Agile Software Development Ecosystems
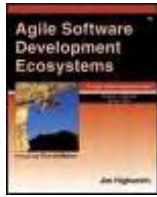
Foreword by **Tom DeMarco**

Jim Highsmith

Agile Software Development Ecosystems

By Jim Highsmith

Publisher :	Addison Wesley
Pub Date :	May 26, 2002
ISBN :	0-201-76043-6
Pages :	448

In a highly volatile software development environment, developers must be nimble, responsive, and able to hit a moving target--in short, they must be agile. Agile software development is designed to address this need for speed and flexibility. Agility describes a holistic, collaborative environment in which you can both create and respond to change by focusing on adaptability over predictability, people over process. Agile software development incorporates proven software engineering techniques, but without the overhead and restrictions of traditional development methodologies. Above all, it fulfills its promise of delivering software that serves the client's business needs.

Written by one of the leaders of the Agile movement, and including interviews with Agile gurus Kent Beck, Robert Charette, Alistair Cockburn, Martin Fowler, Ken Schwaber, and Ward Cunningham, *Agile Software Development Ecosystems* crystallizes the current understanding of this flexible and highly successful approach to software development. It presents the key practices of all Agile development approaches, offers overviews of specific techniques, and shows how you can choose the approach that best suits your organization.

This book describes--in depth--the most important principles of Agile development: delivering value to the customer, focusing on individual developers and their skills, collaboration, an emphasis on producing working software, the critical contribution of technical excellence, and a willingness to change course when demands shift. All major Agile methods are presented:

- Scrum
- Dynamic Systems Development Method
- Crystal Methods
- Feature-Driven Development
- Lean Development
- Extreme Programming
- Adaptive Software Development

Throughout the book, case stories are used to illustrate how Agile practices empower success around the world in today's chaotic software development industry. *Agile Software Development Ecosystems* also examines how to determine your organization's Agile readiness, how to design a custom Agile methodology, and how to transform your company into a truly Agile organization.

Table of Content

Copyright

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

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

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

Dedication

To Wendie

Foreword

The 1990s were the Decade of Process for IT. We prostrated ourselves before the CMM and ISO. We sought not just perfection in the way we built software, but total predictability. We concluded that it wasn't enough just to do things right; we also had to "call our shot"—to say in advance exactly what we intended to do, and then do exactly that. Nothing more and nothing less. We were determined (in CMM terms) to "plan the work and work the plan."

A big part of our process focus in the 1990s had to do with our obsession with all things Japanese. Think back to 1990 and remember your own take on the subject at that time. There was a general malaise and a feeling that the West had lost its momentum and the Japanese had a lock on the future. After all, they worked harder, stayed later, had far more rigorous discipline, and were regularly achieving levels of quality that the rest of the world could only dream of. The Japanese phenomenon, the relentless advance of the "Pacific Tiger," was all the talk that year. If we're going to survive, we thought, we had better become more like the Japanese. And so we moved to a more Japanese kind of rigor and process.

But the 1990s were not kind to Japan. By the end of the decade, the Japanese economy had been in a slump that was as bad as, and had lasted as long as, the Great Depression. Somehow, hard work, discipline, efficiency, and rigor—the very qualities that had been essential in the 1980s—were not a winning combination for the 1990s. What mattered in the 1990s was being able to turn on a dime. The Internet changed everything, and only those who were ready to change quickly with it would prosper. Agility: 1, everything else: 0.

Similarly, the process obsession did not stand its adherents in very good stead. The list of companies most successful at climbing up the CMM ladder early in the decade reads like a Who's Who of downsizing by the end. Process rigor was simply not the right recipe for an era in which everything was changing.

Predicting in advance what your every step would be ended up seeming like a dumb obsession. What sense does it make to predict your steps in advance when you're not even sure where you're headed?

Today the era of fat process is over. As Jim Highsmith has said, "Thin is in." To optimize for speed and responsiveness, we need to put process on a diet. We need to shed paperwork and bureaucratic burden, eliminate endless code inspections, and invest in our people so they can steer themselves sensibly through the chaotic maze that is a modern-day IT project. This is the genesis of the Agile approaches to software development.

Jim Highsmith has put all this together into a kind of survey introduction to the Agile methodologies. He has had the good sense not just to present this as a discussion of a concept, but to tell it as a story. As in any good story, it is the people who matter most. And so he tells us about Agile approaches by telling us about their principal advocates and inventors, people like Kent Beck, Alistair Cockburn, Martin Fowler, and the other "light methodologists." These are the minds that are changing how IT gets done. Their story is what *Agile Software Development Ecosystems* is all about.

Tom DeMarco
Camden, Maine

Preface

From February 11 to 13, 2001, at the Lodge at Snowbird ski resort in the Wasatch Mountains of Utah, 17 people met to talk, ski, relax, and try to find common ground. What emerged was the Agile Software Development movement. Representatives from Extreme Programming (XP), Scrum, the Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Crystal Methods, Feature-Driven Development (FDD), Pragmatic Programming, and others sympathetic to the need for an alternative to document-driven, rigorous software development processes convened. What this meeting produced—*The Manifesto for Agile Software Development,* signed by all 17 of the participants—was symbolic. It declares that:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.

This value statement has a number of fascinating aspects, not the least of which was getting 17 people to agree to it. Although this was a group of experienced and recognized software development "gurus," the word *uncovering* was selected to indicate that the authors don't have all the answers and don't subscribe to the silver-bullet theory.

The phrase "by doing it" indicates that the authors actually practice these methods in their own work. Ken Schwaber told the story of his days of selling tools to automate comprehensive, "heavy" methodologies. Impressed by the responsiveness of Ken's company, Jeff Sutherland asked him which of these rigorous methodologies he used for internal development. "I still remember the look on Jeff's face," Ken remarked, "when I told him, 'None. If we used any of them, we'd be out of business.'" The authors want to help others with Agile practices and to further our own knowledge by learning from those we try to help.

The value statements have a form. In each statement, the first segment indicates a preference, while the latter segment describes an item that, while important, is of lesser priority. This distinction lies at the heart of agility, but simply asking people to list what's valuable doesn't flesh out essential differences. Roy Singham, CEO of ThoughtWorks, put it well when he said that it's the edge cases, the hard choices, that interest him. "Yes, we value planning, comprehensive documentation, processes, and tools. That's easy to say. The hard thing is to ask, 'What do you value *more*?'" When push comes to shove—and it usually does—something must give, and we need to be clear about what stays and what gives.

The first value statement recognizes the importance of processes and tools but stresses that the interaction of talented individuals is of far more importance. Rigorous methodologies and their business process reengineering brethren place more emphasis on process than people. Agile development reverses this trend. If individuals are unique, and we believe they are, then processes should be melded to individuals and teams, not the other way around.

Similarly, while comprehensive documentation is not necessarily harmful, the primary focus must remain on the final product—working software. This means that every project team needs to determine, for itself, what documentation is absolutely essential to delivering working software. Working software tells the developers and sponsors what they really have in front of them—as opposed to promises of what they *might* have in front of them. Working software can be shipped, modified, or scrapped, but it is always *real*.

Contract negotiation, whether through an internal project charter or an external legal contract, isn't a bad practice, just a seriously insufficient one. Customers want a product that conforms to their needs—at the time of delivery. "Contract negotiation encourages the addition of buffers [contingency]," says colleague Ron Holliday. "This makes projects take even longer, drives up costs, and reduces responsiveness to change. A collaborative arrangement is a team effort between customer and supplier to deliver the best possible solution." Contracts and project charters provide boundary conditions within which the parties can work, but only through ongoing collaboration can a development team hope to understand and deliver what the client wants.

No one can argue that following a plan is a good idea—right? Well, yes and no. In a world of business and technology turbulence, scrupulously following a plan can have dire consequences, even if it's executed faithfully. However carefully a plan is crafted, it becomes dangerous if it blinds you to change. As you will see in several of the case studies presented in this book, few, if any, of the projects delivered what was planned for in the beginning. And yet they were successful because the development team was Agile enough to respond again and again to external changes. In the Information Age, planning is important, but accepting that deviations from any plan are "normal" is even more important.

The meeting at Snowbird was incubated at a spring 2000 gathering of Extreme Programming leaders organized by Kent Beck. At that meeting in Oregon, a few "outsiders" but "sympathizers," such as myself, were invited, and attendees voiced support for a variety of "light" methodologies. During 2000, a number of articles referenced the category of "light" or "lightweight" practices. In conversation, the soon to be "Agile" authors didn't like the moniker "light," but it stuck at that time.

In September 2000, "Uncle Bob" Martin of Object Mentor in Chicago started what was to become the Agile ball rolling with an email: "I'd like to convene a short (two-day) conference in the January to February 2001 time frame here in Chicago. The purpose of this conference is to get all the lightweight method leaders in one room. All of you are invited, and I'd be interested to know who else I should approach." Bob set up an online group site, and the discussions raged. Early on, Alistair Cockburn weighed in with an epistle that identified the general disgruntlement with the word "light": "I don't mind the methodology being called light in weight, but I'm not sure I want to be referred to as a lightweight attending a lightweight methodologists meeting. It somehow sounds like a bunch of skinny, feeble-minded lightweight people trying to remember what day it is."

The fiercest debate was over location! There was serious concern about Chicago in wintertime—cold and nothing fun to do. Someone suggested Snowbird, Utah—cold, but fun things to do, at least for those who

ski on their heads, as Martin Fowler tried on the first day. Someone else mentioned Anguilla in the Caribbean—warm and fun, but time consuming to get to. In the end, Snowbird and skiing won out.

The Agile Manifesto value statements represent a deeper theme that drives many, but not all, of the Manifesto's authors. At the close of the two-day meeting, Bob Martin joked that he was about to make a "mushy" statement. Although tinged with humor, no one disagreed with Bob's sentiments—that we all felt privileged to work with a group of people who held a set of compatible values, based on trust and respect for each other, promotion of organizational models based on people, and the desire to build collaborative communities in which to work. At the core, I believe Agile developers are really about mushy stuff—about delivering products to customers while operating in a vibrant, sustaining workplace. So in the final analysis, the meteoric rise of interest in—and criticism of—Agile Software Development is about the mushy stuff of values and culture.

For example, I think that, ultimately, XP has mushroomed in use and interest not because of pair programming or refactoring but because, taken as a whole, the practices define a developer community freed from the baggage of Dilbertesque corporations. Kent Beck tells the story of an early job in which he estimated a programming effort to be six weeks for two people. After his manager reassigned the other programmer at the beginning of the project (effectively halving the resources), Kent completed the project in twelve weeks—and felt terrible! The boss, of course, harangued Kent throughout the second six weeks. Somewhat despondent because he was such a "failure" as a programmer, Kent finally realized that his original estimate of six weeks was extremely accurate—for two people—and that his "failure" was really his manager's failure to take the responsibility for removing project resources. This type of situation goes on every day with individuals who don't want to make hard tradeoff decisions, so they impose irrational demands.

The Agile movement is not anti-methodology; in fact, we seek to restore credibility to the concept of *methodology*. We want to restore a balance. We accept modeling, but not in order to file some diagram in a dusty corporate repository. We accept documentation, but not hundreds of pages of never-maintained and rarely used tomes. We plan, but recognize the limits of planning in a turbulent environment. Those who would brand proponents of FDD or Scrum or any of the other Agile approaches as "hackers" are ignorant of both the approaches and the original definition of the term hacker—a programmer who enjoys solving complex programming problems, rather than one who practices either ad hoc development or "cracking." Agile Software Development incorporates proven software engineering techniques, but without the overhead of traditional methodologies.

The Agile Alliance was born in early 2001, but the history of the various approaches and the people who developed them goes back 10 to 15 years. This book describes these practices and the principles behind them, but more important, it delves into the people—the people who are developing the practices and the people who use them to deliver business value to their customers.

Finding a Balance

The left side of the Agile Manifesto value statements indicates what Agilists consider *more important* than the items on the right side. This should not be construed as indicating that tools, process, documentation, or contracts are *unimportant*. There is a tremendous difference between one thing being more or less important than another and being *un*important. Judicious use of tools is absolutely critical to speeding software development and reducing its costs. Contracts are one vital component to understanding developer-customer relationships. Documentation, in moderation, aids communication, enhances knowledge transfer, preserves historical information, and fulfills governmental and legal requirements.

But Agilists take a certain *perspective* on these topics. Try this exercise. For each of the Manifesto value statements, construct two questions along the lines of the following:

- Could we have a successful project by delivering documentation without working software?
- Could we have a successful project by delivering working software without any documentation?

By looking at these two end points, we can better grasp relative importance. Although there has to be a balance—documentation *and* working software, contracts *and* collaboration, responsiveness *and* planning, people *and* process—we have to delineate the extremes, the end points, so that organizations, teams, and individuals can find their own balance points. If we start out trying to find the middle ground, we never will.

One of the great contributions of XP's "extremoes"—Kent Beck, Ron Jeffries, and Ward Cunningham—is that they have staked out positions that have stirred debate in ways that taking moderate positions never would. When Ron says, "Great designs emerge from zero anticipatory design and continuous refactoring," he is challenging himself, and us, to rethink our assumptions about software development. We have to understand the limits before we can understand the balance points.

So although I realize the value of documentation, contracts, plans, and processes, there are numerous sources of material about these subjects. My intention is to identify and define Agile Software Development, to articulate its practices and principles, so you can make your own decision about where on the spectrum you, or your organization, need to be.

Fundamental Questions

There are three fundamental questions that this book answers: (1) What kinds of problems does agility solve best? (2) What is agility? and (3) What are Agile Software Development Ecosystems (ASDEs)?

What Kinds of Problems Does Agility Solve Best?

Problems characterized by change, speed, and turbulence are best solved by agility. Some people argue that good practices are good practices (pair programming, customer focus groups, or feature planning, for example) and therefore ASDEs should not be "limited" to a particular problem type. While true in part, the question asks what problems Agile practices *best* solve, not *just* solve. So, while XP, Crystal, or Scrum can surely be used for a wide range of projects, they are particularly relevant to extreme or complex projects—those that have an accelerated time schedule combined with significant risk and uncertainty that generate constant change during the project.

As the level of change caused by rapidly evolving technology, business models, and products increases and the need for delivery speed accelerates, ASDEs' effectiveness increases quickly over rigorous methodologies.

What Is Agility?

Agility, like any other complex concept, has a number of definitions, but for me, the most clearly focused definition is

Agility is the ability to both create and respond to change in order to profit in a turbulent business environment.

Rather than shrink from change, Agile organizations harness or embrace change by being better than competitors at responding to changing conditions *and* by creating change that competitors can't respond to adequately. However, companies must determine what level of agility they require to remain competitive. Agility is only an advantage relative to competitors—a copper mining company doesn't need to be as agile as a biotechnology firm.

Other aspects of agility are also important: nimbleness or flexibility on the one hand, and balance on the other. Agile organizations are nimble (able to change directions quickly) and flexible (able to see how things that worked for them last week may not work as well next week). An Agile organization also knows how to balance structure and flexibility. If everything changes all the time, forward motion becomes

problematic. Agile organizations understand that balancing on the edge between order and chaos determines success.

What Are Agile Software Development *Ecosystems*?

I began writing this book about Agile Software Development *methodologies,* but I kept worrying about the word "methodology" because it didn't fit with the focal points of Agile development—people, relationships, and uncertainty. Furthermore, by using the word "methodology," Agile practices are instantly compared to traditional software development methodologies—thereby using the wrong measuring stick for comparison. So I use the term "Agile Software Development Ecosystem" to describe a holistic environment that includes three interwoven components—a *"chaordic" perspective, collaborative values and principles,* and a *barely sufficient methodology—*and the term Agilists to identify those who are proponents of ASDEs.

Some people think that "Agile" means fewer processes, less ceremony, and briefer documents, but it has a much broader perspective, which is the primary reason for using the word *ecosystem* rather than methodology. Although fewer processes and less formality might lower development costs, they are not enough to produce agility. Focusing on people and their interactions and giving individuals the power to make quick decisions and to self-adapt their own processes are key to Agile ecosystems.

The *American Heritage Dictionary* defines ecosystem as "organisms and their environment: a localized group of interdependent organisms together with the environment that they inhabit and depend on." The *Oxford English Dictionary* extends this definition to include a constant interchange within the system, including both organic and inorganic elements. The word *methodology* conjures up a vision of things— activities, processes, tools. These are not inconsequential elements, but incomplete ones. The word *ecosystem* conjures up a vision of living things and their interactions with each other. Within an organizational context, an ecosystem can then be thought of as a dynamic, ever-changing environment in which people and organizations constantly initiate actions and respond to each other's actions. The word ecosystem focuses us on the dynamic interactions of individuals and teams rather than on the static lines on organization charts.

A Chaordic Perspective

To fully understand ASDEs, we need to understand each of the three components and how they relate to each other. First, Agilists share a view that organizations are *chaordic*—that every organization exhibits properties of both *cha*os and *ord*er that defy management through the use of linear, predictive planning and execution practices. Viewing organizations as chaordic means understanding that the predictability upon which traditional project management and development life cycle practices are built is a root cause of dysfunctionality between customer, management, and development organizations. A chaordic perspective impacts both how we respond to change and how we manage project teams and organizations.

In day-to-day project work, a chaordic perspective creates two outcomes that are 180 degrees out of sync with rigorous methodologies.

- Product goals are achievable, but they are not predictable.
- Processes can aid consistency, but they are not repeatable.

Although ASDEs involve careful planning, the fundamental assumption remains that plans, in a turbulent environment, are not predictable, at least at the level of project scope, schedule, and cost. Plans are hypotheses to be tested rather than predictions to be realized. However, the product goals of the business *are* achievable, in large part because Agile people adapt. They can "adapt" to an articulated vision and a schedule, scope, or cost goal through tradeoffs in the other two dimensions. Second, while process can aid people in working together, in volatile environments the idea of driving out process variation through measurement and correction—statistical process control—becomes an unworkable hypothesis. Changes

that are the result of knowledge gained during the project, knowledge not discernable early in the project, require processes that can respond to change, not ones that attempt to eliminate it.

As Martin Fowler points out, the two fundamental characteristics of ASDEs are their focus on adaptability rather than predictability and on people rather than process (Fowler 2000). As you read through this book, you will see just how fundamental—and challenging to the status quo—these two principles really are. Being Agile means accepting that outcomes are not predictable and that processes are not repeatable. It even implies that as process *repeatability* increases, project *predictability* decreases.

Peter Senge uses the term "mental model" to identify the perspective, set of assumptions, stories, and beliefs that each of us carries in our mind that provide a context for thinking (Senge 1990). In organizations, the collective set of mental models defines an overall cultural context. Companies that are heavily sales oriented differ from those that are heavily engineering oriented. Companies whose driving strategy is customer intimacy differ from those whose driving force is product innovation. Companies whose mental model includes linearity, cause and effect, hierarchy, predictability, and control will operate very differently from one whose mental model includes collaborative networks, emergence, decentralization of power, and acceptance of unpredictability. One is Newtonian, the other chaordic.

A chaordic perspective draws on a complex adaptive systems model in which decentralized, independent agents (each of us) interact in self-organizing ways, guided by a set of simple, generative rules that create complex, emergent results. This perspective is examined in detail in my book *Adaptive Software Development* (Highsmith 2000) and is explicitly embraced in the philosophies of Agilists Ken Schwaber, Bob Charette, and Kent Beck.

XP provides an example of how methodology and culture fit together. At one level, XP is defined by a system of practices—pair programming, test-first development, refactoring, coding standards. But the values and principles of XP define a collaborative culture—how developers work together with customers, how individuals interact and treat each other as human beings.

Collaborative Values and Principles

The second piece of the interconnected web that defines ASDEs is the statement of *collaborative* values and principles. While it is difficult to characterize the Agile Manifesto in one word, "collaborative" seems to be the best single adjective. Values and principles shape the ecosystem. Without a set of stated values and principles, an ecosystem is sterile, reflecting practices but not the people who interact within it.

A collaborative culture includes people and their relationships within a development team and with customers, management, and partnering teams within or external to their own company. Human dynamics, communications, and collaboration may be known as the "soft" sciences, but in practice, they may be the hardest to master. Principles and values help define a culture—the environment in which people want to work.

A Barely Sufficient Methodology

The final component of an ASDE is methodology. The traditional definition of methodology includes things such as roles, activities, processes, techniques, and tools. Alistair Cockburn summarizes these components when he defines methodology as "the conventions we agree to"—the ways in which people work together on a project. In *The Social Life of Information,* John Seely Brown and Paul Duguid (2000) discuss the major differences between process (as used by the business process reengineering movement) and practice. Processes are described in manuals; practices are what happen in reality. Process centrists relegate people to second place; practice centrists place people first. Process centrists focus on explicit (written down) knowledge, while practice centrists focus on tacit (internal) knowledge. The ASDE model provides a practice-centered, rather than a process-centered, approach to methodology.

There are two reasons to pursue barely sufficient methodologies: value and innovation. Streamlined methodologies concentrate on those activities that create value and ruthlessly eliminate non-value-adding activities. Programming usually adds value; process management often adds overhead. Bare sufficiency means keeping the former and eliminating the latter. Second, innovation and creativity flourish in chaordic environments, not orderly ones. Barely sufficient methodologies are cauldrons for breeding innovation.

Methodology also relates to organizational model. Agile methodologies contain minimal processes and documentation and reduced ceremony (formality). Agile methodologies are labeled "barely sufficient" (Cockburn) or "a little bit less than just enough" (Highsmith), or "minimal" (Charette). However, this streamlining of methodology isn't based just on reducing work effort but, more important, it is based on understanding the chaordic world view—one in which emergent (innovative) results are best generated at the "edge of chaos," perched midway between chaos and order.

Practices (or techniques) are the lifeblood of methodology. Whether it's pair programming, Scrum meetings, customer focus groups, or automated testing, the practices of ASDEs, carried out by talented and skilled individuals, produce results.

Changing Perspectives

In the software profession, we've used the words "methodology" and "process" for so long that they roll off the tongue and the pen without trouble. "Ecosystem" will take getting used to, but then, that's why I'm using the word—to foster a different perspective. "There is accumulating evidence that corporations fail because the prevailing thinking and language of management are too narrowly based on the prevailing thinking and language of economics," says Arie De Geus (1997). "They forget that their organizations' true nature is that of a community of humans."

To change thinking, we must change the language we use, so I use the word "ecosystem" to change our thinking about how software projects should be viewed. A *chaordic perspective,* a *collaborative set of values and principles,* and a *barely sufficient methodology* all combine and interact to form an Agile ecosystem. We cannot separate the three, at least in my mind, and I think most Agilists would agree. One could have a streamlined methodology but a linear, Newtonian view of organizations, and the result would not be Agile. One could have a streamlined methodology but a hierarchical, control-based work culture, and it would not be Agile. One could have a collaborative, people-oriented work culture but a rigid, predictive approach to planning and managing projects, and it would not be Agile. Agility requires all three.

The ultimate objective of this book is to describe new ways of working together to deliver business value to software customers. The heart of ASDEs is a core belief in people—their individuality and their interactions. It's impossible to discuss people and their ways of working together (eco system) without discussing values and principles. It's impossible to discuss values and principles without also discussing assumptions about how organizations do, or should, work. It's impossible to compare Agile approaches with non-Agile approaches using "methodology" as the only mechanism for comparison.

In closing, I need to state that I don't speak for anyone in the Agile community other than myself. I may interpret what Ken Schwaber says about Scrum, what Jeff De Luca says about FDD, or what Alistair Cockburn says about Crystal Methods, but they are my interpretations. In making generalizations about ASDEs, I'm sure I've made statements that would generate disagreement from 1 or more of the other 16 authors of the Agile Manifesto. However, I have talked to, corresponded with, or worked with all these authors, so although they are my own interpretations, they also reflect a deep sense of being part of, and contributor to, this community.

Although 17 individuals authored the Agile Manifesto, thousands support this effort. Many, many individuals have signed the Manifesto Web page, and an array of Web sites prominently display the statement "We support the Agile Manifesto." Agilists have stirred a healthy debate within the software development and project management communities. I hope this book will contribute to that debate and encourage others to join in it.

Jim Highsmith
Salt Lake City, Utah
January 2002

Introduction

Agile Software Development Ecosystems contains four parts.

Part I, Problems and Solutions, addresses the key characteristics of our change-driven, Information Age economy and discusses why traditional Rigorous Software Development approaches are insufficient for success in this environment. Chapter 1, The Change-Driven Economy, describes our turbulent economic conditions and explains that the future is unlikely to be less turbulent. Chapter 2 contains a case story about IDX Systems Corporation that illustrates both the turmoil companies face and the results that can be obtained using an Agile Software Development Ecosystem. Chapter 3 outlines in broad brush strokes the solution to the problems raised by speed and change—agility.

Part II, Principles and People, delves into the values and principles that characterize ASDEs and the people, authors, and key thought leaders who created the major ASDEs. Chapters on principles are interwoven with ones on the people who have articulated the principles. What better way to gain a depth of understanding of the Agile values and principles than to discuss them with the key players? These chapters are not organized strictly along the principles of the Agile Manifesto, but by my own categorization. The interviews were not directed at explanations of each Agile approach, but at how the individual's experiences had shaped his understanding of software development.

Scratch below the surface, and you don't need to scratch very far with Kent Beck or Alistair Cockburn or Ken Schwaber or other Agile leaders, and you find individuals committed to making the part of the world related to software development and information technology a satisfying and enjoyable place to work. One can't really understand the Agile movement without understanding the originators' views on working relationships and their deeply held cultural values. The interview chapters are intended to aid in that understanding.

Also in Part II, each principle chapter begins with a case story from an organization that has successfully used one of the ASDEs on a project. These case stories cover a wide range of project types from around the world—New Zealand, India, Singapore, Canada, the U.S., and Germany.

Part III portrays each individual Agile Software Development Ecosystem. The chapters describe the basics of a particular approach and identify key contributions of each one. There are chapters on Scrum, Dynamic Systems Development Method, Crystal Methods, Feature-Driven Development, Lean Development, Extreme Programming, and Adaptive Software Development.

Part IV, Developing an ASDE, discusses how an organization could proceed with analyzing its culture to determine how an ASDE might fit, and then how to design and customize a methodology framework and practices for itself.

Book Organization and Conventions

There are always a myriad of dilemmas in organizing a book. One such dilemma in this book was deciding whether chapters on individual Agile approaches should go first, or whether the chapters on principles should take precedence. I opted for the latter, because ultimately I believe values and principles are more important than practices. If curiosity gets the better of you, skip ahead and read a chapter, or chapters, on the individual ASDEs first. I have also included a paragraph introducing each ASDE, and its key developers, in this introduction.

There are a couple of conventions that I have used throughout the book. First, I think that every software development project is a *product* development project. Over the years I have worked with IT organizations that are developing software for internal customers and software product companies that are developing software to sell in the marketplace. In both cases, the development staff is delivering a product to a customer. Keeping this notion of customers and products in mind helps IT organizations as well as product companies.

Second, when I use the word *developer,* I am referring to all of the people who are directly involved in delivering a product to the customer—programmers, testers, documentation specialists, requirements analysts, and others. I didn't want to refer to four or five roles every time I want to refer to those who deliver working software. Although delivering working software is a key value of Agile development, programming is not the only role required to accomplish that objective.

When I refer to software development or software development practices, I am usually implying some level of project management and collaboration practices in the sentence. Constantly using the phrase "software development, project management, and collaboration" practices takes up half the sentence before saying anything useful, so I will shorten the words, but hopefully not the intent.

I pondered what to call non-Agile approaches and finally decided on using the word "Rigorous," in part because I view it as a nonjudgmental word. Agility means balancing between structure and flexibility, so rigor is a vital part of any development process. Agility focuses on the flexibility side of the definition and rigor focuses on the structure side—thus I've used the two words as contrasting styles of development. Too much structure, and Rigorous becomes rigid. Likewise, too much flexibility, and Agile becomes chaotic. I use the two words—Agile and Rigorous—as a contrast, to emphasize the fundamental characteristics of each. Although I try to be nonjudgmental about their use, my bias toward Agile should be obvious.

Rather than continually use terms like Agile Ecosystem Proponent or Agile Methodologist, I've elected to create the term "Agilist." While far from perfect, using this term cuts down on words that would rapidly become repetitive.

Finally, all the case stories in the book come from my own experiences with clients or from the experiences of other individuals that I interviewed. Some companies and individuals were reluctant to be identified, so I have used pseudo names in some cases. My thanks to all of those "unidentifiables" who provided these stories.

The Major Agile Ecosystems and Leaders

This section identifies each of the major ASDEs and their primary developers and provides a brief synopsis of each of the approaches.

Scrum

Scrum, named for the scrum in Rugby, was initially developed by Ken Schwaber and Jeff Sutherland, with later collaborations with Mike Beedle. Scrum provides a project management framework that focuses development into 30-day Sprint cycles in which a specified set of Backlog features are delivered. The core practice in Scrum is the use of daily 15-minute team meetings for coordination and integration. Scrum has been in use for nearly ten years and has been used to successfully deliver a wide range of products. Chapter 8 contains an interview with Ken Schwaber, and Chapter 17 describes Scrum. Ken, Jeff, and Mike were all coauthors of the Agile Manifesto.

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method was developed in the U.K. in the mid-1990s. It is an outgrowth of, and extension to, rapid application development (RAD) practices. DSDM boasts the best-supported training and documentation of any ASDE, at least in Europe. DSDM's nine principles include

active user involvement, frequent delivery, team decision making, integrated testing throughout the project life cycle, and reversible changes in development. The description of DSDM and an interview with Arie van Bennekum, a member of the DSDM consortium board of directors and a coauthor of the Agile Manifesto, are found in Chapter 18.

Crystal Methods

Alistair Cockburn is the author of the "Crystal" family of people-centered methods. Alistair is a "methodology archaeologist" who has interviewed dozens of project teams worldwide trying to separate what actually works from what people say should work. Alistair, and Crystal, focuses on the people aspects of development—collaboration, good citizenship, and cooperation. Alistair uses project size, criticality, and objectives to craft appropriately configured practices for each member of the Crystal family of methodologies. Chapter 6 contains an interview with Alistair, and Chapter 19 describes Crystal. Alistair is also a coauthor of the Agile Manifesto.

Feature-Driven Development (FDD)

Jeff De Luca and Peter Coad collaborated on Feature-Driven Development. FDD consists of a minimalist, five-step process that focuses on developing an overall "shape" object model, building a features list, and then planning-by-feature followed by iterative design-by-feature and build-by-feature steps. FDD's processes are brief (each is described on a single page), and two key roles in FDD are chief architect and chief programmer. FDD differs from XP in its "light" up-front architectural modeling. Jeff, an Australian, provides two case stories on very large (for Agile development) projects of more than 50 people, in Chapter 20 on FDD. Jon Kern, director of product development for TogetherSoft, Peter Coad's company, was a coauthor of the Agile Manifesto.

Lean Development (LD)

The most strategic-oriented ASDE is also the least known: Bob Charette's Lean Development, which is derived from the principles of lean production, the restructuring of the Japanese automobile manufacturing industry that occurred in the 1980s. In LD, Bob extends traditional methodology's view of change as a risk of loss to be controlled with restrictive management practices to a view of change as producing "opportunities" to be pursued using "risk entrepreneurship." LD has been used successfully on a number of large telecommunications projects in Europe. Chapter 16 contains an interview with Bob, and Chapter 21 describes Lean Development.

