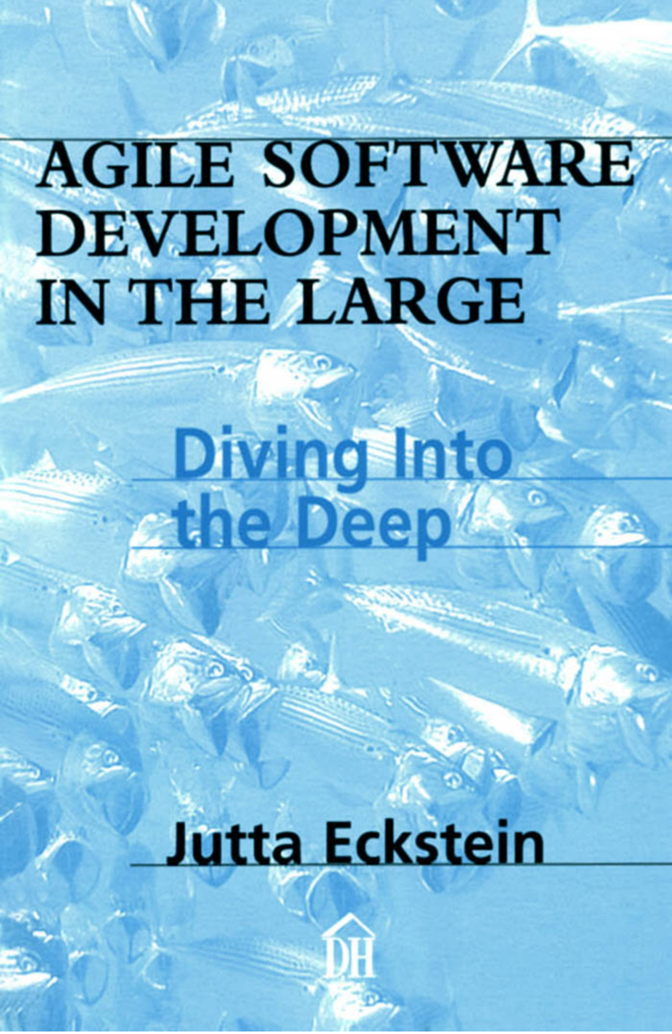


AGILE SOFTWARE DEVELOPMENT IN THE LARGE

Diving Into the Deep

Jutta Eckstein

AGILE SOFTWARE DEVELOPMENT IN THE LARGE

DH *Also Available from Dorset House Publishing*

Adaptive Software Development: A Collaborative Approach to Managing Complex Systems
by James A. Highsmith III foreword by Ken Orr
ISBN: 0-932633-40-4 Copyright ©2000 392 pages, softcover

The Deadline: A Novel About Project Management
by Tom DeMarco
ISBN: 0-932633-39-0 Copyright ©1997 320 pages, softcover

Endgame: Mastering the Final Stage of Software Development
by Robert Galen
ISBN: 0-932633-62-5 Copyright ©2004 288 pages, softcover

Five Core Metrics: The Intelligence Behind Successful Software Management
by Lawrence H. Putnam and Ware Myers
ISBN: 0-932633-55-2 Copyright ©2003 328 pages, softcover

Hiring Technical People: The Artful Science of Getting the Right Person for the Job
by Johanna Rothman foreword by Gerald M. Weinberg
ISBN: 0-932633-59-5 Copyright ©2004 416 pages, softcover

Peopleware: Productive Projects and Teams, 2nd ed.
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-43-9 Copyright ©1999 264 pages, softcover

Project Retrospectives: A Handbook for Team Reviews
by Norman L. Kerth foreword by Gerald M. Weinberg
ISBN: 0-932633-44-7 Copyright ©2001 288 pages, softcover

Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency
by Tom DeMarco
ISBN: 0-932633-61-7 Copyright ©2001 240 pages, hardcover

Waltzing with Bears: Managing Risk on Software Projects
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-60-9 Copyright ©2003 208 pages, softcover

For More Information

✔ Contact us for prices, shipping options, availability, and more.

✔ Sign up for *DHQ: The Dorset House Quarterly* in print or PDF.

✔ Send e-mail to subscribe to *e-DHQ*, our e-mail newsletter.

✔ Visit Dorsethouse.com for excerpts, reviews, downloads, and more.

DORSET HOUSE PUBLISHING
An Independent Publisher of Books on
Systems and Software Development and Management. Since 1984.
353 West 12th Street New York, NY 10014 USA
1-800-DH-BOOKS 1-800-342-6657
212-620-4053 fax: 212-727-1044
info@dorsethouse.com www.dorsethouse.com

AGILE SOFTWARE DEVELOPMENT IN THE LARGE

Diving Into the Deep

Jutta Eckstein

DORSET HOUSE PUBLISHING
353 WEST 12TH STREET
NEW YORK, NEW YORK 10014

Library of Congress Cataloging-in-Publication Data

Eckstein, Jutta.
Agile software development in the large : diving into the deep / Jutta Eckstein.
p. cm.
Includes bibliographical references and index.
ISBN 0-932633-57-9
1. Computer software--Development. I. Title.
QA76.76.D47E28 2004
005.3--dc22
2004010164

Front Cover Photograph: Stefan Krahe
Cover Design & Interior Illustrations: Katja Gloggengiesser, www.gloggengiesser.com

Distributed in the English language in Singapore, the Philippines, and Southeast Asia by Alkem Company (S) Pte. Ltd., Singapore; in the English language in India, Bangladesh, Sri Lanka, Nepal, and Mauritius by Prism Books Pvt., Ltd., Bangalore, India; and in the English language in Japan by Toppan Co., Ltd., Tokyo, Japan.

Printed in the United States of America

Library of Congress Catalog Number: 2004010164

ISBN: 0-932633-57-9 12 11 10 9 8 7 6 5 4 3 2 1

Digital release by Pearson Education, Inc., June, 2013

ACKNOWLEDGMENTS

A mountain is not in the need of a mountain,
But the human is in the need of a human.
—Basque proverb

A book like this is always the result of many years of development. That's why it is so difficult to thank all the people who supported its creation. I would like to apologize in advance to all those people I may have accidentally forgotten.

First, though, I would like to thank Frank Westphal, who suggested that I accompany him to a remote island to concentrate on writing (Frank was writing his book on test-driven development). My sister, Eva Eckstein—a librarian—gave us the idea of hiding on the island of Hiddensee in the Baltic Sea, where many German writers—including several winners of the Nobel Prize in Literature (Thomas Mann, Gerhard Hauptmann, Günter Grass, among others)—created their marvelous works. Thus, we sat on Hiddensee and waited for the same thing to happen to us that happened to one of Günter Grass's protagonists.[1]

> As soon as he arrived at Hiddensee, he got the irresistible urge to . . . write.

[1] Günter Grass, *Ein weites Feld* (Göttingen, Germany: Steidl Verlag, 1995), p. 333.

And, in fact, it worked! However, once most of the text was written, I needed the help of many other people to finally make a book out of it.

First, I would like to thank my family. Preceding all others is my partner, Nicolai M. Josuttis, who gave his moral support as well as his thoughts in uncountable discussions that helped shape the book into what you have in your hands right now. Next, my cousin Katja Gloggengiesser, whose illustrations give the book that personal touch. Finally, someone I consider a member of my family, my long-time flatmate Monika Bobzien, for the endless discussions on our balcony. Again and again, she flagged me down to broaden my horizons.

Furthermore, I would like to thank the following:

The authors of the expert boxes, who enriched the book with their personal experiences: Alistair Cockburn, David Hussman, Diana Larsen, Dierk König, Joshua Kerievsky, Nicolai M. Josuttis, and Stefan Roock. All the reviewers, who helped to shape the book with their comments, some brief and some extensive: Daniel Schweizer, Dave Thomas from Pragmatic Programmers, Dierk König, Eberhard Wolff, Frank Maurer (who wasn't afraid to use the very first draft of the book as course material at the University of Toronto), James Noble, Jon Kern, Ken Schwaber, Martin Müller-Rohde, Mike Cohn, Robert Wenner, Stefan Roock, and Vanita Shroff.

Jens Coldewey, whose wonderful birthday present helped me to find the appropriate introductory quotes and proverbs for the individual chapters.

Stefan Krahe, one of my dive instructors, who took the magnificent picture of the Indian mackerels for the front cover.

And not least, a special thank you to the team at Dorset House: I would like to point out Vincent Au, who made the book readable, and David McClintock, for his belief in the project right from the beginning.

Finally, I would like to thank all of those who accompanied me on my journey through different projects, conferences, training, and workshops, and who shared with me the experience of learning.

CONTENTS

Preface xiii

1 Introduction 3

Questioning Scaling Agile Processes 4
Examining Largeness 5
Raising Large Issues 7
Specifying the Projects in Focus 8
Detecting the Agile Method for Scaling 9
Identifying the Reader 10
Revealing the Structure of the Book 11

2 Agility and Largeness 12

Fundamentals of Agile Processes 13
  The Agile Manifesto 15
  Agile Methods with Respect to Largeness 18
Magnifying the Agile Principles 19
Processes Masquerading As Agile 21
  Almost Extreme Programming, by Alistair Cockburn 22
People Shape the Process 24
  Culture of Change 24
  Adaptation 28
  Communication 30
Mistrust in Applicability 31
  Documentation 32
  Design 35
  Testing 36

Refactoring 39
Summary 41

3 Agility and Large Teams 42

People 43
Responsibility 45
Respect and Acceptance 48
Trust 50
Team Building 51
Building Teams and Subteams 52
Requirements Channels, by Stefan Roock 55
Team Roles 57
Team Jelling 58
Interaction and Communication Structures 61
Open-Plan Office 63
Open-Plan Offices, by Nicolai M. Josuttis 63
Flexible Workplace 65
Encouraging Communication 66
Communication Team 67
Trouble-Shooting 69
Virtual Teams 72
Distributed Teams 73
Distributed Teams, by David Hussman 73
Open Source 77
Open Source, by Dierk König 78
Summary 83

4 Agility and the Process 84

Defining the Objectives 85
Providing Feedback 86
Short Development Cycles, Iterations, and Time-Boxing 87
Planning 92
Result-Oriented Planning 93
Planning Tools 96
Integration 97
Integration Strategy 99
Integration Team 99
Tools for Configuration Management and Version Control 101
Retrospectives 102
Attendance 103
Techniques 104
Learning to Become Great, by Joshua Kerievsky and Diana Larsen 109
Getting Started with an Agile Process 111
Learn from History 112
Start Small 114

Finalizing the Architecture 115
Grow Slowly 117
Culture of Change 118
Learning and Change Processes 118
Introducing Change 120
Force Courage 122
Summary 123

5 Agility and Technology 125

Architect and Architecture 126
Architectural Lead 127
Simple Architecture 129
Architecture As a Service 131
Avoid Bottlenecks 132
Architecture and Largeness, by Nicolai M. Josuttis 134
Ownership 139
Choosing Technology 142
Techniques and Good Practices 144
Testing 145
Refactoring 148
Standards 150
Summary 151

6 Agility and the Company 153

Communication and Organization Structure 154
Project Planning and Controlling 156
Planning 156
Controlling 157
Fixed-Price Projects 158
Enterprise-Wide Processes 160
Process and Methodology Department 160
Formation of a Process 164
Certification and Adaptation of a Process 164
Enterprise-Wide Tools and Technology 166
Quality Assurance and Quality Control 169
Departments on the Edge 171
Human Resources 171
Legal Department 172
Marketing 172
Production 173
The Customer 174
The Role of the Customer 175
Integrating the Customer 176
Company Culture Shapes Individuals 177
Skills 178

Providing Training 179
Establishing a Learning Environment 182
Resources 185
Insourcing 185
Outsourcing 187
Full-Time and Part-Time Project Members 190
Summary 191

7 Putting It All Together: A Project Report 193

The Previous History 193
The Customer 194
The Team 195
Organizational Departments 196
Process and Methodology 196
Tools and Technology 198
Quality Control and Assurance 199
Project Planning and Controlling 200
Starting Off 201
Growing the Team 204
Learning from Previous Problems 205
Training 208
Establishing Short Iterations 210
Learning to Reflect 211
Enabling Communication 212
Managing Outsourced Teams 214
Unresolved Issues 215
Summary 216

Afterword 219

References 223

Index 229

PREFACE

A book is always a prevented dialogue.
—Hans Urs von Balthasar

Looking back on all my years in software development, I discover something extraordinary: Besides the technological progress, revolutions, and unfulfilled promises, the occupational image has changed. In the past, I was a software developer, solving local technical problems. Now, my job is to make sure that the outcome of a whole software project makes sense. The reason for this change is that too many projects are failing, and this failure isn't caused by technology, but by social, structural, and organizational deficits.

