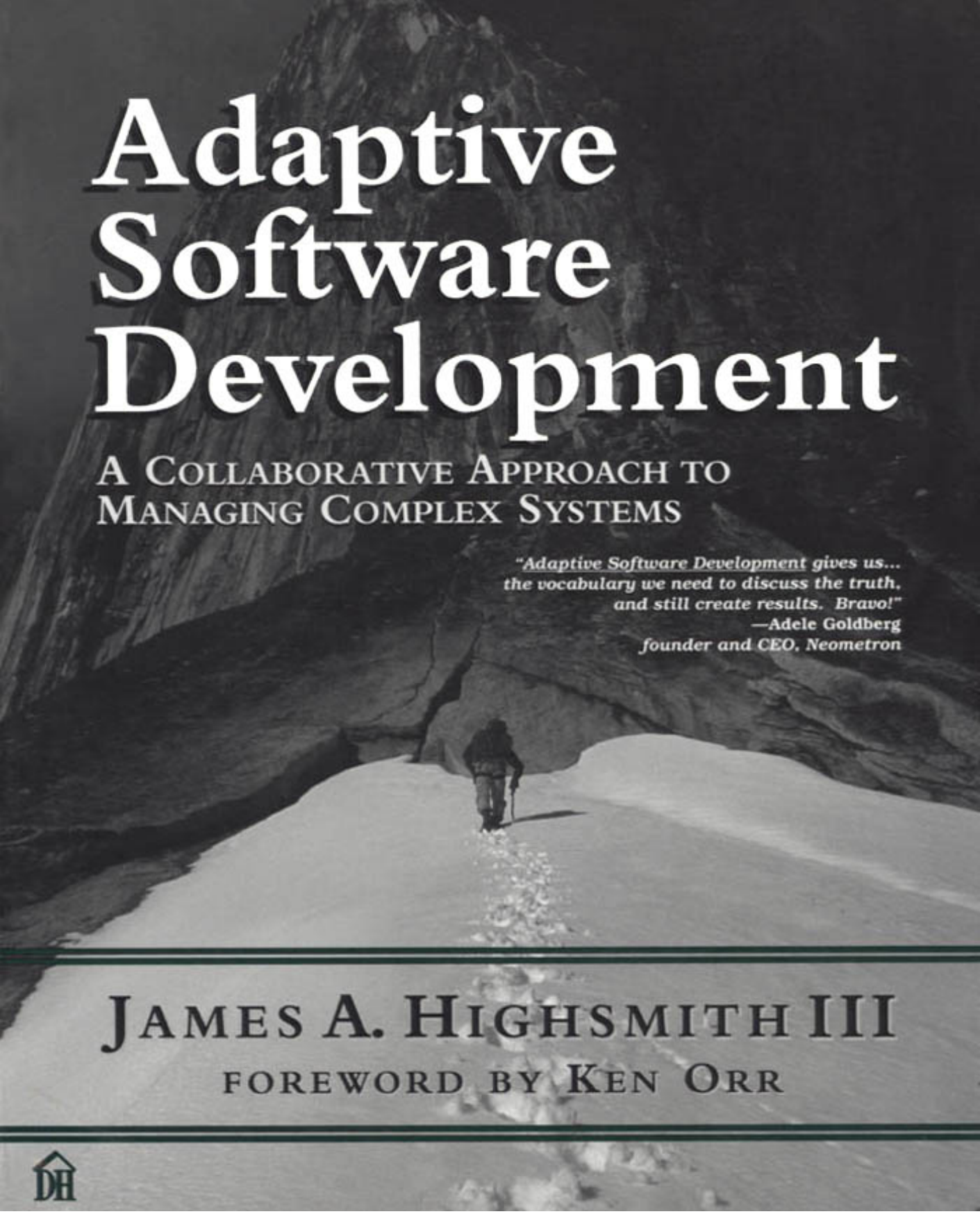
Adaptive Software Development

A Collaborative Approach to Managing Complex Systems

"Adaptive Software Development gives us... the vocabulary we need to discuss the truth, and still create results. Bravo!"
—Adele Goldberg
founder and CEO, Neometron



James A. Highsmith III

foreword by Ken Orr

Adaptive Software Development

A Collaborative Approach to Managing Complex Systems

Also Available from DORSET HOUSE PUBLISHING CO.

Are Your Lights On? How to Figure Out What the Problem Really Is
by Donald C. Gause and Gerald M. Weinberg
ISBN: 0-932633-16-1 Copyright ©1990 176 pages, softcover

Becoming a Technical Leader: An Organic Problem-Solving Approach
by Gerald M. Weinberg foreword by Ken Orr
ISBN: 0-932633-02-1 Copyright ©1986 304 pages, softcover

Complete Systems Analysis: The Workbook, the Textbook, the Answers
by James & Suzanne Robertson foreword by Tom DeMarco
ISBN: 0-932633-50-1 Copyright ©1998,1994 624 pages, softcover

Creating a Software Engineering Culture
by Karl E. Wiegers ISBN: 0-932633-33-1 Copyright ©1996 384 pages, hardcover

Exploring Requirements: Quality Before Design
by Donald C. Gause and Gerald M. Weinberg
ISBN: 0-932633-13-7 Copyright ©1989 320 pages, hardcover

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

The Practical Guide to Business Process Reengineering Using IDEF0
by Clarence G. Feldmann foreword by John V. Tieso
ISBN: 0-932633-37-4 Copyright ©1998 240 pages, softcover

Quality Software Management Series by Gerald M. Weinberg

Vol. 1: Systems Thinking
ISBN: 0-932633-22-6 Copyright ©1992 336 pages, hardcover

Vol. 2: First-Order Measurement
ISBN: 0-932633-24-2 Copyright ©1993 360 pages, hardcover

Vol. 3: Congruent Action
ISBN: 0-932633-28-5 Copyright ©1994 328 pages, hardcover

Vol. 4: Anticipating Change
ISBN: 0-932633-32-3 Copyright ©1997 504 pages, hardcover

Find Out More about These and Other DH Books:

Contact us to request a Book & Video Catalog and a free issue of *The Dorset House Quarterly,* or to confirm price and shipping information.

DORSET HOUSE PUBLISHING CO., INC.
353 West 12th Street New York, NY 10014 USA
1-800-DH-BOOKS (1-800-342-6657) 212-620-4053 fax: 212-727-1044
dhpubco@aol.com http://www.dorsethouse.com

Adaptive Software Development

A Collaborative Approach to Managing Complex Systems

James A. Highsmith III

foreword by Ken Orr

Dorset House Publishing
353 West 12th Street
New York, NY 10014

Library of Congress Cataloging-in-Publication Data

Highsmith, James A.
Adaptive software development : a collaborative approach to managing complex systems / James A. Highsmith, III ; foreword by Ken Orr.
p. cm.
Includes bibliographical references and index.
ISBN 0-932633-40-4 (softcover)
1. Computer software--Development. 2. Computer Systems--Management. 3. Management information systems. I. Title.

QA76.76.D47 H55 1999
658'.0551--dc21 99-052212

Cover Design: David McClintock
Cover Photograph: Stuart Ruckman Photography
Author Photograph: Robert Munk Photography

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: 99-052212

ISBN: 0-932633-40-4 12 11 10 9 8 7 6

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

Dedication

To William Byron Mowery, whose writing influenced me in subtle ways I am just beginning to understand. Bill Mowery wrote several dozen books and published hundreds of articles from the 1920's through the early 1950's. He taught creative writing at New York University. Although I never met him, he was my grandfather, and he passed on a love and understanding of writing to my mother, Dodie, who is an accomplished writer herself. I am indebted to her for a lifetime of encouragement, and to him for a legacy I've always wanted to contribute to in my own way.

This page intentionally left blank

Contents

Acknowledgments

The book before you is a result of creative collaboration. While I take full responsibility for the content, its quality would be greatly diminished without the efforts of my colleagues and clients—the people who have contributed in a variety of ways toward the book's completion.

Ken Orr, with whom I have worked for nearly twenty years. I have always been in awe of Ken's ability to conceptualize and create visions of the possible. Over the years, my relationship with Ken has spanned that of customer, employee, coworker, and colleague—but most importantly, friend.

Lynne Nix, with whom I also have worked for many years. Lynne is not only one of the best project managers I have ever known, but she is *the* best at conveying her knowledge to others. A number of the project management ideas in this book originated with Lynne.

Jerry Weinberg, whose writing has influenced me since the early 1970's. I've known Jerry since the mid-1980's and have always been inspired by his emphasis on and insight into the human side of our technological world.

Sam Bayer, who co-developed many of the RAD practices in this book and with whom I've worked on a number of accelerated develop-

ment projects. As in any good collaborative effort, distinctions as to which of the ideas were mine and which were Sam's have long since been lost. Sam has an uncanny ability to extract the essence of a complex situation and communicate that essence in a cogent, simple way. His comments on draft versions of this book helped me focus more clearly.