Extreme Programming (XP)

Extreme programming, or XP to most aficionados, was developed by Kent Beck, Ward Cunningham, and Ron Jeffries. XP preaches the values of community, simplicity, feedback, and courage. Important aspects of XP are its contribution to altering the view of the cost of change and its emphasis on technical excellence through refactoring and test-first development. XP provides a *system* of dynamic practices, whose integrity as a holistic unit has been proven. XP has clearly garnered the most interest of any of the Agile approaches.

Chapters 4, 10, and 12 contain interviews with Kent, Ward, and Martin Fowler, another major contributor to XP and advocate of Agile development. Chapter 22 provides an overview of XP. Kent, Ward, Ron, and Martin were all coauthors of the Agile Manifesto.

Adaptive Software Development (ASD)

Adaptive Software Development, my own contribution to the Agile movement, provides a philosophical background for Agile methods, showing how software development organizations can respond to the turbulence of the current business climate by harnessing rather than avoiding change. ASD contains both

practices—iterative development, feature-based planning, customer focus group reviews—and an "Agile" management philosophy called Leadership-Collaboration management. Chapter 14 contains Alistair Cockburn's interview with me, and ASD is described in Chapter 23. I was also a coauthor of the Agile Manifesto.

Acknowledgments

Any book is a collaborative effort, but this one was more so than most. A host of people contributed to *Agile Software Development Ecosystems,* from those who were gracious enough to spend several hours being interviewed for key chapters to those who have influenced my thinking.

Alistair Cockburn and I have been talking about Agile development for several years. We have swapped ideas about how software gets developed, have coauthored several articles, are coediting the Agile Software Development series for Addison-Wesley, and constantly talk about an array of topics, occasionally as we ride up the ski lift together. It has been a great partnership, and I particularly thank Alistair for contributing the chapter on my ideas to this book.

Two people have had enormous long-term influence on my thinking: Tom DeMarco and Ken Orr. I've been colleagues and friends with both for nearly 20 years. They both contributed directly and indirectly to this book. Sam Bayer and I started working together in the early 1990s on rapid development practices and continue our ongoing collaborations. I just wish he would stop posing such hard questions!

I want to thank the people who endured long interviews for the profile chapters and other major stories in this book—Kent Beck, Arie van Bennekum, Bob Charette, Alistair Cockburn, Ward Cunningham, Jeff De Luca, Martin Fowler, and Ken Schwaber. Their involvement helped me bring a personal element to the discussion of Agile principles and values.

The *Agile* movement emerged from a meeting in February 2001, when a group got together to discuss what had been previously referred to as "light" methodologies. The collaborations of this group sparked the Agile movement. This group comprises the 17 authors of the Agile Manifesto—Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, myself, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Bob Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. My thanks to all of them for their thoughts and deeply held convictions.

Rob Austin, Ken Orr, Tom DeMarco, Sam Bayer, Laurie Williams, Dirk Riehle, Ron Holliday, Jens Coldewey, and Lou Russell spent time reviewing the manuscript that evolved into this book. All their comments and questions were tremendously helpful.

Many people contributed in some way to the material in this book. They include Dane Falkner, Don Wells, Donna Fitzgerald, Ken Auer, Larry Constantine, Jason Leedy, Lise Hvatum, Lowell Lindstrom, Martyn Jones, Anne Mullaney, Michele Marchesi, Norm Kerth, Pete McBreen, Scott Ambler, Paul Szego, Sistla Purushotham, Michael O'Connor, Bruce Graham, Debra Stenner, Chris Pickering, Doug DeCarlo, Tom Bragg, Warren Keuffel, Ram Reddy, Bill Ulrich, Alan MacCormack, Barry Boehm, Barry Foster, Borys Stokalski, Ellen Gottesdiener, Geoff Cohen, Geoff Flamank, Peter O'Farrell, Ed Yourdon, Tim Lister, James Bach, Glen Alleman, Ivar Jacobson, David Garmus, Michael Mah, and Steve Palmer.

My special thanks goes to Karen Coburn, president of the Cutter Consortium, for her support of Agile ideas and permission to include material I wrote for various Cutter publications in this book.

And last, but (to use the cliché) not least, my thanks to Mike Hendrickson and Ross Venables at Addison-Wesley for their support and encouragement on this project, to Karen Pasley for her wit and wisdom in the editing process, and to Jennifer Kohnke for her wonderful sketches.

The Agile Software Development Series

Among the people concerned with agility in software development over the last decade, Alistair Cockburn and I found so much in common that we joined efforts to bring to press an Agile Software Development Series based around relatively light, effective, human-powered software development techniques. We base the series on these two core ideas:

1. Different projects need different processes or methodologies.
2. Focusing on skills, communication, and community allows the project to be more effective and more agile than focusing on processes.

The series has the following main tracks:

*   *Techniques to improve the effectiveness of a person who is doing a particular sort of job.* This might be a person who is designing a user interface, gathering requirements, planning a project, designing, or testing. Whoever is performing such a job will want to know how the best people in the world do their jobs. *Writing Effective Use Cases* (Cockburn 2001) and *GUIs with Glue* (Hohmann, in preparation) are individual technique books.
*   *Techniques to improve the effectiveness of a group of people.* These might include techniques for team building, project retrospectives, collaboration, decision making, and the like. *Improving Software Organizations* (Mathiassen et al. 2002) and *Surviving Object-Oriented Projects* (Cockburn 1998) are group technique books.
*   *Examples of particular, successful Agile methodologies.* Whoever is selecting a base methodology to tailor will want to find one that has already been used successfully in a similar situation. Modifying an existing methodology is easier than creating a new one and is more effective than using one that was designed for a different situation. *Crystal Clear* (Cockburn, in preparation) is an example of a methodology book.

Two books anchor the Agile Software Development Series:

1. This book, *Agile Software Development Ecosystems,* identifies the unique problems in today's software development environment, describes the common principles behind Agile development as expressed in the Agile Manifesto, and reviews each of the six major Agile approaches. My previous book, *Adaptive Software Development: A Collaboration Approach to Managing Complex Systems* (Highsmith 2000), expresses my thoughts about Agile software development using the vocabulary of complex adaptive systems.
2. Alistair's book, *Agile Software Development* (Cockburn 2002), expresses his thoughts about Agile development using several themes: software development as a cooperative game, methodologies as conventions about coordination, and families of methodologies.

You can find more about Crystal, Adaptive, and other Agile methodologies on these Web sites:

*   www.crystalmethodologies.org
*   www.jimhighsmith.com
*   www.agilealliance.org

Part I: Problems and Solutions

Chapter 1. The Change-Driven Economy

The digital age has its own uncertainty principle: Issues get fuzzier as their parts get more precise.

- *Bart Kosko,* Heaven in a Chip

The future of our Information Age economy belongs to the Agile, those companies that have the capacity to create change, and maybe even a little chaos, for their competitors. If you can innovate better and faster—you create change for your competitors. If you can respond quickly to competitive initiatives, new technology, and customers' requirements—you create change for competitors. If you are slower, less innovative, less responsive—you are doomed to survival strategies in a sea of chaos imposed by others. Is your company going to set the pace of change, or are competitors going to set it? In our Information Age economy, a company's ability to set the pace, to create change, lies in its ability to develop software. In a world of constant change, traditional rigorous software development methods are insufficient for success.

Agile Software Development Ecosystems will flourish for two reasons. First, they match the business need to deal with relentless speed and change, and second, they forge the workforce culture of the future. In recent years, software technology has moved from supporting business operations to becoming a critical component of business strategy. It drives new product development, from automobiles with hundreds of chips with embedded software to cellular phones and other wireless devices that are extending the definition of "distributed" systems.

ASDEs are tuned to innovation and response—to creating new knowledge that delivers value to businesses and to responding quickly to competitive challenges. Rigorous Software Methodologies (RSMs) are useful, but only for a set of problem domains that is shrinking. Many of the techniques from RSMs can be effectively employed by ASDEs, but the framework and philosophy of the two are different. Agile approaches are best employed to explore new ground and to power teams for which innovation and creativity are paramount.

Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battening down the business hatches to ride out fearsome competitive "storms." It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many companies now fear (Goldman, Nagel, and Preiss 1995).

Forget the dotcom debacle; the Information Age economy remains alive and kicking. The new economy is not about the Internet, although the Internet certainly qualifies as a key driver. The new economy is about change—both the acceleration of change and the emergence of multiple types of change. One measure of this increasing change and turbulence is the attrition rate of companies from the Fortune 500 list over the past 25 years. From 1976 to 1985, only 10 percent of the Fortune 500 dropped off the list. From 1986 to 1990 the attrition rate rose to 30 percent, and it jumped again to 36 percent in the next 5-year period to 1996 (Pascale, Millemann, and Gioja 2000).[1] But while everyone admits to the increased pace of change, far fewer actually understand its implications.

[1] Attrition includes (1) failure to grow and therefore falling below 500, (2) mergers and acquisitions, and (3) bankruptcy.

People have focused on the title, rather than the subtitle, of Kent Beck's ground-breaking book, *Extreme Programming Explained: Embrace Change* (Beck 2000). These individuals, especially managers in large organizations, are put off by the word "extreme" and its connotation of daredevils swooping in on their coding snowboards. Or, if they manage to force themselves past the word "extreme," they land on the word "programming" and relegate the material to that "mechanical" stuff that the geeky people do. Although the words in Kent's book may talk about programming, and he may even advocate such *extreme* practices as

testing one's own code, the strategic issue surrounding XP, and all other ASDEs, concerns *embracing change*.[2]

Agile organizations create chaos for their competitors, first by creating change so fast that competitors are left gasping for breath; and second, by responding quickly to competitors' attempts to change the market. Just imagine what it feels like to have a competitor with a new product-development cycle of 12 months to your 18; furthermore, every time you introduce a breakthrough feature, they match it in their next product release. They are attacking, you are always on the defensive. That's the essence of Agile organizations—create change that you can live with and your competition can't. "The source of our competitiveness in this industry is our ability to manage in a chaotic environment," says Silicon Graphics' CEO Ed McCracken. "But it's more proactive than that. We actually help create chaos in the first place—that's what keeps a lot of potential competitors out" (Iansiti 1998). Agile organizations don't just respond to change, they generate it!

Whether it's rampant dotcom mania, the sharp business decline in the aftermath of that mania, or the economic aftermath of the horrendous attack on the World Trade Center, turbulent change will continue to dominate corporate economics. Furthermore, this change is emergent, complex, and messy. "Its 'messiness' lies not in disorder, but in an order that is unpredictable, spontaneous, and ever shifting, a pattern created by millions of uncoordinated, independent decisions," writes Virginia Postrel (1998). The battle lines between proponents of ASDEs and RSMs are drawn from of deep-down, fundamentally different cultural assumptions, or perspectives, about how the world and organizations work.

"How we feel about the evolving future tells us about who we are as individuals and as a civilization: Do we search for *stasis*—a regulated, engineering world? Or do we embrace *dynamism*—a world of constant creation, discovery, and competition?" asks Postrel. "Do we crave predictability, or relish surprise? These two poles, stasis and dynamism, increasingly define our political, intellectual, and cultural landscape. The central question of our time is what to do about the future. And that question creates a deep divide" (Postrel 1998). While it may be stretching to equate software development ecosystems with wider cultural issues, my interviews with key ASDE authors indicate that their long-term visions are just that broad in scope.

ASDEs reflect a nontraditional set of ideas about organizational culture. The fundamental issue—related to Postrel's notion of dynamism versus stasis—revolves around whether an organization has a Command-Control view of management or, as outlined in *Adaptive Software Development,* a "chaordic" Leadership-Collaboration view. I believe that ASDEs are brought into organizations for two reasons. These organizations believe that ASDE principles and practices:

1. Support *innovative product development*.
2. Align with an organizational environment *in which people want to work*.

But perhaps, since the dotcom bubble burst, could turbulence be a phenomenon of the past, but not the future?

Turbulence: Bubbles versus Trends

The rise and fall of the dotcoms created economic turbulence. For example, Priceline.com's stock price rocketed from $69 per share at its IPO on June 30, 1999, to $107.50 on July 1, 1999, and then plunged to $1.46 per share by January 1, 2001. Many stocks went through this roller-coaster ride. But while stock prices magnified the turbulence, the underlying trend of a business environment racked with continuous change remained.

Everyone predicted the demise of the dotcoms, so it's somewhat of a mystery as to why these same people—on Wall Street in particular—were surprised when so many of them failed. A reasonable analysis of major organizations shows that the dotcoms intensified an eBusiness revolution that was already under way, and that the loss of a few (well, many) billions of dollars in stock market capitalization isn't going to put the eBusiness genie back in the bottle.

But which part of dotcom mania was the bubble (the wild speculation on questionable business models) and which part was the trend (the longer-term implications of the Information Age)? Companies such as General Electric, General Motors, and Fidelity Investments—and many more—have extensive, long-term commitments to eBusiness. The pace may slow from blistering to merely brisk, but the ripple effects of dotcom mania will continue to create great opportunities.

We were in the eye of a hurricane during much of 2001; a lopsided hurricane for sure, but a hurricane nonetheless. On one side, during late 1999 and most of 2000, the hurricane was ferocious—full of tsunami-like waves and thunderous winds. The year 2001 was spent in the eye; the winds died down, and people made a quick assessment of the damage. Some, as always, not understanding the nature of hurricanes, thought the storm was over. Although the backside of the storm will be less ferocious, it will be considerably longer lasting.

Our metaphorical hurricane is, of course, the storm of the Internet economy, driven by the dotcom frenzy. Many companies think the storm tides are gone forever, but they have just begun. During the MSNBC Silicon Summit II in spring 2001, CEOs of major high-tech companies pointed to the difference between the speculations in the stock market and the basics of businesses: delivering products and satisfying customers. Part of the learning process will be separating the bubbles from the trends, the pure speculation from the real changes in delivering products and satisfying customers.

The pacesetters for the next wave are those companies that can blend the best of both worlds—bricks and clicks, technology and people, existing legacy systems with the B2s (B2B, B2C, etc.), and traditional corporate cultures with dotcom cultures. I am reminded of the comments by Lloyd Darlington of the Bank of Montreal, who observed that the new information economy defines the most pervasive change in banking in 300 years. He also offered keen insights into the strengths and weaknesses of traditional banks versus dotcom banks. "In the long run," said Darlington, "we figure that while some people might be happy to accept a cheap loan from an unfamiliar provider on a Web site, few would be willing to hand over their life savings to such a provider" (Tapscott et al. 1998). His message was clear: Banks must adapt, but they don't have to remake their entire structure overnight. Darlington's comments were made at the height of the dotcom frenzy, during which a number of Internet-only banks were established. They are all gone. However, their legacy lives on in the "click" side of bricks-and-clicks banking.

A bubble-versus-trend example can be seen in analyzing Delta Airlines and Priceline.com. As mentioned earlier, from January 1999 to January 2001, the stock of Priceline.com went from $69 to $107.50 to $1.46 per share, while Delta Airlines' stock traded in a narrow range of $45 to $60 per share. At one time during 1999, Priceline.com's market capitalization surpassed that of Delta's. The bubble in this situation was the greatly exaggerated stock price for Priceline.com. The Delta stock price trend reflects the nature of the airline business—high fixed costs for salaries and fuel, over capacity, and high capital investment for airplanes and facilities. Fancy ticket pricing didn't overcome the basics of the airline business.

Those companies that have tasted the first round of eBusiness success are forging ahead and warning others to back off at their peril. When the stock market bubble burst, "It was like the pressure was off. I think that's going to lead to a significant reduction in effort," one CEO told *Business Week*. "We think that's a huge mistake." Even though sales forecasts are being reduced, the article indicated the areas in which companies are investing, or are expected to invest heavily: eCommerce ($6.8 trillion in 2004), eMarketplaces ($2.8 trillion by 2004), procurement ($2.8 trillion by 2004), knowledge management ($10.2 billion by 2004), and customer relationships ($12.2 billion in 2004). The same article projected B2B eCommerce to grow from $.5 trillion in 2000 to $1.1 trillion in 2001 to $3.5 trillion by 2003 (Hamm et al. 2001).

Although the backside of the dotcom storm will be less fierce, in the long run it will be fraught with more devastation and more opportunity: devastation for those who retreat to their traditional modes of operation, and opportunity for those who can blend the old and the new, champion the skills of innovation and improvisation, and build the culture required to sustain the ship through the ravages of the years ahead. The front side of the storm, the dotcom side, lasted two to three years, depending on one's point of view. The backside will take considerably longer—another ten years at least.

Turbulence is here to stay. Whether reacting to the rise of the dotcoms, creating new products or business models, rapidly cutting costs in response to a business or industry downturn, or figuring out how to deal with cyber-terrorism, companies that want to survive must take to heart the pop-culture phrase—"deal with it!"

Exploration versus Optimization

At a presentation to senior executives at an oil services company, I asked the question, "What would you think of a oil exploration group that discovered a production-quality well 99.5 percent of the time?" The answer: "The company would go broke. They would not be taking enough risk." Succeeding at oil exploration is a tricky business—take too many risks and the cost of dry holes skyrockets; however, take too few risks and the potential big field will never be found. Dealing with the turbulent economy means, first, understanding the differences between exploration and optimization.

Drilling production oil wells is an optimization, not an exploration, activity. Production drilling involves extracting the very last barrel from a known oil reservoir—it involves efficiency and cost control, getting better and better at something that is relatively well known. Exploration drilling, on the other hand, involves risk management: using well-known engineering practices, but ultimately trying to find the optimum balance between too much risk (costly dry holes) and too little risk (leaving potential revenue in the ground). Oil companies must do both—exploit known oil fields and explore for new ones. Other companies have to do both also, but the difference today is that the balance of exploration versus optimization has changed and companies are not geared to the degree of exploration required to survive and thrive in an Information Age economy.[3]

[3] Thanks to my brother, Rick Highsmith, a 22-year veteran of Texaco's exploration department for insights into the differences between exploration and production drilling.

Oil exploration and production organizations are quite different—they have different goals and reward systems. Exploration groups seek new reserves by going into new countries, new geographic areas, or thinking about how to use a new technology. Exploration departments require great creativity, attracting a high percentage of Ph.D. geologists and geophysicists who organize around skill teams. Exploration groups are often viewed as "strange" by the rest of the company.

Exploration and production people have different experiences because exploration decisions often have to be made with limited information. Most exploration drilling results in dry holes. A typical figure in exploration drilling would be a ten percent chance of discovering a viable oil field after spending $50 to $100 million. Although exploration groups use extensive statistical analysis to probe the risk of a project, in the end there is an element of "rolling the dice." Other groups within oil companies are much less comfortable with dealing so explicitly with risk and uncertainty—they think deterministically.[4]

[4] Uncertainty can be defined as unquantifiable risk. Both risk *and* uncertainty characterize oil exploration.

Market and technology turbulence creates opportunity, but one must have the right mindset to take full advantage of it. Exploring requires a very different perspective than optimizing. For example, optimizing focuses on reducing normal operating costs, in particular the cost of making frequent changes. Exploration, however, requires that we reduce the cost of change rather than reduce the amount of change itself. Optimization practices attempt to reduce change because practitioners view it as costly both in time and money. Explorers, on the other hand, are trying to maximize experimentation; they are trying to find the

path that works out of a myriad of potential paths. Rather than use the high cost of change as a deterrent, exploration geologists try to reduce the cost of change so that exploring multiple options will be cost effective.

Change generates either the risk of loss or the opportunity of gain. Risk management helps mitigate risk; risk entrepreneurship (a term originated by Bob Charette) helps turn opportunity into profit; and failure to change causes, well, failure. Many companies view change as a necessary evil rather than as a competitive opportunity. *Risk entrepreneurship* means managing a portfolio of projects with varying degrees of market uncertainty (Is there a market? Is our timing right?), product uncertainty (Do we have the right product?), technology uncertainty (Did we pick the right technology for our product?), and business uncertainty (Can we make a profit on this product?).

Generally, the higher the uncertainty, the higher the probability of failure (or everyone else would be trying it) *and* the greater potential rewards. So we need both project portfolio management (product selection and funding) and Agile project management and software development practices that limit the cost of failures (there will always be losses if we are aggressively seeking new opportunities) while increasing the probability of success. ASDEs are garnering increasing interest not because they attempt overturn 25 years of software engineering progress, as some critics contend, but because they address the issues germane to turbulence, risk, and uncertainty.

Recognizing the need to change or even the opportunities offered up by change is not enough, however. Companies and project teams must have appropriate practices in place to harness those changes, and therein lies a problem for many companies. *Optimization* processes—those traditional means of software development and project management—are fundamentally change-resistant processes, and rightfully so given the problem domain they address. Exploration practices, on the other hand, are fundamentally change-embracing—they are geared to encourage and harness change rather than tolerate only small increments of it. However, just as oil companies must explore for new oil reserves and *then* optimize production, companies need to explore technology and then optimize to improve performance and scalability.

It is not just development groups but also management that must undergo a change in perspective. A client asked me to speak to his senior executives about a new, leading-edge Web-based project. The folks in the IT department thought they had been successful in delivering a working application in the face of constant requirements changes that arose as their customers learned about the business issues of dealing with the Web. Senior executives were concerned only with the cost and schedule overrun. As we explore new business areas, measuring success from a traditional cost-schedule-scope perspective is only one of the cultural assumptions we need to reevaluate.

The move toward ASDEs reflects many of the same drivers that caused manufacturers to transition from mass production to lean production methods. In the 1980s, Japanese auto manufacturers used lean production techniques to simultaneously improve quality, lower costs, and increase speed to market. Success with lean production didn't come from plant automation or rigid assembly processes, but from a focus on people.

The truly lean plant has two key organizational features: It transfers the maximum number of tasks and organizational responsibilities to those workers actually adding value to the car on the line, and it has in place a system for detecting defects that quickly traces every problem.

Lean production calls for learning far more professional skills and applying these creatively in a team setting rather than in a rigid hierarchy (Womack, Jones, and Roos 1990).

Lean production didn't sacrifice quality for speed—it improved both. Similarly, ASDEs aren't an excuse for ad hoc development, as many critics contend; their objective is significant improvements in speed, quality, and cost. ASDEs address the exploration drilling projects of the IT world. They require engineering-like production practices, but they also require something else—a sense of adventure and a personality that thrives on "hanging out" closer to the edge.

Exploratory Projects

Part of any disruptive technology (such as the personal computer's "disruption" of the mini-computer market) or emerging market involves definitions and distinctions—the endless arguments over new terminology and whether there are meaningful distinctions. Rather than debate terminology, however, a couple of real-life examples may shed light on the subject of what I will refer to as "exploratory projects." The following examples are all from clients.

"We tried the heavy approach to software development in the late 1980s and early 1990s," related a development manager for a large utility company. "The reaction to those efforts was to retreat to an ad hoc approach, which of course didn't work either. We are now searching for the practical, middle ground."

A project manager at a large telecommunications firm in Canada explained, "Our parent corporation is solidly in the CMM camp. However, my project has critical time constraints and the requirements are evolving constantly, during the project, because of turbulence in the wireless telecommunications market, so that generating and maintaining the range of documentation required for a typical corporate project would doom this one. We're still disciplined—in rigorous testing, for example—but in different ways. We just hope we get this project finished before the CMM auditors show up."

"We have to return to our entrepreneurial roots," muses the manager in charge of revamping "methodology" for a systems integration company that deals with Wall Street financial firms. This firm's clients want rapid response, not months of up-front requirements specification and architectural planning.

A small, rapidly growing software company needs a framework for growth—a framework with enough process to keep the growing chaos at bay, but not so much that its highly skilled and experienced development staff loses its innovative edge to formality and documentation.

These companies are not well served by RSMs—either for software development or project management. But what are the characteristics that make exploratory projects unique?

Exploratory projects are frontier (research-like), mission critical, time driven, and constantly changing.

There are significant words in this statement. The first of which is "frontier"—as in out on the frontier where the "arrows are flying." Frontier projects are characterized by risk and uncertainty—risk relating to things that you anticipate might go wrong, and uncertainty relating to things that might go wrong that you don't even know about yet. The manifestation of risk and uncertainty is change, so frontier projects typically are managed to accommodate these changes and explore the unknown. There is a maxim that says, "Stay away from bleeding-edge (frontier) projects and technology." But if your company isn't drilling for new oil out on the frontier, how will it find new sources of revenue and critical cost reductions? Frontier projects don't have to be Internet projects. Any project that addresses a new business initiation, whether it is an eCommerce application or an innovative new approach to cutting costs, can qualify as frontier.

Second is "mission critical." Are "frontier" and "mission critical" possible at the same time? Unfortunately, they are both necessary for many exploratory projects. Teams explore the unknown, while at the same time they must deliver low defect levels and extensive integration with other business applications. Because eBusiness applications often reflect significant business initiatives, a high percentage can be characterized as mission critical.

The third significant phrase is "time driven." There is no question that the pace of business has accelerated, and its pace is unlikely to return to sedate levels. When I ask groups if the slower economy has lengthened product schedules, they just grin. Product development life cycles continue to shorten.

Fourth is "constantly changing." Internet technology today is the same as having a 2,000-piece Lego set and no instruction manual—with pieces continuously morphing into new shapes. Project teams are constantly learning about new technologies, new products, and new components, and the urge to incorporate them into one's project is always seductive. Business models are rapidly evolving as companies try one approach and then evolve it. Following a predetermined project plan in this environment may get you somewhere, but is it still where you need to be when you arrive?

A colleague from Italy relayed an anecdote from one of his clients, a telecom company. One of the managers insisted that the development team commit to a five-year plan, listing all the product features that would be implemented within that time period. Of course everyone agreed that the current product looked nothing like the plans from five years ago, but *admitting* to the unpredictability of the future was just not an acceptable position—way too "radical." It was acceptable to look back and say, "Well, we missed the plan again just like always," but it was unacceptable to admit, in the beginning, that plans were fallible. The problem with failing to admit the unpredictable nature of the future is that it restricts organizations from adopting processes and practices that deal appropriately with that unpredictability.

The core issue is not whether CMM-like or Agile-like practices are *best,* the core issue is building an organizational culture with a balance of practices that support innovation, discipline, and adaptability. Although Silicon Valley's dotcoms may be innovative and entrepreneurial, they often lack the discipline to scale up. Process or structure of any kind is so distasteful to these firms that growth is compromised. Highly disciplined, control-oriented company cultures often discourage the innovation required to explore new business initiatives. Neither innovative nor disciplined behavior guarantees that an organization can be flexible or adaptive when needed. A small, "gung ho" innovative team may shun discipline completely. An adaptive team understands how to balance innovation with discipline and flexibility with structure.

As Tom DeMarco (2001a) pointed out, the history of war has swung back and forth between the ascendancy of armor over mobility and vice versa. Until the business and technology turbulence subsides, mobility will reign supreme. Fast, Agile project teams will win out over slower, efficient ones.

The discipline to build quality software is important. Many technical and management abilities are necessary to deliver results. Ultimately, however, success rests on creating an innovative culture, one that can respond quickly as the unknown morphs into the partially known. Exploratory Projects push the limits of organizations' capabilities.

Command-Control versus Leadership-Collaboration Cultures

A second major reason that ASDEs are being implemented in organizations is to forge the workforce culture of the future. Management practices have also experienced the bubble-versus-trend dichotomy. The bubble seemed to indicate that traditional business precepts could be ignored in the digital world of tomorrow, and that the key to success was finding managers under 30 years old who were not burdened by the lessons of the old economy and traditional management knowledge. But just as companies are learning that success at eBusiness depends on the integration of Internet technology *and* existing business models, they are discovering that they need to blend management styles, and in particular, to draw on elements of the Internet culture—fast response, agility, adaptability, improvisation, reliance on emergent results, and recognition of unpredictability.

How does a ship, tossed by waves and wind, keep from sinking? How does a company, tossed by dizzying new technology and a rapidly evolving business, sustain itself for the long term? Sailing ships have a heavily weighted keel, tons of lead that counterbalance the winds and waves, which rights the ship again and again. I think that the heavily weighted keel for organizations lies in their people, in an adaptive culture that can bind people together in effective collaborative relationships, and in leadership that understands that traveling broadside toward raging seas is a poor strategy.

Interest in ASDEs has mushroomed not because of pair programming or customer focus groups, but because, taken as a whole, their principles and practices define a developer community freed from the baggage of Dilbertesque corporations. Agile cultures are more attuned to the Information Age and have

created ground swells at the developer level that organizations are being forced to deal with—at least those that want to retain talented staff. ASDEs reflect how software *really* gets developed, and this appeals to developers.

In summer 2001, the Cutter Consortium's Agile Project Management service conducted a survey of IT and software organizations worldwide. One of the most significant data points from that survey was that project teams employing ASDEs rated 22 percent higher on employee morale than teams using RSMs.

The real challenge for organizations is not blending bricks-and-clicks technology, but knowing how to build a blended culture, incorporating the best of dotcom and traditional qualities. Many IT organizations have created fledgling Web development groups and separated them from the traditional organizations or outsourced Web development completely. Although this approach may, in fact, get projects delivered quickly initially, it does nothing to address the longer-term issue of evolving a culture attuned to constant turbulence. Setting up separate organizations (skunk works) has long been an accepted strategy for isolating new initiatives from the ponderousness of existing organizations, but it does nothing to change the existing organization. Is it enough, long term, to create a 100-person Internet project group that is fast, Agile, and innovative, but relegate the remaining 2,500-person IT organization to its traditional culture?

Managers in the process-centric community value people, but they still consider process more important. For those who might argue with this assessment, I offer the following paragraph from a *ComputerWorld* article about Ashutosh Gupta, CEO of Deutsche Software India, a subsidiary of Deutsche Bank AG.

"There is this myth that software development is a creative effort that relies heavily on individual effort," *says Gupta from his air-conditioned office high above the din of traffic-clogged Mahatma Gandhi Road.* *"It is not. It is just very labor-intensive, mechanical work once the initial project definition and specification stage is past" (Vijayan and Anthes 2001).*

Agilists totally reject this viewpoint. It's the old treat-people-like-cogs-in-the-machine view. Although this may not reflect the view of all Control-Culture managers, this process-centric, mechanical view of people remains a bias in many organizations. It is easy to see that implementing an Agile approach in Deutsche Software India would be an impossible task given the cultural norms expressed in this statement. Without first acknowledging that an exploratory process is required for a particular problem set, companies such as this one will have a very difficult time breaking new ground, although they will continue do well with traditional optimization projects.

Thriving at the Edge

Given moderate uncertainty, either an ASDE or a RSM can be effective. If your organization is fundamentally top-down and hierarchical, then a rigorous approach may fit better. Similarly, XP, Scrum, or Crystal will fit well in a naturally collaborative culture.

However, "the vast uncertainty of the Internet environment has deeply influenced the nature of research and development," says Harvard Business School professor Marco Iansiti. "Gone are the days of clear objectives, frozen specifications, and proven technologies" (Iansiti 1998).

So, while either Agile or rigorous approaches work for certain problem domains, as turbulence and uncertainty heat up, as the competitive cauldron that your company must compete in bubbles hotter and hotter, RSMs rapidly deteriorate in effectiveness. When the water boils, you must be Agile enough to keep from scalding. And there are times, of course, when the water becomes so superheated that nothing works.

