relevant: *Innovation Games* and *Agile Product Management with Scrum*.
- For planning with small or large groups, *Agile Estimating and Planning* by Mike Cohn is an excellent, practical resource.
- *Waltzing with Bears* by DeMarco and Lister is informative and entertaining; it emphasizes iterative—rather than sequential—development as a key risk-management practice, and explains how to apply Monte Carlo simulation in estimation.

Coordination

- “Bridging the Boundary: External Activity and Performance in Organizational Teams,” by Deborah Ancona and David Caldwell. One of the early research articles that explored teams within their context and how external activity—boundary spanning—relates to team performance.
- *Leading Teams*, by Richard Hackman. Still one of the best references on teams and self-managing teams. It also covers how teams manage their boundaries.
- *Succeeding with Agile*, by Mike Cohn. Scrum coach Mike Cohn covers some team coordination topics in his book.

Requirements & PBIs

- *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing* by Gojko Adzic was also recommended in the *Test* chapter; it emphasizes acceptance TDD, requirements by examples, and includes a chapter on agile requirements workshops.
- *Requirements by Collaboration* by Ellen Gottesdiener describes how to organize and facilitate requirement workshops.
- *Writing Effective Use Cases* by Alistair Cockburn is an excellent book on use cases, and widely considered the de facto standard for this subject.
- *Patterns for Effective Use Cases* also includes useful tips.
- A *Use Cases* chapter, based on the Cockburn system, is available in the *Articles* section of www.craiglarman.com.
- *User Stories Applied*, by Mike Cohn, is a great introduction.
- In the classic *The Design of Everyday Things* and his more recent *Emotional Design*, Donald Norman emphasizes that human factors need to be front and center.
- Interaction design is a fast-moving field of publication. Broadly, search for material that emphasizes lightweight modeling, iteration, prototyping, and cross-functional teams. For example, see *Sketching User Experiences* by Bill Buxton.
- *Agile Modeling* by Scott Ambler emphasizes lightweight and collaborative approaches to modeling.
- *Applying UML and Patterns* demonstrates a variety of modeling techniques, including domain models, activity diagrams, and state-machine models.

Design & Architecture

- The site www.codingthearchitecture.com emphasizes the need for architects to be master hands-on active developers.
- Many of our clients have vast quantities of messy legacy code that is difficult to test in isolation and difficult to evolve. Michael Feather's *Working Effectively with Legacy Code* is an important antidote, covering the techniques that allow developers to start designing a more agile architecture within their existing code base.
- A key element of technical agility is design patterns. Consider these texts: *Design Patterns*, *Pattern-Oriented Software Architecture* (five volumes), *Applying UML and Patterns*, and *Pattern Languages of Program Design* (five volumes).
- Two books by Bob Martin encourage a more agile architecture: *Agile Development, Principles, Patterns and Practices* and *Clean Code: A Handbook of Agile Craftsmanship*.
- Two more useful quality-code-oriented books include *Code Complete* by Steve McConnell and *Implementation Patterns* by Kent Beck.
- *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce reinforces a culture of growing rather than specifying "the architecture."
- *Domain-Driven Design* by Eric Evans encourages thoughtful iterative design, shared understanding, and a domain model that must be well-expressed in the code.

- The paper *Agile Product Development* [TR98] explores the business value of product development and design agility, and how development flexibility can be quantified.

Legacy Code

- *Working Effectively with Legacy Code*, by Michael Feathers. Concrete advice on how to gradually improve your legacy system at code level.
- *Refactoring: Improving the Design of Existing Code*, by Martin Fowler. The classic work on improving existing code.
- *Refactoring Workbook*, by Bill Wake. A concrete guide for becoming better at refactoring code.
- *Refactoring to Patterns*, by Joshua Kerievsky. In this book, Joshua explains how to gradually refactor your code to standard, robust design patterns.
- *Refactoring in Large Software Projects*, by Stefan Roock and Martin Lippert. Large systems might need large refactorings. This book explains how to do these in as small steps as possible so that your systems stays stable.
- *Enterprise Scrum*, by Ken Schwaber. Chapter 9 of *Enterprise Scrum* is one of the few descriptions explaining the relationship between customer promises and the creation of legacy code.
- *Sustainable Software Development: An Agile Perspective*, by Kevin Tate. This book does not cover many new techniques but provides an excellent overview of the practices for creating software in a sustainable way.
- *The Pragmatic Programmer: From Journeyman to Master*, by Andrew Hunt and Dave Thomas. Classic book on modern software craftsmanship.
- *Software Craftsmanship*, by Pete McBreen dives in craftsmanship approach and compares it to the traditional software engineering perspective.
- *Agile Development, Principles, Patterns and Practices*, by Bob Martin. Also known as *Agile PPP*, links good code, modern practices, and eternal design principles to explain what it means to be a craftsman.
- *Clean Code: A Handbook of Agile Craftsmanship*, by Bob Martin. The subtitle says it all. *Clean Code* is the code-focused prequel to *Agile PPP*.

Continuous Integration

- *Extreme Programming Explained*, by Kent Beck. The term CI was first coined in the Extreme Programming method.
- *Continuous Integration*, by Martin Fowler. Probably the best CI description available.
- *Managing Projects with GNU Make*, by Robert Mecklenburg. When working with C/C++, you will probably use Make. This book provides a great overview of Make and also talks about Make in large-scale development.
- *Ant in Action*, by Steve Loughran and Erik Hatcher. When working with Java, you will probably use Ant. This book's focus is Ant, but it covers other topics. *Maven*—another popular build automation tool—is also covered.

- *Groovy in Action*, by Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, Jon Skreett. Groovy is a recent JVM-based dynamic programming language. It has some excellent build automation support.
- *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Apps*, by Mike Clark. A small book that covers lots of technology related to automating Java builds.
- *Continuous Integration: Improving Software Quality and Reducing Risk*, by Paul Duvall, Steve Matyas, and Andrew Glover. The focus of this book is on the automation of builds more than on the practice of CI.
- “*Scaling Continuous Integration*,” by Owen Rogers in *Extreme Programming and Agile Processes in Software Engineering 2004 Conference Proceedings*. Although a little dated, this is among the best of the available material (other than this chapter) related to scaling CI.

Inspect & Adapt

- *The Birth of Lean*, edited by Shimokawa and Fujimoto, offers a glimpse into the evolution and adoption of lean production and thinking at Toyota. For example: “*At a time when all of us are struggling to implement lean production and lean management, often with complex programs on an organization-wide basis, it is helpful to learn that the creators of lean had no grand plan and no company-wide program to install it.*”
- *Fearless Change: Patterns for Introducing New Ideas* by Mary Lynn Manns and Linda Rising comes from authors with experience in change initiatives and knowledge of agile development; they emphasize a bottom-up approach to change.
- The site www.solononline.org, from the *Society for Organizational Learning*, contains many learning resources and recommended readings related to organizational improvement.
- Taiichi Ohno, in his *Workplace Management*, conveys a sense of the importance—for creating a lean culture—of leaders who truly grasp lean thinking, and relentlessly coach others in this.
- There are several good (and more bad) books on team building; some are of the better ones are recommended in the *Teams* chapter of the companion book. Two mentioned in this chapter include *The Five Dysfunctions of a Team* and *Overcoming the Five Dysfunctions of a Team* by Patrick Lencioni.
- *Teamwork Is an Individual Skill: Getting Your Work Done When Sharing Responsibility* by Chris Avery emphasizes taking personal responsibility for creating an effective team, and shares tips for how to do so.
- *The Fifth Discipline: The Art & Practice of The Learning Organization* by Peter Senge, is a classic in systems thinking, learning, and the qualities needed by effective leaders for sustainable, high-impact organizational improvement.
- *Agile Retrospectives: Making Good Teams Great* by Esther Derby and Diana Larsen covers core retrospective skills. And *Project Retrospectives* by Norm Kerth explores how to do retrospectives with larger groups.
- *Agile Coaching* by Rachel Davies and Liz Sedley captures many practical tips for ScrumMasters and other agile coaches, from two experienced coaches.

Multisite

- Erran Carmel's books, *Global Software Teams* and *Offshoring Information Technology*, are two of the better high-level books that explore multisite development.
- Jutta Eckstein's *Agile Software Development with Distributed Teams* is written by a consultant and coach with hands-on experience in both agile and multisite development.
- Keith Braithwaite and Tim Joyce summarize key principles and practices in their paper *XP Expanded: Distributed Extreme Programming*. Although written in the context of Extreme Programming, it applies to all agile development approaches.

Offshore

- All the recommendations in the *Multisite* chapter are relevant, such as *Offshoring Information Technology* and *Agile Software Development with Distributed Teams*.
- Two of the largest agile-offshore outsourcers in India are Valtech and ThoughtWorks. In Martin Fowler's online article *Using an Agile Software Process with Offshore Development* (at martinfowler.com) he describes lessons learned at ThoughtWorks, which parallel those at Valtech.

Contracts

- Mary and Tom Poppendieck, thought leaders in lean software development, have organized several contract workshops over the years, and collected and share “lean and agile contract” papers and presentations at their website www.poppendieck.com. Following a theme similar to this chapter, the Poppendieck’s own material on agile contracts emphasizes the underlying issues of trust, collaboration, and transparency related to contracts.
- Some people new to the subject assume that contracts that encourage flexibility, collaboration, and alignment of interests ('agile' contracts) are a novel concept, but in fact much has been written and promoted in this area over the years, including within the USA government (for example, see *Administration of Government Contracts*). There are dozens, if not hundreds, of books and websites that discuss a variety of contract models.
- There are several ‘public’ contract models that we have reviewed, explicitly supporting iterative, evolutionary, or agile development. However, Valtech and ThoughtWorks—and other agile outsourcers that we know of—write their own contracts rather than use these models. We discourage “copy-paste” contract writing, but these are worth study for ideas.

Feature Team Primer

- The *Feature Teams* chapter in the companion book.