It is all made worse by tool providers who promise heaven and earth, by clients who create the illusion of unrealistic schedules, and by market pressure that loads significant risks on every large project.

However, when I compare the projects I'm working on, only trivialities come to mind. But those trivialities actually hold the key to project success: Instead of making inappropriate decisions, I place value in common sense and create an environment that enables constructive communication.

However, the times are changing. It is just not possible anymore to finance failed large projects. Those who want to stay in business have to deliver fast and timely solutions. Agile processes help developers concentrate on the essentials. Unfortunately, there are only a few real-life examples of how agile processes can support large projects.

This book is intended to close that gap, drawing on the essence of my experience in large agile projects. The book mirrors my discovery that the social aspects of development always outweigh the technical ones. This is why I sometimes call myself a communication manager (although my technical background has proven necessary for being taken seriously not only by management but also by developers).

I hope you have fun reading the book. I invite you to visit the book at my Website, http://www.jeckstein.com/agilebook, and to contact me at agilebook@jeckstein.com.

April 2004 J.E.
Munich, Germany

AGILE SOFTWARE DEVELOPMENT IN THE LARGE

This page intentionally left blank

3

AGILITY AND LARGE TEAMS

Trust is the sister of responsibility.
—Asian proverb

The reasons for implementing a system with a large team are varied. The most common one is that the scope of the project is too large for a small team to handle. However, some large projects would be better off if implemented by a small team. Even if the scope is large, a small team may be faster or more effective, mainly because communication is not as likely to prove a problem as it is in a large team.

Sometimes, the use of a large team is politically motivated. The size of a team may signify the importance of the project and of the project management itself. Author and consultant Tom DeMarco discussed this problem during OOPSLA 2001.[1] He indicated that surprisingly often, the manager of a failed but large project will be valued higher than the manager of a successful but small project.

Furthermore, the project may be shaped and sized to suit a team that is already established. For instance, I witnessed a situation in an organization where a lot of people just sat around, waiting for the project to start. Nobody questioned if this mass of people was really required for the project. Instead, everybody tried to shape the project in a way that kept all these people busy. Granted,

[1] OOPSLA is an ACM SIGPLAN conference: Object-Oriented Programming, Systems, Languages, and Applications.

for some companies (in some countries), it might be easier to shape the project according to the team's size than to get rid of the employees—mainly because of legal issues—but this is not usually the case.

It is always worth questioning the reasons for working with a large team, but this is neither the topic of the book in general nor of this chapter in particular. Instead, the assumption is that the project will be run by a large team and you want to use an agile process. When changing to agile development with a *large* team, you have to deal with several issues involving people, teams, interactions, and communication structures.

This chapter focuses on those aspects of agile processes that work differently in large teams than in smaller teams. First, we look at the people aspect. We discuss how taking up responsibility can work in a large team and what kind of consequences respect, acceptance, and trust have for successful collaboration. Next, we consider how a large team can be divided into subteams and what kind of team roles have to be occupied. In the section on interaction and communication structures, we focus on encouraging communication in large teams. Then, in the section on trouble-shooting, I present typical team problems and their possible solutions. Finally, we look at the difficulties that can occur when developing with dispersed teams.

People

Size matters. The size of a team provides a special risk—a team that is too large can hinder a project. One reason is that the quality of the decision-making typically suffers. For example, the larger the team, the more often you will find that decisions are unclear or postponed. The main reason for this is that within large teams, you will often find a tendency among people to shun responsibility. Because there are so many people on the team, there is a collective mentality that "someone else will decide."

Unclear or postponed decisions confuse the team and make it difficult for team members to decide which direction to take. This leads either to project paralysis, because nobody has the courage to move on without being told, or to a lot of individual decisions as one thinks best. Often, those individual decisions contradict each other, which in turn leads to a form of project stagnation, based on contradictory development. Both symptoms are very frustrating for the whole team. I once consulted on a restart of a failed project. I interviewed team members about what trap they believed would most likely ensnare the restart. Interestingly enough, most people named a lack of clear decisions as the highest risk.

Therefore, although it might seem unusual, it is preferable to make a clear but eventually wrong decision and to correct it later. Making a wrong decision enables you to learn; postponing a decision does not. If you postpone a decision, you do not know until it has been made whether it is the right or the wrong one. However, if you make the wrong decision, you will learn from the consequences and will have the possibility of correcting your mistake, based on your new experience.

Making decisions is one side of the coin; the other is making sure that they are not only communicated to everybody involved but are also carried out. A decision that is made but not carried out is essentially the same as a postponed decision.

Although this all sounds very obvious, it is common to find the same problems popping up over and over again, which is a sure sign that those decisions either have never been clearly made or have not been realized.

As I mentioned earlier, the main reason for the poor quality of decisions on projects with large teams is probably based on an aversion to taking responsibility. You will find that the more people there are, the harder it is to tell who took responsibility for which task. Often, this results in an *undefined task zone,* which is defined by

- **Multiplicated task responsibility:** A lot of people are responsible for the same task. The problem is that they do not know about one another. Therefore, if you are lucky, this task will be carried out repeatedly. If you are unfortunate, they will do the task in ways that contradict each other.
- **Null task responsibility:** Nobody takes responsibility for the task. Everybody assumes that it is someone else's job. This can result in everybody blaming everybody else for not taking the responsibility.

To make things worse, you can be assured that with each additional team member, the risk will rise and more of such problems will arise.

Responsibility

Due to the departmental organization, people in large companies are not usually used to having complete responsibility for any particular task. This is because there is almost always somebody higher up the hierarchy who has *ultimate* responsibility. This is especially true for developers. They often see themselves as only doing what somebody else tells them. When somebody "accidentally" gives them the responsibility for a specific task, they feel uncertain. They are not used to having responsibility, and they do not know what it implies.

On the other hand, agile processes require everybody to be responsible for his or her task, and for the effects that task might have on the whole project. In addition to individual tasks, there is also the shared responsibility for the ultimate performance of the whole system, the project, and even the process of development. Thus, each team member is responsible in some way for every task, even those assigned to other team members.

For example, Extreme Programming has a practice called *collective ownership,* which refers to a shared responsibility for all kinds of things: the code, the integration, the process, and so on. Best known among these shared responsibilities is probably collective code ownership, which enables and obliges everybody on the team to improve every piece of code, no matter whether he or she is the original author of the code or not.

With collective ownership, every team member bears the same responsibility for all aspects of the project. However, allowing everybody to steer the project at the same time is a challenge and, some fear, a big burden. For instance, every developer would want to have a hand in shaping his or her development environment. At the same time, this increased responsibility is likely to increase the developers' fear of making the wrong decisions.

When people first sign up for a task but aren't used to the responsibility it entails, you have to lead them gently into this new territory. For example, ask the developers which task they want to be responsible for, and then assist them in estimating the task. Not only should you make yourself available to answer any questions they may have, it is very important that you also ask them regularly if they are doing okay, or if they need any help, because they might be afraid to bring up such issues themselves.

For example, I remember one project I was working on, where people had problems taking responsibility. I visited all the team members regularly and asked them how they were getting along with their tasks. It did not take long before some of them started complaining that they were not able to get their work done, for various reasons. The most common reply was that they were waiting for something from another team: either the other team had not yet provided some interfaces, or the interface it had provided turned out to be different than expected.

The obvious problem was that these people did not have the right mindset for problem-solving. Instead, they complained that their peers were responsible for the problems. The real, hidden problem was that they were not taking enough responsibility. If they had, they would not have complained, but rather would have started solving their problems. In other words, they might have started talking to this other team, found out why the interfaces were not ready, and addressed the situation.

The typical reaction of people not used to responsibility is to get annoyed at the situation without taking any action to change it. Of course, it could be worse. If, for instance, they could neither complain nor take up the responsibility, you would never learn about their problems.

Therefore, you have to be proactive in asking developers about the status of their assigned task. Only then will you have an idea of any problems they may have. I'm not talking about status reports—I'm talking about walking up to the people and talking face-to-face about their current situation. You should encourage them to look at the big picture and regard their assigned task as part of the whole. Explain that even tasks that may only be partly related to their assigned task (if at all) are important for the completion of the project.

If people are spoon-fed responsibility, they will not learn to make an effort to take it up themselves. Or, as an article in *Fast Company* put it,

> Telling people what to do doesn't guarantee that they will learn enough to think for themselves in the future. Instead, it may mean that they'll depend on you or their superiors even more and that they will stop taking chances, stop innovating, stop learning.[2]

Telling people what to do is not enough. They have to commit themselves to their task. The focal point of this philosophy is that the value of team productivity is much more important than the individual effort. Therefore, every now and then, you have to point out that only the team's success is the individual's success. An individual's success without the success of the team is of no value. Among other things, this means that a well-functioning team does not rely on its official manager—it takes up the responsibility itself, whenever the situation requires it. For this approach to become a reality, the organization has to change from management by command and control to management by responsibility, trust, and teamwork.

Trust is the foundation on which such a management strategy is built. When someone takes on a responsibility, you trust that he or she is capable of handling that responsibility. However, at the start of an organization's first agile project, this culture of trust and responsibility will not be in place yet. Most team members will not be able to take up responsibility, because they are not used to it. However, I suggest that you demonstrate to them how you take up responsibility, and that you encourage them to take responsibility even if they do not feel ready. This shows your team members that you trust them, even though at this early stage in agile development they might not be able to justify your trust. When you refrain from giving them any responsibility, you prevent them from ever getting the chance to learn how to take up responsibility. That simply reinforces their own mistrust in their capabilities. Just as Ulrich Sollmann and Roderich Heinze say, you should give people the chance to learn how to deal with responsibility:

[2] Chuck Salter, "Attention Class!!! 16 Ways to Be a Smarter Teacher," *Fast Company*, Issue 53 (December 2001), p. 114.

> The more often you are in an uncertain situation, the better you can handle this kind of situation, or rather the longer it will take till you will again feel uncertain.[3]

If you want to train your team members to take up responsibility, you have to be aware that this is an investment in their future. This "training" is two-sided: You may also have to train leaders to delegate responsibility and to trust their team members. As with every other learning process, it will be some time before you see results, but it is worth the effort.

Respect and Acceptance

A development team is not usually organized like a team, in the strictest sense of the word—assembled by peers with equal rights—it's hierarchical. The typical hierarchy in a development team, found mainly in traditionally led projects, follows Taylor's theory about centralizing a team's knowledge.[4] Individual team members take up specific roles and corresponding tasks. Analysts, designers, developers, and testers often work independently in a linear process.

As a consequence of this separation of tasks and roles, a hierarchy is created. Although perhaps not officially sanctioned, the hierarchy is formed by the different roles in the team, some of which have greater prestige, importance, or acceptance level than others. Often, the acceptance level is defined by the linear development. This means that analysts have the highest acceptance level, while coders, testers, and, even worse, maintainers are at the very end of the acceptance-level chain, doing all the dirty work. This sequence of acceptance levels is just one example, but an oft-encountered one.

The major problem is that nobody wants to be at the low end of this acceptance-level chain. Therefore (as in the example above),

[3] Ulrich Sollmann and Roderich Heinze, *Visionsmanagement: Erfolg als vorausgedachtes Ergebnis* (*Vision Management: Success as the Predefined Result*) (Zürich: Orell Füssli, 1994), p. 32.

[4] Taylorism is characterized by the division of labor, repetitive operations, extreme labor discipline, and the supervision of work.

everybody tries to climb up the ladder from maintainer to designer or, even better, analyst. From another perspective, you will find the largest percentage of novices in maintenance or implementation. Consequently, there are often too few experienced coders on a team.

In contrast, most agile processes require teams to have shared knowledge and shared skills. This means knowledge cannot serve to form a hierarchy. Therefore, the first step in forming an agile team is to get rid of the Tayloristic split. Assemble teams that cover all the knowledge, where each member of the team is aware of the big picture and takes responsibility to contribute to the whole team's success. The individual role of each member is not so obvious, then, in terms of individual knowledge, but is recast in terms of contribution to the team's success. So, acceptance is then based on performance and not on roles.

One of the main differences between small and large agile teams is that in the former, every individual is typically requested to be a generalist. On the other hand, as I discuss later in this chapter, in a large agile team, a whole *subteam* and not necessarily every individual team member should cover this general knowledge.

This implies that agile teams require more generalists than specialists. At the least, everybody should be able and willing to understand the big picture and not become solely interested in digging into some specific details while ignoring the interests of the whole project.

So, as programmer and software expert Don Wells said, you will find that in an agile project,

> Everyone is of equal value to the project.[5]

But this is only true if every team member bears responsibility for the whole project. Of course, each team member will have individual capabilities and abilities, but now he or she will contribute equally to the team and to the project.