Jerry Gordon, whose ideas impacted this book in many ways. I first met Jerry nearly twenty years ago when I was doing "structured stuff." Jerry introduced me to mountaineering and climbing. Many of the climbing stories in the book are based on trips he and I took. During the years we have worked together (on some form of business or another), Jerry was actually willing to *try* some of my ideas about adaptive development when they were still nascent.

Steve Smith, who read this manuscript more times, and in more different incarnations of it, than anyone else. His comments and incisive questions always expanded my thinking and were greatly appreciated. Steve and I share two joys—discussing difficult issues and taking long, arduous hikes.

Others who have contributed directly, and sometimes indirectly, include Cheryl Allen, Jim Davis, Rob Arnold, James Bach, Karen Coburn, Tom DeMarco, Anne Farbman, Dan Larlee, Adele Goldberg, Warren Keuffel, Martyn Jones, Steve McMenamin, Lou Russell, Robert Charette, Larry Proctor, George Engleberg, James Odell, Bruce Watson, George Johnson, and Wayne Collier. I thank them all for the wisdom they have so generously shared.

I had support from many friends in the Salt Lake climbing community who were party to the analogies used in the book. Among the many, I especially thank Amy Irvine, who first taught me that movement skills were more important than strength, and Doug Hunter, who helped me refine those skills. Thanks also to Mac Lund, my great friend and regular climbing, skiing, and hiking partner, and to Bob Richards, Brian Mecham, Dale Goddard, and the crew at Rockreation.

To my wife, Wendie, who encouraged me throughout the ups and downs of the writing process and who put up with my sleepless nights and pirated weekend mornings, I am profoundly grateful. To daughters Nikki and Debbie, and to the *other* Jim, my father, I am thankful for their support and encouragement.

I never truly understood the relationship between authors and publishers—now I do. The staff at Dorset House that worked with me on this book is exceptional. Mike Lumelsky, Matt McDonald, Bob Hay, David McClintock, and Wendy Eakin were all part of the collaborative effort that turned a manuscript into a polished product. My thanks to all of them.

Permissions Acknowledgments

p. xxx: Material from Warren Keuffel, "People-Based Processes: a RADical Concept." *Software Development* (November 1995), p. 37. Reprinted with permission.

pp. 3-26: Material adapted from Jim Highsmith, "Software Ascents." *American Programmer* (June 1992), pp. 20-26. Reprinted with permission.

pp. 8; 37; 285; 286: Material from Stuart Kauffman, *At Home in the Universe.* Copyright © 1995 by Stuart Kauffman, pp. vii; 15; 100; 84. Reprinted by permission of Oxford University Press, Inc., and Brockman, Inc. All rights reserved.

p. 9: Reprinted from W. Brian Arthur, "Increasing Returns and the New World of Business." *Harvard Business Review* (July-August 1996), pp. 100, 101, 102, 107. Reprinted with permission.

pp. 9ff.: Material adapted from Jim Highsmith, "Messy, Exciting, and Anxiety-Ridden: Adaptive Software Development." *American Programmer* (April 1997), pp. 23-29. Reprinted with permission.

pp. 10; 329: Material from George Johnson, *Fire in the Mind: Science, Faith, and the Search for Order.* Copyright © 1995, pp. 235; 130. Reprinted by permission of Vintage Books, Random House, Inc. All rights reserved.

pp. 11; 58; 59; 143; 202; 205: Reprinted from Arie de Geus, *The Living Company: Habits for Survival in a Turbulent Business Economy.* Boston: Harvard Business School Press, 1997, pp. 11; 155; 46; 157; 140; 3. Reprinted with permission.

Foreword

If I could, I would give a copy of Jim Highsmith's book to everyone involved in developing large systems—end users, managers, IT professionals, and most especially IT project managers! Jim's message is simple but vitally important: Large information systems don't have to take so long, they don't have to cost so much, and they don't have to fail. Unfortunately, as simple as Jim's message is, making it happen is an enormously difficult undertaking in most large organizations.

While many project management books deal extensively with the need for building systems faster in today's business workplace, Jim's is the first book I've read that addresses what must happen when management is faced both with the need for high-speed delivery and with business requirements that are rapidly changing as well!

Jim's solution to both these problems is straightforward—a radical form of incremental development. But actually developing large systems incrementally is a considerably more difficult business than just talking about it. Historically, large organizations tend to attack all problems by breaking them into pieces and assigning each of the pieces to different organizational units for parallel development. Management then places its faith in managing to predefined budgets and schedules, but usually does so without any clear idea of what exactly

the completed project will produce or how the major pieces, the subprojects, ultimately will fit together.

Jim's approach combines the best features of techniques that have been used piecemeal for a long time: customer focus groups, versioning, time-boxed management, and active prototyping. Used individually, these approaches can be effective; combined, they are dynamite.

In my own consulting work, I am often called in to turn off the life support on large, failed projects. Most of these systems fail because they lack the right systems strategies and because they take too long. Projects lasting three-to-five years are rarer and rarer these days because organizations know that they have to implement systems earlier to meet changing business needs. On the other hand, unstructured, short-term projects are a maintenance nightmare. So, since large-scale projects won't simply go away, they need to be approached in a different fashion. In order for organizations to develop large systems more successfully, they must develop incremental implementation plans and incremental architectures, and they must begin to build individual subprojects in small, short-term pieces. Jim Highsmith's book provides the framework for doing just that.

August 1999
Topeka, Kansas

Ken Orr

Preface

George Johnson's *In the Palaces of Memory* weaves a fascinating story about one of the most complex of all biological phenomena: human memory. From Johnson's stories of neurobiologists who study the neuron and its components—axons, synapses, dendrites—and try to model how neural cells excite each other to create patterns, to tales of computer scientists whose explorations of the mind use neural networks and artificial intelligence, the reader is drawn to the conclusion that how memory works is still shrouded in mystery.

Software development may be as close to a purely *mental* activity as any complex business undertaking. Just as we are puzzled by the workings of the mind, we are caught up in the enigma of how software products emerge from the minds of their creators.

Software seems so simple. A few operators, a few operands, and *voilà*—a program is created. Combinatorial mathematics, however, insures that programs of any length have infinite potential variety—perpetual novelty.

Chess also seems simple. With fewer than two dozen rules and despite several hundred years of play, chess's capacity to generate perpetual novelty has not diminished.

Perpetual novelty is one measure of complexity. However, *complex* is not the same as *chaotic*. Chaos is random. Complexity contains patterns—patterns that peek through the perpetual novelty and can be used by people in their struggle to thrive in our world. This book is about the complex human endeavor of building computer software. It is about using the emerging field of complexity science (or, more specifically, the field of complex adaptive systems theory) to aid us in our pursuit of ever-more-complex software products. At its core, software development may be about how we turn *memories* of our world into computer models.

Neurons are pieces that create the fabric of memory, but how this fabric is woven is unknown. However, not understanding the weaving does not keep us from using the fabric. The infinite fascination of chess arises from simple rules. We cannot *predict* where a chess game will go, but we can learn *patterns* of play that bring success. Both chess and memory provide instances of a phenomenon that in the language of complex adaptive systems is called "emergence." Emergence is a property of complex adaptive systems in which the interaction of the parts creates some greater property of the whole that cannot be fully explained from measured behaviors of the agents. These results come from exploiting patterns. Harnessing emergence is a major theme of this book.

Goals of the Book

Adaptive Software Development has five primary goals. Detailed in the paragraphs below, the goals define the essence of building better software in a world where high speed, change, and uncertainty are key characteristics of its intensifying complexity.

The first goal is to offer an alternative to the belief that optimization is the only solution to increasingly complex problems. Optimizing cultures believe they are in control, that they can impose order on the uncertainty around them. Imposed order is the product of rigorous engineering discipline and deterministic, cause-and-effect-driven processes. The alternative idea is one of an *adaptive* culture or mindset, of viewing organizations as complex adaptive systems, and of creating emergent order out of a web of interconnected individuals. An adaptive approach raises the acknowledgment of an uncertain and complex world from being a manager's death knell to being part of a recognized and accepted strategy.

The second goal is to offer a series of *frameworks* or *models* to help an organization employ adaptive principles. The Adaptive Development Life Cycle, for example, provides a framework that reinforces the concepts and details a practical way of moving from the conceptual to the actionable. While there are other types of iterative life cycles, they have not been based on underlying adaptive concepts but rather on short-cycle determinism. There is a certain synergy in linking iteration with adaptive ideas to combat complexity. The frameworks are supplemented by a discussion of various techniques (customer focus groups, for example). However, the focus is on the frameworks, not on a compendium of techniques.

The third goal is to establish *collaboration*—the interaction of people with similar and sometimes dissimilar interests, to jointly create and innovate—as the organizational vehicle for generating emergent solutions to product development problems. To be effective at feature-team, product-team, and enterprise levels, collaboration must be addressed in terms of interpersonal, cultural, and structural relationships.