BIBLIOGRAPHY

- ABCP02 Adolph, S., Bramble, P., Cockburn, A., Pols, A., 2002. *Patterns for Effective Use Cases*, Addison-Wesley
- AC92 Ancona, D., Caldwell, D., 1992. "Bridging the Boundary: External Activity and Performance in Organizational Teams," *Administrative Science Quarterly*, Dec 1992, Vol. 37, No. 4
- Accenture08 Accenture, 2008. "Accenture Delivery Centers In India," *Accenture website*, at <https://www.accenture.com/NR/rdonlyres/9BFC7780-F3C2-47C0-8733-DB51E9DA0B3E/0/IndiaLowresSept2007.pdf> (accessed on July 10, 2008)
- AD72 Adler, M. J., Van Doren, C.L. 1972. *How to Read a Book*, Simon & Schuster, Inc.
- Adzic08 Adzic, G., 2008. *Test Driven.Net Development with FitNesse*, Neuri Limited
- Adzic09 Adzic, G., 2009. *Bridging the Communications Gap: Specification by Example and Agile Acceptance Testing*, Neuri Limited
- AKB04 Appleton, B., Konieczka, S., Berczuk, S., 2004. "Continuous Staging: Scaling Continuous Integration to Multiple Component Teams," *CM Crossroads*, at <http://www.cmcrossroads.com/articles/agilemar04.pdf>
- AKSMW09 Ancona, D., Kochan, T., Scully, M., van Maanen, S., Westney, D., 2009. *Managing for the Future*, South-Western
- Ambler02 Ambler, S., 2002. *Agile Modeling*, John-Wiley
- Armstrong07 Armstrong, J., 2007. "A History of Erlang," *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*
- AS95 Argyris, C., Schon, D., 1995. *Organizational Learning II: Theory, Method, and Practice*, Prentice Hall
- Atwood08 Atwood, J., 2008. "Does More Than One Monitor Improve Productivity?" at <http://www.codinghorror.com/blog/archives/001076.html>
- Austin96 Austin, R., 1996. *Measuring and Managing Performance in Organizations*, Dorset House
- AV07 APLN, Version One, 2007. "2nd Annual Survey. The State of Agile," *VersionOne website* at http://www.versionone.com/pdf/stateofagiledevelopment2_fulldatareport.pdf

- AWT01 Avery, C., Walker, M. A., O'Toole E., 2001. *Teamwork Is an Individual Skill: Getting Your Work Done When Sharing Responsibility*, Berrett-Koehler Publishers
- BA03 Berczuk, S., Appleton, B., 2003. *Configuration Management Patterns*, Addison-Wesley
- Bach94 Bach, J., 1994. "The Immaturity of CMM," *American Programmer*, Sept '94
- Bach00 Bach, J., 2000. "Session-Based Test Management," *Software Testing and Quality Engineering Magazine*, Nov 2000, also available at <http://www.satisfice.com/articles/sbtm.pdf>
- Bach03 Bach, J., 2003. "Exploratory Testing Explained," at www.satisfice.com/articles/et-article.pdf
- BB09 Bach, J., Bolton, M., *Rapid Software Testing*, training material at <http://www.satisfice.com/rst.pdf>
- BC99 Beck, K., Cleal, D., 1999. "Optional Scope Contracts," at www.jarn.com/about/OptionalScopeContracts.pdf
- BCK98 Bass, L., Clements, P., Kazman, R., 1998. *Software Architecture in Practice*, Addison-Wesley
- BDSSS99 Beedle, M., Devos, M., Sharon, Y., Schwaber, K., Sutherland., J., 1999. "Scrum: A Pattern Language for Hyperproductive Software Development," *Proceedings of Pattern Languages of Programs '98*, also in [HFR00]
- Beck99 Beck, K., 1999. *Extreme Programming Explained: Embrace Change (1st edition)*, Addison-Wesley
- Beck01 Beck, K., 2001. "Aim, Fire," *IEEE Software*, Vol. 18, Issue 5, Sept/Oct 2001
- Beck03 Beck, K., 2003. *Test-Driven Development: By Example*, Addison-Wesley
- Beck04 Beck, K., 2004. *Extreme Programming Explained: Embrace Change (2nd edition)*, Addison-Wesley
- Beck08 Beck, K., 2008. *Implementation Patterns*, Addison-Wesley
- Beck09 Beck, K., 2009. "Approaching a Minimum Viable Product," *Thoughts on Programming Blog*, at <http://www.threeriversinstitute.org/blog/?p=333>
- BH07 Baker, M., Hart, S., 2007. *Product Strategy and Management*, Prentice Hall
- BHS97 Brink, C., Kahl, W., Schmidt, G., 1997. *Relational Methods in Computer Science*, Springer
- BHS07a Buschmann, F., Henney, K., Schmidt, D., 2007. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Addison-Wesley
- BHS07b Buschmann, F., Henney, K., Schmidt, D., 2007. *Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, Addison-Wesley
- BI05 Brown, J., Isaacs, D., 2005. *The World Café*, Berrett-Koehler Publishing

- BJ05 Braithwaite, K., Joyce, T., 2005. "XP Expanded: Distributed Extreme Programming," *Proceedings of the XP2005*, Springer
- BJP02 Buwalda, H., Janssen, D., Pinkster, I., 2002. *Integrated Test Design and Automation*, Addison-Wesley
- BMRSS96 Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. *Pattern-Oriented Software Architecture*, Addison-Wesley
- Boehm00a Boehm, B., Abts, C., Brown, W., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., Steece, B., 2000. *Software Cost Estimation with Cocomo II*, Prentice Hall
- Boehm00b Boehm, B., 2000. "Spiral Development: Experience, Principles, and Refinements" *Special Report CMU/SEI-2000-SR-008*
- Boehm06 Boehm, B., 2006. "Keynote: Product and Process Architectures for Integrating Agile and Plan-Driven Methods," *XP2006 Conference Keynote*
- Booch96 Booch, G., 1996. *Object Solutions. Managing the Object-Oriented Project*, Addison-Wesley
- Brooks87 Brooks, F., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer* Vol. 20, Issue 4, Apr 1987
- Burnstein02 Burnstein, I., 2002. *Practical Software Testing*, Springer
- Buxton07 Buxton, B., 2007. *Sketching User Experiences: Getting the Design Right and the Right Design*, Morgan Kaufman
- Buzan96 Buzan, T., Buzan, B., 1996. *The Mind Map Book: How to Use Radiant Thinking to Maximize Your Brain's Untapped Potential*, Plume
- Cagan08 Cagan, M., 2008. *Inspired. How to Create Products Customers Love*, SVPG Press
- Carmel99 Carmel, E., 1999. *Global Software Teams*, Prentice Hall
- CG09 Crispin, L., Gregory, J., 2009. *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley
- CH01 Cockburn, A., Highsmith, J., 2001. "Agile Software Development: The People Factor," *IEEE Computer*, Vol. 34, No. 11
- CH05 Coplien, J., Harrison, N., 2005. *Organizational Patterns of Agile Software Development*, Pearson Prentice Hall
- CKS07 Chrissis, M.B., Konrad, M., Shrum, S., 2007. *CMMI, 2nd edition*, Addison-Wesley
- Clark04 Clark, M., 2004. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps*, The Pragmatic Programmers
- CNN06 Cibinic, J., Nash, R., Nagle, J., 2006. *Administration of Government Contracts, 4th edition*, CCH Inc
- Cockburn97 Cockburn, A., 1997. "Parts: Precision, Accuracy, Relevance, Tolerance, Scale in Object Design," at <http://alistair.cockburn.us/Parts:+precision,+accuracy,+relevance,+tolerance,+scale+in+object+design>

- Cockburn99 Cockburn, A., 1999. "Characterizing People as Non-Linear, First-Order Components in Software Development," at <http://alistair.cockburn.us/Characterizing+people+as+non-linear,+first-order+components+in+software+development>
- Cockburn01 Cockburn, A., 2001. *Agile Software Development*, Addison-Wesley
- Cockburn02 Cockburn, A., 2002. *Writing Effective Use Cases*, Addison-Wesley
- Cockburn04 Cockburn, A., 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams*, Addison-Wesley
- Cockburn07 Cockburn, A., 2007. *Agile Software Development: The Cooperative Game*, Addison-Wesley
- Cohn04 Cohn, M., 2004. *User Stories Applied*, Addison-Wesley
- Cohn05 Cohn, M., 2005. *Agile Estimating and Planning*, Addison-Wesley
- Cohn07 Cohn, M., 2007. "Advice on Conducting the Scrum of Scrums Meeting," *ScrumAlliance Articles* at <http://www.scrumalliance.org/articles/46-advice-on-conducting-the-scrum-of-scrums-meeting>
- Cohn09 Cohn, M., 2009. *Succeeding with Agile: Software Development using Scrum*, Addison-Wesley
- Cone08 Cone, L., 2008. "CMM Level 5 — So What!" *A Day in the Life of a Project Manager Blog*, at <http://it.toolbox.com/blogs/coneblog/cmm-level-5-so-what-5534>
- Conway68 Conway, M., 1968. "How Do Committees Invent?" Datamation Magazine, Apr 1968
- Copeland08 Copeland, L., 2008. *A Practitioner's Guide to Software Test Design*, Artech House Publishers
- Cottmeyer09 Cottmeyer, M., 2009. "Product Owner by Proxy," *Leading Agile blog*, at <http://www.leadingagile.com/2009/03/product-owner-by-proxy.html>
- Crosby80 Crosby, P., 1980. *Quality is Free*, Mentor
- CS95 Coplien, J., Schmidt, D., 1995. *Pattern Languages of Program Design*, Addison-Wesley
- CSSD05 Ceschi, M., Sillitti, A., Succi, G., De Panfilis, S., 2005. "Project Management in Plan-Based and Agile Companies," *IEEE Software*, May/June 2005
- CT05 Carmel, E., Tija, P., 2005. *Offshoring Information Technology*, Cambridge
- CY00 Cusumano, M., Yoffie, B., 2000. *Competing on Internet Time: Lessons from Netscape and Its Battle with Microsoft*, Free Press
- Cybernews09 Cybernews, 2009. "Scrumming RealXtend," News article at <http://www.cybertechnews.org/?p=338>
- DBT05 Dalcher, D., Benediktsson, O., Thorbergsson, H., 2005. "Development Life Cycle Management: A Multiproject Experiment." *Proceedings of the 12th International Conference on Engineering of Computer-Based Systems*, IEEE Computer Society

- DeMarco95 DeMarco, T., 1995, "Conversation with Tom DeMarco," *Computerworld*, Dec. 4. 1995
- Deming82 Deming E. W., 1982. *Out of the Crisis*, MIT Press
- DL03 DeMarco, T., Lister, T., 2003. *Waltzing with Bears: Managing Risk on Software Projects*, Dorset House
- DL06 Derby, E., Larsen, D., 2006. *Agile Retrospectives: Making Good Teams Great*, Pragmatic Bookshelf
- DM89 de Meyer, A., Mizushima A., 1989. "Global R&D Management," *R&D Management*, Vol. 19, Issue 2. 1989
- DMG07 Duvall, P., Matyas, S., Glover, A., 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley
- DS09 Davies R., Sedley, L., 2009. *Agile Coaching*, Pragmatic Bookshelf
- Eckstein04 Eckstein, J., 2004. *Agile Software Development in the Large*, Dorset House
- Eckstein10 Eckstein, J., 2010. *Agile Software Development with Distributed Teams: Staying Agile in a Global World*, Draft. In the companion book this book was referred to as *Agile in the Face of Global Software Development*.
- Edmondson99 Edmondson, A., 1999. "A Safe Harbor: Social Psychological Conditions Enabling Boundary Spanning in Work Teams," in [Wageman99]
- EE06 Elshamy, A., Elssamadisy, A., 2006. "Divide After You Conquer: An Agile Software Development Practice for Large Projects," *Proceedings of the XP2006*, Springer
- EMH05 Eckfeldt, B., Madden, R., Horowitz, J., 2005. "Selling Agile: Target-Cost Contracts." *Proceedings of Agile 2005 Conference*
- Evans04 Evans, E., 2004. *Domain-Driven Design*, Addison-Wesley
- Feathers04 Feathers, M., 2005. *Working Effectively with Legacy Code*, Addison-Wesley
- Feathers05 Feathers, M., 2005. "A Set of Unit Testing Rules, at <http://www.artima.com/weblogs/viewpost.jsp?thread=126923>
- Festa00 Festa, P., 2000. "Netscape 6 Ships after 32-Month Gestation," *CNET News.com* at <http://news.cnet.com/2100-1023-248549.html>
- FG99 Fewster, M., Graham, D., 1999. *Software Test Automation*, Addison-Wesley
- FK06 Fleming, Q., Koppelman, J., 2006. *Earned Value Project Management, 3rd edition*, Project Management Institute
- FL97 Fleischer, M., Liker, J., 1997. *Concurrent Engineering Effectiveness*, Hanser Gardner Publications.
- Fowler99 Fowler, M., 1999. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley
- Fowler04 Fowler, M., 2004. "Inversion of Control Containers and the Dependency Injection Pattern," at <http://martinfowler.com/articles/injection.html>

