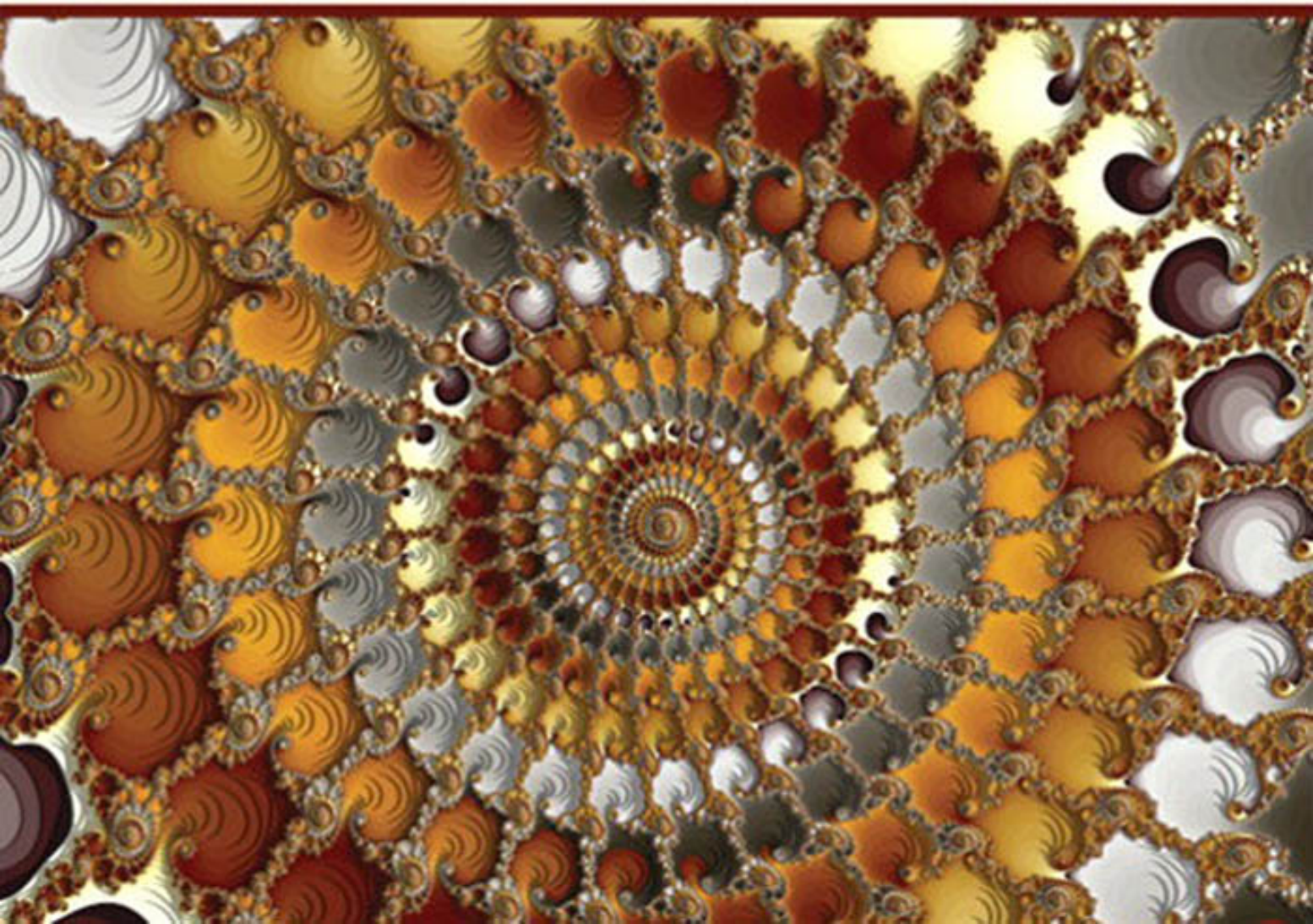
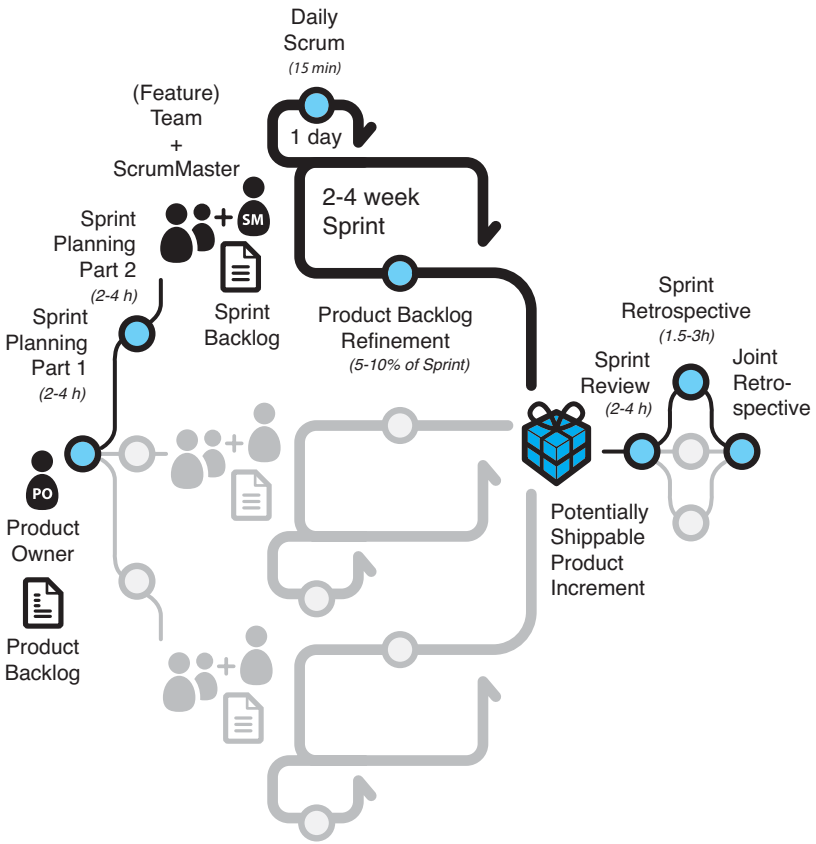
Practices for Scaling Lean & Agile Development