[5] Don Wells, "Transitioning to XP or Fanciful Opinions of Don Wells," (International Conference on eXtreme Programming and Agile Processes in Software-Engineering 2001, Sardinia, Italy, 2001).

Trust

It is natural for people to be skeptical of a change like switching to an agile process. The team members themselves, along with a lot of people only partially involved in the project, might not have trust in the success of this new process. The possibility that the team can change the process over time is often even more frightening than following a defined but indigestible recipe.

The best argument against this mistrust is working software. Therefore, try to complete the first, low-functional version of the software as early as possible. Another strategy for building trust is transparency. Make everything transparent for everybody involved in the project.

Different practices help to make things more transparent:

- **Shared ownership:** Ask everybody on the team to take responsibility for all kinds of things (for instance, the code or the process). This shows your trust in them.
- **Shared knowledge:** This practice is often based on shared ownership. The knowledge about the information—for example, the system—is transferred from one team member to another. This makes the system more transparent and understandable for everybody, and helps in turn to build confidence in the system.
- **Shared skills:** The team has people with a variety of backgrounds and skills. This knowledge is accessible not only for the individuals, but for the whole team. Using nonagile processes, the individual guards expert knowledge from the whole team. Making knowledge transparent makes the team more trustworthy. Furthermore, it allows every team member to add new skills to their repertoire.

It is important that this transparency is always open and honest. Do not hide any negative information. Knowing about the bad things makes it easier to deal with them. Moreover, everybody should be invited to comment on the information and to help improve the situation. Thus, transparency includes representatives from control, audit, and, most importantly, the customer.

Occasionally, when coaching a project, I find that project members assume that transparency stops right before the customer. For example, I sometimes have to lead long discussions in order to open the project's wiki Web for the customer, because the customer will then have full access to the project.[6] Often, when asked for more transparency, project managers tell me they're afraid the customer will find out about the problems inside the project. This is exactly the point! The customer should always be aware of the problems, because the customer is paying for the project. These arguments are typical when discussing the impact of having the customer on-site. As soon as the customer becomes something of an unofficial project member, the fear disappears from both sides: from the team's side, because team members realize that the customer is a real person, and from the customer's side, because he or she understands the difficulties the project members are facing.

This reminds me of how I was before I started scuba diving: I liked swimming in the open sea, but I was always a bit afraid of the creatures underneath me, and I was pretty sure that sooner or later one of them would bite me. As soon as I started scuba diving, I did not even fear sharks or other predators. Being close to these creatures gave me the feeling of actually being a part of the living sea.

Team Building

A large team is hardly manageable as a whole. Thus, in order to establish a flexible team, the team is usually divided into subteams of no more than ten members.

The typical structure used by large teams (and in large companies) is still based on Taylor's theory of building teams according to their knowledge, as I mentioned earlier. Therefore, you will often find an analysis team, a design team, a test team, and so on. The developers are typically further subgrouped into smaller subteams, each responsible for a specific function like presentation, database, network services, and the like. This Tayloristic split is also known as *horizontal* team division. Taylorism works quite well for jobs that are repeatable. It doesn't work as well if a lot of cre-

[6] Originally developed by Ward Cunningham, a wiki Web is a Web-based collaboration platform that allows interactive communication and vivid documentation by editable HTML pages. See Bo Leuf and Ward Cunningham, *The Wiki Way: Collaboration and Sharing on the Internet* (Reading, Mass.: Addison-Wesley, 2001).

ative and holistic thinking is required. You can furthermore consider defining *vertical* teams, which are focused around business functionality. These teams are also known as domain or feature teams, as Peter Coad terms them in the Feature Driven Development process. On the other hand, if you are dividing the team vertically, you might find that not every team has all the necessary skills, or even worse, that every team might start to address the same problems.

Therefore, do not make this an *either-or* decision, but an *as-well-as* one. For example, if you start with a small team and build slowly, you will come to the conclusion that on future projects, your starting team should be staffed with people who have good domain knowledge and a major technical background. This starting team most often defines the first architecture and verifies that the system can actually be built. Furthermore, it can serve as a model for the formation of the other teams. The horizontal and more technically focused teams should then support these new (vertical) subteams.

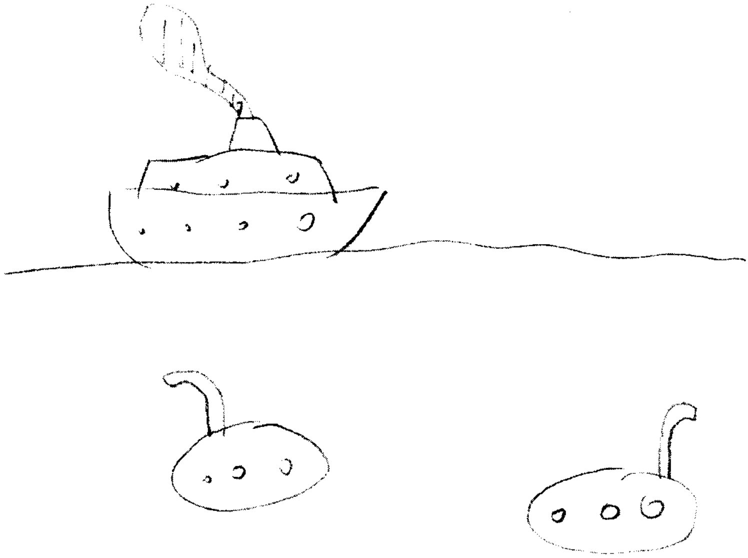
Building Teams and Subteams

As mentioned earlier, dividing the whole team into several subteams should not be a decision between vertical *or* horizontal divisions. Instead, it should be an *as-well-as* decision, to provide a better mix of knowledge in the teams.

Either virtual or real technical service teams could be installed to further support those vertical, domain teams.[7] For example, on one of my projects, we defined domain teams focusing on a specific domain area in banking, with one team focusing on accounting and another one on customer management. Each team had the knowledge needed to implement the features belonging to its domain, including the graphical user interface, the connection to the host, the business logic, and all the other required technology. If, for instance, the accounting team required some functionality from the customer management team in order to implement a feature, the

[7] In some ways, a virtual team is not recognizable as a team. The team members may not be co-located, communicating only by electronic media, or the team members may in fact belong to different teams and just get together every now and then for work on a specific task.

accounting team would just bilaterally discuss the requirements with the customer management team. The customer management team then in turn would provide the required service within the development cycle.



Subteams ...

We established in this case real (not virtual) technical service teams that were responsible for supporting the domain teams by providing some base functionality. For example, we assembled an architecture team responsible for the business logic, and a presentation team for all graphical user interface aspects. Those technical service teams were requested to visit all the domain teams regularly. On request, members of a technical service team supported domain teams as regular team members for a specific amount of time.

Technical service teams should always regard themselves as pure service providers for the domain teams. For instance, the technical service team responsible for building and supporting the architecture should always shape the architecture according to the requests of the domain teams, not vice versa, since the domain teams have to use whatever the architecture team creates, as is often the case.

Depending on the actual size of your team, you will establish either virtual technical service teams or real technical service teams. The members of the virtual teams are usually regular members of domain teams. In contrast, members of real teams usually lack a close connection to the domain teams. For this reason, you have to ensure that real teams do not develop the *best* architecture, but the most adequate. You have to avoid features that are implemented just because somebody *believes* they are needed. Technical teams have to think of themselves as service teams, delivering services to *their* customers, the domain teams. The big advantage of this strategy is that the architecture only contains what is required. This makes the architecture much easier to maintain and, as a side effect, cheaper. Additionally, it eliminates the oft-occurring social discrepancies between the technical and domain teams. One often gets the impression that those teams are working on different projects (not least from the way they talk about one another). Unfortunately, this impression is seldom wrong, and those teams have different objectives. Where technical teams' objective is to make use of a specific technology and develop perfect frameworks not requested by the domain teams, the domain teams' goal is to implement the domain, not caring if they can profit and learn from one another (or from the frameworks the technical teams provide).

But how do the technical service teams know which service is required and, more importantly, which requested service has the highest priority? The team has to come up with a strategy. Not every requirement from each and every domain team will be implemented, because certain requirements might contradict each other. Or, worse, implementing these requirements will cost so many resources that other teams will not be able to get their (more important) requirements done.

Therefore, like real customers, the domain teams have to speak with one voice. Retrospectives can serve as a forum for deciding on new or changed requirements since all teams are present (or at least represented) and the focus of the retrospective is the project's status and progress, anyway.[8] If one team states that it cannot proceed because it needs some special technical service, all teams can decide jointly if this is a requirement they support; if approved, the

[8] A retrospective is a reflective group meeting that is held at the end of a development cycle (see Chapter 4).

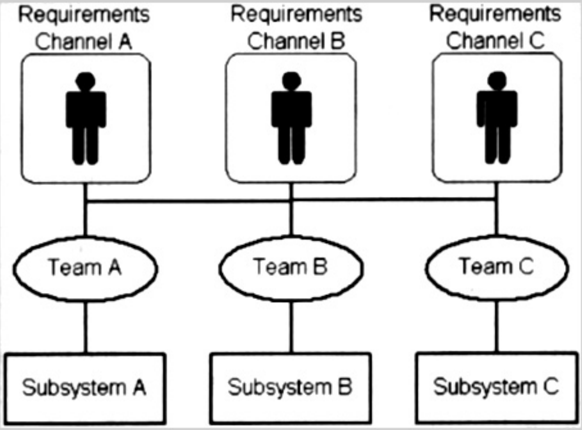
service will be a joint requirement for the technical service team. Otherwise, the requesting domain team has to implement the service on its own. These requirements are then scheduled in the same way the domain teams schedule their requirements. Thus, the technical service team schedules requirements with the highest priority first and does not schedule more than it can accomplish within the next development cycle. It might have to negotiate workload with the domain teams. At the beginning of the project, especially, the domain teams define many requirements for the technical service team, but at other times, there may be few requests, if, for example, the architecture can just be used as is. During "high season," you should ensure that the technical service team does not accept more work than it can accomplish. During "low season," you should ask the members of the technical service team to join the domain teams instead of implementing unnecessary additional features.

Requirements Channels, by Stefan Roock

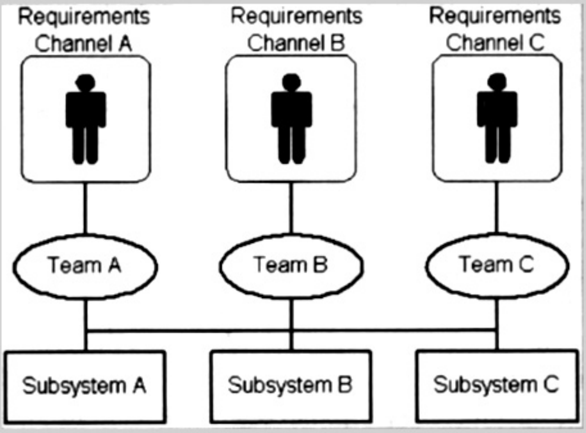
In this project, we had to implement a system supporting multiple channels for different user groups, with various front-end technologies (desktop, Web, laptop). Our starting project team consisted of five people from the development company and two consultants. With seven people, it was a size typical for an agile project. We had all the Extreme Programming practices in place when the project had to scale up and accept additional manpower—mainly developers. The goal was to have about twenty-five people in the project.

When scaling up, we had to address the issue of project structure. It became clear that it would not be possible to integrate all these people in one large team in the project. Therefore, we decided to split the project up into teams. But, we asked ourselves, what are the criteria for the division of teams? Do we use the architecture as the structuring mechanism and assign each subsystem to a team? Or do we assign each requirements channel to a team? In the first case, each requirements channel had to talk with every team. In the second case, each team had to modify classes

all over the system. Since the planning games seemed to be too complex in the former, we chose the latter.[9]



One team per subsystem ...



One team per requirements channel ...

We then got three teams for the three requirements channels, and a technology support team. The teams were

[9] The planning game is an Extreme Programming technique. The customers select and prioritize the tasks for the developers for the next development cycle and the developers estimate the effort for those tasks.

rather small (four people), which supported taking over responsibility.

One thing we learned was that reorganizing teams takes more time than we thought. When we changed the organizational structure, the developers needed several weeks to get used to the new structure and get up to their development speed again.

Because teams were not assigned to subsystems, every developer was able to modify every part of the system. This was no problem because the developers were able to master the code base (about twelve-hundred classes).

As time went by, additional developers joined the project and the code base grew. We ended up with about thirty developers with different programming skills. Now some developers weren't able to modify every part of the system without it breaking. Our first step was to tag core classes and the very complicated parts of the system as "expert code," which had to be modified by a so-called system expert.