- Fowler06a Fowler, M., 2006. "Continuous Integration," at <http://www.martinfowler.com/articles/continuousIntegration.html>
- Fowler06b Fowler, M., 2006. "Using an Agile Software Process with Offshore Development," at <http://martinfowler.com/articles/agileOffshore.html>
- Fowler07 Fowler, M., 2007. "Mocks Aren't Stubs," at <http://martinfowler.com/articles/mocksArentStubs.html>
- Fowler09 Fowler, M., 2009. "Feature Branch," at <http://martinfowler.com/bliki/FeatureBranch.html>
- FP09 Freeman, S., Pryce, N., 2009. *Growing Object-Oriented Software, Guided by Tests*, Addison-Wesley
- GH88 Gelperin, D., Hetzel, B., 1988. "The Growth of Software Testing," *Communications of the ACM*, Vol. 31, No. 6, June 1988
- GHJV94 Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley
- Gilb88 Gilb, T., 1988. *Principles of Software Engineering Management*, Addison-Wesley
- Gilb05 Gilb, T., 2005. *Competitive Engineering*, Butterworth-Heinemann
- Gorchels06 Gorchels, L., 2006. *The Product Manager's Handbook*, McGraw Hill
- Gottesdiener02 Gottesdiener, E., 2002. *Requirements by Collaboration: Workshops for Defining Needs*, Addison-Wesley
- Grady92 Grady, R., 1992. *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall
- Grenning02 Grenning, J., 2002. "Planning Poker or How to Avoid Analysis Paralysis while Release Planning," at www.objectmentor.com/resources/articles/PlanningPoker.zip
- Grenning10 Grenning, J. 2010. *Test-Driven Development in C: Modern C Programming for Embedded, Mobile, Open Source and You* (forthcoming). The Pragmatic Programmers
- GW89 Gause, D., Weinberg, G., 1989. *Exploring Requirements: Quality Before Design*. Dorset House
- Hackman99 Hackman, R., 1999. "Thinking Differently about Context," in [Wageman99]
- Hackman02 Hackman, R., 2002. *Leading Teams*, Harvard Business School Press
- Hall76 Hall, E., 1976. *Beyond Culture*, Anchor Books
- Hayashi08 Hayashi, N., 2008. "Top Engineer Explains How Toyota Develops People," *Nikkei Business Online*, Translated at http://www.gembapantarei.com/2008/08/toyotas_top_engineer_on_how_to_develop_thinking_pe.html
- Hendrickson08 Hendrickson, E., 2008. "Driving Development with Tests: ATDD and TDD," at <http://testobsessed.com/wordpress/wp-content/uploads/2008/12/atddexample.pdf>

- Hendrickson09 Hendrickson, E., 2009. *Exploratory Testing in an Agile Context*, at <http://www.qualitytree.com/ebooks/et.pdf>
- Herzberg87 Herzberg, F., 1987. "One More Time: How Do You Motivate Employees?" *Harvard Business Review*, Sept/Oct 1987
- Hetzelt73 Hetzel, B., 1973. *Program Test Methods*, Prentice Hall
- Hetzelt88 Hetzel, B., 1988. *The Complete Guide to Software Testing*, Wiley-QED
- HFR00 Harrison, N., Foote, B., Rohnert, H., 2000. *Pattern Languages of Program Design 4*, Addison-Wesley
- HH05 Hofstede, G., Hofstede, G.J., 2005. *Cultures and Organizations: Software of the Mind*, McGraw-Hill
- HM03 Herbsleb, J., Mockus, A., 2003. "An Empirical Study of Speed and Communication in Globally Distributed Software Development," *IEEE Transactions of Software Engineering*, Vol. 29, No. 6, June 2003
- Hock99 Hock, D., 1999. *The Birth of the Chaordic Age*, Berrett-Koehler Publishers
- Hohmann03 Hohmann, L., 2003. *Beyond Software Architecture: Creating and Sustaining Winning Solutions*, Addison-Wesley
- Hohmann06 Hohmann, L., 2006. *Innovation Games*, Addison-Wesley
- Hohmann08 Hohmann, L., 2008. "Why Prioritizing Your Product Backlog for ROI Doesn't Work," *Insights-Tools*, at <http://www.enthiosys.com/insights-tools/prioritizeforprofit1of3/>
- Horowitz74 Horowitz, E. 1974. *Practical Strategies for Developing Large Software Systems*, Addison-Wesley
- HP08 HP, 2008. *HP website*, at <http://h20219.www2.hp.com/services/cache/602196-0-0-225-121.html> (accessed July 10, 2008)
- HT99 Hunt, A., Thomas, D., 1999. *The Pragmatic Programmer*, Addison-Wesley
- HW01 Hoffman, D., Weiss, D., editors, 2001. *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley
- IBM08 IBM, 2008. *IBM website*, at <http://www-935.ibm.com/services/us/index.wss/casestudy/imc/a1025831?cntxt=a1000056> (accessed July 10, 2008)
- ISQTB07 ISQTB, 2007. *ISQTB Syllabus Foundation Level v2007*, at <http://www.istqb.org/download.htm>
- ISV09 Iyer, A., Seshadri, S., Vasher, R., 2009. *Toyota's Supply Chain Management: A Strategic Approach to Toyota's Renowned System*, McGraw-Hill
- IXP04 Industrial Logic, 2004. *Industrial Extreme Programming*, at <http://www.industrialxp.com>
- James07 James, M., 2007. "A ScrumMaster's Checklist," at <http://www.scrummasterchecklist.org>

- Jeffries01 Jeffries, R., 2001. "Essential XP: Card, Convention, Confirmation," *XProgramming.doc. An Agile Software Development Resource*, at <http://xprogramming.com/xpmag/expcardconversationconfirmation/>
- Jeffries02 Jeffries, R., 2002. "Foreword by Ron Jeffries," in *Agile Modeling* [Ambler02]
- Jeffries04 Jeffries, R., 2004. *Extreme Programming Adventures in C#*, Microsoft Press
- Jeffries09 Jeffries, R., 2009. "Re: Version One," *ScrumDevelopment mailing list*, at <http://groups.yahoo.com/group/scrumdevelopment/message/39103>
- Jones08 Jones, C., 2008. *Applied Software Measurement*, McGraw-Hill
- Jorgensen08 Jorgensen, P., 2008. *Software Testing: A Craftsman's Approach, 3rd edition*, Auerbach Publications
- JPZ96 Janicki, R., Parnas, D., Zucker, J., 1996. "Tabular Representation in Relational Documents," published in [BHS97]. Reprint in [HW01].
- Kahn04 Kahn, K., 2004. *The PDMA Handbook of New Product Development, 2nd edition*, John Wiley
- Kanigel05 Kanigel, R., 2005. *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*, MIT Press
- Kao08 Kao, C., 2008. *Pushmi*, at <http://search.cpan.org/~clkao/Pushmi-v0.994.0/lib/Pushmi.pm>
- Kato06 Kato, I., 2006. *Summary Notes from Art Smalley Interview with Mr. Isao Kato*, at http://artoflean.com/documents/pdfs/Mr_Kato_Interview_on_TWI_and_TPS.pdf
- KBP02 Kaner, C., Bach, J., Bettichord, B., 2002. *Lessons Learned in Software Testing*, Wiley
- Kerievsky05 Kerievsky, J., 2005. *Refactoring to Patterns*, Addison-Wesley
- Kerth01 Kerth, N., 2001. *Project Retrospectives: A Handbook for Team Reviews*, Dorset House
- KGKLS07 Koenig, D., Glover, A., King, P., Laforge, G., Skeet, J., 2007. *Groovy in Action*, Manning
- KJ04 Kircher, M., Jain, P., 2004. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, Addison-Wesley
- Koch05 Koch, A., 2005, *Agile Software Development. Evaluating the Methods for your Organization*, Artech House
- Koch04 Koch, C., 2004. "Software Quality: Bursting the CMM Hype," *CIO.com News Article*, March 2004
- Kohn93 Kohn, A., 1993. *Punished by Rewards*, Houghton Mifflin
- Koskela08 Koskela, L., 2008. *Test-Driven*, Manning
- Koskinen03 Koskinen, J., 2003. "Software Maintenance Costs," at <http://users.jyu.fi/~koskinen/smcosts.htm>

- KP99 Koomen, T., Pol, M., 1999. *Test Process Improvement*, Addison-Wesley
- Kylmäkoski03 Kylmäkoski, R., 2003. "Efficient Authoring of Software Documentation using RaPiD7," *Proceedings of the 25th International Conference on Software Engineering*
- Larman03 Larman, C., 2003. *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley
- Larman04a Larman, C., 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Prentice Hall
- Larman04b Larman, C., 2004. "Chapter 6: Use Cases," *Applying UML and Patterns*, at http://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf
- Laukkanen06 Laukkanen, P., 2006. *Data-Driven and Keyword-Driven Test Automation Frameworks*, Helsinki University of Technology, Master's Thesis, at <http://code.google.com/p/robotframework/>
- Lecht67 Lecht, C., 1967. *The Management of Computer Programming Projects*, American Management Association
- Leffingwell07 Leffingwell, D. 2007. *Scaling Software Agility*, Addison-Wesley
- Lencioni02 Lencioni, P., 2002. *The Five Dysfunctions of a Team: A Leadership Fable*, Jossey-Bass
- Lencioni05 Lencioni, P., 2005. *Overcoming The Five Dysfunctions of a Team*, Jossey-Bass
- LH07 Loughran, S., Hatcher, E., 2005. *Ant in Action*, Manning
- LH08 Liker, J., Hoseus, M., 2008. *Toyota Culture: The Heart and Soul of the Toyota Way*, McGraw-Hill
- Liker04 Liker, J., 2004. *The Toyota Way*, McGraw-Hill
- Link01 Link, J., 2001. *Unit Tests Mit Java: Der Test-First-Ansatz*, Dpunkt, translated as [Link03]
- Link03 Link, J., 2003. *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufman
- LM06a Liker, J., Meier, D., 2006. *The Toyota Way Fieldbook*, McGraw-Hill
- LM06b Liker, J., Morgan J., 2006. *The Toyota Product Development System*, Productivity Press
- Lundgren08 Lundgren, M., 2008. "The Agile Organization" presentation at Scrum Gathering 2009, Stockholm
- LV08 Larman, C., Vodde, B., 2008. *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, Addison-Wesley
- LV09 Larman, C., Vodde, B., 2009. *Lean Primer*, at www.leanprimer.com
- LW05 Lehmann, D., Winer, R., 2005. *Product Management*, McGraw Hill