The fourth goal of *Adaptive Software Development* is to provide a path for organizations needing to use an adaptive approach on larger projects. Because Rapid Application Development (RAD) approaches had a reputation of eschewing all discipline and rigor, they were relegated by many developers for use only on noncritical *toy* projects. *Real* projects require rigor and discipline; RAD was for playing. So, the last chapters of this book show how adaptive development works in real-life situations in which uncertainty and complexity create the need for an approach that is both adaptable *and* scalable. A major component of meeting this challenge is the ability to move from a workflow, process-oriented development life cycle to one based on work*state* and information.

The idea for this book began as one about RAD projects, written to answer such questions as, "Why does RAD work?" and "How does what works scale up?" I asked these questions because my experience and that of colleagues like Sam Bayer in implementing individual RAD projects was highly successful. I consulted with numerous large software companies that used RAD techniques on very large projects. I also worked with several IT groups that could never seem to leverage success on individual projects into larger successes. I tried to look not only at why certain practices worked, but also at the circumstances in which they were applicable.

Circumstances are defined by culture. The culture of Command-Control management has become outdated, in part because of wider

cultural trends toward flexible management styles, wider participation in decision-making, and empowerment, but also because of a single pragmatic fact: Command–Control management cannot process knowledge and information quickly enough in the new economy. So, the last goal of this book is to offer a new, adaptive management style, which I label Leadership–Collaboration, to replace Command–Control. The ability to adapt and move quickly requires that "leadership" replace "command" and "collaboration" replace "control."

Our business culture and styles of management have been built around an optimizing mindset that values stability and predictability. Our management tools (derived from this belief) work—or seem to work. Consequently, we fail to understand the sharp disparity between orderly and complex systems. Orderly systems can be extremely *complicated,* but complicated and complex describe different *classes* of problems. Complicated problems yield to optimizing techniques; complex problems do not. Tools that work in orderly realms are actually antithetical to complex situations. The disparity between orderly and complex problems requires not only new tools, but a very different mindset—a significant transition for many organizations.

In an optimizing culture, increased rigor (process improvement) and stabilization are the end goal. Optimizing cultures tend to see the world as black or white, with little room for gray. If it is not rigorous, it must be chaotic (or *immature,* to use the Software Engineering Institute's parlance).

An adaptive culture recognizes gray. Researchers have coined a phrase for this turbulent gray area between order and chaos—*the edge of chaos.* It is in this variable, messy, exciting polyglot that emergence happens. In the field of evolutionary biology, for example, scientists postulate that major evolutionary advances occur in this complex region at the edge of chaos.

In an adaptive culture, the goal of rigor is to maintain balance on this edge, providing just enough stabilizing force to keep away from chaos—but no more. Adaptive organizations understand the need for *just enough rigor.* This balance does not come from a compromise of principles, but from an understanding of how the forces of optimization draw down the very energy sources that are needed to fuel emergence. Too little rigor yields chaos. Too much stifles emergence and innovation.

So, while one goal of this book is to offer emergent order as an alternative to a belief in and a dependence on imposed order, emergent

order is not a complete replacement but the basis for a new, additional set of tools for managing complexity. I believe the difference in viewing optimization as a *balancing force* rather than as a goal in itself provides a significant shift in perspective.

For me, this book was a journey—one that goes on. I invite you to join and hope it engages you as it did me.

September 1999 J.A.H.
Salt Lake City, Utah

This page intentionally left blank

Introduction

Adaptive Software Development is written for several audiences. First, it is intended for project teams that have been struggling with high-speed, high-change projects—extreme projects—and are looking for ways both to improve performance and to moderate burnout—especially as the projects they undertake get larger and the teams become more distributed.

Next, the book is written for project teams that have been assigned a high-speed, high-change project to support a critical new business initiative. These team members know their standard approach probably won't work and need a better alternative that will enable them to deliver in a culture geared to more sedate projects, often in an environment that actually is hostile to the practices required to succeed.

The third audience consists of project teams that need to accelerate schedules for small to medium-size projects. Many of the techniques described in the book were derived from this type of project. Although the limitation to scaling typical RAD techniques up to work on larger projects is change itself, adaptive development addresses this issue.

A final audience includes people who need ammunition to fend off requests for high-speed, high-change projects. Even with the best prac-

tices and effective management, extreme projects are risky. Clarifying that risk, and learning when to avoid it, is important.

Reviewing an early version of this book, columnist Warren Keuffel described how the approach differs from more procedural approaches:

> *RADical Software Development . . . is a framework within which the intelligent project manager is expected to fit the practices that have been proven to work.* —W. Keuffel [1995], p. 37.

Adaptive Software Development does not provide a set of prescriptive rules or tasks, but a framework of concepts, practices, and guidelines.

The book is divided into three parts. Part 1 consists of Chapters 1 and 2. Chapter 1 provides background material and introduces the three major models: the Adaptive Conceptual Model, the Adaptive Development Model, and the Adaptive Management Model. Chapter 2 delves into the concepts of complex adaptive systems (CAS). It then introduces the Adaptive Development Life Cycle and explains how that life cycle approach incorporates the concepts of CAS.

Part 2, Chapters 3 through 6, explains the components of the development life cycle—speculating on direction, adaptive cycle planning (detailed speculating), collaboration, and learning. The chapters of Part 2 introduce additional aspects of complexity, and then propose practical techniques based on those concepts. The chapters focus on accelerated delivery from single work groups or feature teams.

Part 3, Chapters 7 through 11, describes the adaptive management culture and practices that I have grouped under the banner of Leadership–Collaboration management. Chapter 7 provides an overview of the Leadership–Collaboration Management Model and the rationale for its use. Then, Chapters 8 through 10 explore components of the model. These chapters emphasize problems of and solutions for scaling adaptive development to larger projects. They also focus on collaboration as it embodies the concepts of emergence across multiple groups, and on the cultural and structural aspects of collaboration. Chapter 11 covers project management topics, including an explanation of a project management framework, or life cycle, and time-boxing. Project management provides a boundary of imposed order within which emergent order can flourish. The chapter treats the topics of risk assessment and other practices contributing to successful adaptive projects.

The final chapter of the book, Chapter 12, provides some parting thoughts—about dawdling, Marshall McLuhan's technology views, assessing organizational growth, and operating in thin air. Although seemingly unrelated, these topics review the message of the book in terms of a discussion for applying adaptive systems concepts.

The technology community drives unrelenting change—change in communications, in business practices, and even in business strategy. But somehow, while admonishing our businesses to adopt technology more rapidly, we, as the purveyors of technology, have failed to anticipate the impact that the speed and turbulence we have created has had on our own practice of management. *Adaptive Software Development* is ultimately about rethinking how to manage in the turbulent times we have brought upon ourselves.

This page intentionally left blank

Adaptive Software Development

A Collaborative Approach to Managing Complex Systems

This page intentionally left blank

CHAPTER 3
The Project Mission

The object of your mission is to explore the Missouri river, & such principal stream of it, as, by it's course and communication with the waters of the Pacific ocean, whether the Columbia, Oregan, Colorado or any other river may offer the most direct & practicable water communication across this continent for the purposes of commerce.

—T. Jefferson, as quoted in S. Ambrose [1996], p. 94.

With these words, Thomas Jefferson launched the Lewis and Clark expedition of discovery, the most notable event in American exploration. For thirty months, these words drove the members of the expedition on—through brutal physical challenges, frightening Indian encounters, isolated winters, and life-and-death decisions. Writing a mission statement is one thing; understanding the scope, meaning, subtleties, ambiguities, and limits of a mission is another thing altogether.

Meriwether Lewis was Thomas Jefferson's secretary for several years before the expedition began. The men knew each other well, talked long about the expedition and its goals, and exchanged extensive correspondence on various aspects of the journey. At a time when

the Federalists were attacking Jefferson for his purchase of the Louisiana territory, Jefferson and Lewis shared a vision of a unified continental United States. They agreed on the purpose of the expedition, viewing it as one means to achieve their grander continental vision. Lewis passed their vision on to William Clark and the other explorers—a vision consisting not only of words, but also of the passion and inspiration to bring the words to reality.

A broad mission statement is important, for it gives purpose and meaning to a task, but it needs to be supplemented with specifics. Jefferson's mission statement gave specific instructions about mapmaking: ". . . you will take careful observations of latitude & longitude, at all remarkable points on the river, & especially at the mouths of rivers, at rapids, at islands . . ."; about Indian ethnology: "to learn the names of the nations, and their numbers, the extent of their possessions, their relations with other tribes . . ."; and about flora and fauna: "to notice and comment on the soil, the plant and animal life, . . . dinosaur bones, and volcanoes" (Jefferson, in Ambrose96, pp. 94, 95).