That solved the problem, but it didn't seem to be a very smart solution since there was no way to guarantee that only system experts modify the crucial part of the system.

Currently, we are searching for better mechanisms for assigning code to teams. The main idea is to take the layering of subsystems into account. Some subsystems are specifically for a requirements channel and should be assigned to the relevant team. Other subsystems are relevant to several user groups and can't be assigned to one of the existing teams. These subsystems are assigned to a virtual "base subsystem" team, which is created on demand from the system experts sitting in the existing teams.

Team Roles

The idea is that a team must have members that possess all the required knowledge. In that sense, each team is a generalist in its domain. For instance, a domain team will be assembled by domain experts, graphical user interface developers, and database developers. But although the team consists of these different experts, those experts will not work solely in their field of speciality. Instead, the team members must take different roles. For example, it is rather typical for the database developer to learn from the graphical user interface specialist how to build the presentation, and to then contribute to the user interface development. Thus, the goal of having generalists rather than specialists on a team is attainable by spreading the available knowledge.

The goal of this approach is not egalitarianism of all team members. Distinct skills and experiences are still necessary for specific tasks. However, the goal is to avoid the general tendency toward thought monopolies and to spread knowledge and skills.

Additionally, each agile team also possesses the required administrative knowledge necessary to perform, for example, integration and configuration management. The person who takes this role concentrates mainly on issues based on internal team integration and configuration, but will also be this function's contact person for people external to the team. However, individual team members may have multiple roles: For instance, the person responsible for integration and configuration may be the domain expert, too.

It is very helpful to establish a team lead for every team. This person acts as a contact person for the whole team. Often, the team lead coordinates who will attend a specific meeting, such as a retrospective.

Team Jelling

Ideally, the whole project team pulls together, all team members communicate honestly and openly, and everybody has the same big picture in mind. As Tom DeMarco puts it, the team *jells*.[10] The pulling together must especially be supported so it becomes natural. In addition to the more formal aspects of project development, other, more enjoyable and motivational, tactics must be employed to keep your project on track:

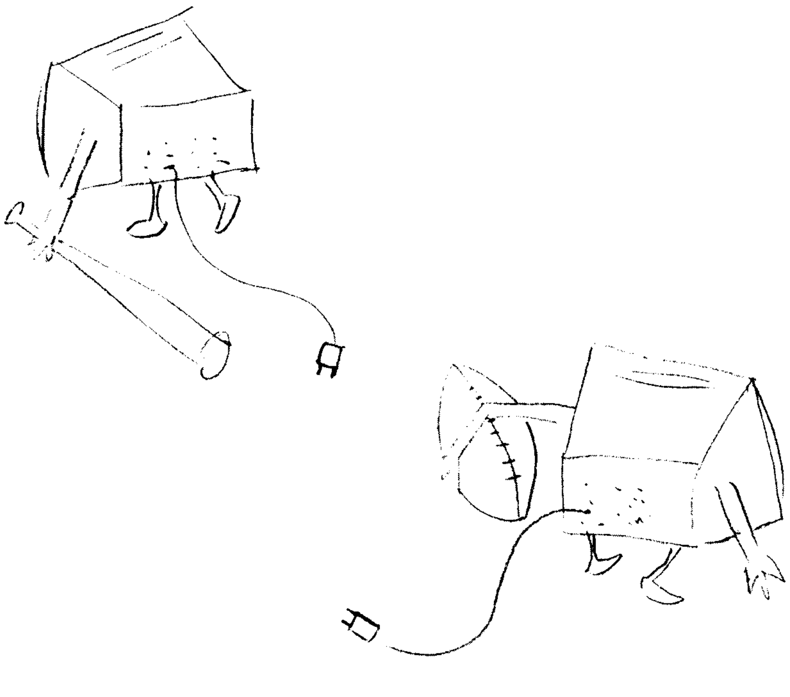
[10] A team jells when it has a good chemistry, comparable to the chemistry good jelly has. For more on this subject, see Tom DeMarco, *The Deadline: A Novel About Project Management* (New York: Dorset House Publishing, 1997).

- **Food:** If you provide food, or just snacks—healthy or otherwise—the area where you place the food will soon become an extremely popular part of the office. And when groups of people are there, taking advantage of the free food, they will start talking. You might also want to make use of team lunches, although you should ensure that lunch time is also a break time that allows the team members to relax and recover from their work. On the other hand, breaking bread together always helps people get closer to one another.
- **Party:** Organize a party once in a while—after the delivery of a major release, for example. This does not have to be something big. It would be enough to serve some sandwiches and beverages for a couple of hours or so. This will help people who wouldn't otherwise have the chance to sit and talk to each other. Try to convince the company of the importance of such project parties, so it will approve them.
- **Recreation:** Organize some sort of recreational outing. It can be a sporting event, such as a volleyball match, or some other social event, such as bowling, go-cart racing, or something along those lines. Doing something as a group will help team members get to know each other, especially when people are asked to team up with someone they do not work with regularly. This will hopefully reinforce respect and acceptance among all. Ensure that everybody can participate in the event, taking into account team members' disabilities, for example.
- **Project identity:** Encourage the team members to cultivate a sense of project identity. Authors Mary Lynn Manns and Linda Rising stress the importance of having a group identity in *Fear Less: Patterns for Introducing New Ideas into Organizations.* They recommend a separate pattern, called *Group Identity.*[11] Special T-shirts, project-specific food and beverages, or even project-specific phrases and slogans help to develop a project culture. On one project, we even came up with a project cocktail. However, the project should not

[11] Mary Lynn Manns and Linda Rising, *Fear Less: Patterns for Introducing New Ideas into Organizations* (Boston: Addison-Wesley, 2004).

alienate itself from the outside; the group identity should help newcomers to identify themselves with the project.

- **Regeneration:** Ensure that project members have time to regenerate. Even when people are under pressure to deliver, make sure that they take their vacations and are not working overtime. A project is comparable to a marathon, not to a sprint.



Regeneration ...

- **Communication of results:** You cannot overestimate how motivational it is for teams to have reports on the growth of the system or the customer's feedback. Therefore, make sure everybody knows about the project's progress.

All the strategies suggested (just a sample of the possibilities) reinforce communication and will ensure that your team members will get to know each other better and, more importantly, learn to respect one another. Try to ensure that members from different subteams interact with each other. For example, if you organize a sporting activity, you can request that each side contain no more than two people from the same subteam. It is astonishing how much this contributes to a sense of communal identity among team members, and this usually results in projects that run more smoothly.

Some strategies are not readily implemented in certain companies. For instance, organizing a party that needs temporal and financial support could be a problem. This is a sure sign that the importance of communication is still underestimated. You will need to convince the organization otherwise. It is worth the effort.

Interaction and Communication Structures

Communication is the most important factor in the success or failure of the whole project. Communication is difficult, even when only a few people are involved, but it gets exponentially more challenging as the number of people involved increases. When setting up a communication structure for a large team, you have to consider the following constraints:

- **Direct communication:** This the safest form of communication, and you know immediately if the receiver of your message understood what you said. However, the more people involved in a communication effort, the harder it is to get a message across. One reason for this is that there will not be enough time for everybody to actively participate in the conversation. Another reason is that participation may be dominated by a few extroverts, whereas all the introverts will accept the message out of discomfort with discussing anything in big groups.
- **Different sensory modalities:** Every person obtains information differently. Some people, known as visuals, learn most effectively by watching; auditories, by listening; and kinesthetics, through action.
- **Overdose on communication media:** There must be a law stating that as soon as a communication path works, it will be abused until it doesn't work anymore. For example, if

messages are exchanged via e-mail, you will read your e-mails and respond to them. However, once your inbox begins to overflow with new e-mails when you get to work each morning, either you will be very selective about which messages you read and respond to or you will ignore them all. This, of course, is bound to eventually result in your getting in trouble for not reading an e-mail that the sender assumed you read.

Changing communication channels ...

Therefore, you should also be agile and flexible with communication. Use various modes of communication that address different persons differently, respecting their unique sensory modalities. Change the communication channels from time to time. That said, though, a manageable, average-sized agile project will always require direct communication.

Open-Plan Office

Ideally, the whole team sits in one room together with the customer. As Craig Larman writes in *Applying UML and Patterns,*

> Having a team on another floor of the same building has as much impact as if it were in a completely separate geographical location.[12]

However, in a large project with a team of a hundred or more members, space constraints make it difficult to have everyone in one office. Open-plan offices are valuable in both creating space and enhancing communication. They can be created by removing cubicles or by positioning the cubicles around teams rather than individuals. Open-plan offices can sometimes accommodate forty to fifty people. So, if you can have two or three such offices next to each other, project members will be sitting in quarters that are as close as possible.

Open-Plan Offices, by Nicolai M. Josuttis

At the start of my professional life, open-plan offices had a bad reputation. They represented the idea of treating human beings like machines, which can be located close together in a big hall to save money on walls. And in fact, in a work environment that assigns each employee a stupid and almost communication-free task, there is a lot to be said against putting large groups of people together in a huge room, like in a laying battery or factory floor—especially if phone calls annoy one's neighbors, and one has to fight tediously for each square meter of space.

After my first large agile project, however, I began to look at open-plan offices from a different perspective. The situation changes tremendously when the job focuses on

[12] Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design* (Englewood Cliffs, N.J.: Prentice Hall, 1998), p. 448.

teamwork that enables several people to create something together. All of a sudden, moving to another room is painful. All of a sudden, it is important to know what your colleagues are working on. All of a sudden, it is important to work *together,* collaboratively and physically. The value of this kind of communication can't be overestimated.

However, this does not mean that it is best for an agile project to pack all project members together in a dreary open-plan office. It is important to be able to have a meeting without disturbing others. Adequately soundproofed meeting rooms are a necessity. Also, individual workplaces are important for people who need a quiet place to think, design, or make a phone call. Glass works well for this purpose: Vitreous meeting rooms, individual vitreous workplaces, or vitreous walls between teams allow the necessary transparency without raising the noise to a degree that disables serious work.

In a sense, an agile workplace is an intelligent mix of everything, which is again typical for agility in general.

Always ensure that the individual subteams can sit together, even inside an open-plan office. Although this might seem like common sense, it is not as common a situation as it should be. Again, whatever your constraints, the distance between team members has a major influence on the success or failure of your project. Be aware that this distance does not necessarily have to be physical. For instance, if certain team members listen to music through headphones while they work, the headphones establish a distance between them and their peers. The least you can do is to make it possible for all members of the team to be on the same floor or, at the very least, in the same building. But everything that improves the seating situation pays off during development time.

Some people argue that the noise levels are too high in open-plan offices. This is not usually an issue. Mainly because everybody is concentrating too intensely on his or her work to be disturbed by the conversations of others. However, you may have some individuals in the team with particularly loud voices. In that case, you should ask them to lower their voice. If this is not possible, you should consider locating said individuals to a place where they will not disturb their peers. But this is a highly unlikely situation. As I said earlier, in my experience, the noise level on projects is almost always acceptable, and the advantages of the close proximity of team members far outweigh those of having a quiet environment.

Flexible Workplace

Nowadays, some companies do not support assigned office space. Instead, they use a system known as *flexible workplace* (also known as floating desks or desk-sharing), where people just sit wherever they find some space. Team members either use cell phones or have calls transferred to wherever they are sitting on a particular day. Typically, filing cabinets are mobile, so team members can have all their papers with them at all times. The underlying idea of the flexible workplace is that it requires less space than a more traditional, assigned-space system. The logic being that, on any given day, certain employees will not be in the office: People call in sick, take vacations, make on-site visits to customers, and so on. Utilizing flexible workplaces, then, is a very efficient way to use office space and to save money on workplaces.

However, the catch is that you will never know, for sure, where to find a specific person, and this is a communication problem. Another problem is that at certain times (during the less popular vacation months, for example), some people may spend their day wandering around looking for an empty space to sit.

Remarkably, this problem sometimes becomes known outside of flexible workplaces. One time, in a taxi I was taking from the airport to a customer's office, the driver asked if she should speed up to make sure I would have a place to sit at the office. (It turned out not to be necessary, since we had plenty of time to spare; it was just eight o'clock in the morning.)

Another risk that teams utilizing flexible workplaces face is that people may get to work late and not be able to sit with the other members of their team. At that point, flexible workplaces are not very beneficial.

If you cannot avoid a flexible workplace arrangement, try to establish an acceptable working environment, for example by defining (flexible) team zones within the constraints of your office's seating arrangement. Be aware that in your attempts to do so, you might get in trouble with the "office police." In such a case, you have to fight this out, because as mentioned earlier, the importance of efficient communication cannot be valued high enough.