We all sympathize with Dilbert and silently rant against his pointy-haired boss. However, Dilbert is part of the problem—and the solution. The Dilberts of our world (and there must be millions of them, based on the number of cartoons on office walls) have an obligation to push back, to change our mindless corporate bureaucracy and, where it exists, the people-as-machine-parts culture. In the final analysis, this is the core of Agile Software Development Ecosystems.

Chapter 2. IDX Systems Corporation

The turbulent nature of the worldwide economy, technology, and new product development plays out in field after field—from telecommunications to insurance. In looking for case stories to illustrate both the success and the challenges of using ASDEs, I talked to Debra Stenner of IDX Systems Corporation. Not only did Stenner use Scrum successfully, but her story epitomized the challenge of today's business environment. Other case stories are embedded in chapters on ASDE principles, but the IDX case story so richly illustrated these challenges, and how an ASDE helped the company overcome them, that it deserved its own chapter. This case story also provides a bridge from the explanation of the *problem,* speed and constant change, in Chapter 1, to an explanation of the *solution,* agility, in Chapter 3.

The IDX Story

"It's a case where, if there is any such thing, we were almost too successful," said Stenner, vice president of business planning and product development for IDX Systems Corporation. She was talking about how IDX transformed its strategic vision into reality using Scrum. Over the years, I've worked with a number of companies that have struggled to move a successful product off of one technology platform onto another. It has usually been an ugly struggle, often lasting years longer than originally intended, and not infrequently ending in failure. IDX not only made the transition, but has emerged with a much stronger product line to carry the company into the future. IDX, a medium-sized company with $340 million in annual revenues, supplies the health-care industry with financial, administrative, and clinical software.

This case story shows both an Agile approach to software development and an Agile approach to the business itself. Ultimately, if Agile engineering processes are to be useful and widely applied, they must be tied to business goals that stretch the boundaries of tradition and therefore require agility, rapid response, and innovation.

In 1996, IDX began a five-year transition from vision to reality, using the Scrum development methodology (jointly developed by Ken Schwaber and Jeff Sutherland). The company had a very successful product but an aging technology platform. IDX's applications ran on Digital Equipment (DEC) hardware, a MUMPS database, and the VMS operating system. "Market research said we were never losing deals on functionality," Stenner noted, "but losing them because of 1982 vintage technology." In addition to the aging technology platform, IDX's main product line—administration and finance-oriented software—was not integrated either technologically or strategically with its radiology software (the Radiology Information System, or RIS). The company needed a strategic product vision that included both new technology and an integrated product vision, and it was overwhelmed at the prospect of implementing both concurrently.

At the time, IDX had a comprehensive, enterprise-wide set of applications for person indexing, scheduling, maintaining electronic medical records for clinical results, and more, but radiology, pharmacy, and pathology were viewed as ancillary systems. The folks at IDX needed to come up with a synergistic strategy that made sense when viewing both their enterprise and radiology systems. In January 1996, they started the strategic planning process for radiology, made extensive customer visits, and asked their customers three key questions: "Should we be in this market?" "How can we make money (20 percent growth in revenue and profits)?" and "How can we make the radiology products synergistic with our enterprise applications?" IDX customers helped the company put together an enterprise-wide vision for a top-down image and information distribution system.

Medical digital imaging—such as CT scans and MRIs—requires massive data storage. "One CT scan with 100 slices can take the equivalent of 13,000 text pages," said Stenner. "These are huge image blobs." Since doctors need to compare current scans with prior ones, a single study can take the equivalent of 60,000 to 100,000 pages of text. In institutions that might conduct 1,000 to 2,000 procedures a day, "That requires a huge data-pipe to move the images around."

Unfortunately, digital imaging has run into problems in many facilities. Since it was so costly to move the data around on demand, especially for referring physicians, facilities had to produce film as well as store the images—thus eliminating the anticipated cost savings. Customers told the staff at IDX that they were in a good position to solve these problems. By having all the patient and enterprise information already on hand, the health-care facilities could use a workflow technology created by IDX to "pre-stage" images (based on when and where the physicians would need the data). For patients who hadn't been seen in some time, this usually meant staging the data from archival to online storage. This strategy would enable IDX to integrate its enterprise applications with a new set of radiology applications and move into the rapidly expanding imaging market.

But the transition would not be easy, as it required three parallel but integrated projects. The first project would rewrite the existing radiology information system to run on a new technology platform—Windows NT/2000, SQL, and Web client pages. This type of transition can flounder badly, taking years longer than originally planned (usually based on hopes rather than realistic plans).

The second project would build a suite of modules to interface with all the imaging equipment in the market—equipment from companies that had a major market presence, such as GE and Siemens. Here again, new technologies emerged for the team members to learn. While the information processing parts of their applications used an industry standard protocol called HL7, the imaging market used DICOM (digital imaging, communications, and medicine), an OO-based standard. The third project would build the workflow engine, allowing for user-defined staging rules for imaging information.

"These were three big things," Stenner said. "Just the porting project was a huge job, plus we had to learn all this new stuff that we'd never done before in the imaging world. The plan was approved in August 1996, but we still had to wrap up existing projects and start staffing and training for the new projects. So when Ken got involved in December 1996, I was sitting there sort of wide-eyed with this great vision and opportunity, trying to figure out how to bring in key technical resources, how to wrap up our existing projects, and how to get on with the new projects."

The first item on the agenda for Ken and Scrum was wrapping up the backlog of client/server projects that had dragged on and on. "They were just haunting us," said Stenner. "So Ken brought Scrum in and spent time educating us. We identified some clear goals and deliverables so we could get the development group out of the beta-support mode that it had been in for nearly nine months." The radiology group spent about six months wrapping up these projects and then about three months in training on all the new technologies for the upcoming products. In September 1997, it got started on the three new vision projects.

"With Scrum, we were able to move pretty quickly in a couple of areas," Stenner said. "One, Scrum really gets you away from the traditional waterfall approach, with its long, drawn-out, up-front processes." Instead, the entire development team was shipped off to customer sites. Stenner recognized that the team was having a difficult time translating the vision into requirements and then into what to do first. "I think it's hard, almost a disadvantage, when you have a clean slate and new technologies, to figure out how to start. With no guidelines, it's overwhelming for a lot of people."

She then called six customers and sent out the imaging team (this was the area in which requirements were new to IDX) to find out how the technology could improve the customers' business processes and help them maximize their huge financial investment. IDX needed to address how its customers could become not only filmless, but paperless. The development group—consisting of functional experts, software engineers, and documentation staff—was split into two subteams, each visiting three customers. The teams then convened each night to compare notes. Within a week after the customer visits were completed, the combined team had finished a requirements document. From beginning to end, the requirements process took about a month. By the time the team got finished, Stenner said, "Everyone understood the requirements."

From the requirements document, the team created its master backlog, a complete list of features from which each 30-day development "Sprint" would be taken. The first team then started on a series of Sprints to deliver functionality. Members of the second team, which was called the 10.0 team because it was

developing version 10.0 of the baseline product, figured out how to chunk the existing product into modules so they could port it to the new technology. The imaging and 10.0 teams each consisted of ten to twelve people. The workflow engine project consisted of just two people—one an outside consultant—so they didn't use Scrum.

When I asked Stenner how the developers felt about Scrum, her reply was emphatic: "They love Scrum. They really, really love Scrum. And I'll tell you why—they don't feel like they are always changing direction." In Scrum, once the work for the 30-day Sprint has been decided upon, it doesn't change. Stenner said that everyone felt they got to concentrate, and they really worked as a team and had good synergy. "They like the small, focused group," she reported. "They like getting a list of things to work on and knowing that the list isn't going to be pulled out from under them. They like that by going to one 15-minute meeting each day that they aren't then pulled into endless other meetings." She agreed that time fragmentation is a terrible waste, both demotivating and fatiguing. Constantly pulling people away from the level of detail they are working with can be exhausting for them.

But beyond Scrum, IDX also put into place a very interesting bonus system for these projects. We hear all the time about fostering teamwork and collaboration, but very seldom is that message enhanced with compensation geared to support these principles. IDX instituted a two-tier bonus system designed to balance cooperative and individual efforts. The first tier was geared to incent the team to meet its basic Sprint goal of completing the planned backlog, which was considered "doable" if everyone worked a 40-hour week. If the team met its Sprint goal (every 30 to 40 days), each team member received $250. However, the catch was that *everyone* had to meet their assigned task plan; if a single person came up short, no team member got the bonus. "It wasn't really the money," Stenner said, "but people really helped each other out. We had developers helping QA finish up work and so on." At one point, Stenner had to tell a distressed developer—distressed because he felt he had let the team down by not getting his features developed and thus the team wouldn't get its bonus—that those were just the rules of the game.

While the first-tier bonus focused on teamwork and getting the basics finished, the second tier focused on individuals and getting extra work accomplished, such as new items or backlog items left over. Individuals could sign up to work on extra backlog items. If they didn't get them finished, there was no penalty, but completion resulted in a $500 bonus. (People couldn't start on second-tier work until the first-tier work was finished.) "People really liked this because they could choose," Stenner remarked. "Some of the married engineers said that they could tell their husband or wife that they had to work a weekend or two, but could show a direct monetary benefit. Overtime work became much more palatable." As the ship deadline created additional pressure, the bonus amounts were increased.

"Scrum worked very well for us," Stenner said. "We had a release coordinator (which Ken calls the Scrum Master) who made sure the Scrums were happening, that the backlog was defined, and that the necessary planning was occurring. Where we ran into trouble—and it wasn't really the fault of Scrum—was once we started into implementation. The development team's days became very unpredic table because of constant interruptions for implementation work." Stenner concluded that the team had not done a good job of technology transfer to other parts of IDX; therefore, as sales started rolling in, the development staff got heavily involved in implementation. During this time, the bonus program had to be abandoned because interruptions were causing the team to miss Sprint goals. The program had begun to demotivate rather than motivate. But all the turmoil was for a good reason—increased sales.

By early 2000, new sales continued to destabilize development efforts. "For instance," Stenner noted, "last year [2000] our average sales person was 180 percent of quota and some were at 400 percent. So the vision has been incredibly, phenomenally successful. But those sales raise new pressures." Before the 10.0 rewrite was even completed, IDX committed to a new version running under Oracle and another major multi–time zone feature enhancement. As of spring 2001, the 10.0 rewrite product, the largest of the projects, is live at 9 sites with 30 more in the installation process. Agile practices can often help companies be successful, but sometimes that success brings its own new set of problems. "We were almost too successful," Stenner said. "We needed to have done a better job of capacity planning for the installation and enhancement work."[1]

[1] This points out that improving the software development process often causes other processes to become the new bottlenecks.

At one point, I asked Stenner if she thought that the short-cycle emphasis of Scrum ever got the team in trouble because the members couldn't back off and see the big picture. She indicated that they had some problems in this area, "but it was not so much Scrum as it was that we didn't have enough discipline with practices like reviews for overall technical quality." A couple of engineers also said that because the Sprint task lists became the focus, Scrum prevented them from "doing the right thing" for the product. Stenner thought this was a valid concern, but she also thought that most of the problems occurred near the end of the project, when time pressures were so great.

When I mentioned the XP practice of refactoring—periodic redesign to maintain code quality—Stenner responded, "We could have really benefited from that for two reasons. One, the short cycle time fed the pressure to get planned tasks done, particularly as other features arose over the life of the project. Two, we made a decision to do this project in the summer of 1997, and we all know how much Microsoft technology has evolved since then." Stenner went on to give the example of code they had to write under Windows NT that turned out to be unnecessary because the functions were built into Windows 2000. "So we should have pulled things out of our system and let 2000 do them naturally," she said. "We could have used a way [refactoring] to encourage revisiting our design on a regular basis."

I observed that one problem is that, over time, the whole system degrades, and then the next set of changes is even harder to make. "Exactly," agreed Stenner. "And I think the discipline is very hard to implement because there are not many people who understand it well. You have a ten-year-old waterfall mindset in which you think through all the issues on paper before you do any coding. The new mindset is more 'cowboy'-like coding, and it's trying to bring together the best of these two worlds."

Stenner fought not to get paralyzed in analysis on these projects. She related the story of another project she had been involved with that spent six years getting the first version of a product out, only to find the competition already ahead. She and architect Ron Keen vowed not to make the same mistake, committing instead to getting something out the door.

Finally, one of the most difficult things in most organizations is to propagate good practices, or sometimes even *any* practices, to other groups within the company. IDX seems to have encountered such barriers to the wider use of Scrum. The radiology systems group and one other are using the full Scrum approach, but others seemed to be using the label "Scrum" only for meetings. "I don't know the exact reasons for that, although it does require discipline to define the backlog and stick to your plan for 30 days," observed Stenner. "Although it doesn't sound like a lot, it's amazing how much energy it can take."

An Agile Group in Action

I spent considerable time with the IDX story because it offers a reality message about ASDEs—a reality that is mostly positive but also shows that Agile approaches, like all other approaches, are not silver bullets.

IDX had to explore in two different ways. First, there was a complete technology platform transition with all the issues of learning not one, but multiple new technologies. And second, while one of the projects involved converting current functionality from one platform to another, the other two projects required exploring entirely new requirements—requirements that had to address aspects of their customers' business processes that were new and innovative.

Over the years, I've worked with a number of clients that were attempting to transition a product from one technology platform to another. Some succeeded, some didn't, but these types of projects are consistently underestimated by an order of magnitude. I remember telling one client, struggling to deliver a product in 6 months with five part-time people, that my rough, back-of-the-napkin calculations (based on the existing system's lines of code) indicated that the project would take at least five full-time people about 30 months. I dwell on the difficulty of this type of project, which usually faces incredible management and sales organization pressure, because it indicates just how unusual the IDX project success was—technology

conversion plus major new functionality. Who says you can't complete a three-plus-year project in 30-day increments? IDX did it!

Chapter 3. Agility

Agility does not come in a can. One size does not fit all. There are no five common steps to achievement.

- *Rick Dove,* Response Ability

If turbulence and turmoil define the problem, then agility is key to the solution.

We software developers and high-tech managers often look at ourselves as being in the forefront of innovation. Agile approaches appear, from inside our profession, to be the cutting edge. But if we look across the aisle into the realm of manufacturing, we might be considered on the trailing edge rather than the cutting edge. Witness, for example, the rise of lean manufacturing practices ignited by the Japanese automobile manufacturers and well documented in *The Machine That Changed the World: The Story of Lean Production* (Womack, Jones, and Roos 1990).

Looking back nearly ten years, the foundation of what it means to be agile was described in *Agile Competitors and Virtual Organizations,* by Steven Goldman, Roger Nagel, and Kenneth Preiss—published in 1995! Even though this book addresses manufacturing (the foreword is written by former Chrysler chairman Lee Iacocca), the agility issues it addresses are the same today as they were then.

Bob Charette, originator of Lean Development, stresses that the original concepts behind lean production in Japan have been somewhat blurred in the North American and European implementations, which focus on streamlining and cost control. The Japanese origins of lean production stressed the human and philosophical aspects of the approach. The translation from the Japanese concept to the word "lean" tends to leave out this human-centric flavor, partially, Bob says, because the main driver in U.S. businesses has been cost reduction. The Japanese word is better translated as "humanware." Whereas lean production and other lean derivatives have focused on reducing staff, the original Japanese concept has more of the Agile concept of working effectively with people.

The definition of agility offered in *Agile Competitors* remains as valid today for software development as it was ten years ago for manufacturing: "Agility...is a comprehensive response to the business challenges of profiting from rapidly changing, continually fragmenting, global markets for high-quality, high-performance, customer-configured goods and services." *Agile Competitors* considers the pursuit of agility to be a strategic issue for survival in today's market. Look at the following list, compiled by the authors, of forces that threaten companies:

- Market fragmentation
- Production to order in arbitrary lot sizes
- Information capacity to treat masses of customers as individuals
- Shrinking product lifetimes
- Convergence of physical products and services
- Global production networks
- Simultaneous intercompany cooperation and competition
- Distribution infrastructures for mass customization
- Corporate reorganization frenzy
- Pressure to internalize prevailing social values (Goldman, Nagel, and Preiss 1995)

Would a list drawn up today differ? Not by much. The Information Age economy has increased the pressures on these issues, but they are still critical to business success. And according to *Agile Competitors,* agility is *the* critical characteristic, the overarching skill required—of both corporations and individuals—to address these issues. If software development organizations—internal IT groups, systems integration consultants, and software product companies—are to fulfill their vision of helping business thrive in this Information Age, they must determine how to transform this vision of agility into reality.

Agility

To become Agile, most organizations will need to change their perspective. Peter Drucker, often characterized as the father of modern management theory, wrote an extensive article titled "Management's New Paradigms," which he introduces by saying:

Most of our assumptions about business, about technology and organizations are at least 50 years old. They have outlived their time. As a result, we are preaching, teaching, and practicing policies that are increasingly at odds with reality and therefore counterproductive (Drucker 1998).

Agility isn't a one-shot deal that can be checked off the organizational initiative list. Agility is a way of life, a constantly emerging and changing response to business turbulence. Critics may counter, "Agility is merely waiting for bad things to happen, then responding. It is a fancy name for lack of planning and ad hoc-ism." But Agile organizations still plan; they just understand the limits of planning. Although the defining issue of agility involves creating and responding to change, there are three other components that help define agility: nimbleness and improvisation, conformance to actual, and balancing flexibility and structure.

Creating and Responding to Change

Is agility merely a matter of reacting to stimuli, like an amoeba, or is it something more? My preferred definition of agility has two aspects:

Agility is the ability to both create *and* respond *to change in order to profit in a turbulent business environment.*

Agility is not merely reaction, but also action. First and foremost, Agile organizations create change, change that causes intense pressure on competitors. Creating change requires innovation, the ability to create new knowledge that provides business value. Second, Agile organizations have an ability to react, to respond quickly and effectively to both anticipated and unanticipated changes in the business environment.

Agility means being responsive or flexible within a defined context. Part of the innovative, or proactive, piece involves creating the right context—an adaptable business organization or an adaptable technology architecture, for example. Innovation is understanding that defined context and anticipating when it needs to be altered. The problem with many traditional software development and project management approaches is that they have too narrowly defined the context; that is, they've planned projects to a great level of task detail, leaving very little room for agility to actually happen.

"I think Agile is both being able to respond quickly (which means you have to know that you have to respond quickly—not an easy thing to do) as well as being able to anticipate when the rules or principles are changing or, more importantly, can be changed," says Bob Charette. "Organizations that do this are often called 'lucky'—in the right place at the right time. Some undoubtedly are, while a few have created the necessary infrastructure to exploit the change."

Part of the cost of agility is mistakes, which are caused by having to make decisions in an environment of uncertainty. If we could wait to make product development decisions, they might be better ones; however, the delay could well obviate the need for the decision at all, because aggressive competitors may get their product to market while we dither. "This fact leads to a sort of organizational uncertainty principle: The faster your decision-making cycle, the less assurance you can have that you're making the best possible decision," says David Freedman (2000).

Nimbleness and Improvisation

16

In our volatile economy, companies need to enhance their "exploration" skills at every level of the organization. Good explorers are Agile explorers—they know how to juggle and improvise. Indiana Jones was a good explorer, somehow living through every outlandish adventure. Agility means quickness, lightness, and nimbleness—the ability to act rapidly, the ability to do the minimum necessary to get a job done, and the ability to adapt to changing conditions. Agility also requires innovation and creativity—the ability to envision new products and new ways of doing business. In particular, IT organizations have not done an adequate job of balancing the needs of exploration and optimization.

The actors in the movie *Crouching Tiger, Hidden Dragon* display incredible agility—running lightly along the tiniest tree branches and showing extraordinary dexterity in sword fighting. Beginning martial arts students are clumsy, not Agile. They become skilled, and Agile, from long hours of training and effective mentoring. Sometimes their drills are repetitive and prescriptive, but only as part of learning.

Agility also requires discipline and skill. A skilled software designer can be more Agile than a beginner because he or she has a better sense of quality. Beginners, with little sense of what is good and what is not, can just revolve in circles—lots of change but not much progress.

"I view the issue as one of almost 'disciplined messiness,'" remarked Bob in an email exchange. "You need to have a very good discipline in place to be able to respond in turbulent times, yet simultaneously know when to be 'undisciplined.' I view anticipation to be actively seeking situations where the generally accepted guiding rules or principles no longer apply, or where shortcuts are the least risky approach to take to gaining some objective. To be able to understand when the rules don't apply, you need to completely understand when they do." For example, Picasso had to become an accomplished fine arts painter to get to a point where he was able to move past that perception of "good art" and create abstract painting. He had to be skilled before he could be Agile.

Agile individuals can improvise, they know the rules and boundaries, but they also know when the problem at hand has moved into uncharted areas. They know how to extend their knowledge into unforeseen realms, to experiment, and to learn. When critical things need to get done, call on the great improvisers.

Improvisation makes great jazz bands. From a few key structural rules, jazz bands improvise extensively. Having a solid foundation enables their tremendous flexibility without allowing the music to degenerate into chaos. The proponents of business process reengineering and software engineering methodologies probably blanch at the thought that improvisation, rather than carefully articulated processes, is key to success. Yet in today's turbulent environment, staff members with good balancing, judging, and improvisational skills are truly invaluable.

Conformance to Actual

"OK," muse the critics, "but how do I *control* development projects in this environment?" There are two answers to this constant question. First, control focuses on boundaries and simple rules rather than prescriptive, detailed procedures and processes. The second aspect of control, though simple in concept, is completely foreign to many managers.

Agile projects are not *controlled by conformance to plan but by conformance to business value.*

"The major problem with planning," say Shona Brown and Kathleen Eisenhardt (1998), "is that plans are virtually always wrong." If we accept the notion of constant change and turbulence, then plans are still useful as guides, but not as control mechanisms. In high-change environments, plans—and the traditional contracts that are derived from plans—are worse than useless as control mechanisms because they tend to punish correct actions. If we deliver a working software product, with features our customers accept, at a high level of quality, within acceptable cost constraints, during the specified time-box, then we have delivered business value. Developers don't get to say, "This is valuable." Customers do. Every three to six weeks, customers tell developers that acceptable business value has been delivered—or not. If it hasn't, the project is cancelled or corrective action is taken. If it has, the project continues for another iterative cycle.

We may look at the plan and say, "Well, because we learned this and this and this during the iteration, we only got 23 of the 28 planned features done." That is useful information for planning our next iteration, but not for control. When we started the project three months ago, we planned only 18 features for this last cycle, and half the features we did deliver were different than the ones in the original plan. We accept that talented people, who are internally motivated, who must work in a volatile environment, who understand the product vision, will do the best they can do. If they don't conform to the plan, the plan was wrong. If they delivered business value, then whether the plan was "right" or not is immaterial. If they conformed to the plan and the customers aren't happy with the delivered business value, it doesn't matter that they conformed to the plan.

An entire generation of project managers has been taught, by leading project management authorities, to succeed at project management by conforming to carefully laid out, detailed task plans. Conformance to plan means locking ourselves into a outdated, often irrelevant plan that some project manager cooked up in great haste (which, of course, precluded talking to developers) 18 months ago, when the world was different. Conformance to plan may get a project manager brownie points with the tightly wound project management authorities, but it won't deliver business value in volatile, high-speed environments.

An exploration perspective contrasts with that of optimization. Take, for example, the use of schedule deadlines. While a schedule may appear to be a schedule, each perspective utilizes dates as a control mechanism in different ways. Optimization cultures use dates to predict and control—they view schedule as achievable, along with the other objectives, and see deviations from the plan as poor performance. Exploration cultures, however, view dates much differently. They basically see dates as a vehicle for managing uncertainty and thereby helping to make necessary scope, schedule, and cost tradeoffs. Exploration cultures, to be sure, use dates as performance targets, but the primary focus is bounding the uncertainty, not predicting dates.

Balancing Flexibility and Structure

It would be very easy to either "plan everything" or "plan nothing," but the really interesting question is how to juggle the two, how to define the context narrowly enough to get something meaningful done, but not so narrowly that we fail to learn and adapt as we go along. The fundamental issue remains one's primary perspective: to anticipate or to depend on resilience and responsiveness. Aaron Wildavsky, a social scientist, writes about this issue and offers an example of the difference between Silicon Valley and Boston. Basically, he believes, the reason Silicon Valley is the primary technology center of the world is that it depends on its resilience to be able to respond to rapid change, whereas the Boston crowd leans more heavily toward anticipation (Postrel 1998). Does that mean no one in Boston ever responds to change and no one in Silicon Valley ever plans? It's not so much either/or as it is a fundamental, underlying philosophy within which situations are juggled accordingly.

Being Agile means trusting in one's ability to respond more than trusting in one's ability to plan. Good explorers both anticipate when the rules of the game have changed (or are about to change)—that is, they define the context—and operate flexibly within a given rule set. Obviously, if the rule set itself is too prescriptive, it leaves no room for agility. Without some rule set, however, agility can become rudderless reaction.

"Agile" Studies

In Geoffrey Moore's rendition of the technology adoption life cycle, the buying cycle progresses from technology enthusiasts to visionaries to pragmatists to conservatives to skeptics. As he says, "The visionary strategy is to adopt the new technology as a means of capturing a dramatic advantage over competitors who do not adopt it" (Moore 2000). In the main, Agile approaches are still in the technology enthusiast and visionary domains—they haven't penetrated into the mainstream pragmatists' marketplace. Visionaries buy ideas; pragmatists buy proof. Will Agile approaches be the methodology equivalent of Clayton Christensen's disruptive technology (Christensen 1997), or will they become another July 4th rocket, bursting upon the scene, only to fizzle out and return to earth? Realistically, we don't know yet.

Although much of the literature on ASDEs remains anecdotal, there have been several academic studies that have pointed to the efficacy of ASDEs. Laurie Williams, an assistant professor at North Carolina State University, wrote her doctoral dissertation on her study of pair programming. Two other studies, both done at Harvard Business School, provide keen insight into the issues surrounding Agile development, although they are not about Agile development per se.

Product Development in Internet Time

"Now there is proof that the evolutionary approach to software development results in a speedier process and *higher-quality* [emphasis added] products." This is the tag line from an article in the Winter 2001 issue of the *MIT Sloan Management Review*. The article, "Product-Development Practices that Work: How Internet Companies Build Software," was written by Alan MacCormack, a professor of technology and operations management at Harvard Business School. Much of the material written on Agile methods, iterative development, and other such practices is based on practical experience about what has worked, but MacCormack provides us with explicit research results.

MacCormack and his Harvard Business School colleague Marco Iansiti have been investigating processes that work best in complex, uncertain environments. The question that the research project addressed was, "Does a more evolutionary development process result in better performance?" The study included 20 projects from 17 companies and a panel of experts to evaluate relative "performance" factors between products.

MacCormack writes, "The most striking result to emerge from the research concerned the importance of getting a low-functionality version of the product into customer's hands at the earliest opportunity. The differences in performance are dramatic. That one parameter explains more than one-third of the variation in product quality across the sample—a remarkable result." These are pretty bold statements from an academic researcher. As he points out in the article, there are so many variables that impact performance that finding one that has such a striking impact is rare in research circles.

MacCormack points to four development practices that spell success:

1. An early release of the evolving product design to customers
2. Daily incorporation of new software code and rapid feedback on design changes
3. A team with broad-based experience of shipping multiple projects
4. Major investments in the design of the product architecture

Now, to those who have been practicing Agile techniques for years, these statements may sound ho-hum. However, to those who are just embarking into this arena of exploratory approaches, or to those who are trying to "sell" these approaches to their management, these research findings are significant.

The study found that releasing early versions of products not only increased performance as MacCormack states above, but that the uncertainty associated with Internet software development "dictates short micro-projects—down to the level of individual features." This finding supports both short, iterative cycles and driving projects by features rather than tasks, as Agilists recommend.

MacCormack's study shows that daily incorporation of new software code and rapid feedback on design changes have a *positive* impact on quality (admitting that there are obviously many factors here that determine quality) and that the quicker the feedback (as in hours, not days), the higher the quality. One of the reasons for this finding may be that if a project has very short feedback cycles, the team is led into continuous, automated testing. "None of the projects with extremely long feedback times (more than 40 hours) had a quality level above the mean," writes MacCormack. On the issue of quality versus feedback time, the study found the highest-quality projects cluster around 2- to 12-hour feedback cycles, with only a couple of data points in the 20- to 40-hour range that were above the mean in terms of quality.

As with any research effort, and when thinking in general about different forms of evolutionary development, understanding the relevant problem domain is critical. MacCormack studied companies operating in complex, uncertain environments—Netscape, Microsoft, Yahoo, and others. To the extent that an organization (or a development project) operates in a slower-paced, more predic table market, some form of evolutionary development may be less critical.

"Heavy" Agile Projects

Isn't "heavy Agile" an oxymoron? I was sitting through Alistair Cockburn's tutorial on designing methodologies at the Software Development 2001 conference when something clicked. I realized that the transition from the label "light" to "Agile" had a secondary benefit; namely, we could now more easily differentiate between small, single-team projects that needed to be Agile due to requirements uncertainty (or other factors) and larger, distributed-team projects that needed to be Agile but also needed additional ceremony (documentation, formality, tools) because of their size.

MacCormack and Iansiti's work at Harvard Business School focused on Internet and software companies whose high-risk profiles are obvious, but what about larger IT projects? Two other Harvard professors, Rob Austin and Richard Nolan, have been investigating just this issue with respect to major enterprise projects, particularly very large enterprise resource planning (ERP) systems. The title of an in-depth report by Austin indicates their emerging conclusions: "Surviving Enterprise Systems: Adaptive Strategies for Managing Your Largest IT Investments" (Austin 2001).

"As the twenty-first century dawns, we are finally learning to obtain value from these very large IT projects," Austin writes. "The old project approaches do not work in this new space.... New and better analogies are based on activities like adaptive software development or new venture investment."

Austin's report first documents several high-profile horror stories. One company, which obviously wanted to remain unnamed, was a major manufacturing company whose ERP installation plan fell apart after spending $30 million, getting board approval for spending $175 million, and then quickly finding out (from the software vendor and the systems integrator) that the price tag would be more like $300 million. After getting board approval, the ERP vendor surprised the company's IT executives by informing them that the "off-the-shelf" package would meet only about 35 percent of their stated requirements.

Dell Computer backed out of an ERP project after spending more than two years and $200 million. The folks at Dell couldn't make the software work (for them); they considered the system too monolithic, and they then opted for a best-of-breed approach.

The survey data in Austin's report came from participants in the Harvard Business School's summer executive education program from the three years 1998 to 2000. This particular program attracts senior IT and general managers from more than 80 companies worldwide. The survey found that up to 88 percent of the respondents (depending upon the year) considered their ERP projects to be large or very large, several responding that they were the largest projects that their companies had ever undertaken.

Besides size, an overwhelming percentage of respondents considered the projects to have considerable technical, organizational, and business risk. The categorization of risk was interesting. Technical risk was defined as the risk of the software's failure to meet business requirements. Organizational risk was defined as the risk that the organization would be unable to make the required changes in order to effectively "use" the software, and business risk was the risk that implementing the system would actually hurt the company rather than help it. While technical risk concerns declined from 1998 to 2000, concerns over organizational and business risks remained very high. However, in 2000, the survey indicated that companies were starting to achieve the long-anticipated benefits. This led to the second set of study questions: "What are the characteristics of successful projects, and how do they differ from failures?"