The thoroughness of the mission statement was critical to the success of the expedition. Because the explorers would be out of contact with the rest of the world during most of the trip, the mission statement needed to be specific and yet flexible enough to allow the leaders to make key decisions as their party explored the unknown. (Indeed, by the spring of 1806 and almost two years into the journey, many people had given up on the expedition as "lost in the wilderness.") Jefferson's instructions focused on objectives and broad directives but left the implementation to Lewis and Clark. Lewis and Clark didn't succeed because of a good mission statement, but they would not have succeeded without one.

Good mission statements don't come easily. Jefferson's mission statement took years of study and represented the collected wisdom of many colleagues. How many project teams, when asked if they have a mission, would answer, "No. We just fumble around all the time"? Most team members probably think they have some sort of mission, but a good mission is not a thing, not words on a page or on a flip chart. A good mission is shared passion. At its most basic, a mission is a product goal worth striving for. At its best, a mission touches each team member's sense of a wider purpose beyond producing some *thing*.

There are three steps to fashioning a statement of mission. The first is to *identify the mission*, to understand what constitutes a good mission. The second is to *create mission artifacts*, to define specific mission docu-

ments and to develop their contents. The third step is to *share mission values,* to go beyond the words on paper or in digital images so that each team member shares the passion and purpose of the mission.

Identify the Mission

A good mission statement is many things, particularly in a complex environment. It must be concrete, yet still invite people to speculate on different scenarios. It must also facilitate the flow of information. The objective of a development effort is to build a usable software product, so a good mission statement needs to be specific. However, if it is too specific, the project team may be overly constrained. A mission statement needs to help the team converge on a solution while still keeping the team open to divergent innovation. A mission statement needs to direct or define the scope or boundaries for the effort, not detail the final outcome.

Characteristics of a good mission statement are threefold. The first is that it will establish a sense of direction. At a project or product level, a mission establishes a framework for action, a scope for what is to be accomplished, and a theme for design.

A second characteristic is that a good mission statement is inspirational. People's best efforts arise from their being passionate about what they are doing. A mission statement answers questions such as, Why do I care about this project? Why should the team care about this project?

There is a third aspect to a mission statement. It guides implementation, suggesting how the project should be approached and providing a framework for decision-making. Steering development processes and practices requires knowing what one is steering toward—the mission. Engineering is in some fundamental way a process of synergy and trade-off, of trying to derive a design incorporating often disparate requirements and making technical trade-offs based on business criteria.

Hundreds of decisions—some major, many minor—are made during a software project. Particularly in an adaptive, iterative project where team members and clients are encouraged to learn and change, a lack of boundaries defined by a mission statement would create chaos. Decisions must be made. Trade-offs are a necessity for moving forward. High rates of change mean that even more decisions must be made, and quickly. Good mission statements guide decision-making.

"[U]nity *is the master principle of great art. And I have seen over and over that unity is the master principle of great software. . . . The* theme *of your software is the dominant idea that constitutes the basis of the design. . . . You've got to have a purpose for your product, and 'unity of purpose' is a good phrase to describe the impact of having a theme."*
—J. McCarthy [1995], pp. 80, 81.

"A mission is a sense of purpose that lures you into your future. It unifies your beliefs, values, actions, and your sense of who you are. . . . Most of all, a mission is fun."
—J. Andreas and C. Faulkner [1994], p. 80.

It is not enough to have a two-sentence vision statement: Different *levels* of mission are needed to make different levels of decisions. Broad-sweeping, generalized mission statements are useless when detail-level decisions must be made. Narrow, precise mission statements don't allow for innovation and flexibility. Part of the ability to develop a good mission statement lies in the development team's ability to understand this ambiguity.

In a complex environment, a mission statement is a speculation about the future, and everyone involved must remember that this speculation is the best guess of the future available at the time, something to be continually tested against reality and adjusted accordingly. At a conceptual level,

> **A mission statement facilitates collection of information that is relevant to the project's desired result.**

Any product development effort is, at its core, the gathering, analyzing, and reconfiguring of information. One of the defining characteristics of a complex environment is the high rate of information flow. In a stable environment, information flows slowly and predictably. In a complex environment, information bombards the project team from numerous and often unforeseen sources. How do team members extract the information they need from the vast possibilities?

Industrial robots use "rules" to manufacture automobiles. Insurance service administrators use well-defined "processes" to initiate a policy. Chess players use "patterns" of play to outsmart their competitors. In software engineering (and in business), the word "process" has become synonymous with a carefully controlled procedure for arriving at planned results. What we need is a new word to connote insight without certainty—and the word becoming more widely used is "pattern." A design pattern is a framework for helping someone transfer knowledge, but it is not a recipe to be followed by rote. Success on complex projects requires that we comprehend the difference between processes and patterns. The first prescribes activities, the second organizes thinking. A mission statement should suggest a direction rather than a destination and must help us select the patterns that will increase our chances of success.

To me, the word "process" suggests a mechanical world, while the word "pattern" suggests an organic one. Adaptive Software Development is not a series of processes, but an assemblage of patterns that can assist development teams in their thinking but that also have limitations.

Innovation and creativity are not the result of processes, but of patterns. Management and organizational patterns are emergent; we cannot always explain exactly how they work, but we recognize that they often (although not every time) produce results.

A mission statement helps us navigate through and understand a complex project environment and facilitates the collection of information. Because of the high information-flow rate in a complex environment, we cannot anticipate exactly what a team will need. However, by identifying a credible direction and instituting appropriate patterns of inquiry, we should be able to collect relevant information to produce the correct product. If the direction is too broad, we will find that processing all the captured bits will be too time-consuming. If the direction is too narrow, there will not be enough diversity to build a viable product. In short, to identify the mission, software project management must focus on direction and it must establish appropriate priorities to facilitate reaching that goal.

A Need to Focus

In the July 1995 issue of *American Programmer* magazine, several well-known authors supported the opinion that, to be successful, a project must meet *all* of the following criteria. The project must

- meet business objectives
- meet quality expectations and requirements
- stay within budget
- meet its time deadline
- deliver actual benefits
- provide the team with professional satisfaction and the opportunity to learn

With all due respect to these highly regarded authors, I view this stance on multiple criteria for measuring project success to be pure rubbish! There is no way to satisfy *all* of the above criteria on any reasonably sized project, much less on an *extreme* one. By making everything top priority, these authors leave managers no effective way to manage change. Change requires trade-offs, trade-offs require an understanding of priorities. My objection does not mean that any one

of the criteria listed above is unimportant, but that we have to *choose one criterion as the most important.* By setting up such impossible multiple criteria for measuring success, we do not give truly successful projects their due.

The siren cry in marketing is Segment, Segment, Segment. In business strategy, it is Focus, Focus, Focus. Why should software management be any different? It is not, as I show in Table 3.1, which contains a matrix to help project leaders manage change by establishing appropriate priorities for desired project results.

A Need to De-Focus

"You do not navigate a company to a predefined destination. You take steps, one at a time, into an unknowable future. . . . In the final analysis, it is the walking that beats the path. It is not the path that makes the walk."
—A. De Geus [1997], p. 155.

The Speculate–Collaborate–Learn Adaptive Development Model Life Cycle, first depicted in Fig. 2.3, has small offshoot arrows that represent breakthrough or emergent learning, suggesting directions or product feature sets not planned but that could potentially improve the product. If project management has not even considered a potential project direction, it will find it difficult to accept data relevant to that direction. So concerned with reducing "scope creep," the seemingly constant change in product requirements, we as managers may fail to examine innovative new directions.

Software development in complex environments is difficult because it is not deterministic—that is, it is not easily controlled by simplistic (which must not be confused with *simple)* rules. While I encourage project team focus through development of mission statements, a singular focus will doom projects in a complex environment. Every project team goes through periods of focus—of converging on a solution—and then de-focus—diverging away from that solution because of new ideas, problems, or conflicting opinions. Both activities are healthy if not carried to extremes. It would be simple to say, "Always focus." It is much more difficult to say, "Sometimes focus, sometimes de-focus." Knowing when to focus and when to de-focus requires judgment—a skill often in short supply. Thinking of a mission as a boundary rather than as a destination, as a pattern rather than as a single point, can assist a project in this balancing act.

In the preceding chapter, a fitness landscape was defined as a three-dimensional representation of a project's success criterion. Thought of in terms of the fitness landscape, focusing is analogous to climbing a peak and striving to reach the top. De-focusing is analogous

to stopping to consider the possibilities of jumping to an adjacent peak, one that offers greater potential because it is a higher peak but which also offers risk because it necessitates abandoning the current peak.

If we need to both focus and de-focus, we need to know when to do each. Determinists want rules such as, "When x occurs, focus. When y occurs, de-focus." This merely replaces one simplistic rule (Always focus) with a short set of simplistic rules. Unfortunately, no one rule, or even one *set* of rules, can be applied to all situations.