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

Craig Larman
Bas Vodde

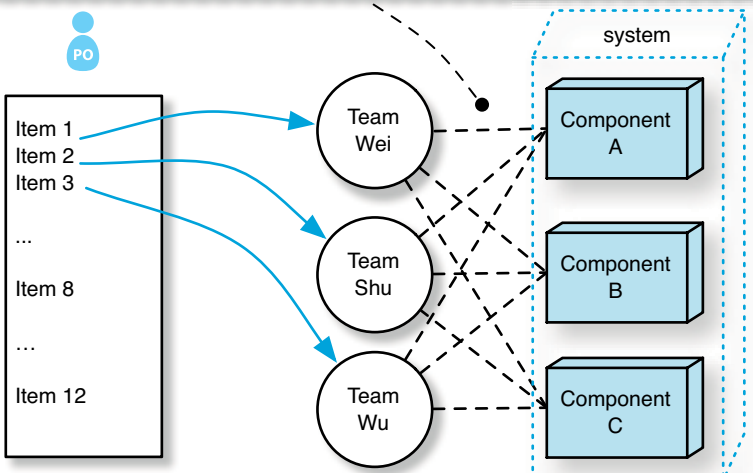


One Example Framework for Large-Scale Scrum



Feature Teams

With feature teams, teams can always work on the highest-value features, there is less delay for delivering value, and coordination issues shift toward the shared code rather than coordination through upfront planning, delayed work, and handoff. In the 1960s and 70s this code coordination was awkward due to weak tools and practices. Modern open-source tools and practices such as TDD and continuous integration make this coordination relatively simple.