Flexible workplaces create an infrastructure that allows people and teams to solve communication problems by relocating easily. However, if your company has the philosophy that every associate should have the same desk over many years, you might discover unbelievable resistance when transitioning to flexible workplaces to better support the project.

Encouraging Communication

The real difficulty of working with a large team is looking for ways to ensure efficient communication. I have found that the following steps are valuable in setting up a communication structure:

- All project members should sit as close together as possible without crowding each other.
- The retrospectives performed after each iteration and release cycle serve as a forum for direct communication. Typically, the topics of optimizing the space and improving direct communication for the daily work will be discussed regularly until they are resolved.
- Regularly scheduled meetings for *all* project members are essential. Such meetings are primarily a mode of information transfer. In my experience, too many people attend these meetings for there to be any effective feedback or extensive discussions, but they work well for one-directional information transfer. Therefore, every project member should have the possibility to contribute—in the form of a lecture about a specific topic, for example. It is a good idea to announce the contents of the contributions in advance.
- Provide a *wiki* on the intranet, not only as a means for documentation, but as a means for communication.[13]

[13] The term "wiki" is Hawaiian for "quick," which in this context represents the ability to make quick changes. For more information on wikis, see Leuf and Cunningham, op. cit.

The philosophy of a wiki is to allow all kinds of discussion on the Web. Everybody has the right to make changes to the Websites. This is possible through editable HTML pages. The wiki Web only knows collective ownership, so everybody has the same responsibility for the contents. This helps to establish a community of trust. Furthermore, no deep knowledge of HTML is required to contribute to the wiki Web. You can even contribute by writing plain text. If the wiki Web is also used to document the project, you can be sure that this will always be a good source of project documentation.

- Establish different e-mail distribution lists that allow you to address everyone involved in the project, as well as specific groups of people.

Communication Team

Be warned, however, that even making use of these different channels will not eliminate your communication problems. Another very effective way of improving your team's communication is to establish a separate (virtual) communication team. Depending on the size of your team, the communication team could consist of just one person. The communication team is responsible for visiting all the teams regularly, obtaining feedback, and discovering deficiencies and (potential) problems, possibly solving them immediately. It is important for the team to proactively approach the project members. You will recognize problems sooner this way than by waiting until they escalate or are reported officially. Typical topics and tasks of the communication team are

- **Unified project culture:** The goal is to establish a common culture regarding guidelines, tests, patterns, and the like.
- **Refactoring:** Uncovering sources for refactoring not only improves the quality of the code but also provides learning opportunities for everybody.
- **Common understanding:** The communication team needs to ensure that all information, decisions, and announcements are understood by all the teams.
- **Problem discovery and treatment:** Problems should be detected and, at best, solved immediately and in a simple

> manner. The communication team has the advantage of having an overview of all the teams. This way, the communication team can establish contact or point to solutions other teams might have found. If several teams have the same problems, general strategies are required for solving these problems (extending/adapting the framework, or providing patterns for the solution). Furthermore, the communication team suggests ways that the process could help overcome or eliminate the encountered problems.

The members of the communication team should never act as supervisors or controllers, but instead more like a team of ombudsmen. These ombudsmen should be sensitive to the hopes and fears of the individual team members and should collect suggestions for process improvements. For example, ensuring that the team members understand the decisions enables them to either accept the decisions or to suggest a solution that supports them better.

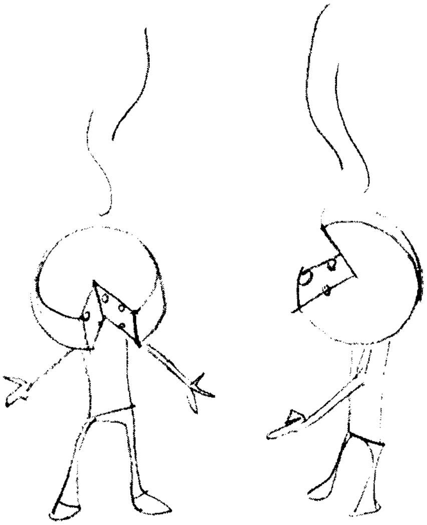
It is very important that the members of the communication team have a good overview, are well-trusted people with good communication skills, and are widely accepted and respected by the rest of the project team. These people should be able to take matters into their own hands, able to manage the project as a whole, but also have good connections to the individual persons. In smaller teams, the communication team will consist of just one person, with tasks that cross boundaries—running reviews, retrospectives, coaching the process, and so on. In larger teams (with more than fifty people), this will always be a full-time job for one or even more persons.

You will rarely find a project organization that is aware of the necessity of this role. This makes it difficult to establish this position. I often call these people communication managers or, simply, *catalysts*. Ideally, as Tom DeMarco and Timothy Lister write, these are people whose mere presence is enough to ensure that a project runs smoothly.[14]

[14] Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams*, 2nd ed. (New York: Dorset House Publishing, 1999).

Trouble-Shooting

Sometimes, you need to act quickly—for example, if one of the teams is completely under stress, one team stops talking to another team, or two teams start continuously blaming one another and are not able to work together anymore. In such situations, you face two difficult tasks: One is to look at the problem and see exactly what kind it is, and the other is to solve the problem.



Smells ...

The first task is more difficult, because it depends a lot on the team's culture. Here are some typical problem signs:

- **Cynicism and sarcasm:** Humor is a sign that everything is right on track and that people are having fun doing their jobs. But if the humor turns into sarcasm, this is a clear sign that the team does not jell and does not believe in what it is doing.
- **Blame:** This sign is much more obvious and therefore easier to tackle. The teams or people blaming each other usually have problems respecting and understanding each other. Sometimes, though, blame can be a sign of difficulty in communicating.

- **Lack of feedback:** This is often a sign that the people have given up. They do not believe in reaching the goal and they do not believe that anybody has an interest in their opinion or in their effort.

Whatever the reason is, you can neither accept nor ignore the situation. All these circumstances will slow down the project's progress significantly. Therefore,

- If a team is under stress and complains that it cannot get its work done because there are too many meetings or its time is spent supporting other teams, protect the team for a couple of hours each day by arranging *quiet times.* It might be necessary to arrange an office-wide quiet time, either temporarily or permanently. For more on quiet times, see Alistair Cockburn's *Agile Software Development,* in which the author suggests defining the period between 10 A.M. and 12 P.M. as quiet time, during which no phone calls or meetings are allowed.[15]

  If instituting quiet time is not sufficient to bring the team back on track, a more rigorous approach is required: Instead of quiet hours, make sure the team will get one or two quiet weeks, with one or two hours of each day as "regular office hours," so that team members can still process incoming requests.

  The most extreme solution is to send the team to a closed meeting for a couple of days. In addition to being extreme, this solution is the most effective and probably the most expensive. Closed meetings are often used in other circumstances: for example, if the team does not jell or has to consider different kinds of solutions. They are most often used as an environment for the project kick-off (for making teams jell) and for the project postmortems.[16]

  Quiet times have a trade-off: They can also lead to a complete lack of communication and should therefore be carefully balanced.

[15] Alistair Cockburn, *Agile Software Development* (Reading, Mass.: Addison-Wesley, 2002).

[16] For more on postmortems, see Norman L. Kerth, *Project Retrospectives: A Handbook for Team Reviews* (New York: Dorset House Publishing, 2001).

- If two teams stop talking or working together efficiently, locate them next to each other. This way, each team will recognize why the other acts as it does, and they will start to respect one another.

  Another strategy is to set up a voluntary exchange program among teams, so that each member switches place with a member of another team.[17]

  Both strategies help to improve the understanding between the teams.
- If a meeting culture evolves where people have to spend more time in meetings than they do working, and if people start complaining about unnecessary meetings, challenge the reason for holding each of the established meetings, especially all regular meetings. Furthermore, you should determine which participants are not required to attend in order for the meeting to be a success.

  Generally, you should introduce the "law of the two feet," as described by the Open Space Technology: Anybody who feels that the meeting is a waste of time is allowed to leave. This might require some sensitivity from the organizer of the meeting: If a participant does not contribute, he or she should be politely invited, outside the meeting, to contribute to the project's success.

  Introducing quiet times is another approach to overcoming the meeting culture.
- Finally, you can at the beginning of each meeting ask one of the participants to excuse himself or herself from the meeting to do something more important (this was first suggested by Tom DeMarco, in *The Deadline*).[18] Take care that a different person is excused each time, and is not the most junior person.
- If a team is not very well integrated—for example, if it is often not well-informed or is often blamed by other teams for incidents that stemmed from a lack of information—then locate food in the team's area. Normally, it is only a matter of hours before other teams find themselves in the

[17] Thanks to Mike Cohn for sharing this approach.

[18] DeMarco, op. cit.

food area and the communication or information flow is reestablished naturally.

Virtual Teams

According to researchers in the area of work life, virtual teams will become the norm in the future. Fewer and fewer real teams will physically come together to work on a project, and more and more teams will be assembled over the Internet. This has a lot of advantages:

- Each individual project member is responsible for his or her own work space and environment. Although sometimes the client will provide the equipment, most often the individual project member will have to use his or her own hardware. This saves the client a lot of money.
- You have much better access to different skills. You are not limited to people from your region or your company.
- You are not responsible for the team in the long run. You only have to pay it for as long as it works for you. You are under no obligation to find its members their next job.
- If the team is distributed all over the world, another advantage is that at any hour, at least one team member is most likely working on the project. There is hardly any project off-time.

The main problem with virtual teams is that they lack the most efficient mode of communication—direct communication. As Erran Carmel and Ritu Agarwal write,

> Distance negatively affects communication, which in turn reduces coordination effectiveness.[19]

In virtual teams, the problem isn't just communication with the customer; it's also communication inside the team. It's very difficult for a virtual team to get a common understanding and to pull together.

[19] Erran Carmel and Ritu Agarwal, "Tactical Approaches for Alleviating Distance in Global Software Development," *IEEE Software*, Vol. 18, No. 2 (March/April, 2001), p. 23.

Distributed Teams

Large teams are always distributed in one way or another, just because they are too large to contain in one room. However, distributed development is somewhat more extreme, in that the project members are distributed over several sites and, as the term suggests, the project itself is developed in a dispersed manner. Sometimes, you might find a single team spread over several sites; other times, several teams are each located at a different site. Outsourcing is one example of a distributed team, as I discuss in Chapter 6.

One problem in this setting is ensuring that everybody on the team pulls together. On projects like this, you will often find that people blame one another, mainly because they do not know each other and therefore do not trust each other. Also, technical topics like version and configuration management are even more complicated in distributed teams. Of course, there are tools that can help manage these more complex areas of development, but they do not make up for the inconvenience and problems caused by distributed development.

If you must have distributed teams, the Internet is likely to be your main form of communication (e-mail, wiki Web, chat rooms), and video conferencing is also a good way to communicate. However, be sure that people working out of different locations are able to meet with each other, at least occasionally. Communicating through the Internet will only work efficiently if people know and trust each other—and there is no better way of building trust than through personal contact.

> **Distributed Teams, by David Hussman**
>
> Most agile practices ask project members to keep communication channels open and filled with honest dialogue, without regard to the message content. Nowhere is this more important than on an agile project with distributed teams. Along with the usual technical challenges of distributed development, agile development brings even more challenges, mostly aimed at those outside the development teams.

Project managers, coaches, and customers need to be vigilant when it comes to listening to and addressing the developer's concerns; tracking successes and failures; the way in which story content is gathered, organized, and presented; and the consistency of the process and the development environments.[20] With distributed teams, the need to embrace change and make the necessary corrections to the direction of the project sooner than later is even greater. Just as the last car of a long train starts moving long after the first, so too does change take longer to move through the distributed teams.

The following list of best and worst practices might help those outside the development group find and address needed changes before the project strays too far from the correct path. Many of the listed items apply to any scaled, agile project, but their importance is heightened when the development teams are distributed:

Best Practices

- **Customer agility:** Ensure that customer teams can make any necessary changes by keeping the ratio of customers to developers as low as possible (one customer to three or fewer developers; the larger the project, the smaller the ratio). The ability of the customer team to react and change direction can be difficult when stories cross team boundaries and teams struggle to bring portions of a software solution to fruition. For example, if five people are writing a book together, with each person working on a different chapter, the lucidity and cohesiveness of the

[20] Extreme Programming uses the term "stories" for requirements, which are defined by the customer for a release cycle. User stories are comparable to use-cases in UML.

book is proportional to the amount of time the authors spend discussing the book with each other.