- MacCormack01 MacCormack, A., 2001. "Product-Development Practices That Work," MIT Sloan Management Review. Vol. 42, No. 2.
- Magennis07 Magennis, T., 2007. "Continuous Integration and Automated Builds at Enterprise Scale," at <http://blog.aspiring-technology.com/file.axd?file=Continuous+Integration+at+Enterprise+Scale.pdf>
- Marick03 Marick, B., 2003, *My Agile Testing Project*, at <http://www.example.com/old-blog/2003/08/21/>
- Martin91 Martin, J., 1991. *Rapid Application Development*, Macmillan
- Martin02 Martin, R., 2002. *Agile Software Development: Principles, Patterns and Practices*, Addison-Wesley
- Martin04 Martin, R., 2004. "Estimating Costs Up Front," *Extreme Programming mailing list*, at <http://groups.google.com/group/comp.software.extreme-programming/msg/9a203fad85f3d363?hl=en>
- Martin08 Martin, R., 2008. *Clean Code*, Addison-Wesley
- MC05 Mugridge, R., Cunningham, W., 2005. *Fit for Developing Software*, Prentice Hall
- McBreen01 McBreen, P., 2001. *Software Craftsmanship*, Addison-Wesley
- McConnell04 McConnell, S., 2004. *Code Complete*, Microsoft Press
- Mecklenburg04 Mecklenburg, R., 2004. *Managing Projects with GNU Make*, O'Reilly
- Meszaros07 Meszaros, G., 2007. *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley
- Meyers76 Meyers, G., 1976. *Software Reliability*, Wiley-Interscience
- Meyers79 Meyers, G., 1979. *The Art of Software Testing*, Wiley-Interscience
- Mironov08 Mironov, R., 2008. *The Art Of Product Management*, Enthiosys Press
- MJ05 Moløkken-Østvold, K., Jørgensen, M., 2005. "A Comparison of Software Project Overruns—Flexible versus Sequential Development Models," *IEEE Transactions on Software Engineering*, Vol. 31, No. 9, Sept 2005
- MM08 Martin, R., Melnik, G., 2008. "Tests and Requirements, Requirements and Tests: A Möbius Strip," *IEEE Software*, Vol. 25, Issue 1, Jan/Feb 2008
- Monson-Haefel09 Monson-Haefel, R., 2009. *97 Things Every Software Architect Should Know: Collective Wisdom from the Experts*, O'Reilly Media
- Moore91 Moore, G., 1991. *Crossing the Chasm*, HarperCollins Publishers
- MR04 Manns, M.L., Rising, L., 2004. *Fearless Change: Patterns for Introducing New Ideas*, Addison-Wesley
- MRB98 Martin, R., Riehle, D., Buschmann, F., 1998. *Pattern Languages of Program Design 3*, Addison-Wesley
- MVN06 Manolescu, D., Voelter, M., Noble, J., 2006. *Pattern Languages of Program Design 5*, Addison-Wesley

- Netscape08 Netscape, 2008. "End of Support for Netscape Browsers," *Netscape Blog* at <http://blog.netscape.com/2007/12/28/end-of-support-for-netscape-web-browsers>
- NN03 Ngwenyama, O., Nielsen, P., 2003. "Competing Values in Software Process Improvement: An Assumption Analysis of CMM From an Organizational Culture Perspective," *IEEE Transactions on Software Engineering*, Vol 50. No. 1, Feb 2003
- Norman02 Norman, D., 2002. *The Design of Everyday Things*, Basic Books
- Norman05 Norman, D., 2005. *Emotional Design: Why We Love (or Hate) Everyday Things*, Basic Books
- North03 North, D., 2003. "Introducing BDD," *Better Software Magazine*, Aug 2003. Also at <http://dannorth.net/introducing-bdd>
- NV04 Newkirk, J., Vorontsov, A., 2004. *Test-Driven Development in Microsoft .NET*, Microsoft Press
- Ohno88 Ohno, T., 1988. *The Toyota Production System: Beyond Large-Scale Production*, Productivity Press
- Ohno07 Ohno, T., 2007, *Workplace Management*, Gemba Press
- OO00 Olson, G., Olson, J., 2000. "Distance Matters," *Human-Computer Interaction*, Vol. 15, Sept 2000
- Owen97 Owen, H., 1997. *Open Space Technology: A User's Guide*, Berrett-Koehler Publishers
- Parkinson57 Parkinson, C., 1957. *Parkinson's Law*, Buccaneer Books
- Parnas72 Parnas, D., 1972. "On the Criteria to be Used in Decomposing Systems in Modules," *Communications of the ACM*, Vol. 15, Issue 12, 1972, also in [HW01]
- Parnas94 Parnas, D., 1994. "Software Aging," *Proceedings of the 16th International Conference on Software Engineering*, also in [HW01]
- Patton05 Patton, J., 2005. "It's All in How You Slice It," *Better Software Magazine*. Jan 2005. Also at http://www.agileproductdesign.com/writing/how_you_slice_it.pdf
- Paulk01 Paulk, M., 2001, "Extreme Programming from a CMM Perspective," *IEEE Software*, Vol. 18, Issue 6, Nov/Dec 2001
- Paulk05 Paulk, M., 2005. "Foreword by Mark Paulk," in *Agile Software Development* [Koch05]
- Pichler10 Pichler, R., 2010. *Agile Product Management with Scrum*, Addison-Wesley
- Poole08 Poole, D., 2008. "Multi-Stage Continuous Integration," at <http://damonpoole.blogspot.com/2007/12/multi-stage-continuous-integration.html>
- Poppendieck04 Poppendieck, M., 2004. "An Introduction to Lean Software Development," at www.poppendieck.com/pdfs/Interview.pdf
- Poppendieck05 Poppendieck, M., 2005. "Agile Contracts" Agile 2005 Conference Workshop, at www.poppendieck.com/pdfs/AgileContracts.pdf

- Poppendieck06 Poppendieck, M., Poppendieck, T., 2006. *Implementing Lean Software Development: From Concept to Cash*, Addison-Wesley
- PRL07 Parsons, D., Ryu, H., Lal, R., 2007. "The Impact of Methods and Techniques on Outcomes from Agile Software Development Projects," *IFIP—Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda*. Springer (draft)
- PS06 Pfeffer, J., Sutton, R., 2006. *Hard Facts, Dangerous Half-Truths And Total Nonsense*, Harvard Business School Press
- Rasmusson04 Rasmusson, J., 2004. "Long Build Trouble Shooting Guide," *Proceedings of XP/Agile Universe 2004 Conference*
- Reeves92 Reeves, J., 1992. "What is Software Design?" *C++ Journal*, Fall 1992
- Reifer02 Reifer, D., 2002. "How Good are Agile Methods?" *IEEE Software*, July/Aug 2002.
- Reinertsen97 Reinertsen, D., 1997. *Managing the Design Factory*, Free Press
- Reppert04 Reppert, T., 2004. "Don't Just Break Software. Make Software," *Better Software Magazine*, Jul/Aug 2004
- RL06 Roock, S., Lippert, M., 2006. *Refactoring in Large Software Projects*, Wiley
- Robot09 Robot. 2009. *Robot Framework User Guide*, at <http://code.google.com/p/robotframework/wiki/UserGuide>
- Rogers04 Rogers, O., 2004, "Scaling Continuous Integration," *Proceedings of Extreme Programming 2004 Conference*
- Rogers08 Rogers, O., 2008, "Beyond Continuous Integration: Continuous Monitoring," at <http://www.hanselminutes.com/default.aspx?showID=131>
- RS98 Reger, G., Schmoch, U., 1998. *Organisation of Science and Technology at the Watershed*, Physica-Verlag Heidelberg
- SB02 Schwaber, K., 2002. *Agile Software Development with Scrum*, Prentice Hall
- Schatz05 Schatz, B., 2005. "Scrum at Primavera," presentation at Scrum Gathering 2005, Boston
- Schwaber04 Schwaber, K., 2004. *Agile Project Management with Scrum*, Microsoft Press
- Schwaber05 Schwaber, K., 2005. *Certified ScrumMaster Course*, version 6.3
- Schwaber06 Schwaber, K., 2006. *Scrum et al.* Video Google talk, at <http://video.google.com/videoplay?docid=-7230144396191025011#>
- Schwaber07a Schwaber, K., 2007. *The Enterprise and Scrum*, Microsoft Press
- Schwaber07b Schwaber, K., 2007. "Scrum Release 2.0?" *Scrum Alliance Articles*, at <http://www.scrumalliance.org/articles/12-scrum-release>
- Schwaber07c Schwaber, K., 2007. Story told in Certified ScrumMaster course.
- Schwaber09 Schwaber, K., 2009. *Scrum Guide, May 2009*, ScrumAlliance, at http://www.scrumalliance.org/resource_download/598

- Schwartz74 Schwartz, J., 1974. "Construction of Software: Problems and Practicalities," published in [Horowitz74]
- SD76 Doyle, M., Straus, D., 1976, *How to Make Meetings Work*, Jove Book
- SEI08 SEI, 2008. *CMMI FAQ: CMMI Appraisals*, at <http://www.sei.cmu.edu/cmmi/start/faq/appraisals-faq.cfm>
- SEI09 SEI, 2009. *CMMI Overview*, at <http://www.sei.cmu.edu/cmmi/general> (accessed Jan. 2, 2009)
- Senge94 Senge, P., 1994. *The Fifth Discipline*, Doubleday Business
- SF09 Shimokawa, K., Fujimoto, T., 2009. *The Birth of Lean*, Lean Enterprise Institute
- Shingo89 Shingo, S., 1989. *A Study of the Toyota Production System*, Productivity Press
- Shore03 Shore, J., 2003. "How I Use Fit," at <http://jamesshore.com/Blog/How-I-Use-Fit.html>
- Shore06 Shore, J., 2006. "Continuous Integration on a Dollar a Day," at <http://jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html>
- SJJ07 Sutherland, J., Jakobsen, C., Johnson, K., 2007. "Scrum and CMMI Level 5: The Magic Potion for Code Warriors," *Proceedings of the 2007 Agile Software Development Conference*, IEEE Computer Society
- Smith07 Smith, P., 2007. *Flexible Product Development: Building Agility for Changing Markets*, Jossey-Bass
- Spolsky04 Spolsky, J., 2004. *Joel on Software*, Apress
- SR07 Stewart, T. A., Raman, A. P., 2007. "Lessons from Toyota's Long Drive: Katsuaki Watanabe," *Harvard Business Review*, Vol. 85
- SSRB00 Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., 2000. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Addison-Wesley
- Stapleton03 Stapleton, J., 2003. *DSDM: Business Focused Development*. Addison Wesley
- Sutherland08 Sutherland, J., 2008. Mail to ScrumTrainer list
- Sutherland09a Sutherland, J., 2009. Personal communication
- Sutherland09b Sutherland, J., 2009. Personal communication
- Sutherland09c Sutherland, J., 2009. Personal communication
- SW07 Shore, J., Warden, S., 2007. *The Art of Agile Development*, O'Reilly
- Tate05 Tate, K., 2005. *Sustainable Software Development: An Agile Perspective*, Addison-Wesley
- TR98 Thomke, S., Reinertsen, D., 1998. "Agile Product Development: Managing Development Flexibility in Uncertain Environments," *California Management Review*, Fall 1998