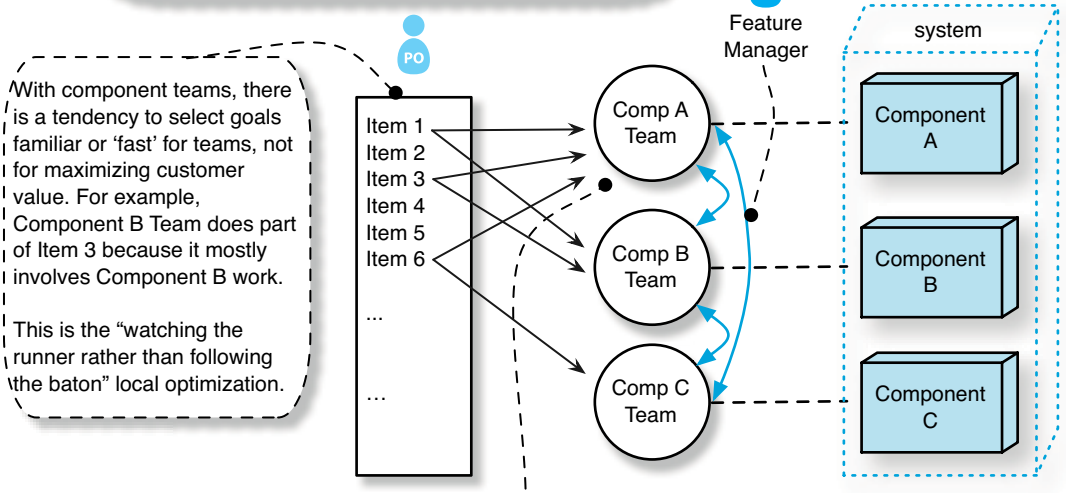


Component Teams

With component teams, a project or feature manager is used to coordinate and see to completion a feature that spans component teams and functional teams.

With component teams, there is a tendency to select goals familiar or 'fast' for teams, not for maximizing customer value. For example, Component B Team does part of Item 3 because it mostly involves Component B work.

This is the "watching the runner rather than following the baton" local optimization.



With component teams, there is increased delay, as one customer feature is split across multiple component teams for programming, and eventually transferred to a separate testing team for verification. There is handoff waste, and multitasking waste—as one component team may work on several features in parallel, in addition to handling defects related to 'their' component.

Practices for Scaling Lean & Agile Development

This page intentionally left blank

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

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

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

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 Alfvin, 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, Lisa 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, Shivakumar 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

Book

Chapter

1

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 unique[4] 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.*

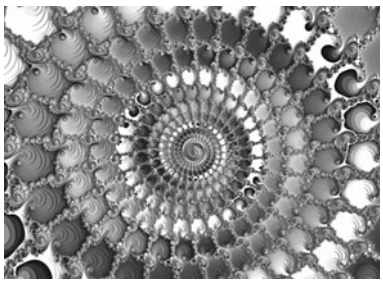
In *Managing the Design Factory*, a similar point is made:

> *…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 Childress.

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.

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

Book

Chapter

2

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 process[1] 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.