Holding anxiety, dealing with paradox and conflicting constraints, continually being poised at the edge, alternately applying techniques to converge and diverge—all are critical to maintaining an environment conducive to emergent results. Knowing when to focus or de-focus a project team is essential to successful management in complex environments; adaptive techniques can assist managers in making these judgment decisions.

One popular technique used for de-focusing is called *scenario planning*. Scenario planning requires project members first to change some basic assumptions about a product, a technology, or a market, and then to explore what this new direction would mean. Feasibility studies often contain an analysis of alternatives, but these are usually alternatives for implementation. Scenarios provide alternatives for significant product feature sets, or even for entire missions.

Monitoring the environment using alternative scenarios keeps the team from focusing too tightly. As in an ant colony that has scouts out exploring the territory, software developers must scour the terrain for the first indications of competitive danger or opportunity.

"Scenarios bring new views and ideas about the landscape into the heads of managers, and they help managers learn to recognize new 'unthinkable' aspects of the landscape even after the scenario exercise is over."
—A. De Geus [1997], p. 46.

Create Mission Artifacts

The second step in fashioning a mission statement is the creation of mission artifacts. Why this step is important is not always clear in the heat of a project, but it stems from the fact that software project failure is endemic. At the start of a project, everyone is in a hurry, either because management has already delayed six months in initiating the project and needs to make up lost time, or because some competitor has just struck, or because . . . just fill in the blank: "Full speed ahead." "Damn the torpedoes." "Don't be a wimp." The beginnings of many projects are injected with a tremendous oversupply of testosterone. Three or four months into the project, someone finally realizes that no

one knows what the project is all about, and a reevaluation period ensues. The following statistics indicate how wasteful this is.

> *Our research shows a staggering 31.1 percent of projects will get canceled before they are ever completed. Further results indicate 52.7 percent of projects will overrun their initial cost estimates by 189 percent. . . . Over one-third of these same challenged or impaired projects experienced time overruns of 100 to 200 percent. One of the major causes of both cost and time overruns is restarts. For every 100 projects that start, there are 94 restarts.*
>
> —J. Johnson [1995], pp. 4, 5.

That 94 percent of all projects will be *restarts* is incredible! According to Johnson, we lose $78 billion a year on canceled IT application development projects. A good set of mission artifacts, developed in a feasibility study process, can reduce the losses substantially.

Mission artifacts comprise the specific documents developed for a project. While the details must be customized for each project, these artifacts need to answer three questions:

1. What is this project about?
2. Why should we do this project?
3. How should we do this project?

A concept is not a project, but a beginning, a thing to be examined and *nurtured into* a full-blown project. Many projects get in trouble because they confuse *starting* fast with *ending* fast. In most situations, taking extra time to get the project started properly pays big benefits in time saved at the end.

What is this project about? To provide context for this question, we might paraphrase Lewis Carroll's *Alice in Wonderland:* "If we don't know where we want to go, then any path will do." One primary reason for project restarts, or outright failure, is the lack of a project mission. Team members, customers, and other stakeholders need a good understanding of the project's fundamental components—its goals, objectives, scope, problem statement, constraints, and vision. A good test of whether project participants understand a project is to walk around and ask them, "What is this project about?" The more complicated the answer, the more trouble the project is in. A crisp, business-

oriented, nontechnical answer usually means the project's groundwork has been well established.

Why should we do this project? This question might be restated as, "Why do *I* care?" The second major question answered by mission artifacts is, "Should the project proceed?" Part of analyzing the mission is determining its feasibility—and one possible option is to not proceed. A significant portion of the $78-billion loss on software projects comes from projects that should have never moved past the feasibility stage, but that have become caught up in a battle of corporate egos and politics. Once the problems and opportunities have been identified, the next task is to define the criteria for an acceptable solution. Feasibility (acceptability) incorporates political, economic, technical, and organizational components.

How should we do this project? Good mission artifacts say more than "Do it." In addition to defining the project's objectives and helping management decide whether to proceed, the artifacts need to provide, at the least, a broad outline of *how* to proceed—giving information on items such as architectural considerations, gross project size (including function points, lines of code, or indications of work effort), major milestones, and estimates of resources needed. A plan of action serves two purposes: It gives the follow-on team a direction and it forces the feasibility team into thinking about critical implementation issues early in the project.

Words such as mission, vision, goals, objectives, requirements, and theme apply to many of the concepts discussed in this chapter. I have cited different authors on what defines a mission, but I will not attempt a single-sentence definition because I believe a single sentence, or even a single document, isn't broad enough to encompass the concept of mission. However, by identifying the basic elements, the artifacts, of a mission, I show that a project's mission then becomes an amalgamation of these artifacts. My intent is not to prescribe exact mission artifacts for each and every project or organization, but to describe *types* of artifacts from which each project's needs can be determined.

"Visions are aesthetic and moral—as well as strategically sound. Visions come from within—as well as from outside. They are personal—and group-centered. Developing a vision and values is a messy, artistic process. Living it convincingly is a passionate one, beyond any doubt."
—T. Peters [1987], p. 486.

The most important mission artifacts are the project vision (charter), the project data sheet (PDS), and the product specification outline (PSO). They are defined briefly below, are illustrated in Fig. 3.1, and then they are discussed more fully in named subsections.

The Project Vision (Charter): The charter provides a short, two-to-ten-page definition of key business objectives, product specifications, and market positioning.

The Project Data Sheet: As originated by my colleague Lynne Nix, this staple of her seminars is a single-page summary of key business benefits, product specifications, and project management information. The sheet serves as a document to help focus the project team, management, and customers.

The Product Specification Outline: The outline inventories the features, functions, objects, data, performance, operations, and other relevant specifications of the product at a high level.

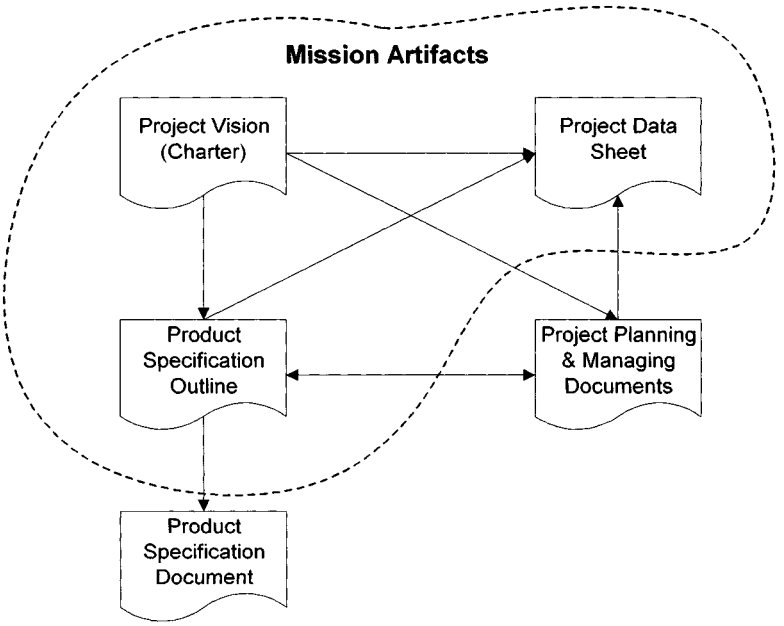


Figure 3.1: Mission Artifacts.

The Project Vision (Charter)

The project vision is recorded in a document, or artifact, which establishes a focus for the project and identifies the foundation on which to build the team's commitment. It establishes which direction to take into the fog of the unknown. It provides boundaries for the exploration phase of the Speculate–Collaborate–Learn Adaptive Development Model Life Cycle. Depending on the situation, a project vision could be

stated as a single sentence or a multi-page document such as a project charter or a project feasibility study report. The specific categories of information within the vision document will be different for each organization, for each size project, and for each release of a product.

Most projects have existing historical information developed by clients, marketers, or developers. This information may range from a detailed market research study in a product company, to a project proposal and cost/benefit analysis put together by an internal client, to previous work done on a project by the IT group. The historical information is a starting point for answering questions such as,

- Who are the customers for the product? What are their needs and how will this product benefit them?
- Is the project a subsequent version of an existing product or a totally new product?
- How much time do we have for this project? What is the trade-off between time and value?
- Do similar products exist within our organization? What can we learn from them about what to do or not do?
- Is there competition for this product? Who is the competition and who are its customers?
- What does our organization expect to achieve by completing this project? What is the value of this project?
- Where does the project fit into the "big picture"? Are there dependent projects?

A vision document should contain a short product-capability statement. This vision statement helps team members pass the elevator test—that is, by using the statement, they can explain the purpose of the project within two minutes. As formulated by Geoffrey Moore, the elements of the elevator test are

- *For* (target customer)
- *Who* (statement of the need or opportunity)
- *The* (product name) *is a* (product category)
- *That* (statement of key benefit—that is, compelling reason to buy)

- *Unlike* (primary competitive alternative)
- *Our product* (statement of primary differentiation)

—G. Moore [1991], p. 161.