Nolan and Austin concluded that there were three dysfunctional elements—in terms of flawed assumptions—in large front-end-loaded projects:

- The first flawed assumption is that it is actually possible to plan such a large project well enough that success is primarily determined by degree of conformance to a plan.
- The second flawed assumption embedded in planning-intensive approaches is that it is possible to protect against late changes to a large system project.
- The third flawed assumption is that it even makes sense to lock in big project decisions early.

"Building a huge new enterprise system is, in many ways, more like building an entirely new venture than it is like managing a traditional IT project," says Austin. Therefore, he and Nolan recommend a staging model, much like venture capitalists use to fund new ventures: spend some money, demand tangible results, spend additional money. Although this "staging" may cost a little extra, it helps companies manage the risk exposure with cost expenditures—much like exploration drilling. Staging can also produce incremental return on investment (ROI). Of the participants surveyed in the 2000 executive education group, 75 percent indicated they used some type of staging strategy. Even more interesting, companies in the planning stage estimated they would need 4 stages; companies in the midst of implementation estimated they would need 7 stages; while companies who had completed implementation said they had taken an average of 12 stages. Tektronix, one of the two in-depth Harvard Business School case studies (the other was Cisco), reported 25 stages, or waves, as it called them.

In the report summary, Austin points to four characteristics of these more successful projects (which I have paraphrased):

- They are all iterative.
- They all rely on fast cycles and insist on frequent delivery.
- They get functionality in some form into business-user hands very early in the project.
- They are preceded by little or no traditional ROI-style analysis of the project as a monolithic whole.

So, in the end, from a strategic perspective, Agile approaches are not about levels of documentation or not using UML or the discipline of pair programming. In the end, Agile approaches are about delivering working products—packaged software, embedded software in a wide range of products, and internal IT products—in environments characterized by high levels of uncertainty and risk. Whether your firm is a dotcom rushing to market or a traditional company slogging through a $50 million ERP system implementation, agility is the key to success.

Agile Software Development Ecosystems

If agility can be characterized as creating and responding to change, nimbleness and improvisation, conformance to actual, and balancing flexibility and structure, then ASDEs should help organizations exhibit these traits. They do so in several ways.

First, ASDEs focus on the set of problems described in Chapter 1—problems characterized by uncertainty and change—which require an exploratory approach. As the degree of uncertainty becomes greater, the probability that an Agile approach will succeed (over a Rigorous approach) increases dramatically, until at some level an Agile approach becomes the only type with a reasonable chance of success. As the level of uncertainty increases dramatically, it becomes unlikely that any methodology can help. No matter how good an exploration geologist you are, no matter how lucky, the inherent risk of exploring uncharted terrain remains. ASDEs improve the odds, but they don't guarantee success.

Second, ASDEs are intensely customer driven, which is both a characteristic and a risk factor. That is, no customer involvement, no Agile approach. It's basically that simple. But ASDEs advocate something even scarier—actually putting the customer in control. If we drive development from the perspective of value first and traditional scope, schedule, and cost second, then customers must be actively involved.

Third, ASDEs focus on people—both in terms of individual skills and in terms of collaboration, conversations, and communications that enhance flexibility and innovation. As Alistair says, "Software

development is a cooperative game." Most traditional "methodologies" place 80 percent of their emphasis on processes, activities, tools, roles, and documents. Agile approaches place 80 percent of their emphasis on ecosystems—people, personalities, collaboration, conversations, and relationships.

Fourth, ASDEs are not about a laundry list of things that development teams should do, they are about the practical things a development team actually needs to do—based on practice. The stories about what methodology manuals contain versus what development teams actually do are legendary. Furthermore, needs change from project to project and from iteration to iteration within a project. Alistair uses the word "embellishment," which means that many methodologies get "embellished" with artifacts, tasks, roles, techniques, and more that creep into the methodology because someone thought they really should be there, even though they themselves could never find the time or inclination to actually do them.

By articulating what ASDEs are and what they are not, people can decide which, if any, of the premises, principles, and practices to use. "Never push Lean Development beyond its limits," is the twelfth principle of Bob's Lean Development. This principle can be extended to all Agile approaches. Every approach to a problem carries the risk of loss. If we don't understand the domain in which some methodology or practice is applicable, if we don't understand the risks associated with a practice, if we don't understand the business risk and opportunity, then we may apply the wrong practice to a particular problem.

One last characteristic of ASDEs—they may be inefficient. Production drilling for oil is about efficiency and cost control. Exploration drilling is about risk management, improvisation, intuition, and occasional lucky guesses. Exploring is often messy, full of fits and starts and rework. Companies that want to explore—to innovate—need to allow for some inefficiency.

Part II: Principles and People

Chapter 4. Kent Beck

Questioner: *"I'm worried about the fact that in XP you don't do any design."*

Kent: *"We do design for a couple of hours each day during an XP project."*

Questioner: *"That's what I'm concerned about—you don't do any design!"*[1]

[1] An oft-repeated scenario, according to Kent.

Extreme Programming has captured the imagination of developers worldwide, often to the chagrin of their managers. XP has a massive following, but it also generates intensely negative responses from a number of people. I was recently in a meeting with a software development executive who went ballistic at the mention of XP, but I didn't get a chance to ask if he had actually used any of the practices. I got the feeling that he was far too emotional about the issue to consider actually *trying* any of them. Pair programming seemed to be the issue that lit his fire.

The threads of thought that evolved into XP were woven from Kent's early association with Ward Cunningham and later, on the famous Chrysler C3 project, with Ron Jeffries. These three—Kent, Ron, and Ward—are the designated "three extremoes" of XP.

I sat down to talk with Kent in the lobby of The Lodge hotel at Snowbird ski resort, watching the final skiers of the day cut through two feet of fresh Utah powder. We chatted about the past, the present, and the future of the XP movement. The conversation wasn't about pair programming or estimating stories; instead, it ran the gamut from his early days as a programmer, to the impact that parenting five kids has on his work, to dealing with life as the leader of a movement that has mushroomed in just a few years, to his views that developer-customer relationships are akin to the codependency between alcoholics and their family members.

Kent, who has a long association with the OO community, has been involved with Smalltalk (he wrote one book on the subject), the pattern movement, and XP. His XP books include *Extreme Programming Explained* (2000) and *Planning Extreme Programming* (with Martin Fowler, 2001). He is currently working on a book on leadership.

JIM: *What were the historical threads that came together to create Extreme Programming?*

KENT: First was my and Ward's emphasis on technical mastery. We spent thousands of hours learning every nuance of Smalltalk and then created CRC [Class-Responsibility-Collaborator] cards as a way to share some of this experience with less-experienced Smalltalk programmers.

Second was the aesthetics of code. Ward and I shared this aesthetic. We just liked writing code. Even if we were in a hurry, we would slow down even more. There was this sense of a level of engineering that most software engineers don't even know exists.

JIM: *When you work like this, with such an emphasis on the quality of design and code, are you going faster and producing higher quality?*

KENT: No question about it, when you operate like this, you are eking every last lesson out of the experience that you have. So you learn a lot faster. You can only have so many ideas; you want to suck as much juice out of each one as you can.

JIM: *Do you mean that there is a "gestalt" to mastery?*

KENT: Mastery comes from inside out, like a tree grows.

The third thread of XP comes from a feeling of oppression. [Kent is referring here to a story about his early career, retold in the preface to this book, when he badly missed a delivery schedule because the manager cut the development team in half but didn't change the schedule.] I thought, I really thought that I was a bad programmer, because I couldn't estimate. I've always kind of been a crusader in that regard, and I'm not sure it's always been the healthiest thing. But it's part of your makeup; it's what you do. Codependence is a great model of the relationship between business and programmers, with programmers playing the part of the alcoholic. The programmers just say, "If I was a better programmer, maybe they would stop beating me up." And I always said this and meant it, but a friend of mine is going through some rough times right now, and getting close to a real situation shows how insidious a codependency can be. The two just tear each other down and have no idea that they are doing it.

The fourth thread of XP comes from helping build software teams at start-ups and their need to maintain flexibility. In one place I worked, there was a compiler writer who wrote five lines of test code for every line of functional code. He was writing a Data Parallel C compiler, while another team was writing a Fortran compiler. He would just outproduce them. The light went on for me about testing, this whole idea of automated testing and that it wasn't an idea of an investment but an accelerator button, that the testing made people go faster. I'd always seen it as a zero-sum game; the more testing you did, the less time you had to develop code. He showed me that this testing moved the whole curve.

I then moved to a project, Well-Fleet, at an employee benefits firm. Very good environment. I came up with three architectures for database mapping, spreadsheet-like calculations, and workflow. I would spend a month developing prototypes and then show them to the client. At some point I was going to tell them about testing, so before one visit, I realized I needed to do something around testing. So the night I left, in classic consulting fashion, I wrote a little framework capturing "here's how I test" (as of four and a half hours ago, of course). I showed them how, and they started this type of testing. The whole project contained a basic set of practices, but not the detailed planning stuff. We had the testing process, and we continually refined the architecture. After we delivered the client/server spreadsheet solution, they realized they could rip out the mainframe calculation and put in the client/server solution at a fraction of the cost of rewriting the mainframe system. Even though the entire environment changed, we were able to "reuse" much of the system easily because of the test cases and the architecture.

So that project was an eye opener. Also, in reading *Chaos* (Gleick 1987) about the same time, I recognized the echoes in the few rules I'd given them and the emergent behavior that happened, without all the hoohah. This was in the 1993–94 time frame.

JIM: *It's interesting that so many of the Agile authors have used complexity theory as a conceptual base.*

KENT: Well, it's the only way to make sense of the world. There is no centralized explanation of how you ought to behave. CAS [complex adaptive systems] theory has a very simple explanation about how you ought to behave in the world. Find this set of rules and then act on them, measure the results, and tweak the set of rules. It's very liberating when you really take that in. You don't have to be in charge of everything,

you don't have to make all the technical decisions. In fact, to the degree to which as a leader you are making technical decisions, you are screwing up the dynamic of the team.

JIM: *But sometimes people want you to make decisions, and you have to push back.*

KENT: Oh, absolutely! Someone comes into your office and asks you for a decision. You ask them, "Which alternative do you think is the best?" They often respond that they don't know. "Well, if you don't know, maybe we'd better try them both and take the hit." When they don't want to take that much time, encourage them to just pick one.

JIM: *In these situations, a control-oriented manager will often use people's reluctance to make decisions to bolster their view that managers have to make the decisions after all. Pushing out that decision making helps everyone learn a new way of operating.*

KENT: Then came the C3 project at Chrysler, where I tried to be much more hands off. I wasn't going to make a single technical decision.

JIM: *You were trying to create the right kind of environment?*

KENT: Yes, Ron and I. We were trying to set up the initial conditions and then tweak them. No heroics, no Kent riding in on a white horse, which was quite a switch in value system for me. I prided myself on being the biggest object guru on the block. I had to deliberately give that up and just say that's not what's going to make me valuable.

JIM: *I've found that one of the problems with this approach to management is that success often seems almost accidental to people.*

KENT: The best manager I ever had was at the Stanford linear accelerator lab. I worked in the control room. This guy spent all his time making Heathkits.[2] Everybody complained about how lazy he was and why he didn't do this or that. But everything was always on time; if you needed parts, they were there. Personality clashes got resolved instantly. I've always wished that I wasn't 20 years old when it happened so I could appreciate that this guy was absolutely masterful. He kept everything running absolutely smoothly. So that's my aesthetic as a coach. If I do everything perfectly, then my contribution is totally invisible to the team. The team says, "We did this."

[2] For those who started life after the 1970s, these were electronic product kits that one assembled.

That's a hard thing. If your view of a leader is Teddy Roosevelt charging up San Juan hill, braving death with the bullets flying around, then it's just impossible to operate like this. But if you have a team that's operating in this self-organizing fashion, it's disorganizing to have someone inject that kind of leadership.

JIM: *What else has contributed to your views about mentoring and leading teams?*

KENT: I think being a parent comes into it, too. I have five kids—you learn when you have a larger family by today's standards just how much stuff really doesn't matter. That's how I enjoy them. I have three stages of emergency that I live by. Number one is seeping blood, which I'll take care of at half-time; dripping blood, which I'll take care of at the next commercial; and spurting blood, which I'll get up off the couch and deal with immediately. And seriously, some of that imperturbability helps me survive large clients where people yell and scream. I just chill out, and I can get through it.

I grew up in a family that hated confrontation. Everything was quiet, and then every six months there would be a big, scary blow-up. And so, part of my personal growth was learning to invite confrontation earlier and learning positive ways of dealing with it. I think XP really goes on the side of putting everything on the table. That mirrors my relationships. Tough as it is, the truth is the way to go. There is a line from [David Mamet's play] *Glengarry Glen Ross* that goes something like, "Always tell the truth. It's

one story that's easy to keep track of." There is a sense in which you are honest because it's the right thing to do. But, in a way, anything else is so complicated, it's just not worth it.

JIM: *When I was in India, I frequently got questions about sign-offs. Do you think they are useful?*

KENT: Sign-offs mean "I promise not to complain if you do something that follows what you think I just wrote." Forget it; if the person with the money doesn't like what you did anyway, it doesn't matter. If we're worried about what the user is going to say about the system, let's show it to them sooner and more often rather than later and less often. That's one of the games I always play—the opposite game. If anyone states something as the truth, I always reverse the polarity and ask myself, "What would be true about the opposite?"

JIM: *What is your long-term vision?*

KENT: Growing a vibrant community around XP is probably my number one goal at the moment, which, because of the way people listen to the things I say, usually means shutting up. What I say takes on inordinate weight, which is sad for me. I would like to be just one of the guys and just do what everyone else does. So I have to be real careful.

I'm no longer interested in the technical practices of XP, the practices around delivering high-quality, high-productivity, high-flexibility, predictable software. There will be a lot of work around making that happen, cranking it up the next factor of two, but it's not something I care about. What I care about is extending XP in scope in a couple of different ways. One is in specialties—user interface design, graphics design, traditional business consulting. Because there has been no change in the way business consulting is practiced, there is an opportunity to really turn that on its head. I want to tighten the integration between business advice and validation with actual implementation. You can spin that up, put everyone on the same cycle. So I'm actively working with Lante to deliver that, to figure out what it is. But it has the potential to really revolutionize business consulting.

Do you remember the commercial where the two consultants are sitting across the table from the CEO, and he says, "OK, do it"? But they don't do that part? That's the opposite of what I'm talking about. Let's implement an itsy bitsy part of your supply chain and then build on that—so, Extreme Consulting. Another aspect of extending XP is scale. I'm working with Iona Technologies, an infrastructure provider, to create a real team out of the many people who contribute to the customer's voice. There are too many layers of marketing and product management. The best "customer" I ever had was a user who actually did the job for seven years. She was so fed up with the application she nearly quit, but the company got her to continue. I strive to get the programmer together with the person who has the problem so what emerge are features that delight the customer. Don't just introduce another layer in the hierarchy.

JIM: *Do you think the objective of Agile approaches like XP is to get orders-of-magnitude better results?*

KENT: Absolutely—on a bunch of different dimensions. I don't want this [XP] just laying someplace different on the curve. We're trying to increase business value by an order of magnitude. While this may include dramatically lower defects, dramatically higher productivity, dramatically more business feedback, or the ability to change business functionality at just insane rates, in the end, business value can't be measured by any single simple metric.

JIM: *With regard to change, it seems to me that we should* encourage *customers to change.*

KENT: Right, we don't want requirements creep; we want requirements to get up and dance. The problem is that they are just creeping. New requirements are caused by evolving businesses, and you want as much of that as possible.

JIM: *You have to make it easy for people to request changes, to think about changes, to implement changes.*

KENT: Right, we want to discover new ways to do that. That's why we don't have prioritized features [in XP]. We just have what we are working on now and what's in the future, because it's all subject to change. We're living in this moment. Prioritizing features [as in high, medium, and low priority] is just not useful.

JIM: *Do you mean we have to view the plan as variable?*

KENT: A plan isn't a prediction. It's there for collaboration. It's there to help make go/no go decisions. It is not there to dictate.

JIM: *In many IT organizations and consulting organizations, people are dissatisfied with processes and their results, and the clients don't like them either, but everyone wants to continue doing things the same way because that's the way it has always been done.*

KENT: Exactly. I would love to lighten it up. If you look at nature, there are three to four rules in emergent systems like anthills—nature doesn't have twelve. Twelve is too many [Kent is referring to the twelve practices of XP]. So I like to go in the other direction, remaining cognizant of Alistair Cockburn's comment "embellishment is the methodologist's trap." We can always find a reason for adding one more thing, one more practice—I don't celebrate that. I'm working with Bill Wake on a "XP 123" or "XP in a box." Just enough material for four iterations, a couple of months. Very simple stuff. It's a fun exercise, deciding what we can cut out, since I'm only going for a six-week project.

Reflections

The idea of social contracts runs deep for Agilists. On the surface, XP seems to be about programming—refactoring, test-first development, coding standards. But listen to Kent, talk to him in a number of situations as I've had the opportunity to do over the past couple of years, and you realize that Kent's most important vision is about changing social contracts, changing the way people treat each other and are treated in organizations. In an email, in which Kent was angry about an article that discussed a "revised" approach to XP, he says, "I started on this crusade sitting on the steps of a building in downtown San Francisco trying to get my blood pressure down after watching a tin-pot dictator belittle and abuse his whole team. Anything that reminds me of that gets my blood boiling again." Continuing on about the article, Kent says, "I was furious that someone would strip out all of the social change and still call it XP. If he wanted to say, 'XP inspired me to make the following changes,' fine, but what he did was emasculate his teams."

Some people think "Extreme" is too extreme, that XP would be more appealing with a more moderate name. I don't think many people would get excited about a book on "Moderate Programming." New markets, new technologies, new ideas aren't forged from moderation, but from radically different ideas and the courage to challenge the status quo. The Agile Software Development movement owes a lot to Kent Beck.

Chapter 5. Deliver Something Useful

HAHT Commerce, Inc.

"These are the days I get the most energy from work," remarked Rowland Archer, chief technology officer of HAHT Commerce, Inc. Archer was referring to the days when HAHT holds focus group sessions with its customers. These sessions are conducted at the end of each project milestone to review progress on a working application and encourage change suggestions. "Getting the customer's voice is absolutely critical," says Sam Bayer, former VP of partner programs and chief evangelist for HAHT. "Furthermore, you have to be physically present with the customer."

Sam should know. He and I codeveloped the original RADical Software Development methodology in 1992 that evolved over the years into Adaptive Software Development. HAHT's development methodology today is a refined version of that original approach. In close to a decade, Sam and I have been involved (both jointly and separately) in more than 150 projects that encompass the basic tenets of short iterative cycles, feature-driven planning, end-of-cycle customer focus groups, and joint kick-off sessions. Everyone at HAHT, from the CTO to individual developers, uses the HAHT rapid application development (RAD) methodology to maintain close contact with customers.

"We sell our technology first," said Sam, "but clients rave about our delivery process." The "process" is a strategic core competency for HAHT. "When we present our core competencies to prospects, our first slides deal with our technology," Sam continued. "But Rowland includes our implementation methodology as a core competency. In fact, he would attribute the methodology to converting all of our clients—all of our clients, without exception—into raving fans of our company, because of what they have learned in the process of implementing our technology about allowing the voice of the customer to come into their projects."

HAHT uses its technology, its RAD process, and its business domain knowledge to build Demand Chain Management Solutions for manufacturers. With its application server technology, HAHT has developed a series of commerce application products that are integrated with enterprise resource planning (ERP) software from SAP and J.D. Edwards. The company works with customers like Minolta, which sells copiers and printers. Minolta needed a Web-based application for its dealer networks, its primary marketing channel. HAHT's out-of-the-box applications usually provide 70 to 80 percent of the customer's required functionality. The other 20 to 30 percent is delivered in one or more 90-day projects that customize HAHT's standard Commerce Suite applications. These 90-day projects are typically staffed with five or six people.

Many companies sell sizzle and attempt to keep prospective customers in the dark about the realities of their product until *after* the sale, when it's too late to change. HAHT's approach to its customers clearly demonstrates that both customers and development organizations work best when all their cards are out on the table in the beginning. HAHT's sales process includes a discovery workshop (a joint application development [JAD] session) in which the customer's business processes are examined through the eyes of one of HAHT's Commerce Suite applications. A detailed functional analysis of the gap between the

software's capability and the customer's needs becomes the input to a function point–based estimating process.

"In the beginning, the sales force hated this approach," said Sam. "It added a day or two to the sales cycle and uncovered the reality—the real dirt about our products—and the sales guys don't like that. Obviously, the customers loved it. They really want to know what it's going to take to implement their required functionality, and they don't want a snow job—they've been there before. Our clients don't *like* this approach, they *love* it!" The sales staff—responding to the customer reaction to this process—have now fully integrated it into their sales cycle and are staunch defenders of the process.

The experience at HAHT points out that Agile approaches impact much more than the development process itself—they change the very nature of developer-customer interaction and the way success may be measured.

The success of HAHT's Commerce Suite and effective delivery process has changed its fundamental business model. In 1997–98, the company began selling its application server product, and applications were an afterthought. However, in a crowded server market, HAHT saw that developing specific applications—primarily those to Web-enable ERP software—would help sell its server product.

In early 2001, HAHT announced that it would make its application products available on IBM's WebSphere server (as well as continuing to sell its own application server). HAHT's core competencies have therefore emerged as its domain knowledge of ERP systems, its expertise in externalizing demand chain systems to the Web, its packaged Commerce Suite products, and its Agile practices for customizing those products to specific client needs.

Change tolerance is an attitude as well as a process. Although becoming change tolerant requires a good process, it's the attitude that often strikes first. In 2000, Sam moved from head of consulting services to business development, bringing on resellers and partners, particularly to address the needs of small to mid-market (less than $1 billion in revenue) manufacturers. In moving to this market segment, Commerce Suite applications needed to be redesigned, since few smaller firms use SAP's ERP software. HAHT brought in new partners along with a few potential customers of their joint offerings and conducted a focus group aimed at uncovering gaps between operating processes at these smaller firms and HAHT's applications.

HAHT's partners were somewhat aghast at first. "What happens if we have this focus group and they uncover unique things in this market?" one lamented. "Of course, this is exactly what we want," Sam said at the time. "Now we can do an analysis of what we need to change based on customers who have been there. Whatever we hallucinated beforehand, in terms of what we needed in the product, can now be changed based on the focus group result."

Sam concluded, "We exposed these partners to our culture of change tolerance driven by customer requirements, and they were fascinated by it. The key is to get the customer voice in, and what we've found over the last three years is that unless you physically get the customer voice in (on site), you get into all sorts of meaningless internal arguments. Furthermore, the longer you go without this direct customer interaction, the more insular you become."

Customer Delivery Principles

This chapter has two themes: the principles and the practices of delivering something useful to customers. These principles are drawn from three principles of the Agile Manifesto:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
- Business people and developers must work together daily throughout the project.

The practices of customer interaction involve understanding the optimal information "interface" between customers and developers.

Delivering Customer Value

In the final analysis, the critical success factor for any method—Agile or otherwise—remains whether or not it helps deliver customer value. Furthermore, we the technology developers don't get to determine value, our customers do. Whether an internal IT customer uses an application to cut costs or a retail customer buys a software package, ultimately customers buy our products—or they don't. It's just as simple, or as complex, as that. A group of the Agile Manifesto authors were concerned that this "satisfy the customer" principle was too obvious, that everyone espouses a similar principle, and therefore it didn't help differentiate Agile Software Development Ecosystems (ASDEs) from Rigorous Software Methodologies (RSMs). However, the overriding issue turned out to be that a customer value statement declares an external focus rather than an internal one. Without this explicit statement, even if it appears obvious, development teams tend to be overly introspective. The levels of the Software Engineering Institute's Capability Maturity Model (CMM), for example, are not based on results, but on the implementation of certain sets of practices (called key practice areas, or KPAs). The underlying assumption—good processes lead to good results—is both a questionable assumption and one that leads to too much introspection about process and too little focus on actual results.

This concern about customer satisfaction as a principle leads to another question: "What *does* separate Agile from rigorous approaches to delivering value?" I think the key issues are that the customers are in control and the relationship must be a collaborative partnership. The subtle, and sometimes not so subtle, message of RSMs is that once the requirements specifications and project plans are completed, the technical development team takes over control. Small changes are acceptable (after review by a change control board) but RSMs' primary objective is to conform to the plan—to deliver a succession of artifacts, not working software.

RSMs have often led to lack of accountability on the part of both customers and developers. Developers claim, "They signed off on the specifications," and the customers counter, "Yes, but we didn't understand all that technical mumbo jumbo—the results aren't what we wanted." Both parties fulfill the prescribed plan, process, and approvals but fail to be accountable for real results.

RSMs have also led to token relationships in which developers involve the customers but retain control. Development teams encourage customers to get involved (developing requirements or writing test cases, for example) but maintain control through the mechanism of "The Plan," which the customers don't understand because it's laced with hundreds of tasks that sound to them like "develop a bumfizzle document" and "provide a gruelsniffle."

Agile approaches present a different, and sometimes scary, scenario to customers. The process is such that the customer is in control *throughout* the project. Every iteration, the customer gets to change priorities based on features delivered, features wanted next, and changes requested from previous iterations. The development team's responsibility is to inform the customers what the impact on cost and schedule will be and to present alternatives that might be faster or lower cost, but in the end, the customers are in control.

Shifting control is scary—for both developers and customers. If I'm a customer and I know that I'm ultimately responsible for the results, I've got to stay closely involved. If I'm a developer and the customer keeps changing the features to be developed, I have a difficult time measuring progress (at least in the traditional manner) because the product is constantly changing. Scariest of all, developers and customers have to trust each other and trust in each other's competence.

Although the customers should be in control, this principle does not, and should not, relieve development teams of being proactive about recommending features (or user interfaces, integration schemes, design enhancements, and other possibilities). Agility has both reactive and proactive components. Customers want to buy two basic things from software providers—technology knowledge and domain knowledge. If a company were selecting a software firm to develop customer relationship management (CRM) software,

and it had two bids from firms with equivalent technical knowledge but one had extensive CRM knowledge as well, which one should the company pick? If software development were a reactive activity only—the customer provides feature requirements and developers implement them—the selection should be a toss-up. But, of course, it is not a toss-up. Clearly the second firm, with the CRM experience, would be preferable.

In a recent workshop, a project manager was questioning the feature approach to iterative planning. "But aren't requirements specifications and architecture documents important?" he asked. "Yes," I replied, "they are important, but what we have to understand is that customers don't care—about documents, UML diagrams, or legacy integration. What customers care about is whether or not you are delivering something useful to them every development cycle—some piece of business functionality that proves to them that the evolving software application serves their business needs."

One last aspect of delivering customer value is time to market. While speed may not be the only component of value, it will certainly remain critical. RAD methods arose in the early 1990s as a way of increasing the speed of software development, which had often slowed to near reverse during the information engineering era. From a methodology perspective, ASDEs draw from both the heritage of effective software engineering techniques and what RAD projects taught us about short iterative cycles, working closely with customers, and effective teaming practices.

Voice of the Customer

There are two assumptions about customers that lurk just beneath the surface for many technologists: that customers don't know what they want and that they are shortsighted. The extensions to these assumptions—"The customers don't know, but we do" and "The customers are shortsighted, so we'd better do the long-term planning for them"—turn out to be very dangerous, particularly in an exploratory project. Many RSMs are built on the assumption that the customers don't know what they want, so the developers want a detailed specification so they can absolve themselves of responsibility: "I built what *you* put in the specification" is a common excuse. Requirements processes are important, but how we view the customer is critical. Agile practices are based on the belief that neither the customers nor the developers have full knowledge in the beginning and that the important consideration is having practices that will allow both to learn and evolve as that knowledge is gained—without ongoing recrimination.

The second dangerous assumption is that the customers are shortsighted, and therefore we (the developers) must build in extra features and extra data so that when the customers finally get a clue, we're already ahead of them. This assumption leads to overdesigned databases, class structures, and cross-application integration schemes that attempt to solve future problems. Many of these "extras" are done in the spirit of future "flexibility." Part of being change tolerant is having the conviction—*and* the development and design practices to back it up—that we are more capable of responding to future changes when they occur than we are of predicting those changes today.

This should not imply that there is anything wrong with designing a database for cross-application integration; however, it should be designated as a feature and prioritized by the customer. If we can't sell it to the customer, it shouldn't be done. There was a series of emails on a recent discussion group that addressed this issue. One particularly relevant response came from Ron Jeffries, responding to the need (or lack thereof) to build standard application programming interfaces (APIs).

> What I'd really say is certainly "don't build anything you don't have a story for." I'd also ask the customer—if he didn't already know it—whether he expected to "facilitate integration activity," whatever that is. If he said no, I'd push, but ultimately accept the answer. If he said yes, I'd suggest that he provide some of those stories early—if they have significant business value—because we would learn something from them.

> However, since (by assumption) the code we create is nearly perfectly modular, I'd expect to be able to put API features in easily whenever they came along, because I've always been able to so far. (The future will be much like the past, because in the past, the future was much like the past.) Personally I

really do try never to write a line of code that is not directly in support of the customer stories I'm working on right now. It's a choice I make to see whether I ever get in trouble, because frankly I'm not afraid of getting in trouble: I expect to recognize trouble early, and I expect to be able to get out of it because I know some bright guys I can call up to help me. As far as I can see, Kent [Beck] does the same thing, but he does it more by habit than as a conscious experiment.

The issue is what one is afraid of: As Kent says, all methodology is based on fear. I greatly fear not keeping my customer happy, and I have very little fear of getting technically cornered. Others may have a higher level of concern about getting technically cornered—perhaps, for example, because the team isn't very good yet at refactoring. They might make a different tradeoff. I have had the luxury of being able to take XP, which I didn't invent, and push it to the limit. I've found that it doesn't break down—IF YOU DO ALL THE PRACTICES. It leads you to a very interesting place where there is little speculation, little tension, just this rhythmic cycle of test/code/refactor that produces wonderful code almost automatically.

How many millions of dollars have been wasted on projects because developers have patronized customers—putting in "stuff" because they thought the customers weren't smart enough to know they were going to need it in the future? These thoughts may not be expressed, but they lurk underneath the surface in development organizations. Read carefully what Ron says: He "tries" to get the customer to think about integration issues but doesn't make the decision. Many people might say that an integration decision is a technical decision, but integration is a feature, and it should be driven by customers' needs. If they want quick-and-dirty, and we've pointed out the negative long-term impact, and they insist on quick-and-dirty—give it to them. ("Dirty" here refers to lack of integration, not defects.)

Now, there may be a number of issues surrounding how customer decisions are generated—single on-site customer, consensus from a focus group, executive sponsor—but ultimately the customer needs to make decisions about features and functionality. It is a development team's professional responsibility to provide information about the advisability of the decision, but not to *make* the decision.