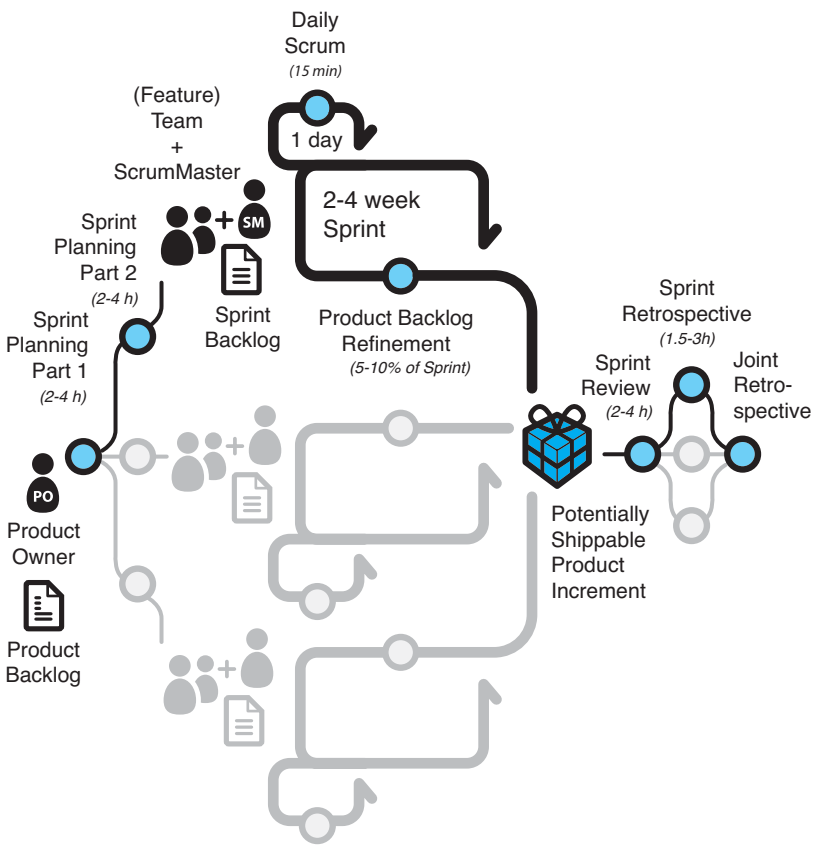
Try…Large-Scale Scrum FW-1 for up to Ten Teams

The first framework is appropriate for one (overall[3]) 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] explains[4]:

> *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

- ❑ Sprint Planning
- ❑ Daily Scrum
- ❑ Product Backlog Refinement
- ❑ Sprint Review
- ❑ Sprint Retrospectives
- ❑ 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

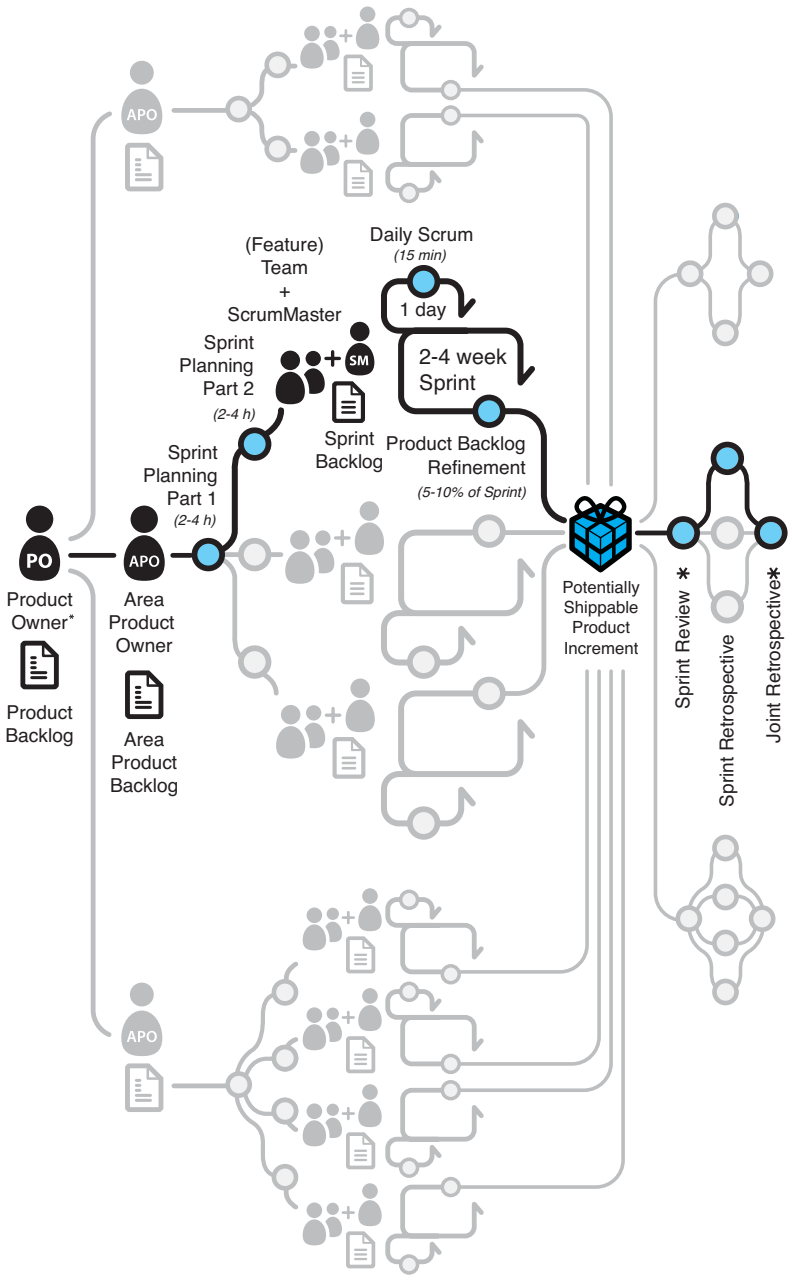
Definition of Done (DoD)—The DoD applies to all Product Backlog items for all teams. See *Planning* for DoD and *Undone Work* tips.