Although the vision statement for an internal product for customers of IT will vary slightly on the above theme, a vision statement for a fictitious, Web-based collaboration tool might be worded as follows:

> *For Fortune 1000 companies' product development groups who need to build innovative products across both geographic and organizational distances, the Adaptive Software* InNovator *product is a Web-based, server-layer collaboration tool that speeds turning creative ideas into innovative products. Unlike other collaboration tools, our product integrates accountability into the too often open-ended creative process.*

By answering the questions posed above, the project team puts together a detailed project vision document, which should contain some subset of the following:

- **Project Background**—Describes the current environment(s) that the project will affect directly or indirectly.
- **Project Vision Statement**—Defines the project vision in 25 to 50 words (the elevator test).
- **Project Scope**—Directly sets the boundaries (resource, schedule, scope) on the project so that it can be done successfully. Also specifies what is *not* included within the application, a detail that is useful in understanding the project's boundaries.
- **Executive Sponsor**—Names who has the greatest stake in the project and identifies who has overall responsibility for a commercial product or for an internal project. This person becomes the sponsor responsible for project costs and benefits.
- **Product Market Positioning**—Describes how the marketing department is positioning this product.
- **Internal and External Customers**—Identifies internal and external customers and how they will use the product in their job.

- **Business Functional Objectives**—Addresses the benefits of the product in terms either of opportunities to exploit or of solutions to current problems.
- **Technical Performance Objectives**—Identifies technical performance criteria and measurements.
- **Project Risks**—Describes the major risks that could adversely impact the outcome of the project.
- **Staffing Requirements**—Identifies the skills and number of staff required to develop the product.
- **Prerequisite/Dependent Projects**—Identifies the project's dependencies on deliverables (such as requirements specifications, architectural constraints, or code modules) from other projects.
- **Constraints**—Identifies limits imposed on the project and outside the project team's control, in the form of staff, budgets, interfaces with other systems, technology, or time.
- **Assumptions**—Identifies all costs, benefits, and situations underlying or having a bearing on the project proposal.

The Project Data Sheet

Where do the project team, other stakeholders in the project, or people with a casual interest in the project go to get a thumbnail sketch of the project? The project data sheet! The PDS is the minimum deliverable from any project initiation activity.

Whatever the detailed contents of a project vision paper, a one-page PDS should also be developed. For some projects, the PDS either may be enough by itself or may need only minimal supporting information in order to constitute a complete mission statement. The PDS captures the essential nature of the project in a simple but powerful fashion.

As the quip "I would have written a shorter letter but I didn't have time" demonstrates, condensing an enormous volume of project information into a single page forces the team to carefully consider and select the most important parts of the project. The very act of sifting through and organizing information helps team members focus on important aspects of the project.

The PDS includes the following details:

- clients/customers
- project objective statement
- features
- client benefits
- performance/quality attributes
- architecture
- issues/risks
- major project milestones
- core team members

The project objective statement (POS) should be specific and short (25 words or less), and it should include important scope, schedule, and resource information. Referring to the vision statement created for the InNovator product, the POS might be "Specify, develop, and prepare for market a new Web-based, server-layer collaboration tool called *InNovator,* by the end of July 2001, for an investment of approximately $500,000."

The Product Mission Profile

The ability to create a mission comes from an understanding of the company's strategic focus—from such dimensions as product leadership, operational excellence, or customer intimacy—and from an understanding of the marketing strategy for a particular product. If, according to Michael Treacy and Fred Wiersema (Treacy95), an entire company, even one as large as Hewlett-Packard or IBM, needs a single strategic focus, surely a product team needs a single focus also. Focusing on customer intimacy does not mean ignoring the other two dimensions; it means concentrating on whatever dimension will offer the company the greatest competitive advantage.

Marketing strategy must be considered in the context of product value. Developing software for a piece of in-flight avionics equipment in which a defect could cause serious injury or death, and developing the next Internet browser pose fundamentally different challenges. The market for each dictates a distinct development strategy. Clearly, these are two unique environments. The first case demands overwhelming excellence in the eradication of defects. No one is going to be overly worried about a few months' delay. This does not mean that *any* schedule will work, but that a reasonable, adequate, "good

enough" schedule is acceptable. In the second case, however, schedule—and a very fast one—is the driver. Defect levels shouldn't be excessively high, but adequate, reasonable, "good enough" defect rates may be acceptable.

The product mission profile is an important tool for documenting focus—a contract between the development group and the executive sponsor or primary customer. While companies focus on strategy, project teams need to focus on the priorities of major product attributes.

Table 3.1 shows a matrix of attributes that give a product its value—its scope (features), delivery schedule, defect levels, and resources (cost, staff, and equipment). The table displays the relative importance of each dimension and provides focus for the development team. The first two columns, Excel and Improve, can contain only one entry each, but the third column, Accept, contains two entries. If the focus of excellence is to ship a product with world-class features, then everything else takes second place. Schedule might be designated as warranting improved performance, but it would be less important than the product's features. (If the schedule slipped seriously, the schedule itself might assume a temporary position of higher priority, but it still would not be the focus of competitive excellence.)

Table 3.1: Product Mission Profile Matrix.

Product Quality Dimensions	Priority Level		
	Excel	Improve	Accept
Scope (Features)	◆		
Schedule		◆	
Defects			◆
Resources			◆

As shown in the table, the Excel column identifies Scope (Features) as the most important characteristic for marketplace success. Given this matrix, trade-off decisions that the team makes during the project would favor feature richness over development speed. The second col-

umn, Improve, indicates the second most important characteristic. Within the constraint specified by the first column, the team tries to improve the indicated characteristic—schedule—as much as possible. In the matrix shown, team members try to improve the schedule unless their action results in unacceptable reductions in features.

An entry in the Accept column indicates that a characteristic can be more variable than the others. If Resources are given an Accept-level priority, a wider variation in staffing or cost would be acceptable. The Accept column indicates the characteristic needs to be "good enough" and is the first considered when any trade-off is required. In the example shown in the table, the team and the product's sponsors have agreed to expend additional *resources* or accept higher *defect* levels (obviously within limits) if necessary to meet schedule and feature requirements. "Good enough" does not give team members carte blanche. An *acceptable* level needs to be defined. For example, the budgeted cost for a project might be $250,000. If cost were indicated as a priority in the Excel column, $250,000 would be a maximum amount and the team would strive to spend less, or at least no more, than this limit. If cost were indicated as an entry in the Accept column, the "good enough" range might be plus or minus $50,000.

Every project team has to adapt to external and internal change during a project's life by making trade-off decisions. These decisions are difficult because of each team member's natural tendency to try to excel in all dimensions. Without explicit priorities, the executive sponsor has to be involved in most change decisions, consuming valuable project time. Explicit trade-off priorities give the team members a basis for understanding how various factors relate to each other so they can make the decisions themselves.

In putting a mission profile matrix together, the team must stay within the bounds of what is reasonable and feasible. Marking a dimension as acceptable and then creating extreme acceptability measures clearly defeats the purpose.

It would be unusual for a complex project to have its highest priority in either the resource or defect dimension. If the focus is on low defect levels, it is improbable the schedule focus could be anything but acceptable. Extremely low defect rates and high speed are generally not compatible. High speed and acceptable defect rates, however, are.

The Product Specification Outline

A short focusing statement such as the project objective statement or vision statement is very helpful to the project team and other involved parties, but it is not sufficient to properly determine scope and size or to understand the product. There must be a level of product specification more detailed than the 25-word project objective statement and yet less detailed than a traditional specification document. This intermediate-level document is called the product specification outline.

The PSO serves several purposes. First, it provides the stakeholders and core project team members with a reasonable understanding of the boundaries and scope of the development effort. It is important to specify what is included and, sometimes even more importantly, *ex*cluded from the product. New projects often suffer from unbounded expectations. Developers and customers see the new product as solving all the old nagging problems. Expectations need to be tempered with a dose of reality.

Second, the PSO is the baseline for size estimation. Whether the team is developing a work-breakdown structure and estimating each feature or using a tool such as function-point estimation, a reasonable project size is necessary for rational project planning. Reasonable sizing requires an understanding of at least an outline level of project requirements.

Third, the PSO facilitates adaptive cycle planning, which is covered in greater detail in the next chapter, and is accomplished by assigning product features to specific cycles (similar to project milestones). In order to develop these plans, the basic features or functions of the product need to be identified. Whereas purely task-based planning can be accomplished (although it is not recommended) by a cursory analysis of the requirements using a task-list template or methodology from prior projects, *adaptive* planning requires knowledge of the features of the application to be built. Outline specifications provide the required information.