- VCK96 Vlissides, J., Coplien, J., Kerth, N., 1996. *Pattern Languages of Program Design 2*, Addison-Wesley
- Venkatesh08 Venkatesh, U., 2008. *Managing Offshore Development Projects: An Agile Approach*, Multi-Media Publications
- VH09 Voos, K., Hileman, A., 2009. "Using Virtual Worlds for Distributed Agile" presentation at Agile2009 Conference, Chicago
- Vodde08 Vodde, B., 2008. "Measuring Continuous Integration Capability," *CrossTalk: The Journal of Defense Software Engineering*, May 2008
- Wageman99 Wageman, R., 1999. *Research on Managing Groups and Teams: Groups in Context*, JAI Press
- Wake03a Wake, W., 2003. "INVEST in Stories, and SMART tasks," at <http://xp123.com/xplor/xp0308/index.shtml>
- Wake03b Wake, W., 2003. *Refactoring Workbook*. Addison-Wesley
- Weinberg71 Weinberg, G., 1971. *The Psychology of Computer Programming*, Dorset House
- Weinberg08 Weinberg, G., 2008. *Perfect Software and Other Illusions about Testing*. Dorset House
- Wipro08 Wipro, 2008. *Wipro: About Us*, at <http://www.wipro.com/aboutus/whoweare.htm> (accessed July 10, 2008)
- WJ00 Weisbord, M., Janoff, S., 2000. *Future Search*, Berrett-Koehler Publishers
- WJR90 Womack, J., Jones, D. T., Roos, D., 1990. *The Machine That Changed the World*, Harper Perennial
- Womack09 Womack, J., 2009. "Why Toyota Won and How Toyota Can Lose," *Lean Enterprise Institute Newsletter*, at <http://www.lean.org/common/display/?o=750>
- WS95 Wood, J., Silver, D., 1995. *Joint Application Development*, Wiley
- Zahran98 Zahran, S., 1998. *Software Process Improvement*, Addison-Wesley

This page intentionally left blank

List of Experiments

Large-Scale Scrum

- Try...Large-scale Scrum FW-1 for up to ten teams 10
- Try...Large-scale Scrum FW-2 for ‘many’ teams 15

Test

- Avoid...Assuming *testing* means *testing* 24
- Try...Challenge assumptions about testing 25
- Avoid...Complex testing terminology 26
- Try...Simple testing classifications 27
- Avoid...Separating development and testing 29
- Avoid...Test department 30
- Avoid...Test department 32
- Avoid...TMM, TPI, and other ‘maturity’ models 32
- Avoid...ISTQB and other tester certification 32
- Try...Testers and programmers work together 33
- Try...Testers not only test 33
- Try...Technical writer tests 34
- Try...Educate and coach testing 34
- Try...Community of testing 35
- Try...Recognize project test smells 36
- Avoid...Separate test automation team 37
- Try...Feature team as test automation team 38
- Try...All tests pass—stop and fix 38
- Avoid...Using defect tracking systems during the iteration 39
- Try...Zero tolerance on open defects 39
- Avoid...Commercial test tools 40
- Try...Acceptance test-driven development 42
- Avoid...Traditional requirement handoff 46
- Avoid...Thinking A-TDD is for testers 47
- Avoid...Confusing TDD and A-TDD 47
- Try...A-TDD match the iteration flow 48
- Try...Discuss in workshop during Product Backlog refinement 49
- Try...Clarification over writing tests 49
- Try...Use examples 50
- Try...Product Owner writes tests 51
- Avoid...‘Optimizing’ the requirements workshop 51
- Avoid...Computers and projectors in the workshop 52
- Try...Condense workflow in business rules 52
- Try...Test the walls 52
- Try...Use table format 53
- Try...Workflow tests 54
- Try...Typical workshop agenda 54
- Try...Distill the tests 55

- Avoid...Multiple requirement descriptions 56
- Try...Use A-TDD coaches and facilitators 56
- Try...Robot Framework 57
- Try...Other A-TDD compatible tools 57
- Avoid...Conventional test tools for A-TDD 57
- Try...Wrap conventional test tools under an A-TDD tool 58
- Try...Show tests in Sprint Review 59
- Avoid...Confusing acceptance and user-acceptance test 59
- Try...Automate all tests 60
- Try...Manual tests 61
- Try...Write “A-TDD tests” for non-automatable requirements 62
- Try...Exploratory testing 62
- Try...Plan and time-box exploratory test sessions 64
- Try...Continuous Integration System 65
- Try...Maintainable tests 65
- Try...Refactor tests 66
- Avoid...Duplication in and between tests 66
- Try...Delete tests 66
- Avoid...Test through the UI 67
- Try...Run tests frequently 67
- Avoid...Traceability 67
- Try...Traceability 68
- Try...Treat non-functionals the same as functionals 69
- Try...Requirement area for non-functionals 70
- Try...Continuously run long-running tests 70
- Avoid...Expensive tests 71
- Try...Expensive tests 72
- Try...Automate expensive tests 72
- Try...Unit testing 72
- Try...CppUTest for C and C++ 73
- Avoid...Unit testing by a separate person 73
- Try...C++ xUnit framework for C 73
- Avoid...JUnit 73
- Try...Test-driven development 74
- Try...Use TDD coaches 74
- Try...Internal and external coaches 75
- Avoid...Write your own xUnit framework 76
- Try...Use a unit test framework in a compatible language 76
- Try...Write your own xUnit framework 76
- Try...Dual targeting 76
- Try...Run tests on the development environment 76
- Try...Run tests on the real hardware 77

- Try...Function-to-function-pointer refactoring 78
- Try...*Learning* tests 79
- Try...Learning tests for hardware 80
- Try...Refactor tests 81
- Try...Small tests that test only one thing 82
- Avoid...Slow unit tests 83

Product Management

- Try...Exploit *business* advantages of Scrum & lean thinking 100
- Try...Understand the changes with Scrum & lean thinking 104
- Avoid...Product management negotiating a “release contract” (scope & date) with R&D 106
- Try...Product management collaborates with R&D *each iteration*, adapting release scope or date 116
- Try...Challenge traditional product-management assumptions 117
- Try...Product Manager is Product Owner 120
- Avoid...Product Manager is *not* Product Owner 120
- Avoid...Fake Product Owner 121
- Avoid...Business manager is *not* Product Owner 121
- Try...Product management owns the product 122
- Try...Product Owner owns the product 122
- Avoid...Short-term product managers or focus 123
- Try...Fake Product Owner 123
- Try...Business manager is Product Owner 124
- Avoid...Believing Product Owner is just an analyst role 124
- Avoid...Believing Product Owner *must* attend the Daily Scrum 124
- Try...Product Owner product manager focuses outward to the market and channels 124
- Avoid...Too ‘inward’ product management & Product Owners 124
- Avoid...Too ‘outward’ product management & Product Owners 125
- Avoid...Us-Them: Product Owner versus Team 125
- Avoid...“Product Owner” 126
- Try...“Product Owner” 127
- Try...Overall product manager is chief engineer 128
- Avoid...Platform group with a “shared infrastructure” backlog 128
- Try...Add and do a cross-product common goal 128
- Try...Product Owners work together to maximize *company* ROI 131
- Try...One and only one Product Backlog 132

- Avoid...Fake team-level “Product Backlogs” 132
- Try...Area Product Owners when many teams 133
- Try...Product Owner Team 134
- Try...Map different scaling terms 134
- Try...Better behavior over ‘better’ PO scaling definitions 136
- Avoid...Try...“Product Owner Team” 136
- Avoid...Too inward-focused Product Owner Team 137
- Try...Product Owner representative (supporting PO) 138
- Try...Value 139
- Avoid...Value 140
- Try...Prioritize with multiple weighted factors 141
- Try...Include total life-cycle cost of an item 142
- Avoid...Feature priority categories 143
- Avoid...False dichotomy yes/no answers to customers 145
- Try...Involve real users or customers in Sprint Review 145
- Try...Product management connects teams and customers 146
- Avoid...Product management or Product Owner between teams and users 146
- Avoid...Multi-level P-M indirection from customers to teams 146
- Try...Shift R&D language toward P-M and user language 146
- Try...Extra help for product-manager Product Owner 147
- Avoid...SMEs not talking to customers 148
- Try...Product Management inspect and adapt 148
- Try...Product management education 149
- Try...Product Managers study Scrum & attend a course 149
- Try...Product managers Go See 149
- Try...Senior product managers coach 150
- Try...Invite displaced people to join product management 150

Planning

- Try...Kickstart large-scale Scrum with *one initial* Product Backlog refinement workshop 155
- Try...Continuous product development rather than projects 157
- Try...*Initial* Product Backlog refinement workshop 158
- Try...Scaling Sprint Planning Part One 163
- Try...Simple Sprint Planning Part Two 166
- Try...Asynchronous or joint Product Backlog refinement 166

- Try...Plan bounded research or learning items 166
- Try...Plan infrastructure items by regular teams 168
- Try...Avoid... Fixing defects 169
- Try...Product-level Definition of Done 170
- Avoid...Definition of Done defined by quality group 173
- Avoid...Undone Work 173
- Avoid...Needing a Release Sprint 173
- Avoid...Needing to 'harden' 175
- Try...Include Scrum teams in a Release Sprint 175
- Try...After one Release Sprint, hand off remaining Undone Work to the Undone Unit 177
- Try...Reduce—and eventually, remove—the Undone Unit over time 178
- Try...Expand the Definition of Done 178
- Try...Expand team-level Definition of Done 179
- Try...Avoid...Early and incremental handoff of Undone Work 179
- Avoid...Try...Planning an 'agile' release train 180
- Try...Estimate with Story Points 181
- Try...Avoid...Synchronize points and range 182
- Try...Combine progress measures 183
- Try...Avoid...Estimate velocity before iteration-1 184
- Try...Adjust duration estimate with Monte Carlo simulation 184

Coordination

- Try...Avoid...Cross-department coordinator 190
- Try...Integrate *all* functions into the teams 191
- Try...Focus on the overall product 193
- Try...Coordinator, ambassador, and scout activities 193
- Try...Team is responsible for coordination 194
- Avoid...External-to-team coordinator 195
- Avoid...Project managers 196
- Avoid...“Fake Scrum” by renaming the project manager role 196
- Avoid...ScrumMaster coordinates 197
- Try...Facilitation (rather than coordination) by ScrumMaster 197
- Try...Focus on overall product measures 198
- Avoid...Competition between teams 198
- Try...Myriad coordination methods 199
- Try...Scrum of Scrums 200
- Try...Use different questions for the Scrum of Scrums 201
- Try...Two-part Scrum of Scrums 202
- Avoid...Scrum of Scrums being a status meeting to

- management 202
- Avoid...Scrum of Scrums being a ScrumMaster meeting 203
- Try...CoP for ScrumMasters 203
- Try...Rotate Scrum of Scrums representatives 203
- Avoid...Frequently rotating representatives 203
- Try...Open Space 204
- Try...Town Hall meeting 205
- Try...Joint Scrum meetings 205
- Try...Joint Sprint Review bazaar 206
- Try...Prefer decentralization solutions over centralization ones 206
- Try...Send chickens to Daily Scrums 206
- Try...Travelers 207
- Try...Communities of Practice 207
- Try...Communication CoP 208
- Try...Increase shared space 208
- Try...Break cubicles and other barriers 209
- Try...Communicate in code 211
- Try...Communicate in tests 211
- Try...Environment mapping 211
- Try...Coordination working agreements 212