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.

Figure 2.2 large-scale Scrum FW-2



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.

see Feature Teams Primer chapter

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

Book

Chapter 3

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 TPI[1] 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:

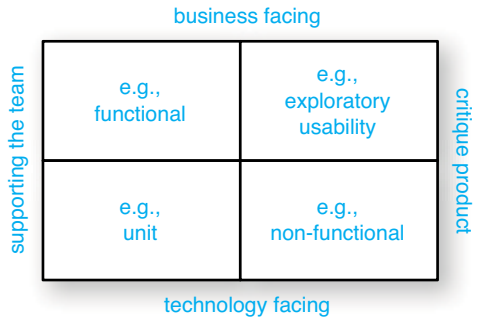
> *You can know the name of a bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird… So let's look at the bird and see what it's doing—that's what counts. I learned very early the difference between* ***knowing the name of something*** *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, Poppendieck06]; [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 book[5] 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* [Hetzel73] 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 of 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 model[6] 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 ISTQB[8] 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 writer[9] 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 its 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 appli-

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

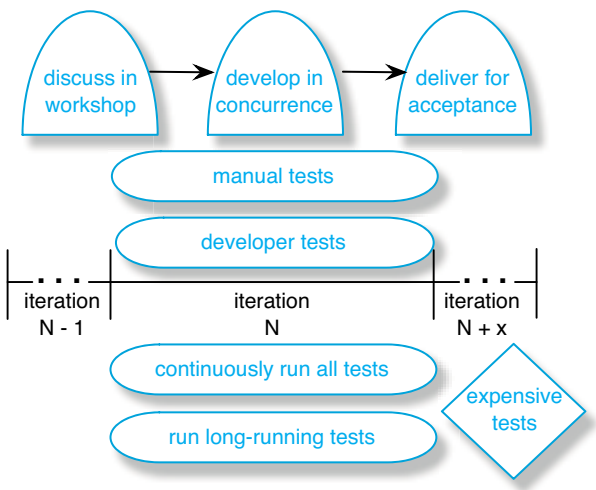
We once coached at a company building a commercial "automated testing" tool—a GUI testing tool. The requested coaching? To learn how to do automated testing for developing their automated testing tool…

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.