Sam tells the story about visiting with both the CEO/founder and the president of a small high-tech company (Bayer 2001). He was trying to convince the company to use focus groups to enhance its understanding of customer needs. In the first meeting, the founder (the technology guru) responded somewhat indignantly, "We've been doing this for five years. We know exactly what our customers want." He went on to articulate a list of the product's technology features. Later, in a meeting with the president of the firm, Sam learned that the president didn't think the company knew either who its customers were or why they bought the company's product.

Listening to the voice of the customer is the most difficult task in product development. Building a partnership of shared responsibility, giving customers control over features and priorities, spending enough time in conversation to understand business processes and the customer's real problems—these are the linchpins of Agile development.

Satisfying the customer through continuous delivery of valuable software is not an easy task. At HAHT, implementing focus groups took the energy and drive of Sam Bayer. Finding the voice of the customer doesn't come easy. Getting the right users on a project can be difficult. Delivering something useful, delivering a product that creates value for customers of software products—whether commercial, PC, or internal IT software—begins with practices that create effective working relationships between development staffs and their customers.

Working Software

Long ago and far away, in my heavy methodology days before enlightenment, I was delivering a seminar on structured analysis and design. The whiteboards were filled with data models, data flow diagrams, and process specifications from four long days. At this point I turned to the group—consisting of analysts and programmers—and asked, "Now, you know how to code from these diagrams, right?" The blank stares informed me that they didn't have a clue, so I then began writing code blocks on the board and drawing

lines back to the locations where the information came from on the model diagrams. I could see the lights begin to go on in people's heads.

I took a profound lesson away from that experience: If software developers couldn't relate models to code, how in the world were nontechnical customers going to relate models to a finished application? The debates have raged for twenty-plus years about which form of system model—data flow diagrams, data models, object models, and so on—are most understandable to users. And the answer is: none of the above. Users relate to the real thing and the real thing only. Paper mock-ups and visual prototypes do serve as effective intermediate markers at times, but real progress is demonstrated only by working software.

The principle "working software is the primary measure of progress" caused intense discussion among the Agile Manifesto authors. Questions ranged from what the term "working software" meant and whether it included prototypes and models to whether working software was the only measure and who got to measure. If we can't show demonstrable, tangible progress to the customer, then we don't really know if we are making progress. Data models, UML diagrams, requirements documents, use cases, or test plans may be markers that indicate some degree of progress (or at least activity), but these markers are not primary measures. Features that implement a portion of a business process, features that demonstrate that a piece of flight avionics software actually drives the required actuators, features that allow a doctor to extract and view a digitized CT scan—these are primary measures. Nothing else qualifies.

The principle doesn't say "compiled binary executable," it says "working software." If working software can be generated completely from models, then the delay between artifacts (models) and working software approaches that encountered between compiler input and compiler output. If we develop a set of UML diagrams but converting those diagrams to tested code takes weeks, then the diagrams (models) do not constitute a primary measure of progress. Similarly, although DSDM uses prototypes, in practice this usage has gravitated toward the definition of a working portion of the application rather than a throwaway prototype. Time-driven projects don't have time for throwaway prototypes.

Good project managers have always strived to base milestones on verifiable deliverables—end products rather than intermediate markers of progress. Traditional software life cycles, however, have placed far too much emphasis on intermediate markers such as diagrams and documents. ASDEs are the third wave of software development practices. The first wave was the ad hoc wave, in which the software itself was in fact the measure of progress. Because software development was in its infancy and making major changes was costly, a panoply of intermediate markers—designed to reduce the cost of errors—arose. This second wave sought to reduce the cost of change by eliminating it. Unfortunately, the primary focus became these intermediate markers, not the working software. As the third wave unfolds, as tools and practices reduce the cost of change, we return to the principle of working software as the primary measure of success.

Frequent Delivery

The fastest development cycle I've run across was in a project managed by a Philadelphia-based software firm. For an Internet project with a large customer base and a very short delivery schedule, this company delivered a new iteration of its product every day to more than 100 users. Given the extremely short time frame for the project and the degree of requirements change, getting daily feedback from the customers was absolutely critical to the company's success.

For many years now, process gurus have been recommending an incremental or iterative style of software development with ever-growing functionality. While this still seems to be less common in the wider world of software development, iterations are characteristic of all Agile projects.

However, "deliver" is not the same as "release." Business people may have valid reasons for not putting code into production every couple of weeks. They may want longer cycles to allow time for training, or they may not want to spend time retraining for frequent releases. They may not find enough functionality in the early increments to be worth deploying. Projects may not reach a critical mass of functionality for a year. Still, that shouldn't stop a rapid cycle of internal deliveries that allows everyone to see what's happening and enables constant learning from a growing product.

Frequent delivery benefits both the customers and the development team. The "frequent delivery" and "working software" principles work in conjunction with each other. Neither frequent delivery of documents nor infrequent delivery of working software conforms to Agile principles. Developers may feel that frequent delivery, especially when it involves practices such as focus groups, is disruptive and involves needless overhead. Customers may resent the intrusion into their busy schedules. But volatile projects can careen off course quickly. Developers begin assuming too much, customers become complacent, and the results can be disastrous.

There is an old project management adage that "bad news gets worse with age." A corollary to this adage might be that "poor features get worse with customer neglect." By the time the members of a development team have worked on a feature set for two or three months or longer, they begin to take "ownership" of those features and form natural ego attachments. Customers, cut off from seeing results, lose track of development efforts and have a tendency toward excess criticism when they do see results. Monthly customer reviews build partnerships. Quarterly reviews are apt to contribute to adversarial relationships.

It's ironic that the more sheer joy and satisfaction we get from our technical work, the easier it is to get distracted from delivering what the customer wants. One of the marvelous attributes of the three extremoes—Ward, Kent, and Ron—is that they are consummate programmers dedicated to high-quality, nonsmelly code, while at the same time they are laser focused on delivering customer value. It's a rare combination.

Work Together Daily

The other key differentiator between Agile and rigorous approaches is the emphasis on collaborative partnerships. At a talk before a large financial institution, I discussed the principle of daily interaction between the development team and the customer. One project manager in the audience stood up and explained how that would be impossible in his company since they rarely saw customers and the customers wouldn't stand for such a time commitment. The response was simple: Don't do exploratory projects or use Agile practices with this client. All methodologies recommend customer involvement in projects, but Agile practices and principles insist on the customer's taking control and being constantly involved.

Agilists don't expect a detailed set of requirements to be signed off at the beginning of the project; rather, we anticipate a high-level view of requirements that is subject to frequent change. Clearly, this is not enough to design and code, so the gap is closed with frequent interaction between the business people and the developers. The frequency of this contact often surprises people. The word "daily" is in the principle to emphasize the software customer's continuing commitment to actively take part in, and indeed take joint responsibility for, the software product.

Years of research studies show that user involvement in projects ranks first or second in importance to project success. Gathering requirements continues to be one of the most difficult challenges of software development. The fact that there are books and conferences devoted to requirements analysis tells us that it's hard and, whatever techniques are used, it will still be very hard to write down everything customers want and expect before beginning work.

During the 1980s, I facilitated dozens of requirements gathering (JAD) sessions for companies. Although they were usually effective in gathering requirements, it wasn't until the early 1990s, when I began combining these sessions with frequent delivery of working software, that customers began showing real enthusiasm for working with development teams on an on going basis. Working together daily and frequent delivery of software work conjointly to build customer-developer partnerships. Without working closely, the wrong requirements get implemented, and without frequent delivery, customers lose the incentive to work with developers.

Finally, does "daily" really mean *daily*? There was debate among the Manifesto authors over whether to substitute "frequently" for "daily," and I think we kept the right word. Of course absolute adherence to a daily schedule may not always work. Working together daily may mean for a couple of hours each morning rather than all day, and it may mean skipping days now and then. However, when daily begins

slipping to every other day, and then to once a week, results will suffer accordingly, particularly if time to market is a driver.

Practices That Deliver Useful Features

The principles that form the foundation of how customers and developers interact must be instantiated with practices, such as XP's on-site customer or ASD's customer focus group. Implementing the appropriate set of practices in an organization depends upon an understanding of the information flow across the customer-developer interface and how the relationship can be formalized in a contract or project charter.

The Customer-Developer Interface

In order to better understand the relationship between customers and developers, let's postulate the equivalent to an API—we could call it a customer-developer interface. Now, for a small (less than ten people), colocated team of developers, what would be the simplest interface that could possibly work? First, the role of customer would be played by a single on-site customer, per XP. Second, the "parameters" that would need to pass back and forth across the interface would be a product vision, a set of identified features, a prioritization of those features (which ones to work on in the next iteration), a set of conversations around those features and prioritization decisions, and acceptance tests. Developers need to know what features are required, and then they need to understand the features. Several conversations, and minimal documentation, may be required to accomplish that understanding. The customer needs to have information with which he or she can make tradeoff decisions—from user interface design to feature alternatives.

The above could be considered the simplest form of a customer-developer interface. Furthermore, no matter how many individuals are on the customer "team" or the development "team," the requirements for the interface remain the same. Now let's ask the question, "What is the simplest thing that will possibly work if the customer side of the interface includes two different departments (sales and order processing), each with several different users?" In this case, the interface remains the same—vision, features, priorities, conversations, and acceptance—but the single on-site user becomes a group of users whose ideas must be gathered and massaged. Now, if we are dealing with a group rather than an idealized on-site user, then the task is to make the output of the group mirror the output of the single user, which is what practices such as facilitated workshops and customer focus groups are designed to accomplish.

XP recommends having an on-site customer. ASD recommends having a customer available for conversations each day. Crystal Clear discusses frequent user interaction. What happens, though, when the number of users (or user departments) grows to 3 or 4 or 50 or 75 million (as in the case of Microsoft consumer products)? Obviously, the XP-specific practice of an on-site customer doesn't scale, but what if we interpret the practice as a single instance (small, colocated team) of an interface? I would argue that no matter the number of customers or developers, the interface itself remains stable.

If we had four user departments and 50 users, we might use a requirements gathering group session (often referred to as a JAD) to create a product vision statement, identify the features on story cards (XP), develop low-precision use cases (Crystal), and develop a backlog (Scrum's prioritized feature list). Some prioritization process such as multi-voting among the user community would also be necessary. As each feature entered development, selected users would be asked to have "conversations" with developers to further articulate requirements. At the end of each iteration, the development team might conduct acceptance tests (XP) and a focus group (ASD) to get appropriate feedback from the original user group. As the number of users increasd, these additional practices would be brought into play, but only as absolutely needed—that is, after determining the simplest set of practices that will fulfill the needs of the interface.

Proxy Users

Many people think that internal IT-based products and commercial products are different when it comes to working with customers. However, the fundamental problem is the same—at some point there are inevitably "proxy" users inserted between the developers and the actual users. For internal products, these people are often business analysts or systems analysts. Their job is to work with tens to hundreds of users (actual users, executive sponsors, business functional experts, and others) in order to create the information flow required by the developers. Unfortunately, the conversations—in which the most important information resides—suffer from the proxies' translations. Proxies don't have the depth of knowledge that actual users have, and the problem is compounded because the proxies are overly prone to substitute documentation for conversations.

Similarly, the "proxy" roles in commercial software firms have titles such as product manager, marketing analyst, and program manager. Regardless of the role, they are still proxy users. In some situations—with consumer products like Microsoft Office—there may be several layers of proxy users. Although proxy users are necessary in certain situations, they always cause information to be skewed. That's why consumer software product companies employ usability labs to occasionally get the pulse of the real customers. That's why companies like HAHT use periodic customer focus group sessions.

The biggest problem with proxies comes when they begin to believe that they actually *are* the customers, and in particular, when they use this belief to block direct contact between developers and real customers. Now, having 75 million customers visit a developer's office may not be feasible, but having the developers view a customer in a usability lab may be. Having 50 users working side by side with a development team may not be feasible, but having 10 to 12 of them in a focus group may be feasible—and very useful.

I've seen systems analysts intervene to stop users from directly talking with in-house developers, and I've been around product marketing managers who took the position that "they" knew what the customers wanted and development shouldn't worry. I've seen program managers (the *specification* gatherers often situated between marketing and development) who thought requirements should be a linear flow from customer to marketing to product management to development—with no feedback from development.

Having conversations is difficult enough; proxies tend to make the problem worse. They serve a valuable and necessary function, but they can't forget that their most important job is not feature identification or prioritization but the conversations that transfer key knowledge back and forth between the actual customers and the development team.

Domain-Knowledgeable Developers

Ten or 15 years ago, a customer (again, either internal or external) could walk into an IT manager's office, spend 30 minutes outlining his or her problem, and expect results with minimal further interaction. Even though the lack of customer involvement was a recognized problem (and many projects failed because of it), a surprising number of projects succeeded with minimal involvement. Why? In many instances the IT analysts and developers or the external consultants had enough domain knowledge to enable them to fill in the gaps. In prior years, development teams asked for customer involvement, didn't get it, and got the job done anyway—at least often enough that we convinced customers that their involvement wasn't as important as we kept telling them.

With new business models, with eCommerce and eBusiness, IT professionals are less able to bridge the domain knowledge gap. This means that continuous customer involvement is no longer merely an option, it's mandatory. If we don't build effective collaborative relationships with customers, no amount of contract negotiation or change order processing will bring success. When no one has the answers in the beginning, the cost of learning has to be shared. If both parties fail to understand the nature of the turbulence, if they don't understand that constant change is the name of the game, if they don't understand that consistent joint decision making and tradeoffs are required, then the chances of ending up with a satisfied customer are Lilliputian.

The best developers have knowledge of the customer's domain. While, in theory, the customer-developer interface just described should work with customers who possess all the problem domain knowledge and

developers who have all the technical knowledge, in reality, developers with domain knowledge reduce the level of conversation required and the rework from misinterpretations. In the past year, I've worked with software groups whose "customers" are income tax specialists, geophysicists, physicists, biologists, banking and financial specialists, eCommerce retailers, and cell-phone chip experts. If these groups had no knowledge of problem domains, their software development efforts would be noticeably poorer and markedly more time consuming. Having domain knowledge, they can understand the terminology used by the customers, recommend features based on their knowledge, and even use their own experience to fill in detail-level requirements gaps.

Although the best practice may be to have a user readily available to the development team, it is not always possible within large, multi-national corporations. Although the best practice may be to have customers make all feature and priority decisions, there are situations in which the development team may inject its own ideas. Business and systems analysts fulfill a vital role on development teams—they possess valuable domain knowledge. Development teams shouldn't take an arrogant, "we know best" attitude, but neither should they take a "customers know everything" one.

The critical point is that customers should

- Establish the vision and scope for the product
- Make *most* of the feature and priority decisions
- Generate *most* of the requirements specifications
- Converse *frequently* with the development staff
- Accept the interim and final features

Customers should be heavily involved, as every product development process needs an external reviewer. However, this should not preclude development staffs from using their domain knowledge to make significant contributions to features and functionality.

An API operates between programs—it should be concrete and standardized. A customer-developer interface isn't the same—there should be room for fuzziness to fit the variations among individual customers and developers. We want to forge a collaborative bond between customers and developers, not a different kind of wall. If we attempt to wall development teams off from the reality and turbulence of the real world, Agile approaches will be no more successful than traditional ones. We need collaboration, accountability, and definition of roles defined by an interface, but we also need to be adaptable to actual conditions within companies.

Contracts: Shaping Customer Relationships

In many organizations, manufacturing and sales departments still play the traditional sales forecasting game. Sales departments make product sales forecasts. Since sales people are traditionally optimistic, manufacturing personnel (having been burned with excess inventory in the past) then reduce the sales forecasts by some percentage. Obviously, the sales staff know this and therefore bump forecasts up by an offsetting percentage. So, everyone understands that everyone else is lying (well, fudging the truth) so the battle of phantom numbers escalates unabated until no one has a clue about reality—until earnings reports arrive. Early generations of manufacturing resource planning (MRP) systems floundered on this political reality.

There are two strategies for "fixing" this type of problem. Strategy number one is to constantly cajole everyone involved to "play nice." This is a poor long-term strategy given human nature, although trying to get everyone to be honest does help. The second strategy, more radical to be sure, is to alter the entire game. Rather than forecast demand, respond to demand. That is, when someone orders a widget, build a widget. Of course there are limits to a completely demand-driven manufacturing operation, but the process of moving from being forecast driven to being demand driven has many additional benefits, such as lower inventory and less risk of overstocking.

Short-cycle, iterative software development has demand-driven characteristics. Rather than try to forecast feature delivery capability far into the future—a thankless task, as we all know—why not respond to demand rather than forecast it? Plan an iteration with a certain number of features, measure how many were actually delivered, tell the customer what the delivery capability will be on the next iteration, and have the customer prioritize the remaining backlog according to that demonstrated capability. If an auto manufacturing plant has the capacity to assemble 550 cars a day, then forecasting 750 cars for March 24 is a useless activity. Software development groups are similarly capacity limited.

Aggressive schedules, evolving technology, rapidly changing requirements, uncertain business models, fixed-priced contracts—pick out the term that seems incongruent with the others. "How do we write fixed-price contracts using Agile approaches?" is a question I hear frequently from clients and workshop participants. "Wrong question," is my usual reply. But while "wrong" in some respects, it is a reasonable question, and the issue of writing "Agile" contracts needs to be addressed.

The fundamental problem is that the business world is increasingly unpredictable, and traditional fixed-price contracts imply predicable results. While we all acknowledge the constancy of change in today's world, we want vendors to bid as if they are better at predicting the future than we are. In many regards, today's contracting process is attempting to guard us against yesterday's problems.

There are really a series of questions involved in delivering projects, and what type of contract to use is last on the list.

- What are the characteristics of the problem domain?
- What is the best process for accomplishing the work?
- How do we establish a good working relationship with the customer?
- What kind of contract do we use?

When waterfall life cycle development prevailed, developers (internal or external) were asked for fixed-price estimates prior to definition of requirements. At best, one estimate or contract was issued for requirements work and a second for development. However, even with seemingly complete requirements, this approach was often fraught with problems and politics and often ended up with extensive changes near the end. And this was at a time when 10 to 15 percent scope "creep" was considered a problem.

Today the problem domain is different. Business models are evolving and morphing constantly, technology is changing rapidly, Internet time demands rapid response, and technical skill levels are suffering as development teams struggle up one technology learning curve after another. In such a turbulent problem domain, short, iterative, feature-driven (specify, build, review) exploratory development cycles are becoming "best practice" for accomplishing the work.

Building good working relationships with customers and users has always been important to building good software applications. However, in today's world of exploratory projects, good working relationships—true partnerships in which both parties understand their roles and responsibilities—are imperative. They are imperative because every change demands that a choice, a tradeoff, be made. Referring to a formal contract or processing a "change order" for each of these choices will slow projects to a crawl.

So if the problem domain is volatile, the best way to accomplish the work is through an evolutionary development process, and establishing a close working relationship with customers is imperative, then what type of contract matches this environment?

The answer, I think, is a contract based on periodic delivery of features—something the customer can determine has value. Some call these fixed-schedule, variable-scope contracts. I like the term delivered-feature contracts. They are sometimes, wrongly I think, considered time and material contracts. Time and material contracts don't—directly at least—tie in features delivered to the customer. With a delivered-feature contract, the development team demonstrates features to the customer at the end of each short delivery iteration. The customer evaluates value delivered, and if acceptable, the development team and customer proceed to replanning the next cycle based on current information. Commitments for the next

iteration are based on delivered value and the next iteration's plan. There are, of course, overall targets for schedule, cost, and scope, but these are not contractual obligations on either party.

Delivered-feature contracts are the ones that make the most sense in this environment, because they best fit the problem domain, how the work should be accomplished, and the necessary working relationship with the client. But actually, if you have the first three, nearly any type of legal contract can be made to work. If you don't have the first three, no type of legal contract will work.

This delivered-feature contracting model fits exploratory development problems for which ASDEs excel. Another model that provides insight into these types of contracts is the model venture capitalists use for funding new companies. According to colleague and Harvard Business School professor Rob Austin, "VCs employ three basic principles to make venture investment success more likely. First, a venture investment must be staged; it must be structured so that dollars invested and risks endured are matched as well as possible over time. Second, incentives and contracts must be arranged so that all of the parties involved in a deal share in the high inherent degree of risk, so all are encouraged to take actions that are in the best interests of the overall project. Third, and most important, the venture must involve the right people to achieve success" (Austin 2001).[1] The venture capital model provides appropriate incentives, and responsibilities, for both parties in an exploratory development situation.

[1] Rob draws many of these ideas from Bill Sahlman, also of the Harvard Business School.

However, although delivered-feature contracts may be the lowest cost and best fit with Agile development, the reality of corporate, government, and national practices may preclude them, at least in the short term.[2] In these cases, companies have a couple of options: combining a dysfunctional contracting process with an equally dysfunctional delivery process (the current situation) or combining a dysfunctional contracting process with a viable, Agile delivery process. In the latter case, success may eventually help bring about future changes in the contracts. Fixed-price contracts can be written (this may require more up-front work than for a purely Agile project), then the delivery process can be Agile. It may not be the best scenario, but it is workable. I know of one software company in which the CEO says the company makes *more* money this way, because it understands, and can manage, the risks better than the clients can. This company writes fixed-price contracts, delivers iteratively, and makes more money!

[2] For example, in Poland, fixed-price contracts are virtually required, as they are in most outsourcing projects with companies in countries such as India.

Obviously It's Not Obvious

If the goal of delivering customer value is so obvious, then why are the processes used to deliver that value dysfunctional in organizations? I think it boils down to four reasons: difficulty, lack of trust, and poor collaboration.

Sometimes we lose track of the fact that building complex software products is just flat-out hard. Like exploratory drilling, dry holes are part of the landscape. If building good customer-developer relationships were easy, everyone would have done it long ago. The fact that it has always been hard and will be hard in the future provides the opportunity for competitive advantage.

There is always conflict in customer-developer (or more widely in buyer-supplier) relationships. Customers always want more, faster, lower cost, and higher quality—that's just the nature of the game. Developers want more time, stable requirements, and additional money—always. At the Cutter Summit conference in May 2001, Kent Beck stated that when customers don't get quite all they want and developers are pushed a little too hard, we've probably reached a reasonable balance.

Conflict may be inevitable, but we can still choose how we deal with it. Conflict handled poorly reduces trust, while conflict handled well improves trust. A series of interactions over time gives rise to either a virtuous or a destructive cycle. Say a customer forces a development group into agreeing to an impossible

schedule. When the group fails to make the schedule, the customer complains that they can't trust development because the group doesn't meet its commitments. Developers, remembering all the requirements changes and lack of adequate customer participation, return the mistrust. What will the negotiations on the next project look like? Not pretty!

The one practice that might improve delivery on the next project—frequent conversations—is inhibited by mistrust, ensuring that the next project will be worse than the last one. It is very difficult to break out of this dysfunctional cycle. It requires drastic action—maybe even an Agile approach to software development.

Chapter 6. Alistair Cockburn

Software development is a cooperative game of invention and communication.

- *Alistair Cockburn*

Alistair Cockburn[1] introduces himself as a ethno-methodologist, someone who studies the ethnology (cultures) of software methodologies. He is also a *cultural relativist,* having lived for 15 years in a diverse set of countries—Sri Lanka, Bangladesh, Sweden, Scotland, Switzerland, and Norway. His work travels have taken him to other exotic destinations, including, for example, lengthy stints in South Africa and Utah. As a kid, he returned from many years of living overseas (his father worked for the United Nations) to live in Cincinnati, Ohio. "Talk about culture shock," he said. "You can pretty well imagine I was beaten up for being different. I was about as different as you can get from the norm in the Midwest—speech, clothes, vocabulary, hairstyle, attitudes."

[1] Pronounced "Co-burn."

Alistair makes the point that most cultures work. Even though one country's or one organization's culture may seem *foreign,* they all have the capacity to get the job done. Not understanding the cultural aspect of methodology, process, and practice implementation leads to failure: India isn't the same as France, New Zealand isn't the same as Japan, Nokia isn't the same as General Electric, and British Telecom isn't the same as Microsoft. Every country is different, every company is different, every company location is different, every project is different.

Although Alistair and I had shared ideas back and forth for a couple of years, the genesis of our collaboration on methodology occurred while traveling up the Germania lift at the Alta ski area in Utah. We found our ideas about methodology, people, and software development compatible. His comment about our two approaches to software development is one of my favorites:

> *To the best of my knowledge, only Adaptive Software Development and Crystal explicitly call for bare sufficiency and for adapting the methodology to the specific project's situation, which leaves the two vying for the seemingly odd title of the lightest, sloppiest-looking, most effective adaptive methodology (Cockburn 2000).*

Alistair has written three books: *Writing Effective Use Cases* (2001), *Surviving Object-Oriented Projects* (1998), and *Agile Software Development* (2002). Alistair's Crystal Methods are actually an array of "methodologies" targeted at specific project size, criticality, and objective domains. His work is usually mentioned in any listing of Agile methodologies.

JIM: *How did Crystal Methods evolve to where it is today?*

ALISTAIR: The story started in 1991 when I was at IBM Zurich Research Laboratory and was looking for a job in the States. The IBM Consulting Group was just getting started, and the people there wanted advice for their consultants worldwide about OO development. They had an information engineering–based methodology, had figured out that objects were not insignificant, and then hired me to write the OO portion of their methodology. They were already using incremental development, risk analysis, and other core techniques—they had some pretty smart people—but nobody knew what an OO methodology was. One nice thing they did got me in the habit of what I do now. They had no vested interest in a particular solution—any answer would be OK, as long as it worked.

I got the books on the subject—about five at the time—and decided I couldn't discern the answers from the books. We decided that I would debrief projects inside and outside IBM to get the answers. What I found very quickly on these projects was that what the people on the projects *did* had no relationship to what was going on in the books. I became a strong cultural relativist in the process. In the end, of course, I flunked my assignment. That job was to produce a set of policies and documents that could be used by all consultants worldwide, and I couldn't do it. I still can't.

JIM: *What happened after IBM?*

ALISTAIR: In 1997, when I was working at the Central Bank of Norway, the company had several kinds of projects: 35-person, Y2K, COBOL; 2-person Java/CORBA stuff; 1-person SQL information retrieval. I would just sit there in my chair and conjure up a theory, look around at the variety of projects, and realize that the theory wouldn't work—it wouldn't cover every project. Before I got there, I already knew that one size didn't fit all, that everything is different, but I had an amazingly difficult time convincing them of that. But just taking a look around that one organization pretty much anchored, irrevocably for me, the fact that there cannot be one answer.

JIM: *If every methodology is a one-off methodology, is it a methodology?*

ALISTAIR: That's the dilemma. The question for me now as a methodologist who studies these things is that what I produce I almost hesitate to call a methodology because I don't think it can be used more than once. What I am finding is that certain skills are transferable across projects. However, most policies[2] are not.

> [2] "Policies" are the organization's rules for projects.

JIM: *What have you learned about how to study methodologies?*

ALISTAIR: As I interviewed project after project, I created a list of what people thought was important and what was not. I used to ask leading questions like, "What techniques would you use?" I would get responses like, "We would use responsibility-based design or incremental development or use cases." What I wasn't attuned to at the time was the fact that they didn't actually *do* these things. I was using leading questions, and they were obliging me. In a sense, if I look at my history, I find immense numbers of filters, blinders, and preconceptions loaded into my interviewing techniques. The last ten years has been a process of detecting those and removing them one at a time.

So, after a couple years of methodology investigation, I finally got an opportunity to try out my ideas on a real project. I took the lightest, most effective ways of working I'd come across—use cases, responsibility-driven design, and increments—and taught people how to do it. Much to my chagrin, they didn't do it! We had a successful project, the software was delivered, and we wrote something like use cases and did a little incremental development, but except in rare instances, the team didn't do responsibility-driven design.

That's when I detected that what goes on in a project is more complex than I could write down. People don't follow instructions. They do, somehow in their heads, solve problems, and they don't necessarily use

even the simplest formal technique. That was one of my first crises: Whenever I go in with a prescriptive recommendation, teach people to work this way, basically they don't. They work, however they work, inside their own heads, and a few people will pick up a few bits of a few of the techniques. So I now accept that as a part of life.

Bottom line of my research: Put four to seven people who exhibit good citizenship [behavior toward each other] in a room, and you will get software out. Playing well together is the bottom line.

JIM: *Can you give me a couple of examples of successful projects?*

ALISTAIR: The first one: 45-person, fixed-time, fixed-price, consulting house work with an IT department, client-server project, GUI, Smalltalk. I did this one by raw instinct. We created functionality teams in which business experts were in close contact with two to three programmers. A business expert (an IT staff member with significant business experience) would sit with users, get requirements, do use cases, and then, by word of mouth, transmit the information to the programmers. The requirements basically grew and lived in the personal memories of the requirements gatherer. So notes on the use cases that a person chose to take and the evolving code base of the programmers made up the requirements documentation, but we made this work because two to three times per day there was a conversation between these people. We were also absolutely hooked on increments.

I wrote this methodology up in my *Surviving Object-Oriented Projects* book, defined its domain as internal IT with 40 people, and called it Crystal Orange. The point is that we dropped the written deliverables, the written documentation, to next to zero. This is the kind of thing that puts me in the "light" methodology camp.

We also had excellent executive support, the latest equipment, and good training, and the executives had already experimented with iterative development and weren't scared by that. Our users were directly involved, we had good designer-programmers, and we had a good collaborative environment and short, rich communications paths. We had all those things that make a project successful as I now define it.

The second project was at the Central Bank of Norway in the 1997–98 time frame. It was a COBOL, mainframe project. I'm sitting there minding my own business, and this person comes in and says, "Alistair, would you please be the technical coordinator of this project: COBOL, VSAM, assembler, LU6.2, nationwide banking project." I looked at this person and said, "Excuse me, I must have this wrong: I don't know anything about COBOL, I don't know anything about banking, I don't speak Norwegian, and I don't know any of the people. Am I on *Candid Camera*? Is this a trick question?"

But they persisted, so I asked about how long and how many people and was told three people and 15 work-weeks. The three people had just finished a similar project and deployed it, so they assured me the new project would be pretty easy. I said, "Wait, we're talking five weeks elapsed time. You don't need a technical coordinator. Just do it and go home!" But they insisted, and I said OK. The first thing I figured out was a technique for assessing risk. I got each of the three programmers in a room and brainstormed all the considerations I could think up that they might encounter, and whether they had dealt with them before. With each one, what I watched for was whether his body language matched his answers. If he said it was easy and they'd done it before, I figured it must be so. I was looking for when he didn't seem sure about something. If it was something he hadn't done before but thought they had ideas about how to do, that went on my risk list.

Anyway, as you might guess, the project ended up being a 14-month project. It got bigger, became a split-site project, and when it got tough, I stepped up the personal interaction—conference calls, emails, visits— I did not step up bureaucracy. In the end, the project was considered a success since it did what it was supposed to, our eventual project plan stayed stable over the last eight months of the project, and there were no problems in bringing the system live.

JIM: *You talk about skills and policies on projects.*

ALISTAIR: Skills go in one dimension, and policies go in another. Project policies—notations, standards, increment length, milestone length—are specific to a project and don't transfer. Skills, however, do transfer—how to code, how to do test-first development, how to do user interface (UI) design. However, some skills aren't learnable or teachable. Project managers need interviewing, communicating, decision-making, creating skills—those aren't skills I'm going to teach someone. I can give them some techniques to get them out of the woods, but they'd better show up with that other stuff.