Requirements & PBIs

- Try...Group items into requirement areas 215
- Try...Group items into themes 216
- Avoid...Feature screening for PBIs 216
- Try...Prune an overgrown backlog 217
- Try...Prefer cell-like splitting over treelike splitting 217
- Try...Maintain at most one ancestor—direct or indirect 220
- Try...Maintain three levels when using Area Backlogs 221
- Avoid...Maintaining more than three levels of split items 222
- Try...Use special terms for size of items 222
- Try...Defer or ignore implementation *and analysis* of sub-items 223
- Avoid...Defect items in the Product Backlog—unless few 225
- Try...Add a single placeholder PBI for all defects—when many 225
- Try...“Undone Work” and system-level NFRs as PBIs 225
- Avoid...Try...Separate “Undone Work” from the Product Backlog 226
- Try...*Genuine* research work as PBIs 227
- Try...Research items quickly lead to customer-centric PBIs 228

- Avoid...Fake research items: regular analysis, ... 228
- Avoid...Giving research items to separate ‘research’ groups 228
- Try...Visual management for the Product or Release Backlog 229
- Try...Traceability with executable requirements as tests 229
- Try...Organize requirement artifacts to include... 229
- Avoid...‘Solving’ requirement problems with a documented meta-model 232
- Avoid...A complex requirements meta-model 233
- Avoid...Describing a simple meta-model in a complex way 233
- Avoid...Separate analysis or specialist groups 234
- Avoid...Separate systems-engineering group 234
- Avoid...Separate interaction design group 235
- Avoid...Separate architecture group 235
- Avoid...Fake team members 235
- Avoid...Product Owner Team as separate analysis group 236
- Try...Write customer-centric requirements (PBIs) 236
- Avoid...Technical task ‘requirements’ (PBIs) 237
- Avoid...Technical task PBIs in team-level “Product Backlogs” 238
- Try...Ask, “Would users understand every PBI?” 238
- Try...Prefer goal-oriented over solution-oriented requirements 238
- Try...Requirements workshops 240
- Avoid...Using computers in workshops 241
- Avoid...A large queue of well-analyzed, fine-grained PBIs 242
- Try...Maintain only a small queue of fine-grained PBIs 242
- Try...Requirements workshops for Product Backlog refinement 243
- Try...Specification by example—usually in tables 245
- Try...Joint requirement workshops 246
- Try...Stop refining an item once it is fully INVESTed 247
- Try...Split Product Backlog items (such as stories) 247
- Try...Ask, “What benefit from splitting in this way?” 250
- Avoid...Adopting user stories because they are ‘agile’ 266
- Avoid...Believing *writing* user stories means *user stories* 266
- Try...Apply user stories with card, conversation, confirmation 266
- Avoid...User stories good; other models bad 267
- Try...Learn *many* analysis skills: user stories, use cases, ... 268
- Try...Explore requirements as automated tests 271
- Try...Prefer PBI titles in C-style user-story format—usually 271
- Avoid...Requirements management and ALM tools—for N years after agile adoption 273
- Avoid...Old-style, centralized, and hierarchical document tools 274
- Try...“Web 2.0” decentralized, networked tools 275
- Try...Baseline and version-control in your “Web 2.0” tools 275
- Avoid...Requirement information in email 276
- Try...Aggregate email and discussion threads on webpages 276
- Try...RSS feeds on requirement page changes 276
- Try...Multiple page labels for a requirement page 276

Design & Architecture

- Try...Think ‘gardening’ over ‘architecting’—Create a culture of living, growing design 282
- Try...Design workshops with agile modeling 289
- Try...Just-in-Time (JIT) modeling; vary the abstraction level 295
- Try...Design workshops each iteration 295
- Try...A couple of days to a couple of weeks of design workshops before first iteration 296
- Try...Design workshops in the team rooms 297
- Try...Joint design workshops for broader design issues 298
- Try...Technical leaders teach at workshops 299
- Try...Architects and system engineers are regular (feature) team members 300
- Avoid...System engineers and architects outside of regular feature teams 300
- Try...Serious attention to user interface (UI) skills and design 300
- Try...UI designers in regular (feature) teams 300
- Avoid...UI designers in a separate UI design group 300
- Try...Architectural analysis *before* architectural design (repeat) 301
- Try...Question all early architectural decisions as final 301
- Avoid...Conformance to outdated architectural decisions 302
- Try...Hire and strive to retain master-programmer ‘architects’ 302

- Avoid...Architecture astronauts (PowerPoint architects) 302
 - Avoid..."Don't model" advice from extremists 303
 - Try...Prototypes in a *different* language 304
 - Try...Very early, develop a walking skeleton with tracer code 305
 - Try...Incrementally build 'vertical' architectural slices of customer-centric features 305
 - Try...Do customer-centric features with major architectural impact first 307
 - Try...Architects clarify by programming spike solutions 308
 - Avoid...Architects hand off to 'coders' 308
 - Try...Tiger team conquers then divides 308
 - Try...SAD workshops at end of "tiger phase" 310
 - Try...Agile SAD with views & technical memos 310
 - Try...Back up "human infection" with an agile SAD workshop 310
 - Try...Technical leaders teach during code reviews 312
 - Try...Experts participate in ongoing design workshops rather than late approval reviews 312
 - Avoid...Approval reviews by experts at the end of a step 312
 - Try...Design/architecture community of practice 313
 - Try...Show-and-tell during workshops 313
 - Try...Component guardians for architectural integrity when shared code ownership 314
 - Try...Component mailing lists 314
 - Try...Internal open source with teachers—for tools too 315
 - Try...Configurable design for customization 315
 - Avoid...Branches for customization 315
 - Avoid...Create 'designs' and then send them for offshore implementation 316
 - Try...Architectural and design patterns 316
 - Try...Promote a shared pattern vocabulary 316
 - Try...Test on the old hardware as soon as possible 317
 - Try...HTML-ize and hyperlink your entire source code, daily 317
 - Try...Lots of stubs, plus dependency injection 318
 - Avoid...Using stubs to delay integration 319
 - Try...Test-driven development for a better architecture 319
 - Try...Dependency injection framework 320
 - Try...Use an OS abstraction layer 320
 - Try...Create a low-level hardware abstraction layer (HAL) API 320
 - Try...Create a mid-level object-oriented HAL 321
 - Try...Create simulation layers for hardware, etc.
- 321
- Try...More FPGAs and fewer ASICs 322
 - Avoid...Big upfront interface design 324
 - Try...Start with some weakly-typed interfaces, then strengthen 324
 - Try...Simplify interface change coordination with feature teams 326
 - Avoid...Freezing interfaces 327
 - Try...Wrap calls to remote components with proxies or adapters 327
 - Try...Start with indirect interaction between major components, then replace as needed 327
- ## Legacy Code
- Avoid...Fixed content with unrealistic deadlines 335
 - Try... Transparency and customer collaboration 337
 - Avoid...Hiring many weak developers 339
 - Avoid...Believing universities teach development skills 340
 - Try...Increase organizational support for learning development skills 340
 - Try...Support more self-study 341
 - Avoid...Trivializing programming 341
 - Try...Raise awareness of the negative impact of legacy code 342
 - Avoid...Rewriting legacy code 343
 - Try...Clean up your neighborhood 346
 - Try...Write both high-level and unit tests 346
 - Try...Rewrite lethal legacy code 347
- ## Continuous Integration
- Avoid...Believing CI is a tool 352
 - Avoid...Large batches of changes 355
 - Avoid...Process preventing developers from checking in 357
 - Avoid... Branching 358
 - Try...Speed up the build 361
 - Try...Add new hardware to speed up the build 362
 - Try...Parallelize the build 362
 - Avoid...Using ClearCase 362
 - Avoid...Treating test code differently than production code 364
 - Try...Multi-stage CI systems 364
 - Try...A mix between feature and component CI systems 365
 - Avoid...Manual promotion 365
 - Try...Visual management with CI 367
 - Try...Add red-green screens to your CI system 368

- Avoid...Large changes 369
- Avoid...Leaving obsolete interfaces in your code 369
- Avoid...'Solving' organizational problems with technical solutions 370

Inspect & Adapt

- Avoid...Adoption with top-down management support 374
- Try...Adoption with top-down management support 375
- Try...Individuals & interactions over processes & tools 376
- Try...Job and personal safety (not role safety) 376
- Try...Patience 378
- Avoid...Adopting "do agile/lean" 378
- Avoid...Being agile/lean without agile/lean practices/tools 379
- Avoid...Agile/lean transformations or change projects 380
- Try...Agile/lean adoption *forever* 381
- Try...Impediments service rather than change management 381
- Try...Human infection 385
- Avoid...Agile/lean adoption targets or rewards 385
- Avoid...Competitive 'improvement' 386
- Avoid...Try...'Easy' agile or lean adoption 386
- Try...Experiment rather than improve 387
- Avoid...Forcing adoption of practices 387
- Try...Encourage experiments; offer coaching 387
- Avoid...Adopting <X> because "agile didn't work here" 388
- Avoid...IBM/Accenture/... agile adoption 388
- Avoid...Adopting agile with "agile management" tools 389
- Try...Transition from component to feature teams gradually 391
- Avoid...Waiting for the organization chart 393
- Avoid...In-line 'ScrumMaster' line- or project managers 393
- Try...Line manager as ScrumMaster of out-of-line team 393
- Try...Break the walls—team areas with whiteboards 394
- Try...Two-week iterations to break waterfall habits 394
- Try...One flip chart for tasks of one Product Backlog item 395
- Try...Repeating large-audience introductions 397
- Try...Open-Space Technology for early-days adoption 398
- Try...Big gatherings to share stories & experi-

- ments 398
- Try...Central coaching group 399
- Avoid...Central coaching group with formal authority 399
- Try...Concentrate the coaching on a few products 399
- Try...External agile coaches 399
- Try...Pair external agile coaches with internal ones 400
- Avoid...Advisors/consultants who are not hands-on coaches 400
- Try...Structured intensive curriculum for all teams 401
- Avoid...Internal agile/lean cookbooks 401
- Try...Joint Sprint Retrospectives 403
- Try...Joint Retrospective big improvements in Product Backlog 404
- Try...Cross-team working agreements 405
- Try...Joint Sprint Reviews 405
- Avoid...Try...Individual team-level Sprint Review 406
- Try...Spend money on improving, instead of "adding capacity" 406
- Try...Lower the waters in the lake 407
- Avoid...Rotating the ScrumMaster role quickly 408
- Try...Reduce harm of policies that cannot yet be removed 408

Multisite

- Try...Fewer sites 414
- Try...Think 'multisite' even when close 415
- Avoid...Believing in multisite *Daily Scrum magic* or that multisite forces are inconsequential 415
- Avoid...Thinking 'distributed' must mean 'dispersed' 416
- Avoid...Thinking distributed pair programming is required 416
- Try...One iteration (Sprint) for the product, not for the site 417
- Avoid...Sites organized by components or functions 417
- Try...Allocate a whole feature to a co-located feature team 418
- Avoid...Dispersed groups or 'teams' 419
- Try...A dispersed feature 'team' only if it really hurts 420
- Try...Gradual transition to co-located Scrum feature teams 421
- Try...Temporary co-location of a new dispersed team 422
- Try...Learn at existing sites, rather than add 'ex-