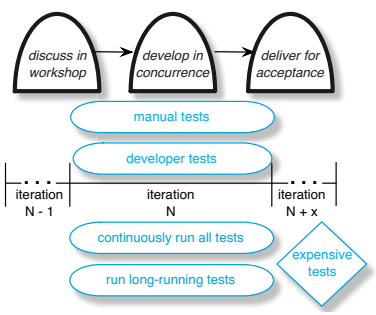
Before release

- 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 workshops" on p. 240.

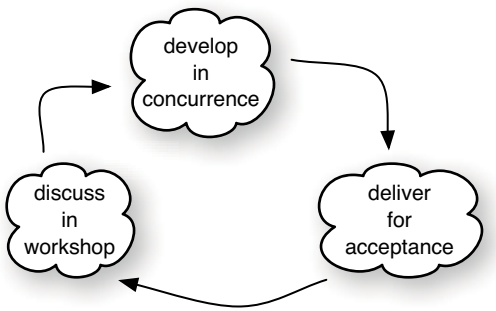
Concurrent engineering—The authors of *Concurrent Engineering Effectiveness* define concurrent engineering as follows: "*There is a tight link between participants in the product development process, such that they can perform much of their work at about the same time*" [FL97]. The main driver of concurrent 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*, Singeo 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 workshop[14]. 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.

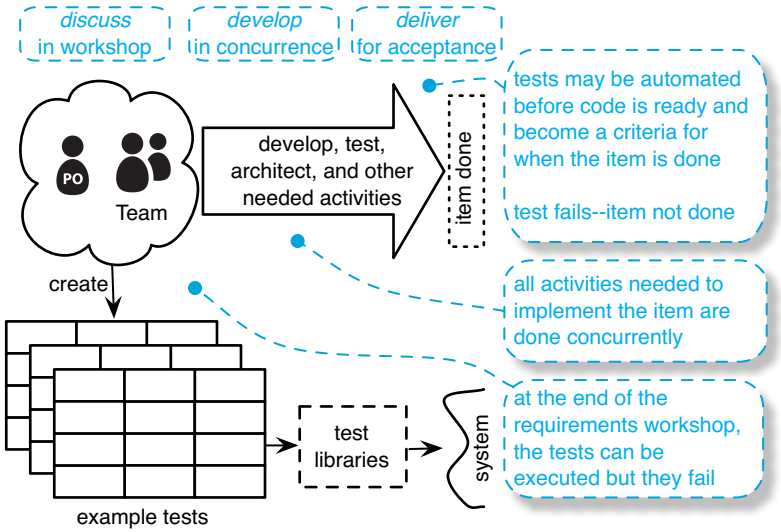
See "Try…Product-level Definition of Done" on p. 170.

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.

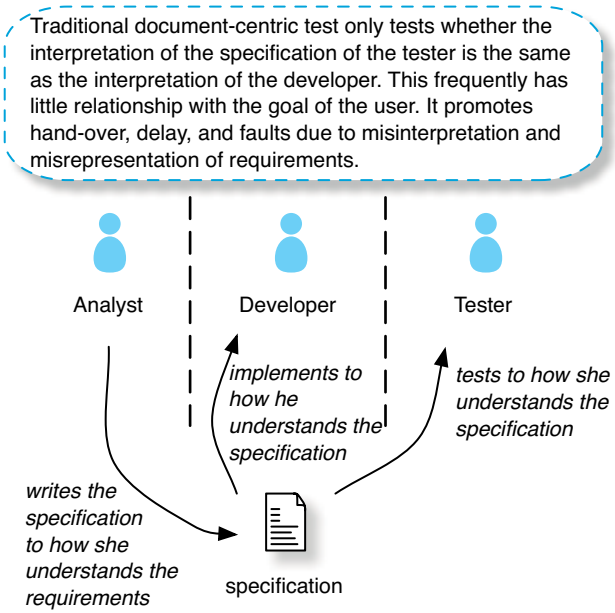


Figure 3.5 conventional document-centric style of requirements clarification

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 unalike. Don't confuse them.