JIM: *Are you saying that the so-called "soft" skills aren't learnable?*

ALISTAIR: Ah, I'm saying that you can add techniques to the soft skills, but the person has to be talented in that area to start. When I look at programming and design, we can teach CRC cards or test-first programming, but have to have an ability for abstract thought—I can't teach that. Just as you can teach paper-based prototyping to UI designers, but you can't teach color sensitivity.

My proudest thing on the banking project, besides the risk management, was working with a new project manager from the user department, who knew nothing about project management or software. I coached her in how to read the body language of the participants in our weekly videoconference. The programmers would mumble to each other, and what she and I had to do was to determine where to insert a question to get clarification on the problems.

At the end of three months, she was better at it than I was. She could understand the tone of the discussion even when she didn't understand the technical language. That, for me, was a successful moment in conveying a large part of what project management is about, and she couldn't have learned it if she didn't already have a talent for watching and listening.

JIM: *You have lived and worked in many countries and seen multiple national and project cultures at work. What have you taken away from these experiences?*

ALISTAIR: First, all these cultures work; they all have different rules. It's not that a particular culture is wrong and needs to be fixed. So actually, now I'm a severe cultural relativist in the sense that I go from project to project and assume that the organization is working or it wouldn't be in business. I always start by saying, "You guys are doing something right."

JIM: *In the knowledge management field, there has been a lot of discussion about tacit knowledge and knowledge transfer. How do you think people learn new things?*

ALISTAIR: We're talking about the motivation for the person who picks up a book or takes a course, or whatever. You cannot predict for any given person what they will pick out. In this searching and picking, there is something special about stories. Stories give a trajectory for actions. Not just a single action, not if-then-else rules, but they tell what different people did over time, and human beings digest that very well.

This project that you are working on now—developing stories of projects [for this book]—is to me simply perfect. Because if you want to influence somebody's mind, a strong way is to tell them a story. You may not guess exactly what they will take out of it, but they will be able to follow the trajectory, they will do an abstraction, learn their own lessons, and then, if you give a digest of what you learned from the story versus what they saw, that has some likelihood of impacting their thinking.

JIM: *So if stories are important, why continue to teach courses on subjects like use cases?*

ALISTAIR: My current theory is to teach techniques, in workshops, schools, on the job, whatever, and that those skills go into people's personal kit bags, which they then use for however many seconds at a time. They internally flip from technique to technique to help solve problems. The focus of a technique is to put something in their kit bag, and that is transferable across projects. The other thing a course does, particularly if a group takes it together, is align their vocabulary.

JIM: *So what is a methodology?*

ALISTAIR: To me, a methodology is the set of communications and coordination policies used on a project. So what a person does is not a methodology to me. How three to *n* people coordinate their activities is methodology. Things like doing increments, milestone-based planning, specifying use case templates, using Microsoft Project—whatever the team agrees upon at a cross-team level are the parts I want to influence.

JIM: *So policies are boundaries. Whereas traditional methodologies specify activities and documents, your approach is to have a few policies that organize the communications and coordination and then depend on individuals to use their "kit bag" of skills as they and the team deem appropriate.*

ALISTAIR: Once people show up on a project, you don't tell them how to write Java. What you need to tell them is who they need to talk to and how their code is to be accepted and what meetings they have to attend. The techniques go into an educational curriculum that you set up for your corporation, which says we want people to have certain skills.

JIM: *You wrote a paper about software development as a cooperative game of invention and communication (Cockburn 2000). How did you arrive at that analogy?*

ALISTAIR: Mostly to get rid of this aberration called software engineering. Actually, the cooperative game idea describes large sections of normal engineering activities. The other parts of engineering involve looking up previous solutions in standards books. But engineering has given us a faulty vocabulary, a vocabulary that leads us in a direction that's not fruitful to a software project. It gives us questions like "Is the model accurate and complete?" Much more interesting are "Do we have our goals aligned as a team?" "Do we have a correct set of skills?" "Is our product meeting the customer's needs?" and "What is the next move in this game, given that our competitor has just announced a new product?"

JIM: *To me, software development is mind work more than anything else I can think of, maybe with the exception of writing.*

ALISTAIR: There is nothing but the idea. I did the thought experiment. Let's say you had a group of people who were hired to write a 20,000-word epic poem for John D. Rockefeller—what would it look like to coordinate this effort? I wrote a story about the person, the poet, who takes this on, gets swamped, recruits four friends to help. They get swamped and hire several more friends. What would happen? What would this project look like? The poet is pulling her hair out trying to coordinate things and eventually hires a manager and buys tools. Then subgroups form—you get some people who are good at descriptive passages, some who are good at fight scenes, some who are good at describing personalities, and then you get a splinter group that goes off and then comes back again. I wrote this up and showed it to a guy at Novell who works on operating systems. He loved the article and took it home to his wife and said, "That's what my life is like at work!"

So software development is a cooperative game of invention and communication. All we do is to invent and then communicate our invention. If we hold that thought and want to make it work, then what we want to do on a project is ask, "How do we enhance people's invention actions?" and "How do we enhance their communications?"

The reason I like this definition of a cooperative game of invention and communication is that it does lead us to issues that I find do show up on project after project and didn't have a home in the process-oriented, software engineering vocabulary.

JIM: *What passion drives your work?*

ALISTAIR: For me, it's understanding the human mind. Even when I designed hardware, I've always had an interest in the interface between the human mind and technology.

There is nothing that happens to me, anywhere, at any time during the day, that I can't apply to a project. I will read a book or have a work experience and move things around on a project as a result. It gives me a chance to see how things work. Programmers are my clientele or my study group. It is a subculture that I used to belong to—I'm still affiliated with it—and one I care for. You can keep everybody else around on a project and send the programmers home, and you won't come up with a system. They are really the heart and soul of the story. So I really like to study them.

Reflections

I've known and worked with many methodologists over the years, nearly all of whom focus on what they thought constituted a methodology. Alistair is the only one in my experience, with the probable exception of Jerry Weinberg, who has conducted his investigation with such a critical eye and unbiased viewpoint on analyzing what works and what doesn't. That's why I think of Alistair as a methodologist and archaeologist—digging deeply into the bone shards of the past, trying to get at the roots of what really makes software projects successful, not depending on the superficial, surface evidence.

As Alistair interviewed people in multiple projects around the world, he began to realize a couple of interesting things. First, almost no one did what the books written by the experts said to do. Most didn't claim they did what the books said, and the few that did make that claim didn't do it anyway.

The second pattern that emerged was that the particulars of languages, testing techniques, diagramming models, tools, and other practices and tools didn't predict performance. Whether teams succeeded or failed seemed to revolve around a single factor: *Did team members play well together?* Even successful teams, who themselves attributed success to using a methodology or some high-powered development tool, inevitably also talked about individual talent and how well the team members worked together. From these revelations, Alistair began talking and writing about software development as a cooperative game.

In one way, reading through this interview with Alistair is depressing—there doesn't seem to be a right answer, or even a reasonably small set of right answers, for software development. The focus of software engineering process specialists and methodologists appears to be far down on the list of project success factors.

Finally, although in one respect Alistair's message is that every project, every project team is different and requires a tailored approach, there are also a few things that come across as essential—small wins through incremental development, constant feedback, and person-to-person conversations.

Chapter 7. Rely on People

Apprenticeship takes a long time because it takes time to absorb the culture and ethics of a craft.

• *Pete McBreen,* Software Craftsmanship

ThoughtWorks

Martin Fowler, coauthor of *Planning Extreme Programming* (with [Kent Beck, 2001](#)) and chief scientist at ThoughtWorks, Inc., has been instrumental in bringing the concepts of XP and other Agile processes to projects. Agility is considered to be a core competency and a competitive differentiator at ThoughtWorks. But no one ever said it was easy.

"Personally, and I can say this as the person who introduced XP to the team and as the team coach, I am beginning to loathe XP," commented Jack Bolles, project manager for one of ThoughtWorks' projects. Now that was too good a quote to pass up! In a series of email exchanges, he explained his rather provocative statement, which in the end turned out not to be a negative comment on XP but rather a lament about how people—their skills, personalities, and quirks—are both critical to success and frustratingly wonderful to work with.

"I said that? Surely I've been misquoted!... Taken out of context!!" Jack replied to my inquiry. "Actually it's true. A lot of that comment stems from working with inexperienced developers who are good at understanding the movie set version of XP but don't understand how the pieces came to be or what they represent; those for whom 'develop' means open up a text editor and a compiler and have at it."

Was the project successful? "Very. We met our dates and delivered the application that is in use today. Along the way, we changed internal team management, as well as other personnel, without skipping a beat," Jack responded. As to the reason for success, "Running away, it was the people. I can't stress enough the importance of team dynamics."

Working with process and tools pales into simplicity when matched with the frustration, messiness, difficulty, and fun of working with people and relying on people to get the job done. Peel back the façade of rigorous methodology projects, ask why a project was successful, and the answer usually matches Jack's: "It was the people." Real people don't have sharp edges and meticulously crafted data interfaces like process flows. Real people are messy, often disorganized, and difficult to understand, while at the same time they are innovative, creative, exuberant, and passionate.

Jack's project was to create a leasing origination application, complete with pricing, legal documentation, and financial modules. ThoughtWorks took over the project from another company, but of course delivery dates remained unchanged. The client had limited industry experience and a fuzzy idea of their requirements. The project was moderate in size, with six to ten developers, two to four analysts, and one to three dedicated customer representatives during various phases of the project. As with most hand-off projects when one supplier falters, the project's beginnings were inauspicious. Many Agile projects begin like this: When all else fails, try something weird!

"I joined the project fairly early in its ultimate life cycle," says Jack, "but with only eight weeks until first delivery. (Between us and another company, the project had been in existence for six months already.) Basically, there was no process. We were sent out to deliver at all costs. The code base and process were in such disarray that those new to the team started applying the tools and practices that had made them successful previously. I couldn't tell what the code was doing, so I refactored. Another person didn't feel comfor table without tests in place, so she put test cases in place. Another person was so frustrated with

out-of-date or noncompiling builds that he tackled that. Eventually, our personal success encouraged others to adopt these practices. After a while, we looked up to recognize, "Hey, we're doing XP!"

So what about the loathing part? "I'm especially wary of any combination of YAGNI ["You aren't going to need it"], do the simplest thing that will possibly work, and refactor mercilessly," Jack explained. "These catch phrases have been understood by some as carte blanche to write crappy code (by crappy I mean code that manages to compile despite itself and pass a few preordained tests) because the only thing that matters is that the tests pass. Doing the simplest thing first is also an argument not to learn what I refer to as the 'context of coding'—OOP, design patterns, language-specific best practices, deployment, and so on. In that sense, my idea of the simplest thing that could possibly work and someone on their first or second J2EE project's idea of the simplest thing is completely different.

"Now don't get me wrong; we hire good people who want to do the right thing. But it was easier to get them to take the extra time to understand the context of coding before XP gave developers a free pass from proving they knew their stuff. And yes, pair programming is supposed to mitigate this to a certain degree, but it only goes so far. One thing that methodologies heavier than XP give you is talking points for bringing in context. Ideally, I can work on a team that doesn't need it, but I've not been that lucky."

Jack brings up another key point—*process is not a substitute for skill,* whether the approach is XP or the CMM. Whether a process is heavy or light, Agile or rigorous, just following a process doesn't create good code (or good test cases, or whatever)—skill creates good code. He continued talking about skills, personalities, and attitudes that constitute the first-order effect on any project (methodology being a somewhat distant second).

"The thing that got me paying attention to this was one of our developers who was tasked with integrating with a third-party system. His stuff worked, and I could see that it was well tested. Come review time, he received high marks. A month later and after he had rolled off the project, we had to go back into the code, and it was a nightmare. Among other things, he was a difficult person to work with, so it was easy not to pair with him. Another developer had come up with an elegant solution using patterns; a bit later another pair came along and refactored it out because they weren't familiar with the pattern and thought the code was too obscure."

Jack talked about the aggravation associated with inexperienced team members and their use of XP practices and principles. While simplicity was desirable, unskilled simplicity wasn't. Just as customer-developer interactions can oscillate rather than converge on a solution, developer-developer interactions can oscillate also—one pair codes a design, and then another pair refactors unnecessarily because the later programmers didn't understand the pattern. Team leaders and coaches (XP uses the coach role extensively) have to be particularly aware of this tendency, especially with inexperienced individuals.

The values that drove Jack's team were straightforward: "Desire to do the best that we could, a strong drive to succeed, and the dream of going home! We laughed, we cried. We had a good team of generally likeable people. ThoughtWorks is very excited about XP. The biggest challenge, I've found, is that XP assumes a certain amount of tacit knowledge and skill that isn't in the books."

"Software development is a cooperative, finite, goal-seeking group game," writes Alistair Cockburn (2002). There are two parts to group games: individual skills and collaboration. Without world-class individual skills, mountaineers should not attempt extreme peaks. But extreme mountaineering also depends on teamwork and collaboration. Similarly, individual skills and teamwork are inseparable on software projects of any size.

Who Are You Calling Average?

Note that the title of this chapter isn't "Rely on Good People" but simply "Rely on People." I take a survey periodically during conference talks by asking the question, "Will all of you who consider yourselves to be *average* please raise your hands?" I do get a few hands, but only a few. The Agile Manifesto value

statement that people are more important than process and tools may sound flippant at first, but there remains a segment of corporations that still views employees as average machine components, and untrustworthy components to boot.

Time and again we see references to "process" being the mechanism for turning average people into reasonable performers. I hate the connotation of average. One of my favorite movies, *Rudy,* coaxes us into thinking about "average" people. By all accounts, Rudy shouldn't be allowed on a football field— especially the hallowed field of Notre Dame. Short, underweight, possessing little athletic ability, Rudy fought to make the taxi squad, practiced every day as the "blocking dummy" for the larger, faster players, and only got to play once in his career—the last game of his senior year, the last play of the game. But since that game in 1975, he remains the only Notre Dame player to be carried off the field in triumph by his teammates. What Rudy lacked in ability and size, he made up for with the determination, desire, and raw guts that fueled the admiration of his teammates. He made the entire team better.

Process doesn't turn people into star performers. People turn people into star performers. To the extent that process-centric mental models steer us away from valuing the Rudys of the world, they will also fail to ignite outstanding performance. Kent Beck once related an experience in which the most experienced architect on a team left—and the team's performance went up! Which led Kent to quip, "I wonder what would happen if we got rid of all the smart people?"

The Agile Manifesto proclaims a focus on people with a value statement ("Individuals and interactions over processes and tools") and a principle ("Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done"). In an article I cowrote with Martin Fowler, we observed:

> *Deploy all the tools, technologies, and processes you like, even our processes, but in the end it's the people that are the biggest difference between success and failure. We realize that however hard we work in coming up with process ideas, the best we can hope for is a second-order effect on your project. So it's important for you to maximize that first-order people factor. The hardest thing for many people to give is trust. Decisions must be made by the people who know the most about the situation. For managers, this means trusting in your staff to make the decisions about the things they're paid to know about (Fowler and Highsmith 2001).*

Value people. Trust them. Support them. These values are the essence of building an Agile ecosystem.

Trust, Mistrust, and Communications

Many project managers abide by the old creed, "If you want something done right, do it yourself." The unfortunate corollary to this creed becomes, "If you have to depend on someone else to do it, you have to pester them until they get it right." This creed—beaten into our psyches by years of repetition—is dead wrong! This creed harbors deeply flawed assumptions that are so ingrained that we don't realize how much they influence our task-oriented, "inch-pebble" approach to project management. The other underlying assumption in this creed is that the "yourself" in the phrase gets to decide what is "right." So, in essence, the creed should read, "If I want it done right, as I personally define right, I have to do it myself." This creed says more about the speaker's inability to communicate than the hearer's inability to perform.

Double-barreled mistrust stalks this creed. First, "I" don't trust you to understand what right means, and second, even if you understood what I mean by right, I don't trust you to deliver it to my standards. There are two fundamental, diametrically different ways of approaching work. The first, the traditional Command-Control style of management, is formulated on a foundation of mistrust: I don't trust you to get the job done correctly; therefore, I have to constantly follow up, keep the pressure on or you will slack off, and micro-measure your performance. The second, the chaordic, Leadership-Collaboration style of management, is formulated on trust and respect: I trust you to get the job done correctly, and in order to assist in getting the work done, we have to interact in order to monitor expectations and performance. The first approach says people won't do a good job unless tightly managed. The second implies that we (both

manager and managee) are working to the best of our ability, but we realize that communication is difficult, and we constantly need to make sure we are in sync.

Managers in organizations either have a fundamental trust of people or they don't—there is no middle ground. Of course we need checks and balances for the minor percentage of people who are actually not trustworthy. Of course we need reviews so that when people stretch the limits of their understanding in order to learn, we provide a safety net for them. We all make mistakes, so we need adequate feedback practices to catch mistakes and learn from them. We need to communicate progress and adjust as needed. But these are mechanisms to catch miscommunications and learning mistakes—not mechanisms to control the "untrustworthy."

Norm Kerth voices a wonderful "Prime Directive" in his book on conducting project retrospectives:

> *Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given what was known at the time, his or her skills and abilities, the resources available, and the situation at hand (Kerth 2001).*

Norm's directive says to trust people to do the best job they are capable of, knowing what they know and understanding their own skills. This doesn't say people are infallible or that their understanding of their work is complete or that their skills and resources are adequate. It says trust people to do the best job they can under the circumstances. It says, "Trust people, mistrust communications."

"I firmly believe that people are almost always sane from their perspective," says colleague Lou Russell. "In other words, the decisions people make are sensible given the information they have at the time. Insanity can come from a bunch of people who are sane from their own perspective but whose sum total is insanity." Insanity arises from failing to distinguish between mistrust and miscommunication.

From a methodology perspective, trust extends not just to getting the job done, but to deciding how the job is to be done. Every project requires a custom-tailored process, and no one is better suited to doing the tailoring than the project team doing the work.

Talent, Skill, and Process

In *First, Break All the Rules,* Marcus Buckingham and Curt Coffman (1999) identify the traits of great managers, but they also point out, "Over the last 20 years authors have offered up over nine thousand different systems, languages, principles, and paradigms to help explain the mysteries of management and leadership." So what makes their analysis different? Research. Buckingham and Coffman work for Gallup (the polling and research people), and their conclusions about good management are the distilled ideas from interviewing 80,000 managers in 400 companies over the past 25 years.

In their second book, *Now, Discover Your Strengths* (2001), the authors delve deeply into the attributes of talent, skill, and knowledge. Their most relevant comments on high-performance working environments come when they address the issue of people- versus process-centered performance management.

> *"Both camps share a belief in the fundamental importance and potential of their employees, but only one of them will create the kind of environment where that potential is realized," say Buckingham and Coffman. "The larger, establishment camp is comprised of those organizations that legislate the* process *of performance.... They try to teach each employee to walk the same path."*

> *They say that the strength-based camp, on the other hand, "focuses not on the steps of the journey but on the end of the journey—namely, the right way to measure each person's results in the three key areas [impact on business, impact on customers, impact on coworkers]."*

"The distinction between the two camps is real. Step-by-step organizations are designed to battle the inherent individuality of each employee. Strength-based organizations are designed to capitalize on it" (Buckingham and Coffman 2001).

These are powerful statements—statements coming not from a single person's ideas about great managers, but from an exhaustive *study* of managers. Extending Buckingham and Coffman's ideas, it's not that organizations that employ RSMs value people less than those that employ ASDEs, it's that they view people and the way to improve performance differently. RSMs are designed to standardize people to the organization, while ASDEs are designed to capitalize on each individual's and each team's unique strengths.

Rigorous, document-centric processes tend to relegate people to a role of interchangeable, marginally reliable machine parts. Prescriptive processes first specify, in excruciating detail, exactly what activities people are to perform, and then they specify meticulous control activities to make sure the prescribed processes are done correctly. Prescriptive processes imply that employees are stupid, while control processes tell them that they are untrustworthy. Companies continue to declare that hiring and retaining talented technical people are critical to business success in today's technology-driven economy. Why then, do companies continue to promote and implement processes built on the same fundamental assumptions that were used to design 1930s automotive assembly lines? That's hardly a winning combination for today's talented, constantly mobile workforce. There is a place for process and ceremony, but how they are implemented within a people-oriented value framework is key to agility.

There are three basic problems with traditional approaches to process:

- Process and skill become confused, and since process seems more objective, it becomes the focal point.
- The artifacts of a process-centric approach become process boxes connected by defined information flows.
- Innovation and creativity are driven out by formality and endless process refinement.

Process versus Skill

One of the frequently voiced criticisms of ASDEs is that they require highly skilled individuals. "Like XP and its Agile teammates, chief programmer teams also enjoyed early victories for a while, but the model ultimately foundered on the shoals, caught between the Charybdis and Scylla of charisma and scale," writes Larry Constantine (2001). "There are only so many Kent Becks in the world to lead the team." Others have cautioned about the necessity of having skilled individuals in order for Agile approaches to work. To summarize the concerns, Agile approaches require skilled leadership and team members. If this statement is true, then the flip side should be true also; namely, that rigorous approaches can be utilized by unskilled leaders and team members. The arguments for and against certain methodologies often founder on this confusion between process and skill.

So, what methods *do* work best with unskilled and inexperienced developers? The issue, of course, is not methodology. Unskilled and inexperienced developers should work on easier problems suited to their skill levels. Complex, difficult problems should be attacked by experienced and skilled developers. The match, or mismatch, has to do with skill versus problem, not skill versus methodology. Methodologies are how teams work together to solve a particular kind of problem. If we attack an optimizing problem with an exploratory methodology, we reduce our chance of success.

Process should assist a project team in delivering the best they are *capable* of producing. At a certain skill level, a team is capable of some things but not others. No amount of additional process will make the members of the team capable of delivering a product that is beyond their talent and skill. While process and structure can support and leverage skill, increasing structure does not make up for skill deficiency. For example, if I have trouble articulating good questions, no amount of "process" will turn me into a good systems analyst; it will merely leave me a poor systems analyst who is now grouchy.

There is also confusion between process and checklists. Processes and procedures are prescriptions for work. Checklists keep us from forgetting something we know. Would I want an unskilled pilot flying my Boeing 747? Not likely. Would I want an unskilled pilot with a prescriptive manual flying my 747? Nope. Would I like for my skilled 747 pilot to have a checklist so he or she doesn't inadvertently forget something? Certainly. An example of an approach that uses checklists for skill building would be Karl Wiegers's *Software Requirements* book (1999). Wiegers isn't prescribing a detailed process for developing requirements as much as trying to impart skills. His 14 steps consist of checklists and skill building rather than prescription.

In a workshop recently, someone asked about the iterative planning approach I was advocating. "But what if you get the wrong people into the planning process?" they asked. "You get a poor plan," I responded. Process doesn't make up for lack of skill, judgment, and experience. Similarly, to the oft-posed question, "But what if you lose a key technical person off your project? Don't you need documentation backup?" I responded, "If you lose a key technical person off your project, and you haven't created a good peer-to-peer knowledge-sharing environment, then no amount of documentation will save you."

"Tacit knowledge cannot be transferred by getting it out of people's heads and onto paper," writes Nancy Dixon (2000). "Tacit knowledge can be transferred by moving the people who have the knowledge around. The reason is that tacit knowledge is not only the facts but the relationships among the facts—that is, how people might combine certain facts to deal with a specific situation." At a simple level, explicit knowledge is codified, or documented, and tacit knowledge is that knowledge still in your head. Best practices may be written down, but it takes tacit knowledge (experience, know-how, thought, and judgment) to apply them. Explicit knowledge provides you with the rules; tacit knowledge provides you with the deeper understanding that tells you when to break or bend the rules. You can no more become a good programmer from reading a C++ book than become a golfer by reading *How I Play Golf,* by Tiger Woods.

Proponents of process and structure assume that knowledge can be written down, encapsulated in little "knowledge" objects (called "best practices"), and plunked into people's brains just as reusable code is plunked into the next application. But whether you are a professional golfer or a professional software developer, the essence of your skill is tacit—it can't be written down and plunked into the next recruit. The essence of talent and skill isn't what we record on paper; it's defined by proficiency, mastery, virtuosity, savvy, and artistry. Transfer of tacit knowledge takes time, energy, and face-to-face apprenticeship.

If skills are so critical, and process isn't a substitute for skill, then a key question becomes, "What is the best way to improve skills?" Laurie Williams, assistant professor at North Carolina State University, based her Ph.D. dissertation on pair programming. More telling than the charts and graphs showing the benefits of pair programming (necessary, of course, for a dissertation) were her comments at an XP panel discussion at OOPSLA 2000. Prior to teaching pair programming, Laurie related how her office was constantly filled with students asking questions about assignments. However, with her pair programming class, her office stayed virtually empty—the students were learning from each other. The two critical requirements for skill building are mentoring (peer-to-peer or expert-to-less-expert) and fast feedback cycles to test learning.

Artifacts and Information Flow

A second problem with a process-centric approach to development involves how information flows are depicted and viewed. A process-centric approach, one that defines explicit process boxes and information flows between the boxes, sways teams toward managing explicit information rather than tacit information. The people interaction becomes lost in the white space between the process boxes, and the information outside that narrowly defined by the flows becomes irrelevant.

We have to get better at *managing the white spaces* between the process boxes (Rummler and Brache 1995). When managing a project that grew in size and team distribution, Alistair made the comment, "When it got tough, I stepped up the personal interaction."[1] When projects get larger and distributed, traditional process-centric managers, assuming that the bulk of the knowledge required to create deliverables resides within the designated document flows, increase the emphasis on formalizing those

flows. Agile managers look first to increasing interpersonal interactions, and then to minimally increasing documentation.

Innovation and Creativity

Agile approaches excel at exploratory problems, those defined by uncertainty, risk, change, and speed. Solving these problems requires, in addition to a certain dash of luck, teams that are innovative and creative—able to think outside the proverbial box. *Adaptive Software Development* explains in a comprehensive way how innovation and creativity flourish in an environment poised on the brink, at the cutting edge between chaos and order (Highsmith 2000). Processes don't encourage emergent, innovative results—people do.

The Fall and Resurrection of Programming

XP in particular, and ASDEs in general, reverse the trend of software engineering's trivialization of programming. Starting in the early 1980s, the rise of waterfall development life cycles and the emphasis on front-end modeling, requirements specification, and database design (all appropriate in moderation) began relegating programming to a mechanistic skill that would be shortly automated. The waterfall life cycle was compared to building construction (a terribly misleading analogy) in which "coding" was compared to bricklaying, as if bricklaying or other construction activities were somehow done by unskilled labor. Noninteractive, waterfall development doesn't work very well in construction either, as any skilled builder will attest. The IT industry began developing specialists—architects and designers—who neither had any experience programming nor thought programming was very important in the overall scheme of software development.

In IT organizations, database administration, architecture, business analysis, and standards and methodology groups gained stature, while "lowly" project teams of developers—burdened with increasing paperwork—struggled to get working software delivered to customers. I recently visited an IT department that had a development project of 80 people, and only a single programmer. When executives finally realized the imbalance, they understood why the project was floundering. While IT and big systems integration vendors played into this version of the world, most software companies, such as Microsoft and Borland, continued to focus on code, and therefore stature remained with actual development teams. The "fall" of programming was, in part, due to the misunderstanding of the boundary between concreteness and abstraction.

There was considerable debate among Agile Manifesto authors over the principle that "working software is the primary measure of progress." Several participants in the debate worked for companies whose modeling tools generate code, so in those cases, does working software mean code or models? In essence, if the model or prototype generates tested code within a time span similar to that in which compilers operate, then the difference is negligible and working software can be defined by modeling efforts. However, then the question becomes, "If we model in sufficient detail to generate code, isn't the modeling process essentially coding?"

Figure 7.1 indicates where the arguments have arisen. For example, a compiler translates one concrete implementation, source code, into another concrete implementation, executable code.[2] The transition from an electrical engineer's chip design to a physical chip would also be a translation. Traditional methodologies have relegated programming to a translation activity—converting models to code. However, it hasn't worked out that way.

As we learn and software becomes more powerful, the line between abstract and concrete may change or even blur a little (because of heuristic translations done by software), but the fundamental abstract-to-concrete analogy still holds.

Figure 7.1. Abstract Models to Concrete Implementations
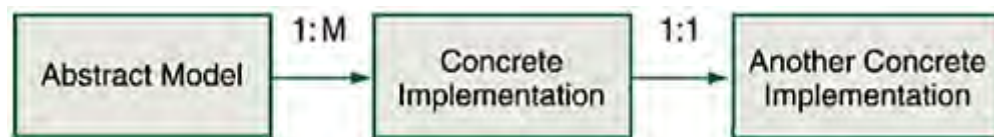
Writing software is more analogous to writing in general (books, articles, manuals) than it is to building construction. Developing abstract software models (whether they be data, object, UML, or other models) is like outlining a book—plotting the story, identifying the characters, establishing the setting, envisioning the scenery. As every author knows, or finds out rapidly, moving from the abstract model of characters, setting, and plot to concrete words on the page is a process fraught with ambiguity, despair, and sheer joy. The transition from abstract model to concrete implementation is a one-to-many mapping: many, many combinations can be generated from the outline. Each "writing model" (outline) can literally become thousands of specific implementations. If the "outline" was specific enough that only one story could be generated from it, it would no longer be an abstract model but a concrete implementation. Translating the final story from English to French would be converting from one concrete implementation to another, not converting from an abstract model to a concrete implementation. Similarly, once an abstract software model becomes specific enough to generate a single implementation, it is no longer an abstract model but a concrete implementation, regardless of whether the form is text, code, or a model diagram.

Software development has gotten into trouble by confusing creation with translation. Abstract models help us organize our thinking and reduce complexity by "abstracting" to higher levels. A data model diagram showing entities and relationships is an abstract model. Add enough detail—attributes characteristics, relationship characteristics—that a database schema can be generated (one and only one concrete implementation), and the diagram itself is no longer abstract but concrete. The process of generating a physical database schema has been reduced to a translation activity—one often automated. Add enough diagrammatic complexity to be able to represent business rules and thereby generate data update and maintenance code, and we now have a concrete model in both data and process dimensions. However, moving from the high-level, abstract data model to a single concrete implementation requires as much serious thought as moving from an outline to a novel. Whether the result of the creation process is a feature-rich diagram (or series of diagrams) or source code is irrelevant as long as the next step in the process is purely translation.

But a problem looms. Anyone who has looked at detailed UML diagrams or multi-symbol business rule diagrams, has asked him- or herself: "*These* are easier to understand than code?" Some people, incorrectly, assume that Agilists—particularly those associated with XP—are anti-modeling. What Agile developers do understand is that the magnitude of the gap between abstract models and concrete implementations is something that modelers fail to understand because they aren't programmers. This gap, minimized by modelers, would be similar to *my* developing a 20-page annotated outline for a book and expecting *you* to write a single implementation of a 500-page novel from it.

When, and if, model diagrams are directly translatable into working software, are as easy to understand as code, and the tools are reasonably priced, then Agile developers will use them. Until then, Agile developers will continue to operate on the belief that the gap between abstract models and concrete implementations is much wider than most software engineers acknowledge.