The specification outline's primary objective is to define the features or functionality of the product. For the purposes of creating a specification and subsequently for planning iterative cycles, I use the term "component" to indicate a set or group of features. Although similar to the concept of component in object-oriented development approaches, here the term is used to define a group of things (objects, business features, the graphical user interface (GUI), or containers, for

example) that are planned and implemented together rather than in the more restrictive sense of components as groups of objects. I use the words "component" and "feature" (a component is a set of features) because Adaptive Software Development does not depend upon, or specify, particular software engineering techniques such as object development, data flow diagrams, or entity relationship data models. Adaptive Software Development is a management approach to delivering software products; it is not a specific development approach.

Three component types are used in ASD—primary, technology, and support. Primary components deliver functionality to the customer. A list management component in a spreadsheet program would contain a number of features to implement that functionality. In a business application, a component contains the functions and data required to implement some business process such as generating orders or producing warehouse stock reports. A primary component could be represented by a process in a data flow diagram or by a use case in an object-oriented analysis document.

Technology components—networks, computer hardware, operating systems, and database management software—are those on which the primary components are built. Often, many of the technology components are already in place and only need to be used by the development team. However, if the technology component is not already installed, it must be identified and installed as one of the project team's responsibilities. For example, if new database software is part of the project, its installation is identified as a technology component.

Support components include everything else, from data models to conversion programs to training materials.

In *Dynamics of Software Development* (McCarthy95), Jim McCarthy provides definitions, paraphrased below, that could be incorporated into a PSO for organizing features:

- *strategic features* are centered around fundamental choices such as operating systems and hardware platforms
- *competitive features* respond to features the competition had or might implement
- *customer satisfaction features* are those frequently requested by customers
- *investment features*, usually of an architectural nature, are those offering long-term benefit, but not much short-term benefit

- *paradigmatic features* are those that change the way people work and therefore have significant competitive implications

Product specification outlines are used in initial estimating and planning, but they may not be detailed enough for development. As each development cycle begins, the team will need to decide how much additional detail is required. In *Rapid Development* (McConnell96), Steve McConnell discusses how specification outlines grow into the *minimal specifications* necessary for the project, a topic germane to the minimalist documentation approach needed for extreme projects.

As McConnell points out, detailed specification writing is often wasted effort because rapid changes occur during the project, making the specs obsolete. However, there are also potential problems with minimal specifications, particularly if the team uses the minimalist concept as an excuse to avoid a seemingly tedious activity. Developing specifications is extremely important, but the specs do not need to fill thousands of pages in order to be effective. Writing a useful spec is another one of the balancing acts required for extreme projects.

Share Mission Values

Documenting mission artifacts is a mechanical task. Creating the shared values that infuse meaning and purpose into the artifacts requires interpersonal effort. Sharing a mission—understanding the subtleties, agreeing on meaning, generating passion—doesn't happen in a two-hour meeting. With the team's concerted effort and patience, shared values grow over time. The core team needs to be aligned, but other stakeholders at various levels of the organization also need to be brought in sync—a formidable task, for a mission is like a magnetic field, with alignment based on its strength of shared values, as depicted by the three variations in Fig. 3.2.

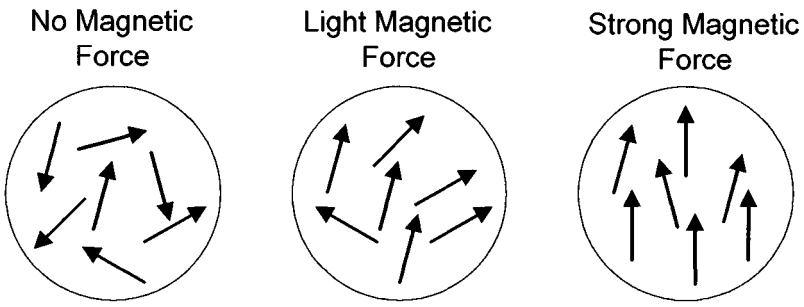


Figure 3.2: Shared Mission Values Show a Magnetic Alignment.

Too many managers and staff think *sharing a mission*—aligning values—means getting together for a couple of hours and jointly writing a mission statement on a flip chart. Unfortunately, sharing a mission is much more complicated than merely writing down a list of values. Its difficulty can be likened to the experience of veteran rock climbers who are always admonishing less experienced climbers to *use their feet*. The reminder is a kind of miniature mission statement. The intent—to get the climber to use leg and core body muscles rather than arm muscles—is correct, but hearing it and understanding it are miles apart. In rock climbing, mastering very precise foot placement, initiating body movement from the legs, and directing momentum through the hips rather than pulling with the arms takes hours and hours and hours of practice. Getting a team to share a mission is equally challenging.

How does an $8-billion company turn virtually on a dime? Of course! Its management writes a new mission statement and off it goes in a new direction! As absurd as it sounds, that is just what Microsoft did in early 1996 (Rebello96). Within a six-month period, the Internet went from being a sideline to the main line at Microsoft. With most 15,000-employee companies, such a radical change would have taken much, much longer—if it happened at all.

What makes Microsoft different? First and foremost is the Microsoft employee's trust in Bill Gates and others in leadership positions—a trust that the mission moves the organization in the right direction, and that, while difficult, it is still achievable. There is a direct correlation between trust in one's management and commitment to a mission.

A second condition present at Microsoft in 1996 was the staff's sense of shared purpose or values. If Gates had outlined a mission to move into the hog-feeding business, the staff's response might have been different. Good mission statements need to tap into the broader corporation's shared values as well as those of the development team.

Teams don't jell overnight—they jell over a period of time. During the process, each team member must ask questions such as

- Does this project's mission connect to something I consider purposeful?
- Do I understand what to do?
- Do I trust the leadership and the other team members enough to collaborate, rather than always insist on "my way"?
- Can I develop a sense of excitement about and commitment to this project?

Not every project manager has the leadership credentials of a Bill Gates. But there are certain attributes that bring a mission to life for a project team:

- The core team works on developing the mission components together. Refining the mission, or reshaping people's perception of the mission, is viewed as a *continuous* activity, not just a checklist item at the beginning.
- The project data sheet is prominently displayed on wall charts in the team area along with documents related to the project. The display of important project information is intended as a reminder, not as a slogan. While slogans attempt to convince the audience of something, the wall charts help remind it of a vision already agreed to.
- When decisions must be reached or controversies need resolution, the mission components are consulted. In the heat of a project, controversy can escalate. Often, the resolution begins by reviewing the mission statement in order to identify an area of agreement that can be used to initiate more constructive discussion.

"[A] cumulative defect removal efficiency of 95 percent appears to be a powerful nodal point for software projects. . . . [These projects] have the shortest schedules . . ., the lowest quantity of effort . . ., and the highest level of user satisfaction. . . ."
—C. Jones [1992], pp. 166, 167.

Quality

Software development teams cannot successfully share values related to mission statements without discussing quality. Few individuals strive to accomplish poor-quality work, yet most individuals are frustrated by the quality of products produced by their organizations. It seems there is always *someone else* whose work degrades *our* quality.

Despite all the words written and delivered in seminars and at conferences about the issue of assuring quality, there are still significant problems in practice, such as the following:

- failure to differentiate among perfection, excellence, and "good enough"
- failure to acknowledge that defects are only one aspect of a product's value
- failure to develop a shared understanding of a particular product's expected quality
- failure to understand that quality is an emergent property that is often ambiguous and uncertain
- failure to understand the unique characteristics of software that negate application of practices used in other functional areas such as manufacturing

For something so deeply valued in our software development culture, quality remains hard to define. For me, the most useful definition of quality is the following:

> *Quality is value to some person.* —G. Weinberg [1992], p. 7.

The simplicity of Weinberg's definition belies its depth. Two critical issues are suggested. The first is that quality is multidimensional. Many software engineers consider defect levels to be the sole dimension on which to measure quality, but Weinberg's use of the word *value* indicates a definition of quality that is broader, to be measured in terms of schedule, scope, defects, or even aesthetics. A second issue raised by the definition is whether quality is an intrinsic characteristic or is dependent on a viewer's perception of the product. Too many definitions of quality leave out consideration of how people are affected by the product and, therefore, are sterile and lifeless.

Many software engineers are unsettled by the idea of value as a perception of the beholder. Their analytical nature wants quality to be intrinsic. One of the most telling arguments for quality as a *perceived* property is the deep emotion it stirs. By defining quality in an analytical way, by reducing it to charts and numbers and processes and rules, we lose passion—the very motivator needed to accomplish it.

The current state of software quality management practices often is governed by the phrase, *Do it right the first time.* But in a complex environment, *Do it right the first time* is a recipe for failure. In essence, it says,

- We can't be uncertain.
- We can't experiment.
- We can't learn from mistakes (because we shouldn't make any).
- We can't deviate from plan.