- pert' sites 422
- Try...Prefer co-location of feature teams and Area Product Owner of one requirement area...but do not restrict this 423
- Try...Treat all sites as equal partners 423
- Try...Continuous integration in "one repository" across sites 424
- Try...Seeing is believing—ubiquitous cheap video technology and video culture 425
- Try...Include diverge-converge cycles in large video meetings 428
- Try...Start early multisite video meetings informally 428
- Try...Multisite planning poker (estimation poker) 429
- Try...Multisite Open Space to replace Scrum of Scrums 430
- Try...Experiment with multisite Scrum meeting formats and technologies 431
- Try...Cross-pollination 432
- Try...Welcoming committees and buddies 433
- Try...Multisite communities of practice (CoP), including a communications CoP 433
- Try...Retrospectives at several levels 433
- Avoid...ScrumMaster representing the team 434
- Try...ScrumMasters acting as and encouraging matchmakers 435
- Try...Improve multisite design with *Design* chapter tips 435
- Try...Basic practices for multisite meetings 435
- Try...Vigilance for shared agile vocabulary and concepts 437
- Try...Cultural education 437
- Try...Vigilance about a common coding style 438
- Try...Multisite tool that records audio or video 438
- Try...Tablets for shared sketching 439
- Avoid...Commercial 'agile' tools for multisite collaboration 439
- Avoid...Commercial development tools; use free tools 440
- Try...Wikis as your share point; employ a Wiki-Gardener 440
- Avoid...ClearCase for multisite continuous integration 441
- Try...Remove barriers between offshore team and onshore client 450
- Try...Matchmakers rather than intermediaries 450
- Avoid...Single point of contact 450
- Try...Seeing is believing—video sessions 451
- Try...Remote Sprint Review 454
- Try...Seeing is believing—client visits team 454
- Try...Team members visit client 455
- Try...Rotating ambassadors 455
- Try...Translator on team 455
- Try...Offshore team speaks English 456
- Try...Clients participate in a Sprint Retrospective 456
- Try...Offshore group first does several iterations onshore 457
- Try...Proactively find and educate an onshore Product Owner 457
- Avoid...Believing 'yes'; ask open questions 458
- Try...Offshore requirement workshops each iteration 458
- Try...Offshore domain and vision workshop 460
- Try...Requirements documentation adaptively 'simple' 461
- Try...Frequent onshore UI prototypes 461
- Try...Semi-detailed requirements documentation for iteration 462
- Try...Detailed requirements with A-TDD 462
- Try...Wiki for all requirements 462
- Try...A-TDD for UAT 463
- Try...Manual (if you must) UAT each iteration 463
- Try...Manual pre-UAT after each feature 464
- Try...Iterative requirements onshore to offshore 465
- Try...Stable offshore Scrum teams 466
- Try...Simple titles map to special titles 467
- Try...Encouraging the teams to say 'no' 467
- Try...A ScrumMaster intent on self-organizing teams 468
- Try...Long-term agile coaching group if high attrition 468
- Try...Outside-the-site agile coaches 469
- Try...Buddy system if high attrition 469
- Avoid...Onshore management, offshore development 469
- Try...Offshoring features, not disciplines or components 470
- Try...Treating the offshore organization as internal partners 470
- Try...Dispersed feature team if us-them is a problem 472

Offshore

- Try...Educate that agile offshore is not just short iterations 446
- Try...Agile guide for sales people and prospects 448
- Try...Kickoff agile workshop to educate customers 448

- Avoid...Unbalanced onshore favoritism or bias 472
- Avoid...“four-year programmer” partners 473
- Try...Experts coach/review rather than dictate design 474
- Avoid...Outsourcers saying “Leave it to us, we *do agile* for you” 475
- Avoid...Outsourcers with top-heavy management 476
- Avoid...“four-year programmer” outsourcers 477
- Avoid...Outsourcers whose environment does not “walk the agile talk” 477
- Avoid...Outsourcers with analysis, coding, or testing ‘factories’ 478
- Avoid...Try...Large outsourcers 479
- Try...Interview outsourcer-programmers by *programming* 479
- Try...The great programmers forever 480
- Try...Improve together with your outsourcer 480
- Avoid...Believing CMMI appraisal or certification means much in creative R&D work 489
- Avoid...Believing ‘agile’—or any—certification means much 493
- Avoid...Toxic CMMI consultants and appraisers 494
- Try...Alternative contract models 494
- Try...Fixed price and fixed scope with agility 495
- Avoid...Commercial tools 495

Contracts

- Try...Share these key insights with contract lawyers 500
- Try...Lawyers study agile, iterative, & systems-thinking concepts 501
- Try...Appreciate a traditional lawyer’s point of view 502
- Try...Debug common misunderstandings when lawyers are introduced to the third agile value 504
- Try...Lawyers study problems arising from silo mentality and lack of systems thinking 505
- Try...Lawyers study the impact of potentially deployable two-week increments on assumptions and contracts 509
- Try...Lawyers study how agility reduces risk and exposure 511
- Try...Heighten lawyer sensitivity to software project complexity by analogies to legal work 513
- Avoid...Incentives and penalties 514
- Try...Share the pain/gain 515
- Avoid...“Quality Management Plan” and “Deliverables List” 515
- Try...Collaborate early and often with lawyers 516
- Avoid...Fixed-price, fixed-scope (FPFS) contracts

- 531
- Try...Variable-price variable-scope progressive contracts 536
- Try...Increase flexibility in project and contract variables 538
- Try...Target-cost contracts 540
- Try...Multi-phase variable-model frameworks 543

This page intentionally left blank

INDEX

A

acceptance test-driven development

- coach 56
- compared to test-driven development 47
- definition 42
- for requirements 271
- for UAT 463
- in iteration 48
- is not testing 47
- offshore 462
- overview 44
- recommended reading 96

adapters 327

adoption

- agile curriculum 401
- avoid cookbooks 401
- large-group introductions 397
- Open Space 398
- overview 373
- project 380
- targets 385

Adzic, Gojko 49

agile modeling 268, 292, 303

- in design workshops 289

ambassador

- activities in coordination 194
- multisite 432
- offshore 455

analysis

- see requirements

analysis group 234

Ancona, Deborah 193

andon 359

appraisals

- CMMI 480

appraisers

- CMMI 494

Arbogast, Tom 499

architect

- active master programmers 288, 302
- astronauts 302
- avoid handing off to programmers 308
- avoid separate review of work 312
- coaches during design workshops 299
- impact if not programming 286

PowerPoint 285, 302

program spikes 308

programmer in tiger team 308

teaches during code reviews 312

architecture

analysis 301

and customer-centric features 307

build vertical slices 305

Community of Practice 313

design 301

documentation 310

see SAD workshops

group 234

integrity 293

outdated 302

question finality 301

see also design

spikes 308

versus growing, gardening 282

Area Backlog 15, 133, 215, 221, 555

Area Product Owner 15, 133, 135, 136, 215, 423, 555

artifacts

see documentation

A-TDD

see acceptance test-driven development

attrition 468, 469

B

backlog grooming

see Product Backlog refinement

best practices 4, 492

branching 358

browser wars 334

bug-free code 39

build speed 361

business advantages 100

business analyst

not the Product Owner 124

business manager

as Product Owner 121

business rules 52

C

- C++ unit testing 73
- career paths 342
- cargo cult 2
- Carmel, Erran 413
- certifications
 - agile 493
 - CMMI 480
- change management
 - contracts 521
- change project 380
- changes
 - large ones 369
- chief engineer 128, 191
- chief Product Owner 135
- clean up your neighborhood 346
- ClearCase
 - avoid 362, 441
- CMMI
 - appraisers 494
 - overview 480
- coaches
 - avoid coaches who aren't hands-on 400
 - external 399
 - external and internal 400
 - offshore 469
- coaching
 - internal group 399
- code
 - HTLM-ize it 317
 - is the design 282
 - multisite 438
 - reviews 312
- coffee 86
- Cohn, Mike 195
- collaboration 116
- co-located team 413
- commitments 190, 335
- committer role 314
- communicate in code 211
- communication barriers 209
- Communities of Practice
 - design/architecture 313
 - for communication 208
 - general 207
- multisite 433
- Community of Practice
 - testing 35
- competition between teams 198
- component guardians 314
- component teams
 - drawbacks 553
 - to feature teams 391
- Concordion 57
- continuous integration
 - developer practice 352
 - how frequently? 356
 - misconceptions 351
 - multisite 424
 - overview 351
- continuous integration system
 - multi-stage 364
 - overview 65, 359
 - scaling 361
 - scaling example 366
- continuous product development 157
- contract game 106
- contract negotiation 106
- contracts
 - acceptance 522
 - agile 518
 - appreciate lawyer point of view 502
 - change management 521
 - collaborate with lawyers 516
 - collaboration 116
 - common misunderstandings 504
 - contract game 106
 - deliverables 525
 - delivery 519
 - fixed price 527
 - fixed-price fixed-scope 531
 - hybrid pricing 530
 - incentives, rewards, penalties 514
 - internal 190
 - key agile insights 500
 - lawyer education 501, 509, 511, 513
 - liability 524
 - multi-phase models 539
 - multi-phase variable-model 543
 - offshore 494

- overview 499
payment timing 526
pay-per-use pricing 529
progressive 536
release contract 106
scope 519
silo mentality 505
target-cost 520, 540
termination 522
thinking about 500
time and materials 527
traditional assumptions 504
value-based pricing 528
variable-price variable-scope 536
warranty 525
- cookbooks 401
coordination
 centralized 200
 cross-department 190
 decentralized 206
 meetings 200
 responsibility for 196
 ScrumMaster's responsibilities 197
 team is responsible for 194
 thinking about 189
 travelers 207
- coordinator 190
coordinator, ambassador, and scout 193
copy-paste 336
CppUTest 73
craftsmanship 337, 339
cross-department coordinator 190
cross-functional teams 191
cubicles 209
Cucumber 57
culture
 multisite 437
 overview 468
- Cunningham, Ward 57
customer documentation 192
customer-facing test 42
customers 145
- D**
- Daily Scrum 14, 124
defect tracking 39
defects
 (to fix) in Product Backlog 225
 zero tolerance 39
Definition of Done 15, 170, 178
demo preparation 59
department interfaces 190
dependency injection 318, 319, 320
design
 multisite 435
 overview 281
 see also architecture
 sending offshore 316
 thinking about 282
 walking skeleton 305
design patterns 316
design workshops
 at the start 296
 each iteration 295
 in team room 297
 joint for multiple teams 298
 overview 289
developer testing 72
development skills 335, 339
discuss-develop-deliver cycle 44
dispersed team 413, 416, 419, 420, 472
distributed teams 413, 416
documentation
 architectural 310
 offshore 461, 462
 requirements
- done
 see Definition of Done
- dual targeting 76
duplication
 between requirements and tests 56
 between tests 66
 code 82
- E**
- education
 for all teams 401

embedded software
learning tests for new hardware 80
testing 77
environment mapping 211
epic
see splitting
terminology 222
estimation
Monte Carlo simulation 184
multisite 429
overview 181
value 139
examples for requirements 50, 245
experiments 2
exploratory testing 62
external coordinator 195
Extreme Programming
see XP