So the arguments about modeling versus coding are really about the value of multiple layers of abstraction and the gap between abstract modeling and concrete implementation. Talk to people in the publishing business. There are people in the publishing business who understand what constitutes a good book idea, they have ideas about whether or not a book will sell, they can help with reorganizing and copy editing—but they also know that they can't write the books. Contrast this with many software modelers, who seem

to think that abstract modeling—coming up with the idea and the plot—constitutes the real work and that the rest is just mechanical coding. The Agile community thinks abstract modeling has gone too far.

Software through People

Recall the quote from the software company CEO in Chapter 1: "There is this myth that software development is a creative effort that relies heavily on individual effort. It is not. It is just very labor-intensive, mechanical work once the initial project definition and specification stage is past." Every value that Agilists believe in says this CEO is completely and utterly wrong. Developers, testers, documentation specialists—these are the skill positions that create products, that deliver working software. Many individuals—analysts, customers, architects—contribute to a software project's success, but Agile approaches emphasize that decision making and control should gravitate to the team that is creating the product, not those in staff and support positions.

Software through People is the motto of the Agile Manifesto authors. The motto epitomizes the idea that people—their skills, abilities, experience, idiosyncrasies, and personalities—constitute the first-order impact on project success. Jack Bolles' lament about "beginning to hate XP" seems more a lament about bringing skill levels up to par than one about XP per se.

Value people. Trust them. Support them. These *values* are the essence of building an Agile ecosystem. Mistrust communications. Put practices in place to bridge the difficulties of communication and collaboration between trusted, valuable people. These are the *practices* that build effective Agile Software Development Ecosystems.

Chapter 8. Ken Schwaber

This book presents a radically different approach to managing the systems development process. Scrum implements an empirical approach based in process control theory. The empirical approach reintroduces flexibility, adaptability, and productivity into systems development.

- *Ken Schwaber and Mike Beedle,* Agile Software Development with Scrum

Ken Schwaber has been at the methodology game for many years, emerging from the rigorous, heavy methodology years of the late 1980s and early 1990s with a deep understanding of the fundamental flaws in detailed, process-centric approaches. His successes with Scrum include the complete revamping of a medical software product at IDX Corporation (Chapter 2) in which he consulted during the early stages of the project and was instrumental in Scrum's implementation.

Scrum evolved from the combined efforts of Jeff Sutherland, Ken Schwaber, and Mike Beedle. It has been used in a wide variety of projects, by a number of practitioners, for nearly ten years. Ken and Mike teamed up to write a new book on Scrum (Schwaber and Beedle 2002), and all three—Jeff, Ken, and Mike—are Agile Manifesto coauthors.

As with several others interviewed for this book, I talked with Ken during the initial Agile Alliance meeting at Snowbird. In the lobby of the Cliff Lodge, after a couple of runs through two feet of fresh Utah powder, Ken and I relaxed and tried to tune out the noise of skiers coming and going.

JIM: *What are the threads of your career that eventually led to the development of Scrum?*

KEN: I've been doing software my entire professional life, nearly 35 years, but it wasn't until I was at Wang Laboratories that I worked on a really big project, about 110 people. This large database system tracked hardware, locations, and maintenance contracts. The whole project was very ad hoc—a lot of meetings and design sessions. I thought there needed to be a better way for everyone to understand what they were supposed to do, and said so. My boss replied, "OK, you are now in charge of processes for systems development at Wang." They asked me to add the stuff we had done to the methodology that they had licensed (SDM), but no one was using it. I couldn't see any benefit to SDM, and updating it was like beating a dead mule.

Wang was trying to sell its systems to MIS groups for software development but wasn't having much success. I suggested that the reason was because people didn't have a methodology, a practice, for building software using its database and forms generator approach, so I proposed building such a methodology and

an implementation service. Wang demurred, and I left to do essentially the same thing with structured methods and CASE tools.

I was approached by Index Technologies, which had CASE tools and wanted someone to provide training and implementation services for structured methodologies. I founded a company, Advanced Development Methods (ADM), which grew to about 20 people. We grew a methodology, which proved to be a good differentiator for consulting companies in those days (1990–91).

Eventually, we built software that automated our methodology and called it MATE (Methods and Tool Expert). We used an early OO development product to build the tool, and people started buying it, which just flabbergasted me. People could customize their methodology using the tool. The benefit was that people learned how to think through data models, and processes, and the context. But we sure took in a lot of money that, when I look back, delivered very little value—and that always troubled me.

JIM: *That was a period of time when people were spending hundreds of thousands of dollars for methodologies, training on methodologies, and methodology management software.*

KEN: Many spent $400,000 to $500,000 on MATE and our services. One of our customers said to us that they liked Coopers and Lybrand's methodology but liked our product for distributing it. They wanted to know if we would customize it with templates, detailed planning, an interface to Microsoft Project, and other things. It would do what we are all now against—very detailed master planning and estimating. For example, estimating time by calculating the numbers of data model entities at so many hours per entity, and on and on.

We ended up with every customer wanting a customized version of MATE. At one point, we had 22 customers and 22 versions of the software—which is enough to drive anyone nuts. A CIO would acquire our product and hand us over to the methodology group. The methodology group had to prove its value, so it would want to customize what the CIO had purchased. It would want its methodology loaded, the Coopers & Lybrand methodology customized, or the product changed based on its vision.

We ended up loading our tool with Coopers' methodology and selling both, getting a royalty on the methodology. We did the same thing with IBM, which used MATE with its methodologies to manage its outsourcing business and some of its internal systems development projects.

At around the same time, Jeff Sutherland and I were doing structured methods, and he was beginning to work with something he called Scrum. He heard of Scrum from the "New New Product Development Game" article (Takeuchi and Nonaka 1986) and the book *Wicked Problems, Righteous Solutions* (DeGrace and Hulet Stahl 1990). Jeff's and my friendship goes back to the early 1980s, and we worked together on systems starting in 1987. At some point Jeff asked me, with all these methodologies—Coopers', IBM's, our own, etc.—which one did we use for building our MATE product? "None," I said. "If we used any of them, we'd be out of business!" So we sat down with our own developers and asked them what they actually did, how they worked. And the answers were quick turnaround, evolving object diagrams, adaptive, requirements evolved, and that everything kept getting better and better.

While Jeff and I were doing this, the IBM people came in and said they wanted to go to the next level. They wanted to do something called virtual management of projects. To all the "stuff" that was already in the methodology, they wanted to add entry criteria, task descriptions, validation criteria, exit criteria, task initiation, so they could virtually manage projects by monitoring all this information. We spent time designing it, then went and talked to others about their methodologies, and none of them had anywhere near the degree of definition we were contemplating.

In my investigation of methodology, I went to the Advanced Research Facility at DuPont, where my brother worked. A friend of his, Babatunde Ogannaike, was in charge of automated process control for DuPont. Tunde, as he was called, had written one of the primary university textbooks on industrial process control. He looked through what I was trying to do and was amazed. He said that this explained why IT seemed to have so much trouble building systems. Tunde said that there were two types of manufacturing

processes in the chemical business. A "defined processes" was where everything was defined in enough detail that you could actually automate the manufacturing process. Set it up and it goes. The other type was called an "empirical process."

With an empirical process, you have a whole different control mechanism. He described a plant in North Dakota where the quality of raw materials coming in varied widely. As complexity theory tells us, the distortion caused by even small changes in the quality of input meant that anything could happen, including the reactor blowing up. Once a problem occurred, they couldn't shut the process down fast enough to keep it from blowing, so they had to have all sorts of gauges and valves on the vessel to constantly monitor it. With empirical processes, you have to monitor constantly in order to make adjustments. [Note: This was the genesis of the daily Scrum meeting].

This made me realize that the methodology industry had gotten seriously off track. I heard a talk by some industry guru shortly after that, about how software development could be just like an automobile production line—we just had to define it.

Using our traditional PERT approach to project management, we always got to 90 percent done—although "done" was very tenuous—and when all the tasks came together, we were probably only 40 percent done. Tunde looked at this and said, "You use PERT charts? You must spend all your time adjusting them." When I mentioned that most people only update charts weekly, he replied, "Empirical processes must have real-time management and measurement controls. The longer you wait, the more things get out of control."[1]

[1] Note that *control* has a different meaning in empirical and defined processes. In a defined process, control means measurements to confirm conformance to plan or prediction. In an empirical process, control measurements are for boundary conditions rather than predictions. In an empirically controlled chemical plant, for example, gauges would monitor pressure and temperature against limits, not attempt to predict pressure and temperature at a given point in time.

JIM: *About this time, in the early to mid-1990s, RAD approaches became popular. However, most of the major methodology providers grossly misinterpreted what RAD was about—they merely pulled a few tasks and documents out of their rigorous manuals and called it RAD. I was doing some work at Microsoft during this time period, and watching their development practices convinced me that RAD practices— iterative cycles, focus on code, feature-driven teams—wasn't just for small projects.*

KEN: Exactly. Around 1995, our main business was still selling those big methodologies. About that time Y2K mania was starting up, and I sold the process management software to a company that wanted to use it for Y2K. Jeff and I also made a presentation about this time at OOPSLA. We presented Scrum and talked about how it could be used on any project.

Scrum allowed us to empirically control the software development process. The daily Scrum meeting was the equivalent of the gauges and dials on a big chemical pressure vessel. It was a part of the replacement for our large project plan. The other part was the inspection of working code at the end of each 30-day Sprint. We had two inspections—the daily Scrum and the end-of-Sprint review. We then added the practices that had projects quickly adapt to the findings of these inspections, such as impediment removal, product backlog modification, and quick decision making. We used Scrum to get out really critical projects where there was a lot of emotion and intensity—"save the company" kinds of things.

JIM: *What type of project could use Scrum?*

KEN: Most of our Scrum implementations have been in situations where people are desperate for the product, and their regular defined approach or hacking has let them down. They need the product for their business to survive. An example was CoreTek, a micro-component fiber-optics start-up that was approached by a large telecommunications company as an acquisition candidate. The owner thought the offer undervalued the company, but since they didn't have a finished product, valuation was difficult. To get the valuation increased, I asked the president, "What are the most important things to valuation?" He

listed off a number of things—improving yields, an upcoming electronics show—but the one thing that seemed to be the most pressing was demonstrating a product at the show for credibility purposes.

With the show a few months away, we started daily Scrums with the team. These meetings have just three questions for each participant: What did you do yesterday? What are you planning to do today? What's in your way? The company had lots of engineers, many of them with Ph.D.s, but they weren't talking to each other. They had just been doing their own thing. These meetings got them talking to each other about what they were doing—they started communicating. The issues involved with this product involved very heavy theoretical problems; the timing of solutions couldn't be predicted.

One thing we found was that they were waiting for parts and spent a lot of time calling the buyers for different parts. We overcame this impediment by hiring someone whose job was to do procurement. We set up four 30-day milestones so we could be successful with some aspect of the product every 30 days.

The 15-minute daily meetings not only aided team communications—they were used to set up meetings in which two or three people got together to resolve specific issues—but they eliminated many other meetings. People found that they didn't have to go to other meetings because they might miss something. Managers attended the meetings to find out how the project was going but couldn't participate. The project removed political maneuvering because this project was number one on the president's priority list.

JIM: *It would seem that sticking to the meeting formats would be necessary, and a little difficult.*

KEN: I, as the Scrum Master, have to be very forceful in these meetings, in a way that people might not tolerate from internal executives. Scrum and all of the Agile processes are a paradigm shift—a whole new way of thinking that requires people to experience it. When people intellectualize about Agile, they do so from their "defined" frame of reference. Since all of the Scrum projects were life or death, I was very aggressive in quickly implementing the paradigm shift. In less-demanding situations, I'd use more example and less force. This is training in how to act in a very high-performance way. People get these stupid rules embedded in their mind and it becomes the way they operate. The Scrum Master becomes a lightening rod for all the team's politics. But the more desperate the company is, the more they tolerate it.

JIM: *It reminds me of a conversation that Alistair and I were having recently. He had a client that asked what they needed to do in order to deliver software faster and be more Agile. Alistair reeled off four or five things, to which the response was, "We can't do that!" To which his counter-response was, "Then you'd better sell your stock." Change is a bitch.*

KEN: This reminds me of another company in which the engineers were working an incredible amount of overtime on a critical project. They had to go downstairs to get coffee—and, no free coffee. I went into the president, who had something against free coffee because of a previous incident. I said, "Look at your company motto: 'We are going to produce world-class software to improve the lives of patients,' unless, of course, it requires coffee. Developers want to work, and you're saying no." He changed his policy.

Every impediment that we bubble up from Scrums we can measure against our company or project mission. We allow stuff to accrete over time and allow it to get in our way. That's the training cycle where people have to learn to mercilessly remove impediments.

JIM: *I worked with a bank in New York City. When we reduced the development releases from years to four months, we ran head on into a development-to-production conversion time of six months. Again, in an IT company, the DBAs at first refused to revise the database schema throughout the project. They wanted a one-shot database design. I finally had to say, "OK, let's go visit the senior vice president (who had approved the project and the approach) and see what she says." They found a way to revise the database schemas on a regular basis.*

KEN: Classic operations management is driven nuts by this stuff. They are used to building a stable, bullet-proof environment. In one case, we had to take over operations in order to release every 30 days.

JIM: *Once you reduce the release cycles, you force other areas of the business to respond more quickly and make their processes more Agile.*

KEN: Give me a company. Give me one project, the most important one, and we can change the company. You have to have someone who is willing to be fired—it's usually me—because you have to turn up the heat further than an internal person can do without making enemies. One other point about Scrum meetings that is critical is that once you get into the rhythm of day-to-day meetings, it almost feels like a metronome—you just crank along.

Another thing that surprised me was business and marketing people working on the backlog—selecting features for deployment. You come to the end of a Sprint demo—a visual presentation of the actual product—and that's when the brainstorming starts happening. The marketing and sales staff look at what's going on in the market and what development has been able to do in 30 days, and they can focus on the features for the next 30 days. Collaboration and partnership are good principles, but you need practices you can use in the heat of battle. Without that, it doesn't get real.

JIM: *What about measurements?*

KEN: Jeff and I go back and forth. He believes in dashboards, and I believe the only thing you should be measuring is the product that you produce, the functionality that is actually developed, and the company's ability to use it to improve the bottom line and market valuation. For example, we've reduced the whole personnel review process to three questions: What have you contributed to increase company value? What have you done to make our customers happy? What do your peers think of you? We ask eight to nine peers to give an estimate of a person's contribution on a one to ten scale. In daily Scrum meetings, you see who's helping and who's not.

JIM: *Are you using Scrum to do more business consulting than software development consulting these days?*

KEN: It seems so. Most of the projects that I've worked on are projects to build commercial products that the company depends on. I've done some IT project implementation work, and I see this increasing as organizations get more fed up with the traditional heavyweight approach. I have a friend who wants to use the Scrum process to build a new-style organization. We want to start with a live project, removing impediments and making things happen. Your alternative is using a big consulting firm and having them do a two-month assessment, recommend changes, then have everyone line up to resist the changes.

Reflections

Ken, and Jeff and Mike, have provided a solid conceptual foundation for empirical processes and specific practices for managing them. Although other ASDEs advocate holding frequent, short, team meetings, Ken helps us understand why they are so important and why "daily" is the right frequency. Scrum reduces project management to its essence—getting people to collaborate and focus on the business goals.

Chapter 9. Encourage Collaboration

Software development is not a solo, intellectual task. Rather, it is a collaborative, social task that requires lots of communication.

- *Pete McBreen,* Software Craftsmanship

The Modern Transport Team at ITL

How do you work with customers who are 8,000 miles away? In January 2001, at the invitation of India Technologies Limited (ITL),[1] I spent two weeks in India, conducting three public workshops and reviewing projects with ITL teams in Chennai (formerly Madras), Hyderabad, and Mumbai (formerly Bombay). The trip was intriguing—a new cultural experience for me and a chance to work and talk with people at a number of organizations at CMM Level 4 and 5. The issues that arose revolved around the advantages and disadvantages of CMM (and ISO) certification for exploratory projects and whether their rigorous process approaches would work on these projects.

[1] India Technologies Limited, Modern Transport, and individual names associated with these companies are fictitious at their request.

As I talked to individuals and project teams, I found them using Agile-like approaches already, although admitting to "relaxing" their rigorous process constraints was proving difficult. At an evening talk to the Computer Society of India in Mumbai, a participant voiced a comment to which many heads nodded in agreement: "Your description of Adaptive Software Development has given us a label, and some justification, for what we have been doing—but we couldn't really admit to it out loud."

One of the most interesting teams I talked with was the Modern Transport project team at ITL. The Modern Transport project was conducted in near-classic "Agile" form. The project leader and his team spent a couple of hours reviewing the project and their approach with me. The project was a multi-featured Web-based system for a transportation industry client. Modern Transport Corporation (MTC) provides a portal through which companies can schedule ground transportation services via the Web, and the member companies receive the requests and then notify drivers. There was one primary customer for the project who had significant software experience and therefore understood the nature of evolving requirements. The project ended up taking approximately 18-plus months and 600 work-months of effort.

The following description of the business has been paraphrased from MTC's Web site:

Modern Transport has developed an online reservation, networking, and e-commerce infrastructure for the ground transportation industry. The Internet allows Modern Transport to offer benefits to its target customer base by revolutionizing efficiency in providing these services. The company has developed several applications for use by both small and large customers.

Modern Transport's services combine two interfaces, or portals, one dedicated to the Service Providers and the other one to Corporate Travelers.

The system contains a wide range of functionality: travel provider profiles, customer profiles, pricing, individual itineraries, dispatch, billing, accounts receivable and payable, payroll, affiliate branding, and Web phone. The technology environment included Windows NT servers running MTS and MS-SQL Server. The development environment was Visual Basic 6.0, D/HTML, ASP, Crystal Reports, Visual Voice, and MS Visual Studio.

The team faced typical Internet-time development challenges: a very tight schedule (originally 9 to 12 months), evolving requirements, lack of an existing system (this was a new "dotcom" business), moderate levels of experience with the technology, work pressure and team motivation, technology changes during project, and knowledge transfer so everyone on the team understood the requirements and design.

The project was a success, working software was delivered, and the customer was satisfied—mostly. So to what did team members attribute success? First and foremost, this team had a great customer! The primary customer understood the nature of the project—high change—and accepted responsibility for those changes. Although the Internet-oriented business was new, the customer had experience in the transportation industry. The customer himself, because of prior programming experience, insisted on a "light" development process. There was also a single customer (which can be tremendously beneficial or not, depending upon the situation). Having the *right* single customer is wonderful. As another ITL team found out, having the *wrong* single customer can be difficult.

Several members of the project team spent an initial six-week period on site with the customer, working on the overall architecture and preliminary requirements. [Note that this time period is in line with several Agile practice guidelines for project initiation work.] One interesting comment on the initial six-week visit was that while the much of the architecture and requirements specified in that period changed dramatically, the relationships established carried throughout the project. Even though the team was 8,000 miles from the customer, they maintained nearly daily contact during the project. The team members used a mix of email, threaded discussion, chat, telephone, and teleconferencing in their collaboration efforts. Both the development team and the customer understood the need for frequent collaboration because of the constant technical and requirements changes. They understood that light methodology didn't mean light conversations.

Members of the Modern Transport team also mentioned that tight source code control (critical in high-change environments) and maintaining a detailed data model (an anchor point for key changes) contributed to their success. Operating within a CMM Level 4 organization, the team worked out variances from standard processes in order to meet the demands of a volatile project environment. For example, team members performed code reviews on a " sample" basis (rather than all code) because of time pressure, although the quality team wasn't comfortable with this "lighter" practice.

A somewhat different story emerged from the customer. After spending several months tracking down a contact, I was able to talk to Shawn, the VP of technology for MTC. Our conversation provided an example of the benefits, and the limits, of process. This was a new business, and in the dotcom heyday of 1999, speed was critical—or at least it seemed that way at the time. The first phase of the project was initiated in July 1999, and the first delivery occurred six months later, in January. So the initial objective of getting the company up and running quickly was achieved. There was significant contact between the development staff and the customer, including the customer's travel to India about once a month.

But the application suffered in several key areas: architecture, defect levels, user interface design, and business workflow design. Shawn, who joined MTC after the initial version of the application was installed, had spent a number of years with a major systems integration firm. "Their [ITL's] CMM implementation seemed to only be skin deep. The architectural issues seemed to be the result of inexperience with Microsoft products. The project suffered from poor testing and difficulty in fixing known bugs." The user interface design and business workflow problems arose from several sources: cultural and language problems, lack of business process understanding on the part of the development team, and inadequate acceptance review on the customer's part. For example, one frequent transaction required several screens and steps, taking three to four minutes, which was unacceptable in a Web interface. The database design was a "nightmare of referential integrity," Shawn said. "The theory of referential integrity outpaced its practical application and impacted both performance and user interface design."

The cultural issues encompassed misunderstanding of business processes (even when everyone thought they understood), nuances of wording and intention, and even spelling differences. Although both customer and developers tried to maintain frequent contact, Shawn felt videoconferencing, online

whiteboards, and MTC's having direct access to the evolving application would have made communications more effective.

One interesting aspect of the MTC project, and other projects that I reviewed while in India, was that even in an environment steeped in a rigorous process approach, the problem type itself—volatile, high speed, high change—dictated alterations to those processes. The experience also showed that organizations focused on delivering results to customers can adjust their rigorous processes, but it is difficult to alter their cultural legacy.

The first observation I would make about the MTC project is that it validates the principle that process—whether a rigorous CMM process or a flexible Agile one—won't overcome levels of experience and skill that are inadequate for the project. Although the ITL staff were skilled, their lack of experience with ASP, MTS, and SQL resulted in architectural decisions that hampered follow-on enhancements. Because of the customer's tight delivery requirements, ITL bypassed some of its QA steps, but whether or not the QA staff had the experience to review the architecture was questionable anyway.

A second observation is that collaboration skills and tools must receive constant attention. For example, Shawn wondered about the interaction even within the development team. "The skills they had didn't seem to transfer well even within the team," he said. "A team that had a skilled VB lead would have inadequate stored procedures, while one with a skilled SQL lead would have inadequate VB code. The sharing of knowledge seemed to be missing." Maybe MTC's relentless pressure for speed caused some of the knowledge-sharing problems, but whatever the cause, knowledge sharing—between peers and from the more experienced to the less experienced—was less effective than needed.

Collaboration between customers and developers is difficult at best. Distance and culture (whether corporate or national) exacerbate those difficulties. In the ITL/MTC project, distance, culture, and lack of business knowledge all contributed to problems. The solutions lie in better collaboration practices, business-knowledgeable team members, and constant feedback on results through practices such as customer focus groups. As an example of the last point, Shawn mentioned that feedback during development iterations was too infrequent.

So was the MTC project a success? A product did get delivered on a very short, customer-specified time schedule, but the results had some significant technical deficiencies. The development team responded to many changes, but those changes became difficult over time as early architectural decisions and lack of refactoring restricted change efforts. Would the team have been better off enforcing its standard CMM processes, or did the more Agile processes allow the team to meet the key customer requirements? It would be easy to say, "If ITC had followed its CMM practices, it wouldn't have gotten into trouble"—but if the underlying issues involved skill, experience, and communications, then additional processes weren't the solution.

Processes help people work together. They can provide a common framework, common language, and checkpoints to help catch problems. But they can't make up for problems in either competency or collaboration

A Cooperative Game of Invention and Communication

Teams using a process-oriented approach to software development focus on the internals of a particular process—the algorithms or heuristics of turning inputs into outputs—and defining, as precisely as possible, the contents of the information flows. Most RSMs, for example, spend considerable time defining the format and contents of multiple information deliverables from each defined process. Herein lies one of the primary differences between a methodology and an ASDE. The latter focuses on interactions—information exchange—between people. As people who are face-to-face, or across the world, begin having conversations about various aspects of a project, they begin to *understand* the problems and solutions. Conversations, back-and-forth debates, questions, and clarifications generate innovative ideas in ways that reading documents never can. Processes and documentation can support innovative environments—conversations and collaboration create them.

Alistair Cockburn says, "The most effective communication is person-to-person, face-to-face, as with two people at the whiteboard.... We should rely on informal, face-to-face conversation, not merely tolerate it. Face-to-face communication should become a core part of your development process" (Cockburn 2002).

Alistair views software development as a cooperative game: "a game of speed and cooperation within your team, in competition against other teams." Adaptive Software Development (ASD) stresses collaboration to solve complex, high-speed, high-change software development problems. Bob Charette declares that "every Lean Development is a team effort," and "success depends on active customer participation." XP offers pair programming, collective ownership, and on-site customer practices, and Scrum emphasizes daily "Scrum" meetings. One recurring theme among all of these ASDEs is that projects depend on talented individuals and their interactions rather than written prescriptions for activities and precise documentation artifacts. Alistair makes the point in another of his principles: "The software development process, as actually performed (as opposed, I might add, to how theoreticians think it *should be* performed) is so complex that we cannot write it down accurately, and if we could, no one could read that description and learn to perform it" (Highsmith 2000b).

The Agile Manifesto addresses the issue of collaboration in two value statements ("Individuals and interactions over processes and tools; Customer collaboration over contract negotiation") and two principle statements ("Business people and developers must work together daily throughout the project; the most efficient and effective method of conveying information to and within a development team is face-to-face conversation").

Effective collaborative groups are dependent on individual talents and how those talented people are brought together to solve complex problems. Building a strong collaborative group, which includes strong leadership and the courage to persist in the face of ambiguity and diversity, is a cornerstone of effective Agile management. Process proponents often state that people are only as good as the process that they are executing. Agilists are adamant about believing the reverse—processes are only as good as the people that are performing them. Skilled people (individual innovation), operating in an effective collaborative frame work (group innovation) are the critical success factor; process can only support them.

In today's distributed, virtual world, it would be foolish to insist that all teams be colocated. However, it isn't foolish to remind ourselves that distributed teams will never be as productive or effective as colocated ones. Study after study has shown that the nuances of close contact—having lunch together, being able to ask questions frequently, discussing the weekend over morning coffee—all contribute to the context within which conversations take place and give meaning to the actual words. Face-to-face conversations are the most effective, and distributed teams will have to work very diligently to mitigate the disadvantages of distance.

Practice versus Process

"Did the focus on process, perhaps, overlook the increasing demand for knowledge in modern organizations?" question John Seely Brown and Paul Duguid (2000). As business has become complex, global, and turbulent, organizations are placing an emphasis on people and their knowledge. High-technology companies are valued for their "knowledge assets" rather than their financial or physical assets. Brown, chief scientist at Xerox PARC, and Duguid, a research specialist at the University of California–Berkeley, discuss the differences between practice and process. Process deals with prescription and formality, whereas practice deals with all the variations and disorderliness of getting work accomplished. Process may be the ideal, but practices define how actual work gets done. The business process reengineering movement began unraveling as people began to understand that even supposedly simple processes like paying insurance claims and repairing copy machines involved a degree of conversation and knowledge sharing that process designers ignored.

Business processing reengineering guru Michael Hammer shows his bias for process over people in his 1996 book, *Beyond Reengineering:* "We now see that the heart of managing a business is managing its processes" (Hammer 1996). Brown and Duguid argue that a process bias has serious problems in a knowledge-centric economy. First, process-oriented approaches leave little room for variations. Variations

are considered to be process "failures," to be driven out by constant vigilance. Second, process-oriented approaches tend to be "relentlessly" top-down. Top-down approaches relegate creativity and innovation to the higher-up thinkers, while those on the bottom of the hierarchy become droid-like doers. Third, process-oriented approaches view people as parts, to be interchanged as needed. And fourth, process approaches discourage lateral links. People have their tasks and the "right" information is supplied to them, often from computer systems, so they have little reason to converse and share knowledge with each other.

Practices, on the other hand, reverse the issues of process orientation. Practices focus on variation as a source of innovation (like mutation in biological evolution). Practices are developed by working with people who are actually doing the work, who know the realities and idiosyncrasies involved. Practitioners understand that what manuals say and what actually happens are different. Practitioners view people as the essential ingredients to success and, furthermore, believe that each individual person brings unique capabilities to the practice. Finally, practice-centered approaches value the sharing of information. Brown and Duguid describe a knowledge-sharing study among Xerox copier repair representatives. For all the formal documentation, they found that difficult problems were often solved through conversations at breakfast before the representatives went out on calls. "Directive documentation [copier repair manuals] wasn't designed for sense making," the authors conclude. "It was designed for rule following" (Brown and Duguid 2000).

Don't we wish that business was simple, that we could describe business processes easily and efficiently? Unfortunately, business is complex. Consider dissemination of best practices, the rage in knowledge management circles in recent years. "Why not identify the best practices, follow them, and live happily ever after?" asks Donald Reinertsen. "After many years of working in product development I have concluded that the idea of best practices is a seductive but dangerous trap. Best practices are only 'best' in certain contexts and to achieve certain objectives" (Reinertsen 1997). Maybe we would be better off if we developed a list of "fairly good" practices, spent time selecting ones appropriate to our organization, and then tailored them to a particular situation.

The pattern movement has also changed how people think about practices. Practices focus on how to "do" something, but the critical piece is often not the "doing" as much as determining "under what conditions" to do it. The pattern movement encourages a format in which practices are collected and analyzed and the how to "do" is bracketed by an explanation of the context in which the practice should take place, including an explanation of the forces that the pattern tries to balance. By understanding the forces, a development team can better understand when a practice might be appropriate, and when not.

Even the idea of a "best" practice for knowledge transfer is morphing into acknowledgment that there are a series of good practices that need to be selected and tailored to the situation. Nancy Dixon, for example, postulates five categories of knowledge transfer: (1) Serial Transfer—a team learning from itself (using reviews, for example); (2) Near Transfer—one team transferring explicit knowledge to another team; (3) Far Transfer—one team transferring nonroutine knowledge to another; (4) Strategic Transfer—one team transferring very complex knowledge to another; and (5) Expert Transfer—transfer of explicit but infrequently used knowledge. Near Transfer, for example, can be facilitated by documentation, while Strategic Transfer requires significant face-to-face conversation (Dixon 2000).

Is software development process or practice? Is software design one and programming another? Does software development involve mostly Near Transfer or Far Transfer of knowledge? The answers to these questions dictate an organization's approach to collaboration and conversation within a team. Processes are fed by defined information flows both in and out. Practices are fed by conversations among project team members, management, and customers. Although a dollop of process may be necessary, it is insufficient without practice.

Documentation Is Not Understanding

"Firms need to shift attention from documents to discussion," say Thomas Davenport and Laurence Prusak (1998). From the early 1990s emphasis on business process engineering and explicit knowledge, the

pendulum is swinging toward an emphasis on people and tacit knowledge—knowledge shared through conversation.

Inevitably, when discussing ASDEs, the topic of documentation arises. In the IEEE Dynabook on Extreme Programming,[2] several critics bear down on the documentation issue. It's enough to make one scream: "The issue is *not* documentation, the issue *is* understanding!" Do the developers *understand* what the customers want? Do the customers *understand* what the developers are building? Do testers *understand* what the developers intended to build? Do software maintainers *understand* what the developers built? Do the users *understand* what the system does for them?