In the early stages of a project, if the delivery time horizon isn't too far out, we may be able to speculate on what the *generally correct* direction is, but defining a single "right" borders on fantasy. Even if we *could* define right, doing it the first time makes no sense for anything other than trivial products. "The first time" assumes we understand the cause and effect, the specific algorithm of getting to the final product from our initial starting position, and the needs of all stakeholders. It says we know it all. What we need is to be willing to *do it wrong the first time* in order to *get it right the last time.*

Several writers have addressed this multidimensional view of quality, most notably James Bach. Bach's terminology set off a minifirestorm of reaction: He used the term *good-enough software development* as a tag line (Bach95). The irony is that "good enough" seems to indicate a compromise position, one of settling for less than the best. It offends many developers whose value systems seek perfection rather than quality.

"The rule 'I must be perfect' is common enough in the general population, but among people attracted to software engineering, it's close to universal."
—G. Weinberg [1994], p. 212.

To me, "good enough" means the combination of attributes providing the *best total value* in a complex environment. With the myriad combinations and permutations of value attributes—scope, schedule, defect levels, and resources—there can never be an optimum value. Not only is the value landscape vast, but competitors are constantly altering its features. "Good enough" suggests synthesis rather than compromise. It does not mean settling for average, but advocates

delivering the best mix of attributes in a given competitive situation. If software development is to be considered a true engineering discipline, then balancing—not compromising—conflicting quality values is part of building high-quality products.

Value is related to use. For example, with a piece of software slated to run medical equipment that could injure or kill a patient, the *value* goal would be *low defects,* no matter how much time the development and test phases take. However, the same value standard would not be applied to a sales application needed for product planning.

The four broad categories of software quality in Fig. 3.3 are the same ones used in the product mission profile matrix (Table 3.1): scope, schedule, resources, and defects. Many project management approaches use only the first three categories, but I include defects as a major category because of the emotional reaction many people have to the issue of quality and the tendency to consider defects to be synonymous with a lack of quality.

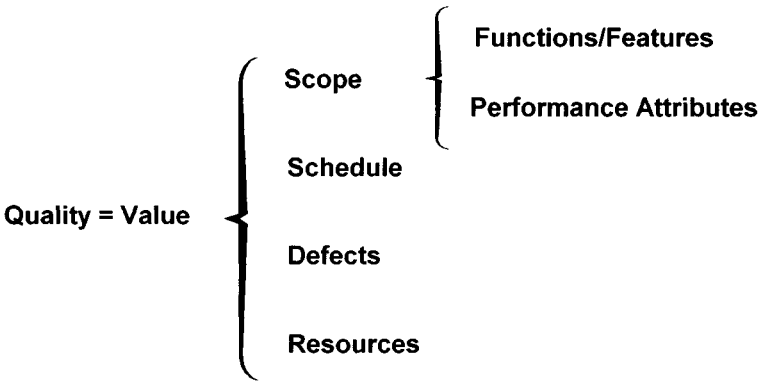


Figure 3.3: Dimensions of Quality.

The belief that a product with zero defects can be achieved is a myth left over from the manufacturing roots of most quality assurance initiatives. In manufacturing a thousand widgets per hour to a statistical quality tolerance of +/−.001 inches, a plant that can manufacture so that no widget exceeds the tolerance would be considered to have a zero-defect level. Would the narrow tolerance have any real meaning if the best machine tools in existence could only produce to +/−.1

inches? Of course not, yet we openly embrace equivalent nonsense when it comes to software.

Depending on what they are designed to do, software products have different tolerances for defects. The tolerances for an operating system, a sales forecasting system, a medical equipment control system, and a 3-D adventure game are very different. Many organizations resist this idea of *tolerance* for software defects, and have little clue as to how many defects their software actually contains. It is interesting to me that the better software developers usually know more about their defects and which ones they are shipping to customers than the developers (and companies) who claim a belief in zero defects know about the defects they ship.

Evaluate the Mission Every Day

Creating a sense of shared values, a sense of a shared mission, requires constant attention by the team members. In the early stages of the project, there may be broad ideas about the feature set, but the details are off in the fog. I once ran a six-month, twenty-plus-person project with a reasonably stated overall mission, but fuzzy details. The project plan contained about eighteen tasks, which we never looked at after the first week. But every single day, in some fashion, the team discussed the end goals, the contents of the final product, and what the product's features might look like. Each team member had "to-do" lists according to his or her own responsibilities but, from an overall management perspective, meeting the cycle dates and refining the component definitions were the main focal points.

In complex projects, no one *completely* understands the final deliverables until they are ready for shipment or installation. The question in everyone's mind, every day, is, How can we improve our understanding of what we are supposed to produce? Shared understanding itself is not a goal, but an ongoing process of interaction and relationship-building.

*"**Great Groups ship.** Successful collaborations are dreams with deadlines."*
—W. Bennis and P. Biederman [1997], p. 214

Focus on Results

The four steps that define the recipe for success on a complex project are: Define the result; Define the quality criteria to measure the result; Create a shared understanding of the destination; and Let the results

evolve in an emergent fashion. A feature team of six people who have their own ideas about what constitutes quality and who are driven to perfection is a recipe for disaster. The same group, driven by a vision and guided by a focus on excellence—that is, a group that understands the pitfalls of perfection—will still be witness to colossal arguments and bruised feelings, but will stand a better chance of shipping a great product. Great groups, and particularly *leaders* of great groups, understand the difference between perfect, excellent, and "good enough."

One of the critical success factors in mountaineering is the climber's ability to focus intensely—on the next step, then on the next, and so on. I've been in situations in which I had to talk myself into pushing ahead. Counting each crunch in the glacier ice, I more than once have felt overwhelmed by the need to take even one hundred steps. Reaching one hundred, my next goal was another hundred—thinking about getting all the way to the summit was just too daunting. By playing games with my mind, I could fool myself into taking the climb step by step. In climbing, just as in product development, you know one hundred steps isn't the real goal, but it provides relief—a small accomplishment to battle the fatigue and the little voice saying, "Go back; you've done enough. It was a valiant effort. The top isn't really so important anyway." Good mission statements help team members battle the doubt, indecision, and despair that is part of any project.

In the beginning, a mission statement is purposefully broad, but as the project's ship-date approaches, mutual understanding narrows the gap so a product can be shipped. A mission is like a mountain, broad at the start, increasingly narrow at the top. The more extreme the software project, the more crisply defined and visual the focal point must be as the end approaches. Head-down, fur-flying ship-mode is not the time to be uncertain about the goal.

Quality is not about time—software archaeology is littered with the bones of very long projects of very poor quality. Quality is about setting the correct mission and the correct objectives. It is about understanding the constraints, managing trade-offs, and displaying courage. If the objective is to produce a certain feature set in three months, and there are no specified criteria for the software to integrate with other applications, then labeling the result as poor quality because of poor integration is spurious. If the plan is to monitor quality through the use of technical reviews and they are bypassed because of time con-

straints, then someone has either made a valid trade-off decision or lacked the courage to take the time needed.

Much of the outcry about poor quality is based on the deterministic view that quality can be predicted and planned, and that specific processes can be put in place to achieve it. In reality, quality is an emergent property, an ever-shifting position on a fitness landscape. Until we learn to view quality not as a point in quality space but as shifts along a continuum of unfolding possibilities, the arguments will continue to focus entirely on the wrong issues.

Summary

- Thomas Jefferson's mission statement to the Lewis and Clark expedition was one of the most famous in history. It combined a specific goal, boundaries of behavior, and wide latitude for implementation.
- A mission statement needs to be focused. Attempting to excel in multiple dimensions usually results in a product being mediocre in all of them and excellent in none.
- The product mission profile forces a focus on the single area—features, schedule, defects, or resources—in which the development team needs to excel. This profile provides the high-level, trade-off strategy for the project.
- A mission statement facilitates collection of information that is relevant to the project's desired result.
- A mission should establish direction, inspire the participants, and provide enough detail for ongoing decision-making.
- The components of mission are a project vision (or charter), a project data sheet, and a product specification outline.
- Writing a mission statement is easy. Creating a sense of shared responsibility for achieving the mission is very difficult. Building a shared vision is an ongoing, never-ending, collaborative team effort.
- The ability to periodically de-focus is important.

- The project vision or charter establishes the focus and key motivational theme for the project. It establishes which direction to take into the fog of the unknown. It provides the boundaries for the exploration phase of the Speculate–Collaborate–Learn life cycle.
- The project data sheet is a one-page summary of the key information about the project. It serves as a focal point and quick reminder of the most important elements about the project. It is simple, but powerful.
- The product specification outline describes the features of the product in enough detail so that developers can understand the scope of the effort, create a more detailed adaptive cycle plan, and estimate the general magnitude of the development effort.
- Quality characteristics are part of the mission definition. Software developers often fail to differentiate between excellence and perfection. They also fail to provide a basis for making necessary trade-offs during a project.
- Quality is in the eye of the beholders. In Jerry Weinberg's words, "Quality is value to some person."

This page intentionally left blank

Index