- **Group speak:** The more often that project managers, coaches, and trackers discuss planning and development issues, the better. Although this discussion can take place over e-mail, at a wiki site, and through non-verbal channels, ensure that at some point there are conference calls or video conferences. If possible, have a different team lead the discussion on every call (this helps all involved to embrace the project). Also, as teams grow and change, each group starts to form its own view of the project picture. Sharing your team's lingo with other teams can aid in maintaining a common view and provide cross-team insight.
- **Common acceptance:** If possible, ensure that each team is building and testing against a common set of hardware and software (if this is not possible, discuss the environmental differences regularly). The closer the environments, the better the level of acceptance testing. Try utilizing a common acceptance testing strategy, and whenever possible share the data sets used for testing. Try to start automated acceptance testing early, and run the tests as often as possible.
- **Developer rotation:** Often times, teams in the same building feel as disconnected from each other as teams separated by states or countries. If possible, move players from team to team. If the players cannot co-locate, explore the notion of rotating the work done by the teams instead of the team members themselves. This may seem like an unrealistic suggestion, but it may be a

way to help maintain a healthy project whose members have a holistic perspective.

Worst Practices

- **Technical stratification:** It is often a natural fit for distributed teams to work on subjects that they are familiar with (this is often why distributed teams are brought into a project). If at all possible, avoid splitting development tasks across technical boundaries. Try instead to plan and develop toward functional goals and subdivide the work from that perspective (this will avoid the classic producer-consumer relationship between teams, where one team has finished all its work and is left with nothing to do).
- **Failure to communicate mock implementations:** As teams may not have everything that they need in order to complete a particular feature, they may choose to mock or stub-out temporary solutions.[21] This is fine (and, if everyone is using the same code base and acceptance tests, it can be quite clear), but make sure that the interfaces to the mocks or stubs are agreed upon by all.
- **Loss of iteration synchronization:** In most cases, it is best to keep all teams on similar iteration boundaries. It may be that the teams (and more importantly, those in charge of the planning) find a steady state with varied iteration boundaries. In either case, ensure that the iteration synchronization is as constant as possible. Again, as with scaled, agile projects, if a team has an iteration schedule slip, even of just a couple

[21] Mock and stub objects allow you to test partial solutions by simulating the missing code.

of days, planning issues may arise that are best dealt with by moving the incomplete features to the next iteration. The difference may seem small, but the consistency will benefit the project plan.

Conclusion

Distributed teams using agile practices face many of the same problems that subteams face when agile projects scale to large numbers. The need for vigilant and constant communication is exacerbated. Small issues can quickly grow large or span several iterations if not known to all team members. One team's frustration can affect other teams without it even knowing it. Strive to keep process, schedules, and environments constant, and when they change (as we know they will), notify all involved as soon as possible. The more that can be shared between teams (source code, data, and so on), the easier this task will be. Ensure that planners listen to the developers, and help them listen to one another.

Open Source

Open source projects are well-functioning examples of virtual teams. Possible reasons for their success are

- All the team members are very idealistic. There is no need to motivate them or try to ensure that they identify themselves with the project. This all comes naturally.
- Everybody feels responsible for the whole project and takes this responsibility very seriously.
- There is a broad community that provides immediate feedback. This feedback is what drives the whole project. There is no difference in the value of feedback whether it comes from peers or from users.
- Everybody who contributes to the project takes pride in his or her work.

The main underlying principle of open source projects, and the reason for their success, is the gift economy (the culture of giving away capacity and information). This means that everybody working on an open source project is doing so voluntarily. A lot can be learned from this approach, especially for use on commercial projects.

Mary Poppendieck once reported that a new project manager asked her for advice on becoming a successful team leader. She asked him if he had ever led a team of volunteers (of any sort). He replied that he had been a successful choir leader. Poppendieck continues,

> I suggested that if he used the same techniques with his project team that he did with his choir, he would be a successful project manager. He said it was the best advice he ever received, and he blossomed as a project manager.[22]

Open Source, by Dierk König

The open source movement derives its name from the practice of sharing the source code of a valuable product among an arbitrary number of developers. This so-called collective code ownership means that the code belongs to all these developers. They are entitled to change it and are responsible for the final result.

In the context of this book, open source projects may be of interest because they share properties of both large and agile projects. They make use of agile practices while suffering from the same problems that large projects have in regard to team distribution.

I'm not so bold as to claim that open source produces better results in general. Sure, lots of open source products are widely known for having excellent quality with zero costs to the user. But I'm the first to admit that there are also numerous sloppy projects out there that will never produce anything useful. However, maybe we can take something from the successful ones!

[22] Mary Poppendieck, personal e-mail correspondence on the gift economy.

Distribution

It is evident that the physical co-location of all of the contributors to an open source project is impossible. However, we have observed that members of the core editing team of an open source project sometimes get together to tackle a special issue, often sacrificing personal time and money. The collaboration then looks like an Extreme Programming pair programming session.[23]

Even the users of open source software hold events so they can get together and share their experience. The Eclipse Code Camp is one example.[24]

If the open source people put so much effort into overcoming the obstacle of distribution, can we—in paid time—go upstairs to pair up with the database guy?

Idiosyncrasies

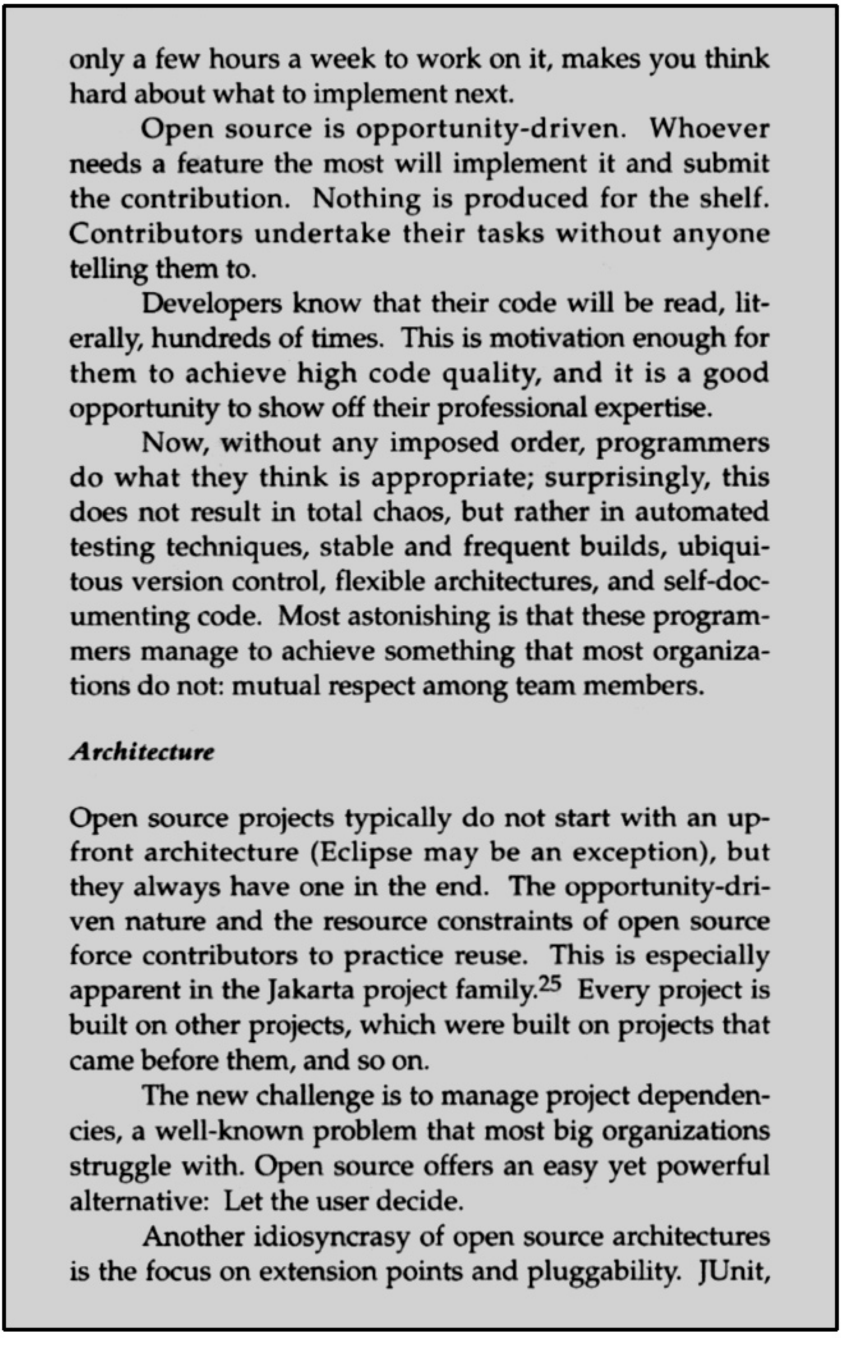
An open source contributor is not forced into anything. If you do not like anything about the project (the setup, the code style, the technology, the people, and so on), you can leave at any time or even *fork* (make a new project based on the old one).

This, and the fact that a lot of people write contributions in their spare time, leads to a project staff that likes the applied work style, or at the very least accepts it. The number of complaints is noticeably small.

One could claim that open source projects are not limited in time, scope, or resources, and therefore do not need the measures of control that are applied to in-house projects. This is not really true. Running an open source project in your spare time, knowing you have

[23] Extreme Programming requires developers to write productive code in pairs. Thus, a continuous review takes place throughout the development by two developers working together.

[24] Developers of the Eclipse development environment meet occasionally in so-called code camps to exchange experiences and develop the product further (see http://www.eclipse.org/).



only a few hours a week to work on it, makes you think hard about what to implement next.

Open source is opportunity-driven. Whoever needs a feature the most will implement it and submit the contribution. Nothing is produced for the shelf. Contributors undertake their tasks without anyone telling them to.

Developers know that their code will be read, literally, hundreds of times. This is motivation enough for them to achieve high code quality, and it is a good opportunity to show off their professional expertise.

Now, without any imposed order, programmers do what they think is appropriate; surprisingly, this does not result in total chaos, but rather in automated testing techniques, stable and frequent builds, ubiquitous version control, flexible architectures, and self-documenting code. Most astonishing is that these programmers manage to achieve something that most organizations do not: mutual respect among team members.

Architecture

Open source projects typically do not start with an upfront architecture (Eclipse may be an exception), but they always have one in the end. The opportunity-driven nature and the resource constraints of open source force contributors to practice reuse. This is especially apparent in the Jakarta project family.[25] Every project is built on other projects, which were built on projects that came before them, and so on.

The new challenge is to manage project dependencies, a well-known problem that most big organizations struggle with. Open source offers an easy yet powerful alternative: Let the user decide.

Another idiosyncrasy of open source architectures is the focus on extension points and pluggability. JUnit,

[25] For Jakarta, see http://jakarta.apache.org/.

ANT, and Eclipse are perfect examples of this approach.[26]

Just think of the effect that applying the principles of open source software would have on your corporate IT projects.

Project Structure

In open source projects, you typically see a core team of editors with write access to the repository. The requirements for becoming an editor differ on every project. Some do not have any restrictions, while others only grant write access to contributors that consistently submit quality work. Some projects have a fixed group of editors.

The core team decides whether contributions from the outside get incorporated or rejected.

Communication channels are highly self-organized. The flow of information typically takes place on mailing lists.

Project Setup

For open source projects, having a self-contained build is absolutely crucial. It is no wonder that open source projects were the pioneers of build automation. The same holds true for the use of versioning systems, nightly builds, and automated self-testing.

As the source code is highly visible, a new degree of rigor is applied to the end result. The source code is subject to excruciating review and refinement. Its compliance to every standard in use will be checked for all platforms the community uses. The contributor has total control over how to achieve this result. Assessing results rigorously but giving developers freedom to use their own work style is a strong agile move that large projects can follow as well.

[26] For JUnit, see http://www.junit.org/; for ANT, see http://www.ant.org/.

Documentation

Typically, in open source software there is not a lot of external documentation. The code must speak for itself. Although it may seem unusual, this strategy works well on open source projects, where the code, especially the test code, must reveal its intention.

Successful projects are often accompanied by articles and even books. The usage documentation that comes with the distribution typically contains only very basic help for the beginner.

It appears that any necessary external documentation gets written on request. For example, in Canoo WebTest, we had a fairly complicated security use-case.[27] A user volunteered to write the documentation for this, provided that someone helped him figure out what to do.

The result seems to be a good balance between the two extremes of lacking all the necessary information and having excessive documentation in which the important information is hidden among pages and pages of unnecessary text.

Planning

Frankly, there is no long-term planning whatsoever. There are exceptions, but for most open source projects, there is only a little short-term planning. The reason for this is to save the developers unnecessary work by not prescribing long-term directions of any greater detail than a common vision. Directions are derived solely from user feedback.