F
false dichotomies 2
Feathers, Michael 73
feature 222
feature screening 216
feature teams
as automation team 38
choosing 554
dispersed 420
from component teams 391
in large-scale Scrum 12
multisite 418
overview 549
transition 555
vs component teams 551
vs project groups 552
Fit 57
FitNesse 57
fixed-price contracts
see contracts
flexibility and specialization 551
Fowler, Martin 351
FPGA 322
function-to-function-pointer refactoring 78
FURPS+ 231

G
Git 358
Grenning, James 97
grooming
see Product Backlog refinement
growing vs building 355

H
Hackman, Richard 198
hardware 317
hardware abstraction layer 320, 321
hardware design 322
hardware simulators 71
Hetzl, Bill 29
Hohmann, Luke 152

I
impediments
backlog 381
service 381
improvement 373
incentives 514
incremental handoff 179
infrastructure work 128, 168
inspect-adapt
overview 373
product management 148
interaction design
see UI design
interaction design group 234
interface API design 323, 324, 326
INVEST test 247
ISTQB 32
iteration planning
see Sprint Planning

J
jidoka 353
JIT modeling 295
joint design workshop 298
joint requirement workshops 246
joint Scrum meetings 205
joint Sprint Retrospective 15, 17, 403

joint Sprint Review 17, 405

K

Klärck, Pekka 57

L

lake and rocks metaphor 407

language 456

large-scale Scrum

- artifacts 13

- definition 9

- framework-1 10

- framework-2 15

- overview 9, 10

- roles 12

law of communication paths 199

law of the inverse relationship between size and skill 339

lead Product Owner

- overall 135

learning debt 336

learning tests 79

Lecht, Charles 334

legacy code

- awareness 342

- lethal 347

- overview 333

- solution 343

line manager 393

M

maintainable tests 65

Marick, Brian 27

Martin, Bob 57

matrix organization 31

MDA 291

MDD 291

meetings

- multisite 428, 431, 435

Meszaros, Gerard 36

milestone 108

mocks 318, 321

modeling

agile 292

avoid extremists 303

just-in-time 295

requirements

tools 291

Monte Carlo simulation 184

moving skeletons 368

multisite

- ambassador 432

- avoid ClearCase 441

- centrifugal forces 413

- coding style 438

- Communities of Practice 433

- continuous integration 424

- culture 437

- design 435

- dispersed vs. distributed 416

- estimation 429

- feature team 418

- is non-trivial 415

- matchmakers 435

- meetings 425, 431, 435

- one iteration per product, not site 417

- Open Space 430

- overview 413

- partner sites 423

- planning poker 429

- Scrum of Scrums 430

- shared space 209

- site organization 417

- teams 420

- thinking about 414

- tools 438, 439

- transition to feature teams 421

- video culture 425, 428

- visits 432

myriad coordination methods 199

N

Netscape 334

non-functional requirements

- in Product Backlog 225

- see FURPS+

O

offshore

- acceptance TDD 462
 - ambassador 455
 - certification 480
 - CMMI 480
 - CMMI appraisers 494
 - coaches 469
 - contracts 494
 - culture 468
 - design problems 316
 - documentation 461, 462
 - domain and vision workshop 460
 - educate customers 446
 - educate Sales 448
 - feature teams 470
 - kickoff workshop 448
 - language 456
 - matchmakers 450
 - onshore partnership 469
 - onshore Product Owner 457
 - overview 445
 - partnership 470
 - planning 470
 - remove barriers 450
 - requirements 462, 465
 - requirements workshop 458
 - ScrumMaster 468
 - Sprint Retrospective 456
 - Sprint Review 454
 - team visits onshore 457
 - teams 466
 - titles 467
 - tools 495
 - translator on team 455
 - UAT 463, 464
 - UI design 461
 - video sessions 451
 - visits both ways 454
- Open Space
- for agile adoption 398
 - multisite 430
 - overview 204
- outsourcer
- avoid factories and factory mindset 478

choose good programmers 479

- choosing 475
- four-year programmers 477
- improve together 480
- poor environment 477
- top-heavy management 476

outsourcing

- and legacy code 341
- choosing a partner 475
- overview 445

overall product focus 193, 198

overall Product Owner 135

overburden 337

P

PBI 215

PDMA 152

penalties 514

personal safety 376

Pichler, Roman 152

planning

- infrastructure 168
- iteration 163
- overview 155
- research and learning 166
- Sprint 163

planning poker

multisite 429

platform departments 191

platform development 128, 168

Poppendieck, Mary 4

potentially shippable 26

potentially shippable product increment 14, 170

PowerPoint architects 302

practices

context dependent 4

pricing

contracts and outsourcing 527

prioritization

- avoid categories 143
- of Product Backlog 139
- of value 139, 141

Product Backlog

Area Backlog 215

- avoid tasks 237
 - avoid team-level backlogs 238
 - creation 155
 - items 215
 - major improvement goals 404
 - one per product 132
 - only one per product 13
 - PBI 215
 - prioritization 139, 141
 - refinement 166
 - themes 216
 - visual management 229
- Product Backlog refinement
- for A-TDD 49
 - initial 155, 158
 - joint or asynchronous 166
 - overview 15
 - workshop 243
- product management
- avoid short-term focus 123
 - changes when adopting Scrum 104
 - collaboration with R&D 116
 - contract negotiation 106
 - inspect-adapt 148
 - overview 99
 - traditional assumptions 117
- product manager 120, 126, 128
- Product Owner 120, 122, 135
- Area 15, 133, 135, 136, 215
 - avoid too inward 124
 - chief 135
 - fake 123
 - has business authority 121
 - help from Team 147
 - interaction with other POs 131
 - lead 135
 - looks outward 124
 - not just an analyst 124
 - offshore development 457
 - overall 135
 - overview 12, 120
 - PO Team 17, 136, 137, 236
 - proxy 135
 - representative 135, 138
 - supporting PO 134, 135
- us-them versus Team 125
- Product Owner representative 135, 138
- Product Owner Team 17, 136, 137, 236
- profit 141
- program manager 190
- project managers 196, 393
- projects
- prefer product view 127, 157
- prototypes 304
- proxies 327
- proxy Product Owner 135
- punching holes 210

R

- refactoring
- large ones 369
- Reinertsen, Donald 4
- relative value points 139
- release contract 106
- release planning 155, 158
- Release Sprint 175, 177
- release train 180
- requirement areas 133, 215
- for non-functionals 70
 - overview 555
 - vs development areas 556
- requirements
- acceptance TDD 271
 - artifacts 229
 - by example 245
 - clarifying by writing tests 49
 - customer-centric 236
 - meta-models 232, 233
 - multiple descriptions 56
 - non-functional 225
 - offshore 462
 - offshore to onshore 465
 - offshore workshop 458
 - overview 215
 - splitting 217, 247
 - tables 245
 - tool 462
- requirements workshop
- A-TDD workshop agenda 54

- for Product Backlog refinement 243
overview 240
so-called optimizing 51
- research
fake 228
in Product Backlog 227
planning it 166
rewards 385, 514
risk 141
- Robot Framework
architecture 87
calling C code 90
example using 83
introduction 57
test library 86
types of tables 86
- room
see team room
- rubber chicken 354
- S**
- SAD workshop 310
safety (personal) 376
SAGE 338
salary 342
scenario 249
Schwaber, Ken 9
Scientific Management, critique 4
scout activities 194
Scrum
see large-scale Scrum
Scrum 2.0 9
Scrum of Scrums
alternatives
Open Space 204
Town Hall 205
format 201
multisite 430
overview 200
rotate representatives 203
rotate representatives too frequently 203
ScrumMaster's role 203
status to management 202
two parts 202
- ScrumMaster
avoid representing team 434
in large-scale Scrum 13
not project manager 393
offshore 468
slow rotation 408
- Second Life 209
secret toolbox 336
shared space 208
simulation layers 321
- Slim 57
small changes 355
specialization 550
spikes 308
splitting requirements 217, 247
splitting user stories
see splitting requirements
- Sprint Backlog 13
- Sprint Planning 163
in large-scale Scrum 14, 17
multisite issues 165
part one 163
part two 166
- Sprint Retrospective
in large-scale Scrum 15
joint 17, 403, 433
multisite 433
offshore 456
- Sprint Review
bazaar 206
in large-scale Scrum 15, 17
joint 17, 405
multisite 454
offshore 454
show tests 59
team level 406
- stakeholders 141
stop and fix 38
- stories
see user stories
- story points 181
strategic alignment 141
stubs 318, 321
supporting Product Owner 134, 135

T

- tables for requirements 245
- target-cost contracts 520
- targets 385
- task-coordinator activities 194
- tasks
 - in Product Backlog 237
- Taylor, Frederick 4
- TDD
 - see test-driven development
- Team (in Scrum) 12
- team room 297, 394
- team size 192
- teams
 - cross-functional 234
 - technical debt 336
 - technical writing 34
 - test 23
 - test automation team 37
 - test department 30
 - test education 34
 - test independence 29
 - test sessions 64
 - test smells 36
 - test tools
 - commercial 40
 - conventional 57
 - wrap conventional tools 58
- test-driven development
 - better architecture 319
 - coach 74
 - internal 75
 - overview 74
- tester certification 32
- testing
 - and Product Owner 51
 - assumptions 24, 26
 - before release 42
 - classifications 27
 - customer-facing 28, 42
 - developer 28, 72
 - in Sprint Planning 41
 - in Sprint Review 42
 - keyword-driven 83
 - legacy code 346
- manual 60, 61
- meaning of 24
- on the hardware 317
- overview 40
- skills 96
- specialization 33
- terminology 26
- thinking about 24
- through the UI 67
- traditional 46
- UAT 463, 464
- using walls 52
- testing community 35
- tests
 - automated 60
 - deleting 66
 - distill 55
 - expensive 71
 - long-running 70
 - non-automatable 62
 - non-functional 69
 - on development environment 76
 - on real hardware 77
 - performance 70
 - refactoring 81
 - reliability 70
 - table format 53
 - user-acceptance 59
 - workflow 54
- testware 37
- themes 216
- tiger team 308
- TMM 32
- tools
 - agile management 389
 - for modeling 291
 - for requirements 273
 - multisite 438, 439
 - offshore 495
 - requirements offshore 462
 - testing 40, 56, 57
- Town Hall meeting 205
- TPI 32
- traceability 67, 68, 229
- tracer code 305

transformation project 380
transition
 component teams to feature teams 391
 overview 373
 to feature teams 421
travelers 207
trivializing programming 341
TTCN 57

U

UAT 463
 pre-UAT 464
 with A-TDD 463
UI design
 importance of 300
 offshore 461
UML 291, 295
Undone Unit 31, 177
Undone Work 173, 177, 179, 225, 226
unit testing
 overview 72
 rules for 73
unit tests
 slow 83
use case 249
user stories
 formats 271
 history 223, 271
 overview 266
 question their use
 splitting
 term 222
user-acceptance test 59
 see UAT

V

value 139, 141
velocity 184
video sessions 451
video technology 425
virtual shared space 209
virtualization of hardware 71
visual management 229, 367

W

walking skeleton 305
weakly-typed interfaces 324
whiteboards 290
wikis 275, 440, 462
wishful thinking 337
workflow test 54
working agreements
 cross-team 405
 for coordination 212
workshops
 design 289, 295, 296, 297
 initial Product Backlog refinement 158
 joint 246
 joint design 298
 multisite 428
 requirements 54, 240, 243
 SAD 310

X

XP 303
xUnit 76