The most challenging part of adopting open source strategies in corporate IT projects may be trusting in the evolution and refinement of a complex adaptive system that is beyond managerial control.

[27] For WebTest, see http://www.webtest.canoo.com/.

Summary

You should never underestimate the value of face-to-face communication, especially with large and distributed teams. You should do everything you can to provide as many opportunities as possible for direct communication. Direct communication, together with transparency, helps to build trust, both inside the team and between the team and the outside world. If team members trust one another, they will not fear taking up responsibility.

A tool like the wiki Web will help you build trust by empowering everybody through shared ownership. In doing so, such tools change the flow of communication from control-driven to collaborative-driven.

When building subteams, you should ensure that the skills within each subteam are mixed. Furthermore, the technical service teams should serve the domain teams. If so desired, they can also be established as virtual teams.

Last but not least, allow the team to have fun. This not only helps teams jell, but more importantly, it also makes the work environment a pleasant one, which will make everybody want to work and succeed with the team. Consider the values of the gift economy the way the open source does; this will also help make work much more enjoyable.

INDEX

Acceptance tests, 37, 39n., 75, 76, 87, 89, 94, 95, 100, 110, 146, 147, 158, 169, 170, 172, 181, 188, 189, 210, 216
Adaptive Software Development, 15, 226
Agarwal, Ritu, 72, 223
Agile Alliance, 15, 17
Agile Manifesto, 13, 15–18, 19–20, 21, 22, 23, 32, 41, 117, 226
Agile methodologies, 9, 10, 12, 32, 36, 38, 220
Agile planning, 93
Agile principles, 10, 12, 19–21, 134
  large teams and, 19
Agile processes, *xi–xii*, 4, 5, 9ff., 21, 22, 23, 28, 34, 35, 41, 43, 45, 49, 84, 86, 87, 88, 92, 94, 95, 102, 103, 111, 112, 118, 119, 122, 124, 126, 161, 162, 164, 166, 169, 171, 176, 188, 190, 196, 210, 214
  adapting, 14–15, 19, 22, 23, 24, 28–30, 103, 124, 160, 163, 164, 165, 193, 214
  applicability of, 12, 31
  establishing, 14, 84, 85, 111, 194
  fundamentals of, 12, 13–15
  identifying, 23–24
  introducing, 28, 29, 112, 118, 119, 139
  large projects and, *xii*, 4, 5, 9, 10, 12, 18–19, 81, 93, 193, 220
  scaling, 4, 5, 8, 9–10, 19, 102, 205, 207, 210
  skepticism about, 12, 50, 200
  switching to, 12, 24, 34, 90, 112, 162, 171, 196, 215
Agile projects, *xii*, 11, 35, 47, 49, 55, 63, 64, 73, 78, 93, 127, 129, 139, 154ff., 163, 169, 191, 192
  scaled, 74, 76, 77
Agile software development, 4, 5, 14, 31, 34, 40, 47, 73, 88, 104, 111, 142, 116, 119, 154, 172, 173–74, 188ff.
  switching to, 41, 43, 120, 162
Agile teams, 17, 49, 58, 88, 97, 139, 140, 157, 162, 190
Agile value system, 8, 9, 10, 11, 17, 41, 220
Agility, 9, 10, 11, 64, 124, 220
Almost Extreme Programming, 22–23, 226
Ambler, Scott, 35, 36n., 224
ANT, 81, 226
Architectural lead, 11, 126, 127–29, 151, 203
Architectural requirements, 115, 131, 211
Architecture, 11, 21, 52, 53, 54, 55, 80–81, 87, 94, 114ff., 125, 126, 129, 131, 132,

134, 138, 150ff., 179, 181, 187, 190, 201, 204, 206ff., 214, 219
 change in, 24, 116, 129, 132, 135, 136, 152, 179, 180, 187, 188, 206, 207, 214
 largeness and, 134–38
 simple, 126, 129–31, 201–2
Architecture team, 53, 131, 132
Artifacts, 16, 20, 23, 31, 36, 41, 87, 139ff., 156

Backward compatibility, 41
Banmen, John, 118n., 226
Beck, Kent, 159, 224
Beckhard, Richard, 118, 224
Blame, 34, 44, 69, 71, 73, 140, 148, 159, 176, 206
Bobzien, Monika, *vi*, 164
Bottlenecks, 97, 98, 126, 132ff., 136, 138ff.
Brooks, Frederick P., Jr., 33, 127, 128, 211, 217, 224
Budget, 4, 7, 104, 157, 173
Bugs, 29, 38, 40, 170
Business value, 13, 41, 92, 93, 94–95, 96, 114, 156, 157, 160, 173, 188
 of refactoring, 148

Canoo WebTest, 82
Capability Maturity Model, 166
Carmel, Erran, 72, 223
Change, 13, 14, 17, 24, 28, 29, 41, 46, 50, 66, 74, 75, 77, 86, 88, 91, 98, 100, 103, 118ff., 126, 132ff., 139, 145, 149, 154, 155, 163, 165, 168, 169, 173, 174, 181, 183, 186, 188, 191, 197, 201, 209, 210, 215, 217, 220
 accepting, 17, 24, 29, 122, 150
 agents, 10, 116, 120, 121, 219
 embracing, 28, 74, 119, 120, 121, 122, 188
 introducing, 24, 29, 91, 120–22, 185, 215
 resistance to, 28, 112, 120, 121, 215, 216
 team size and, 8, 13
Chrysler Comprehensive Compensation, 22, 226
Coaches, 10, 74, 75, 117, 118, 166, 196, 220
Cockburn, Alistair, 9n., 22, 30, 31, 34, 70, 128, 168, 224
Code, 7, 31, 34ff., 50, 57, 67, 76ff., 99, 100, 101, 110, 134, 136, 139, 144, 145, 150, 177, 178, 180, 181, 182, 189, 203, 209, 210
 changes in, 78, 102, 150, 189
 comments and, 22, 23
Cohn, Mike, 71n., 114n., 157, 191n., 223
Collaboration, 16, 17, 43, 51n., 64, 79
Communication, *xi*, *xii*, 5, 7, 8, 10, 24, 29, 30–31, 34, 35, 41ff., 51n., 58, 60ff., 64, 66, 72, 73, 77, 83, 86, 154, 155, 170, 191, 202, 207, 212–14, 219
 direct, 20, 30, 31, 46, 61, 63, 66, 72, 83, 117, 180, 181, 189, 192, 202
 encouraging, 43, 66–67
 flows, 72, 83, 134, 155, 156, 185, 212
 improving, 171, 190, 202
 team size and, 7–8
Configuration management, 58, 73, 101, 102
Contracts, 16, 17, 24, 158, 159, 172, 186ff., 196, 214, 215
 fixed, 159, 172
Conway, Melvin E., 125, 223
Conway's Law, 125, 134, 227
Cooper, Alan, 175, 225
Cost, 6, 28, 30, 39, 88, 94, 116, 136, 158, 187ff., 204
Coupling, 132, 133, 135, 137, 138
Cross-section departments, 153, 154, 155, 161, 163, 164, 191, 196
CruiseControl, 102, 227
Crystal Methodologies, 15, 19, 227
Culture, 12, 25, 26, 41, 69, 85, 103, 120, 177–78, 180, 184, 187, 189, 195
 of failure, 90, 182, 183–84
Cunningham, Ward, 51n., 66n., 225
Customers, *xi*, 3, 5, 6, 13, 16, 17, 20, 24, 30, 32, 33, 37, 38, 41, 51, 54, 56n., 63, 65, 72, 74, 86ff., 92ff., 102, 116, 131, 132, 134, 136, 143, 146ff., 153ff., 163, 167, 169ff., 174–77, 186, 187, 189, 193ff., 198, 207, 208, 217
 collaboration with, 16, 17, 51
 on-site, 51, 174, 176, 177, 195
 representative, 24, 176, 195
cvs, 102, 227

de Geus, Arie, 178, 179n., 225
Deadlines, 85, 86, 92, 94, 158, 159, 169, 187, 211

Decisions, 17, 43, 44, 52, 67, 68, 93, 118, 122, 123, 128, 167, 168, 194, 201, 202, 204, 207, 208, 212, 214
DeGrace, Peter, 225
DeMarco, Tom, 26, 42, 58, 68, 71, 165, 170, 190, 225
Demmer, Christine, 165, 223
Design, 7, 14, 20, 21, 27, 29, 35–36, 38, 64, 95, 114, 125, 127, 136, 138, 145, 148, 149, 182, 188, 207
Developers, *xi*, *xii*, 6, 7, 10, 16, 19, 20, 29, 32ff., 38, 41, 45, 46, 48, 51, 55, 57, 58, 72, 74, 77ff., 85, 89, 95, 98, 100ff., 122, 131, 134, 137, 138, 145ff., 150, 151, 163, 167, 169, 170, 171, 176ff., 181ff., 192, 194ff., 198ff., 208, 211, 212
Development, *xi*, *xii*, 31, 34, 36, 38ff., 44, 45, 57, 58, 64, 73ff., 79n., 86ff., 93ff., 107, 112, 114ff., 123ff., 130, 136, 146ff., 151, 156ff., 176ff., 181, 182, 187ff., 193ff., 199, 203, 207, 209, 214
  incremental, 40, 86, 88, 200
  iterative, 13, 86, 88, 101, 157, 172
  linear, 20, 48, 86, 88, 155, 169, 172, 173–74, 200, 216
  process, 14, 21, 52, 114, 119, 164, 205, 220
  team, 5, 48, 73, 74, 87, 100, 128, 142, 171, 176, 177, 198, 199, 209, 212
Development cycles, 13, 14, 20, 22, 53, 54n., 55, 56n., 85, 87ff., 92, 94, 95, 107, 108, 121, 148, 169, 176, 183, 188, 191, 201, 214, 216
  distributed, 9, 73
  planning, 191, 200, 201, 216
  short, 87–88, 94, 95, 97, 157, 173, 200, 210
Dispersed teams, 43
Documentation, 16, 20, 22, 23, 32–35, 39, 67, 82, 87, 151, 165, 166, 182
DRY-Principle, 181

EasyMock, 147n., 227
Eclipse, 79n., 80, 81, 102, 227
Employees, 20, 26, 43, 63, 65, 154, 168, 179, 185, 186, 190, 196
Errors, 29, 98, 99, 122, 132, 142
Estimation, 46, 56, 89, 90, 107, 108, 149, 156, 157, 189

Extreme Programming, 8, 9, 10, 15, 18, 22, 23, 28, 38, 45, 55, 56n., 79, 84, 99, 109, 110, 131, 145, 176, 180, 227

Failure, *xi*, 26, 30, 36, 61, 64, 74, 87, 89, 94, 177, 182, 183, 184, 194, 201, 205, 206, 217
  learning from, 26, 90, 182ff., 205
Feature Driven Development, 15, 18, 52, 227
Feedback, 13, 17, 18, 60, 66, 67, 77, 82, 85, 86–92, 94, 106, 119, 123, 144, 145, 146, 162, 171, 172, 174, 188, 189, 195, 215, 219
  continuous, 87, 120, 219
  early, 13, 87, 89, 92, 188
Fit, 147n., 227
Flexible workplace, 65–66, 212, 213
Ford, Doris, 157, 223
Foreign element, 118
Fowler, Martin, 28, 39, 157, 224, 225
Frameworks, 21, 54, 68, 131, 147, 166
Freeware, 167, 198
Functionality, 14, 36, 37, 40, 52, 53, 87, 89, 92ff., 114ff., 129ff., 138, 145ff., 160, 173, 185, 188, 194, 200, 210, 216

Gerber, Jane, 118n., 226
Gift economy, 78, 83
Glazer, Hillel, 166, 169, 223
Gold Card method, 184, 227
Gomori, Maria, 118n., 226

Heinze, Roderich, 47, 48n., 226
Hierarchy, 21, 45, 48, 49, 85, 86, 153, 155, 156, 178, 185, 194, 196
Highsmith, James A., III, 14ff., 32, 119, 122, 173, 225
Human resources, 154, 171–72
Hunt, Andrew, 181n., 225

Implementation, 14, 38, 42, 49, 55, 91, 102, 145, 147, 203
  referential, 114, 115, 179, 201, 202
Insourcing, 185–86, 187, 189, 215
Integration, 19, 37, 45, 55, 58, 87, 97–102, 123, 124, 145, 146, 155, 163, 207, 210
  point, 98, 100, 132
  team, 99–101, 146
IntegrationGuard, 102, 227

Interviews, 44, 85, 105, 113–14, 197, 205, 207, 210
Iteration, 66, 77, 85, 87, 88–92, 95, 96, 99, 103, 106, 109, 120, 147ff., 156, 157, 183, 185, 188, 200, 210, 211
  planning, 95, 96, 110, 148, 188
  synchronization, 76–77

Jakarta, 80, 227
Jeffries, Ron, 16, 166, 223
Josuttis, Nicolai M., *vi*, 63–64, 126, 134–38
JUnit, 80, 81n., 102, 227

Kerievsky, Joshua, 89, 109
Kerth, Norman L., 70n., 103n., 206, 225
Knowledge transfer, 32, 33, 34, 35, 50, 66, 117, 180, 214
König, Dierk, 28, 78, 121n., 123n., 170

Large companies, 7, 8, 11, 19, 20, 21, 26, 45, 51, 80, 131, 153, 154, 156, 158, 160, 166, 174, 177, 178, 187, 190, 194, 196, 210, 220
Largeness, 5–7, 18–16, 134–38
Larman, Craig, 63, 225
Larsen, Diana, 109, 110
Learning, 29, 40, 47, 48, 67, 94, 118–20, 121, 122, 150, 159, 178, 180, 182ff., 208–9, 220
  environment, 178, 182–85, 192
  organization, 119, 178
Legal department, 154, 172, 189
Leuf, Bo, 51n., 66n., 225
Lightweight methods, 41
Lister, Timothy, 26, 68, 225

MacCready, Paul B., 129, 130n.
Mackerels, 24–26, 27
Maier, Mark, 145, 152, 226
Maintenance, 32, 48, 49, 87, 145
Management, *xii*, 6, 16, 20, 30, 92, 104, 106, 109n., 117, 119, 122, 123, 185, 186, 201
Manns, Mary Lynn, 59, 226
Marketing, 20, 93, 154, 162, 163, 172–73, 175, 192, 208
Maslow, Abraham, 160
McBreen, Pete, 34, 142, 178, 226
McConnell, Steve, 100, 224
Meetings, 66, 70, 71, 111, 207, 211, 186, 212, 213
Metaplan technique, 105, 106–7
Mistakes, 26, 44, 90, 122, 142, 210
Mistrust, 21, 31, 41, 47, 50, 112
MockObjects, 147n., 228
Models, 34, 36, 52, 206
  programming, 150, 151, 203
Modules, 135, 136, 138, 145
Motivation, 58, 60, 80, 85, 87, 95, 98, 142, 143, 182, 202, 205
  lack of, 194, 202

Non-agile processes, 12, 21–24, 50

Open-plan offices, 8, 63–65, 212
Open source, 11, 77–82, 83, 141, 150
Open Space Technology, 71, 105, 107, 228
Organizational structure, 20, 57, 153, 154, 191
  departmental, 20, 45, 153, 155, 169, 171, 191, 196
  project-oriented, 191, 196
Outsourcing, 9, 73, 104, 154, 158, 172, 185, 187–90, 196, 214–15
Overtime, 20, 26, 60, 203

Patterns, 67, 68, 107, 150
Paulk, Mark C., 166, 224
Planning, 13, 75, 76, 77, 82, 92–97, 111, 123, 146, 149, 156–57, 173, 196, 210
  component-based, 92, 93, 94, 156, 200, 216
  result-oriented, 93–96, 146, 156, 200, 207
  tools, 96, 97, 216
  workshops, 210, 215
Plans, 4, 17, 77, 92ff., 96, 97, 111, 122, 123, 156, 188, 200
Politics, 42, 104, 168, 185, 186
Postmortems, 70, 206
Pritchard, Wendy, 118, 224
Problems, *xi*, 5, 10, 11, 16, 27, 28, 33, 35, 40, 43ff., 48, 51, 52, 61, 73, 77, 78, 80, 85, 90, 101, 107, 108, 114, 116, 117, 122, 123, 130, 133, 136, 144, 147, 159, 160ff., 167, 168, 170, 174, 176, 181, 186, 187, 195, 197ff., 200ff., 208, 213, 214, 216, 219, 220
  learning from, 205–7
  notifying customer of, 51, 170
  solving, 43, 57, 108, 144, 155, 163, 194, 202, 210, 211, 213, 214, 216

Processes, 10, 11, 13, 16, 28, 31, 45, 50, 77, 107, 119, 154, 162, 164, 166, 168, 201, 206, 207, 220
 adapting, 15, 22, 23, 24, 103, 124, 160, 163, 164–66, 214
 business, 136, 137, 138
 changing, 23, 102, 104, 161, 163, 172, 215
 defining, 102, 103, 160, 162, 165
 enterprise-wide, 114, 154, 160–66, 197, 198
 heavyweight, 4, 41
 lightweight, 15
Process improvement, 68, 103, 165, 192
Programmers, 27, 48, 49, 80, 110
Project chartering, 109, 110, 111
Projects, *xi*, 4, 6, 8, 19, 21, 26, 27, 31, 33, 41, 44, 45, 48ff., 54, 55, 59ff., 65, 73, 77, 78, 80ff., 85, 87, 90, 92, 97, 100ff., 110, 113, 114, 118, 147, 152ff., 157, 160ff., 176, 177, 179, 185, 190, 191, 195, 202, 208
 controlling, 41, 79, 153, 154, 156, 157–58, 196
 culture, 59, 67, 180, 189
 failed, *xi*, 5, 30, 42, 44, 61, 64, 112, 122, 136, 177, 201, 205, 217
 fixed-price, 154, 156–60, 172, 188
 large, *xi*, *xii*, 4ff., 10ff., 17, 18, 20, 42, 63, 78, 81, 93, 96, 120, 128, 135ff., 142, 152ff., 178, 185, 193, 220, 221
 mission-critical, 5, 126, 194
 retrospectives, 109, 110, 111
 status, 15, 54, 100, 103, 105, 112, 160
 success, *xi*, 5, 42, 61, 64, 82, 109, 134, 155, 157, 177, 178, 208, 215
 support for, 21, 153, 165

Quality, 14, 78, 87, 100, 127, 143, 158, 159, 160, 165, 168ff., 173, 176, 200, 201, 210
 assurance, 154, 161, 169, 170, 171, 196, 199–200, 210, 212
 control, 20, 146, 154, 169, 170, 196, 199–200

Rechtin, Eberhardt, 145, 152, 226
Recompiling, 132, 133, 135
Refactoring, 29, 38, 39–41, 67, 101, 109, 129, 144, 145, 148ff., 188–89
Referential implementation, 114, 115, 179, 201, 202, 204
Reflections, 14, 21, 90, 105, 115, 119, 165, 204
Release, 14, 59, 85ff., 92, 94, 95, 98, 100, 102, 103, 111, 115, 147, 158, 170, 173, 174, 188, 189, 195, 203, 208
Release cycle, 66, 74n., 85, 87, 88, 89, 92, 95, 96, 120, 183, 194, 210, 211, 216
 time-boxed, 210, 211
Requirements, 3, 4, 5, 15, 16, 21, 22, 24, 28, 37, 38, 41, 53ff., 86, 87, 93, 104, 136, 138, 155, 156, 172, 175ff., 187, 189, 195
 changing, 4, 13, 19, 24, 54, 93, 119, 120, 129, 145, 159, 176, 187ff.
Restart, 44, 112ff., 194, 196, 201, 202
Retrospectives, 15, 28, 54, 58, 66, 68, 90, 102–11, 115, 124, 163, 165, 172, 190, 191, 198, 211–12, 215, 219
Reviewers, 182, 183, 203
Reviews, 68, 101, 182ff., 203
 team, 148, 151, 179, 182–83, 214
Rising, Linda, 59, 226
Risk, 3, 5, 6–7, 13, 24, 26, 27, 39, 40ff., 65, 85, 95, 98, 99, 101, 113, 114, 139, 156, 160, 162, 171, 175, 180, 187, 206, 207, 212
Roock, Stefan, *vi*, 55–57

Salter, Chuck, 47n., 224
Satir, Virginia, 118, 226
Schedule, *xi*, 7, 77, 88, 169, 174
Scope, 5–6, 7, 14, 42, 79, 158, 159, 160, 170, 188, 210
Scrum, 15, 18, 84, 228
Senge, Peter, 109, 226
Service orientation, 131–32
Showstoppers, 134, 159, 194
Smells of code, 40
Software, 3, 16, 34, 39, 40, 50, 74, 75, 80, 86, 92, 94, 102, 103, 126, 130, 132, 151, 152, 157, 171, 174, 175, 203
 delivery, 19, 34, 88, 92, 158, 159, 206
Sollmann, Ulrich, 47, 48n., 226
Staff, 67, 79, 119, 167, 169, 178ff., 183, 185, 186, 200, 206
Standards, 144, 150–51, 152, 165
Stopper, Dieter, 24n., 224
Subsystems, 19, 37, 55, 56, 57, 132, 136, 138

Subteams, 9, 43, 49, 51–55, 61, 64, 77, 84, 96, 100, 101, 114, 117, 128, 156, 157, 177, 212, 220

Success, 41, 47, 49, 50, 74, 202, 204, 208, 219

Systems, 7, 13, 14, 17, 19, 21, 23, 32, 33, 35, 37, 39, 40, 41, 45, 50, 55ff., 82, 86ff., 92ff., 98, 99, 102, 110, 114, 116, 124, 126, 127, 129, 130, 133, 136, 139ff., 145, 146, 149, 171, 173ff., 181, 183, 185, 190, 195, 201, 203, 209, 210
- change in, 36, 40, 92, 139, 144, 149, 209
- growth of, 60, 148, 194
- large, 127, 134, 138
- running, 99, 124, 200

Taylorism, 48n., 51

Team members, 8, 11, 14, 16, 20, 29, 45, 47, 48, 50, 51, 57ff., 64, 65, 77, 80, 85, 100, 119, 121, 128, 140, 141, 148, 162, 167, 171, 181, 183, 195, 201ff., 212, 216
- co-locating, 64, 75, 202

Teams, 4, 15, 29, 30, 43, 45, 47, 50, 55, 63, 67, 83, 97, 109, 111, 134, 137, 144, 148, 149, 180, 181, 208, 210, 211ff., 217, 219
- distributed, 8, 11, 72, 73–77, 83
- external, 149, 150, 190, 214, 215, 217
- growing, 117–18, 132, 193, 204–5
- jelling, 58, 69, 70, 83, 103, 117
- large, 5ff., 17ff., 42ff., 49, 51, 66, 73, 84, 85, 88, 97ff., 122, 123, 126ff., 138ff., 150
- roles, 43, 48, 57–58
- self-organizing, 21, 117
- size of, 6, 13, 54, 113, 114, 126, 152
- skills in, 21, 23, 52, 57, 115, 187
- small, 4, 9, 18, 42, 43, 49, 52, 84, 127, 139, 141, 205
- starting, 114, 115, 117, 131, 201, 202, 204, 205, 208, 212
- virtual, 11, 53, 54, 72, 77, 83
- visitors, 213, 214, 216

Technology, *xi*, 11, 30, 52, 54, 79, 94, 97, 114, 126, 131, 142, 144, 151, 166, 168, 179, 196, 201
- cutting-edge, 126, 142, 152
- morale and, 142
- risks of, 142–43, 144

Testing, 14, 20, 21, 36–39, 75, 144, 145–47, 152, 169, 198

Tests, 26, 36, 37, 38, 39, 67, 98, 145, 146, 169, 181, 189, 219

Thomas, David, 178, 181, 225

Time-box, 13, 85, 87, 89, 90, 91, 185, 201, 211

Time frames, 5, 6, 7, 13, 90

Tools, 13, 16, 26, 30, 31, 36, 39, 73, 97, 102, 141, 149, 167, 168, 198
- illegal use of, 167, 198
- support for, 148, 166, 198, 199

Training, 6, 166, 179–82, 191, 208–10
- material, 182, 209, 210

Transparency, 50, 51, 83, 202, 207

Traps, 113, 114, 142

Trouble-shooting, 43, 69–72

Unified Modeling Language, 35, 36, 134

Unit tests, 23, 29, 38, 39, 40, 100, 110, 145ff., 170, 188, 199, 200, 203

Use-case, 82, 201

Users, 16, 17, 22, 23, 55, 77ff., 94, 175, 177, 195

van der Helm, Peer, 24n., 224

Version control, 101, 139, 141, 149

Version management tool, 149, 150

Waterfall model, 10–11, 86

WebTest, 147n., 228

Wiki Web, 51, 66, 67, 73, 75, 83, 89, 94, 96, 103, 105, 150, 168, 180, 183, 201, 202, 209, 214

Work environment, 63, 65, 72, 77, 83

Working software, 16, 17, 19, 20, 31ff., 50, 87, 93, 159, 187, 201

XPlanner, 97, 228

XpPlanIt, 97, 228

XP Software, 228

YAGNI, 131

Yourdon, Edward, 161, 226

Zaninotto, Enrico, 155