[2] http://computer.org/seweb/dynabook/Index.htm

Many adhere to the assumption, an erroneous one, that documentation conveys reasonably complete understanding. This same mythological assumption may also be at the crux of why reuse, whether of objects or components, has never lived up to expectations. Understanding comes from a combination of documentation and interaction, or conversation—the conversations among people who have a certain knowledge. Furthermore, as the complexity of the knowledge to be transferred increases, the ability of documentation alone to convey that knowledge decreases, and the need for interaction with knowledgeable people increases dramatically. In a complex situation, documentation by itself may provide only 15 to 20 percent of the understanding required.

Let me postulate a simple test, one I've proposed to many groups. If you had a choice, which project would you prefer to manage (and be accountable for results)? On Project I, you would be provided 1,000 pages of detailed specifications, but no access (as in none) to the client or users. On Project II, you would be given a 50-page outline specification and constant access to the client. I've never had anyone pick the first project. Understanding comes from a combination of documentation and interaction, or in formation and conversation, with interaction being far more important. This is not to say documentation is unnecessary— I would not have spent the better part of six years writing two books if I felt documentation was useless— but that documentation is not enough; it is not nearly enough. Furthermore, at some point documentation begins to contribute more to confusion than comprehension—or worse, as we will see shortly.

A little research into the field of knowledge management indicates that efforts to document best practices (software development or otherwise)—filling reams and reams of Web pages with details—have proved of marginal benefit. In thinking about knowledge transfer, one must distinguish between transferring explicit knowledge (written down) and tacit knowledge (in someone's head). My brother Rick was a scholarship tennis athlete in college. He had a booming left-hand serve that was nearly unhitable. One afternoon a middle-aged friend asked Rick, "If I work really hard—take daily lessons and practice several hours each day—how long will it be before I'm as good as you?" Rick's stoic humor was in full swing when he answered, "Never." Transferring tacit knowledge to someone else can take a long time. And if knowledge transfer can be done at all, a mechanism like face-to-face conversation has proven to be much more effective than documentation.

I vividly recall a statement from a project postmortem several years ago: "The specifications for our project were great; we just interpreted them in radically different ways." So the difference between ASDEs and RSMs is not one of extensive documentation or no documentation, but rather the correct mix of documentation and collaboration to convey understanding. Agile practitioners lean toward interaction, whereas rigorous methodologists lean toward documentation.

For an XP team—a small team, colocated, with an on-site customer—understanding can be generated from minimal documentation in the form of story cards and intense customer interaction. When several teams are involved, the teams are distributed, and customers are many and varied, story cards alone will probably not be sufficient. However, the critical question in this latter situation is not what kind of documentation should be generated, but rather how the project manager will facilitate adequate interaction among all the participants. The secondary question is how much documentation will be necessary to capture the information exchanged in those interactions.

While interaction among project participants is the prime contributor to understanding and documentation merely supports that understanding, there is also a downside to documentation that isn't often grasped:

Documentation creates barriers to conversation.

Consider how the waterfall life cycle evolved. First the software development process was divided into phases—planning, analysis, design, coding, testing—and then roles were assigned to each phase—planner, analyst, designer, coder, tester—and then documents were defined for each phase—plan, specification, design document, code, test cases. As roles gave rise to specialized groups and people believed that documentation could convey all the required information to the next group, the barriers to conversation became greater and greater. During the period of waterfall ascendancy, there was also an underlying mechanistic, factory model of software development in which programmers became merely coders—carrying out the precise instructions of the analysts and designers as a factory floor worker carried out the process laid down by an industrial engineer. The more rigorous and documentation-oriented the process, the less the incentive for conversation.

Conversations, when they did occur, were between adjacent role groups—so analysts had conversations with developers, but few developers were allowed to interact with actual users. Groups naturally became very protective of their roles, and cross-functional teams were not the norm. However, when we take documentation away from this environment, conversations increase and the work effort improves, as investigations of successful teams show. For example, one of the most consistent best practices throughout the 1980s, the era of the factory process, was joint application development (JAD) sessions, which brought people together in conversations. Unfortunately, they often brought only users and analysts together, leaving out the remainder of the development teams.

Understanding comes from a combination of documentation and conversation. Documentation can provide content (facts), but conversations are much better at building context. We need both, but our old assumptions about documentation are hard to break. We have to remember that while documentation contributes to understanding (current and future), it also creates barriers to conversation.

ASDEs have demonstrated again and again the primacy of conversation over documentation. Sometimes, in trying to make the point about conversation, Agilists have been overzealous in our denunciation of documentation. However, scratch just beneath the surface, and most of us are not against documentation as much as we are supportive of minimalist documentation.

The Dimensions of Collaboration

"The most effective form of communication (for transmitting ideas) is interactive and face-to-face," says Alistair. Other experts have agreed with his sentiment. However, collaboration encompasses more than face-to-face interactions. Although face-to-face may be the most effective, our virtual world of work forces us to deal with distributed collaboration as well. Collaboration differs from communication. Communication involves the exchange of information from one person to another—as in the exchange of documents or ideas. Collaboration involves joint participation in producing a product, or jointly making a decision—active participation rather than passive exchange.

There are two key questions about collaboration: First, why do we collaborate? Second, what factors contribute to effective collaboration? To the question of why we collaborate, the columns in the Table 9.1 indicate three reasons: to create deliverables (working software), to make decisions, and to share knowledge (learning) in order to create those deliverables or make decisions. To the question of what factors contribute to collaboration, the rows in Table 9.1 indicate three factors: interpersonal (trust and respect), cultural (values and principles), and structural (organization, practices, and technology).

In Table 9.1, the interpersonal characteristic addresses issues of trust, participation, commitment, and respect. Without a degree of trust and mutual respect, group interactions degenerate rapidly. Cultural collaboration addresses the overall management culture and how supportive it is of team collaboration. Competition within an organization can take two basic forms—supportive and cutthroat. Trying to build a collaborative decision-making process in a cutthroat competitive culture would be a waste of everyone's time. Similarly, trying to build collaborative practices into a rigid, hierarchical, control-oriented culture would be folly. Structural collaboration can be both technological and organizational. For example, there are a growing number of good collaboration tools that assist groups in sharing information, documentation, and ideas. An organizational structure that supports knowledge sharing is a community of practice, an informal group supported by the organization and tasked with furthering the knowledge about a certain area.[3]

[3] For information on communities of practice, see Wenger (1998).

Table 9.1. A Collaborative Framework			
	Deliverables	Decisions	Knowledge
Interpersonal			
Cultural			
Structural			

While the left column in Table 9.1 addresses group interactions, the top row addresses the results generated by those group interactions: deliverables, decisions, and knowledge. Groups interact to produce results—deliverables. Whether a small group gathers around a whiteboard to develop a class diagram or two programmers huddle together as "pair programmers" to jointly produce code, one objective of collaboration is to produce demonstrable results.

A second "outcome" of collaboration is decisions. In fast-moving environments, in which information is often voluminous, decisions need to be made quickly and well. Often these decisions need to be made by those with the best grasp of the information, even if they are not managers. Decision making is a core skill being ignored—in any meaningful way—by many software development organizations. Decisions are made, of course, but there is just not much deliberation about who makes decisions, who is involved in the process, what criteria are used, or who makes and is affected by the decisions. Even worse, while few organizations do well at learning about themselves via project retrospectives, far fewer attempt to critically examine their decisions and decision-making process. In the world of distributed project teams, sharing information instantly around the world is easy; however, sharing decision making around the world is formidable.

As management pushes development for faster results, it often denies its own culpability in slowing down the process due to poor decision making. In the late 1990s, I conducted a workshop for a development group in Ireland to help the development staff accelerate delivery. Two things were contributing to their speed or lack thereof: confusion about the roles of project, program, and product management, and a broken decision-making process between the Dublin development center and "headquarters" in the United States. Organizations wanting to excel at Agile management need to examine and improve their decision-making skills.[4]

[4] Decision making is a formidable issue in its own right. Sam Kaner's book, *A Facilitator's Guide to Participatory Decision Making* (Kaner et al. 1996), is an excellent starting place.

Producing deliverables and making decisions require knowledge. When groups are involved in either creating deliverables or making decisions, the individuals must share knowledge. But sharing knowledge is much more than documenting practices or drawing diagrams. The U.S. Navy, for example, emphasizes contextual thinking and case histories in knowledge transfer. In a *Computerworld* article, Larry Prusak of the IBM Institute for Knowledge Management commented on the Navy's program: "Context is not words, and it's not numbers. It's the surrounding meaning of things. One of the failings of expert systems is they don't capture context, just rules. The military used to teach strategy with schematic diagrams—arrows,

circles, and boxes but has largely moved to case studies and stories because they convey context far better" ([Anthes 2000](#)).

Collaboration—whether to create deliverables, make decisions, or share knowledge—is rarely, if ever, viewed as being as important as either project management or software development practices. However, if we are to manage projects in the Information Age, the dominant critical success factor must be constantly reinforced—innovation. An organization's ability to innovate is tied more closely to its collaborative framework than either project management or software development practices

Real *Teams*

"The value of flexible development therefore hinges on the quality of the process for generating knowledge about the interaction among technology, user needs, and market requirements," Marco Iansiti of the Harvard Business School reminds us. "Unlike traditional development projects, where research on user needs provides occasional bursts of input to the development process, projects here use essentially constant feedback on critical features" ([Iansiti 1998](#)).

The most effective form of conversation involves a small group of people standing around a whiteboard discussing an issue. As greater numbers of people and increasing distance enter the equation, four sets of techniques can be used to scale the conversation: documentation, proxies, facilitated meetings, and tools. But these techniques only help us *scale* conversations, not replace them.

RSMs often devote volumes and volumes to process descriptions and documentation, but they are relatively silent when it comes to collaboration practices. Although code reviews and project retrospectives are often recommended, they focus on the processes rather than the interactions of people. Managers and developers who agree that collaboration is necessary, in theory at least, rage against the XP practice of pair programming as too "expensive" (although a growing body of research contradicts this assumption of extra cost). Similarly, while lauding the concept of "teamwork" and "high-performance teams," these same individuals go apoplectic at the thought of XP's communal code practice.

Collective ownership is nothing more than an instantiation of the idea that products should be attributable to the team, not individuals who make up the team. "We believe that the truly committed team is the most productive performance unit management has at its disposal, *provided there are specific results for which the team is collectively responsible,*" write Jon Katzenbach and Douglas Smith in their popular book *The Wisdom of Teams* (1993). The team, not individuals, succeeds or fails in this version of teamwork. Many people laud teams and teamwork in the abstract, but they don't want to be burdened with any real practices that change their ingrained habits. One thing about ASDEs is that their practices instantiate their principles. Whether pair programming and collective ownership are the right practices for every team isn't the point. The real issue is whether managers and team members are mouthing "collaboration" platitudes or implementing practices that match their rhetoric.

Every project has a task plan, many meticulously documented in Microsoft Project. How many projects have you seen that have communications plans, much less full-blown collaboration practices? Collaboration doesn't just happen. Collaboration practices are just like programming practices—they require skills and experience. Practices advocated by Agile approaches include pair programming, collective ownership, daily team meetings, technical reviews, joint planning and requirements gathering sessions, iteration and project retrospectives, customer focus groups, storytelling, facilitated Web meetings, chat, threaded group discussions, collaborative decision making, and face-to-face meetings of multiple types. How many of these are your team using? Was there any analysis of what practices work best for the size and distribution of your team? Does your definition of "team" mean a bunch of people working together, or does it embody the principle that "team" products are the measure of success? Who makes decisions? Are your organization's principles and values about people and interactions embedded in how the practices are implemented? In order to "[encourage collaboration](#)," an organization needs to address all of these questions and more.

Chapter 10. Martin Fowler

Martin Fowler holds the position of chief scientist at ThoughtWorks, a Chicago-based consulting firm. Basically, as chief scientist, Martin runs around the world lecturing and consulting, spends time writing, and assists on ThoughtWorks' projects. Sounds like a pretty cool job.

His books include *Analysis Patterns: Reusable Object Models* (1996), *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (with Kendall Scott, 1997), *Refactoring: Improving the Design of Existing Code* (1999), and most recently, *Planning Extreme Programming* (with Kent Beck, 2001). He is currently working on a book about information systems architectures.

Martin has been a regular speaker at technology conferences, including OOPSLA, Software Development, and ECOOP. He was born in England, moved to the U.S. in 1994, and enjoys hiking and cross-country skiing.

Martin and I shared a pint at the Lodge Bistro lounge at Snowbird after an afternoon of shushing down the slopes. Trying to have a "serious" talk over the music and the growing après-ski crowd was a challenge, but we managed to bear up.

JIM: *What was the evolution in your career that brought you to ThoughtWorks, refactoring, and your work with XP?*

MARTIN: When I got out of college in 1986, I went to work for Coopers & Lybrand and was put into a group doing methodology stuff. The only reason was that they were experimenting with this tool that ran on UNIX, and I was the only guy who knew UNIX. Even though I was the techie, I was interested in what they were doing with the tool. The tool was called Ptech, and the guy responsible for development was John Edwards. He'd had some contact with information engineering and then branched out on his own. He was a brilliant guy, but his crucial mistake was that he believed in keeping everything secret. He didn't tell anyone what he was doing because he was afraid they would steal it, and the results were, of course, no one knew what he was doing and he never got anywhere.

John was convinced that iterative development was the way and was very into diagrammatic approaches. He was very serious about the diagrammatic thing, but he still kind of missed the point [about the limitations of diagrams]. Coopers was really just interested in drawing a lot of diagrams and producing a thick report, as far as I could tell. I got frustrated because they never seemed to actually build anything. In the end, I felt it was all very well to do this design and modeling stuff, but you've got to build something at the end of the day.

I was convinced that the Ptech way of going about things made sense. My background was in electronic engineering, and I was used to the idea of developing a drawing and then building from that. I then joined Ptech for a couple of years—Ptech stood for Process Technology. John was way ahead of his time [this

was prior to the big push into business process reengineering in the early 1990s]. We got the tool to the place where it could build a large part of a system and did a subcontract job where it handled 75 percent of the code—the exception being the user interface. Ptech's edge was an event diagram that you could actually write algorithms in. However, in the end, we ended up with a new programming language. Then you have to ask—how easy is it to learn and to write, and how do you debug and test it?

JIM: *What was the next stop?*

MARTIN: I left in about 1991 and went independent. We had been doing a lot of work for the National Health Service in Britain. We (there were two of us in Ptech in Britain) were producing a clinical model of health care. I kept pushing them to actually build something. We did some interesting modules in Smalltalk, and it proved out one of the things I was already pretty sure of, which was when you actually sit down and code something, you learn things that you didn't get from thinking about them in modeling terms. Though I found the diagrams were useful as a way of exploring what a system might want to do and how it might do it, I favored the whiteboard as my primary tool.

There is a feedback process there—I now think of it as a feedback process—that you can only really get at from executing some things and seeing what works. You realize that—oh—there are other abstractions I can use that you only get from working with them. Smalltalk is fantastic for that; it is an incredible plastic medium. In this regard, Java is not as good.

JIM: *At what point did you move to the U.S.?*

MARTIN: I came to the U.S. in 1994 and continued working as an independent consultant, but prior to that, at OOPSLA in 1992, I listened to a talk by Dave Thomas (of OTI fame), who made a somewhat offhand remark that every object should test itself. It set a little jingling in my head: "Oh, that's an interesting idea." So in my own work I started writing tests for every class, and I found it made a profound difference. I could change things and screw it up—and I screw up lots of stuff—but the tests would quickly tell me that I'd screwed up. But I couldn't get a client to do this. I would suggest that they build self-testing classes, but they weren't interested.

Another very common characteristic of projects that I began to notice was that no one had an idea about how to plan a project. I had learned on the National Health System project that Microsoft Project was a disaster. It forced me to think in this very weird way about a project. All the things that were important to me, I could plan more easily in a spreadsheet.

JIM: *Didn't you get involved in the Chrysler C3 project around this time?*

MARTIN: Yes, around 1993—before Kent. A guy named Tom Hadfield, from Chrysler, was involved. He was a really interesting guy and wanted to replace the creaking old payroll system. I led a modeling effort for them, and I still feel that a lot of the ideas that ended up in C3 were born during some of these sessions—who knows? Chrysler contracted with Good Object Programmers, Inc. [a fictitious name] to do the development work, and it was a complete disaster from the start.

The people at the company were clearly out of their depth. They'd done 2- or 3-person projects in the past, and now they were doing this one with 30 to 40 people. They had no idea about how to plan or design things. They said they were doing iterative development, but they weren't. "Of course we're doing iterative development," they would say, "but we're leaving testing until the end and we're not going to integrate the database until the end either."

I suggested the self-testing idea, but it was way too radical for them. Tom tried to rein them in, but they were also arrogant about their OO knowledge. "We know what we're doing, get out of our face," was their general attitude. They didn't think anyone else knew anything about objects. We used to joke that our biggest fear was that they would actually ship a product, because it was such a mess.

Tom left in frustration in November 1995—he had all the responsibility and no authority. The project sunk further into trouble, and in January 1996 they admitted that they had "performance" problems and called in the known Smalltalk performance guru—Kent. Luckily Tom was involved in writing the statement of work for Kent's work. The statement of work was wonderful; it said, "Come in and help us with our performance problems, but if there's anything else you'd like to chip in while you're here, make sure you do so—comment on the design, etc."

And, of course, Kent completely exploded the project. They'd reached the point where they would fix a bug and five more would pop up. They had a big meeting, and the contractor couldn't predict when the project would finish. So Kent ended up taking it over. This is now the C3 project that everyone talks about. They threw away a lot of code. Started from scratch—well, it wasn't really from scratch because of the fact that they had already built the system once. The developers learned a lot from that experience, so it wasn't as if they started from a clean sheet of paper. The knowledge had been gained, it just hadn't been chunked up properly.

I was invited back to the project. Lots of things about this new project were really wonderful. The planning was really good. It was basically the way it's described in the green book [*Planning Extreme Programming* ([Beck and Fowler 2001](#))]. The plan was always believable. You looked at the plan and thought, "Yeah, that's what it is. It may not be much, but that's what it is." It was the best plan I'd ever run into. I was also very impressed by the testing. One of the things that made me comfortable with Kent taking the project over was when he said that testing was one of his primary things.

This was where I first ran into refactoring—I hadn't really run across it before. And you can see what I thought of that—I spent two years writing a book about it.

The C3 project struck me as very controlled, very disciplined—but at the same time very flexible—and I really liked what Kent had done. But I also bore in mind the fact that a few things were particular to that project. They had built it once and thrown it away, so we really didn't know the full impact of XP. We didn't know how it would work in other teams.

The C3 team members had been shattered by the original failures and basically said they would do whatever Kent told them to do. And I know as a consultant that it is a very rare occasion that the client actually does what you recommend. They did virtually no modeling at all, but I'm still not convinced that modeling is useless.

JIM: *So maybe "light" modeling is reasonable?*

MARTIN: Part of the reason they did OK without it was that it was Smalltalk, they were very disciplined, and they had considerable domain knowledge from the prior project. If you really want to duplicate C3, you have to screw up a project first before you do it properly. Although many successful projects have done just that. Maybe it should be part of the methodology—but I'm not sure I could sell it to management.

JIM: *How did this work with C3 and XP correspond to your use of UML and writing the UML book?*

MARTIN: The *Analysis Patterns* book hit the shelves at OOPSLA 96, and I wrote the UML book at the beginning of 1997—a very fast book. First decision to write to on the shelves was three months—we cooked. UML silenced a whole lot of stupid arguments about notation that were never very interesting in the first place. And that was the only thing it was good for, which meant if we were going to do all these little models, we wouldn't be arguing about what shape boxes to use. It was a huge step forward, but it was no more than what shapes to use. It's like if you want to learn how to ski, it's kind of important that you know how to put your ski boots on—it's that level of importance. Now if you can't do that you're really screwed, but it just moves you on to the interesting stuff. So I'm kind of uninterested in UML in a way— OK, that's done, that was important, very important to deal with, now let's move on to really interesting things. But I am happy to have written a UML book that has gained wide acceptance.

JIM: *Do you see people confusing the form of UML with thinking about how to do analysis and design?*

MARTIN: Yes, that's completely unsaid by many. The problem is how do you build a good design—that's what is really interesting, it's what really interests me. Once you come up with a good design, UML diagrams help explain it. It's a communication tool and valuable for that.

JIM: *What about your work at ThoughtWorks?*

MARTIN: I've been pushing for Agile, lightweight processes—sometimes XP, sometimes others. To me, XP is one instance of an adaptive process. I expect that we will end up with a number of these methodologies with overlapping boundary conditions. We will need to choose which one best matches our profile. We'll start with one of these then rapidly tune it to a particular circumstance.

JIM: *Alistair Cockburn and I are approaching methodology from the same perspective, a series of process templates that need to be tuned by the team that is using them.*

MARTIN: Certainly, anything you pick will only be a starting point and then you tune it to what you need it to be. It's like a ladder that gets you up, but then you can kick it away. [JH: I forgot to ask Martin about getting down!] You need it to get started. Methodologies can also give you good practice ideas.

The other thing I really learned from XP was the way in which techniques interact with each other. Before, I was in favor of best practices—such as those described in Steve McConnell's *Rapid Development* (1996). But what I learned from Kent was that you can't choose the practices in isolation; you have to see how they fit with the others. There might be an excellent practice that makes sense for you, but it doesn't jell with the other practices. Like a code review may still be a valuable practice, but not worth the cost if you are already doing pair programming.

JIM: *What are things about Agile methodologies that you like?*

MARTIN: I think the basics are very easy. In my "New Methodology" paper,[1] the two key things I picked out were adaptive and people orientation—two critical things. Now adaptive grows into self-adaptive, which is the process changing itself. It sort of flows—if you have an adaptive process and you are people oriented, you have to be self-adaptive. Lighter methodologies de-emphasize stuff. It's better to start with something small and accurate. Look at RUP. It has tons of really good stuff in it, but you've got to figure out which bit of stuff you want.

[1] "The New Methodology," www.martinfowler.com.

JIM: *I find that people always have good reasons for "keeping" something, so it's much harder to cut down a methodology than build one up from something simple.*

MARTIN: Right, that's true of Agile methodologies in general. XP added in the testing, which has really huge effects—the automated testing. The planning I really like also. I read stuff about planning and like it, but then ask, What am I supposed to do? [Martin mentioned both Crystal and Adaptive approaches as examples here.] With XP, it says do that. May not be the best thing, so you can tune, but you have a starting place. I can take that and do it. So even if I wasn't dong XP, I'd still push for the planning and the testing. I know how to do the planning and it works very well, and the testing I think makes such an enormous difference.

JIM: *There's this concept of a folded-zippered process. Whereas a waterfall process was linear, with testing coming at the end, Agile processes are folded and zipped together, such that critical activities like testing aren't relegated to the end of the project.*

MARTIN: Yes, with automated testing you just know if you take a backward step. You're going forward all the time. Up-front testing also alters your design. It makes you think about what's a good emergent property. The role of testing has a lot of emergent properties, which makes it a thing I really want to crack a whip over.

JIM: *At OOPSLA 2000, I made a statement in a talk that one of the reasons I thought XP had made such an impact was that the XP principles defined a community that people wanted to work in.*

MARTIN: A very interesting thing with XP is that there are many people who are drawn to it that don't want to do it. A great example of this is pragmatic Dave Thomas, who loves a lot of things about XP but doesn't want to do this practice or that practice. What he likes is the philosophy part of XP. The key thing that XP did was not just provide the philosophy, which frankly you do better in your book [JH: "Thanks."], but it provides an instance of how you can achieve it. The combination of the two is very powerful.

JIM: *I like that about XP also: It's a good, well-defined set of complementary practices—as some have said, a system of practices. I think both philosophy and practices are important. Why do you think XP has gained such notoriety?*

MARTIN: Kent also has a way of shocking people. I've described XP as a cattle prod to methodologists. For example, Alistair's ideas haven't really shocked the industry the way Kent's have, although Kent's and Alistair's ideas are similar in quality. But somehow Kent managed to create a shock effect. The second thing is Kent really managed to create a gathering around him. So you look at XP, and even without Kent, there would be tons of people still pushing it. Some of these people—Bob Martin and Ron Jeffries come to mind—already had a good reputation. A lot of people with good ideas haven't managed to generate the same amount of interest. I don't know why this is, but it's very interesting.

For a while I was worried that XP had sort of sucked the oxygen away from other Agile methodologies, but now I think it has been beneficial. Because XP has caused so much attention, people get pushed toward all Agile approaches.

JIM: *I want to increase the interest in all Agile approaches, figuring that if the total pie is bigger, then each approach will get a larger slice. I've never felt that I was competing for clients with Alistair, Kent, or others. Mostly I compete against the RUPs and CMMs of the world.*

MARTIN: It's always been a world of borrowing from each other, and that's what works well. Because XP has a tight set of practices, you can wind up with this rigidity coming out, which I was exploring in my variations to XP article.[2] But it is only when you actually *do* XP that you can really understand how it works—well, maybe that's too strong. Quite a few people have been surprised by what XP was like when they actually did it—it was different than what they *thought* it would be like.

[2] "Variations on a Theme of XP," www.martinfowler.com.

As a result, some XP people are reluctant to have others second-guess what to do when they haven't experienced it. Ron is a good example. He wasn't XP prior to C3. He was surprised, and I was also. I thought the C3 team would really suffer because they didn't do diagrams early on. I was surprised how well the combination of testing and refactoring worked. The whole thing about YAGNI—I was very uncomfortable with that—but I saw it work, and again I was surprised. So I can understand how someone else comes in and thinks it can't possibly work, but until they have done it, they are missing something.

JIM: *The U.S. Army has a concept it calls "on the ground." That is, to really learn something, you must be there where the action is—on the ground. So try it first, then adapt.*

MARTIN: You've got to do it first, learn from doing, then tune it—become self-adaptive. A fundamental thing about any methodology—in some ways you can only follow it by not following it. When you get to that level, you're clicking. In the beginning you need to adapt, but you don't know what that means until it's clicking. Once it's clicking, then if you deviate in a bad way, you no longer get that clicking and you know to come back again. If you've never had the clicking, you don't know if you are off track.

JIM: *There are a number of these subtleties that come up in actual practice that just can't be written down.*

MARTIN: Yes. For example, there are a large number of people who will never pair program, but there are also a large number who never *thought* they'd pair, but when introduced to it properly, they liked it. You never know which camp you're in until you try it—seriously try to make it work.

JIM: *How much does the success of XP and other Agile methodologies come from working together, from the collaboration?*

MARTIN: Collaboration is key. I mean, Kent makes the comment that communication is the key bottleneck in software development, and he's not joking. Jack Bolles, one of the guys at ThoughtWorks, said this piece really became clear to him in Sardinia [the XP2000 conference]—it really was the collaboration, it was crucial to success. It's absolutely embedded in XP—people oriented as opposed to process oriented. You look at the people and then make the process fit them. You recognize people are human beings and they are also the most important factor in whether you succeed or not.

One of the things I'm always impressed about at ThoughtWorks is that much of the reason the company succeeds is that tying of projects to individuals. You put *this* individual and *that* individual together because of their personalities and their skill sets. Management thinks about people when they are analyzing the teams.

JIM: *What is your vision for the future and your part in it?*

MARTIN: In a way, I'm not very interested in methodology. I'm not a methodologist; I try to resist it. My interest is in what kind of design works. What makes a good design? How do you recognize good design and what do you do with it? I'm very conscious that process has an effect on design, often a stifling effect. I basically want it to get out of the way and let the individuals excel—that's why the people-oriented idea really makes a difference to me. Something that enables people to do their best.

JIM: *I've been talking about the difference—often not recognized—between the terms "formality" and "discipline." You can have a highly disciplined group that's just not very formal—which is how I perceive XP. Given a choice, I'll take discipline over formality every time.*[3]

[3] The idea of formality versus discipline is covered in more depth in Chapter 25 in the section on scaling.

MARTIN: The point is fundamentally that human beings, developers, are not stupid—they are not going to do something that doesn't seem to be adding value. For example, up-front testing doesn't seem to be adding value until you make a big change and the problems pop right out. You get into this rhythm of writing tests first, but you've got to get over the initial hump of proving to yourself that it's worthwhile. The discipline is necessary while you're getting over the hump, then it isn't needed as much because you're going to do it. The problem with heavy process is that they throw all this stuff at you, and you couldn't figure out the benefit—at least none I could ever see.

A wonderful example came to me from ThoughtWorks colleague Jim Newkirk, who was working with a client who was using RUP. The senior guy has to produce an updated architecture document, and Jim asked him who reads this stuff. The guy replied, "I don't have the foggiest idea; I can't even get anyone to review it. But it's required by the methodology, so I have to do it." That's the stuff I really hate. Or CASE tools that force you to enter insanely rigid stuff to draw simple diagrams.

Another thing I learned on the C3 project was how you can really make code itself be communicative. I hadn't realized the degree to which you can do that. You have to try, do a lot of refactoring, and pay a lot of attention to it. But you can really write code so you read the high-level code and it's what the documentation would read like except with funny symbols here and there, but to a programmer they are understandable.

JIM: *With refactoring, how do you guard against continuous refinement that gets out of hand?*

MARTIN: The key is iterative development in which every iteration you have to produce value. That keeps the pressure on everybody to create something of business value. That's what's going to create the balance.

This reminds me about what inspired the paper I wrote, "Is Design Dead?"[4] I was having a conversation with a well-known industry person, and he was saying how much he hated XP. He said that the XP white book [*Extreme Programming Explained* (Beck 2000)] was the most dangerous book published in the last ten years. I began exploring that statement with questions like, "Do you believe in iterative development?" To point after point, he agreed with the principle. I remember wondering what the difference was. Then it came out. He said, "I can't have programmers monkeying around with my design."

[4] "Is Design Dead?", www.martinfowler.com

A key piece of XP is that you have to trust your developers, and the coach's role is to teach them how to do it well, but in the end you have to trust them. And if you haven't got that trust, XP can be threatening. That was the difference. He hated XP because he views his role as an architect as stating how things are going to be—laying out the design and making all the critical decisions and having developers carry them out. That's a very different value system from XP's.

Reflections

As I think about my conversations with Martin, Bob, Ken, Kent, and the others, several thoughts prevail. These no table individuals have rich backgrounds in software and have thought deeply about development practices—most for 10 to 20 years. Their path to Agile methods didn't start in 1997 or 1998, but from years and years of work—from successes, and failures. One of my goals in publishing these interviews is to show that ASDEs aren't a fad (although there has been significant recent press about Agile approaches), but the result of years of careful thought and experimentation.

Chapter 11. Technical Excellence

Self-confident people make it simple, and simplicity makes you fast.

• *Warren Bennis, "Will the Legacy Live On?"*

The PDFS Team at Generali Group

Alistair Cockburn never told me about the "mother-in-law" feature in Crystal. It seems that on a Crystal Methods project in Germany, one of the developers occasionally finished new features during weekends. On Monday, he would return to work having implemented several new features that had been discussed but that no one had time to implement. Jens Coldewey, the project coach, knowing the developer had two kids at home who didn't want dad working away all the time, asked about the weekend work. "Well, my mother-in-law visited us this weekend," the developer replied, "so I took some time with my laptop...." Jens reported that this particular mother-in-law, who probably never heard of the project, is still one of the project's most valuable team members.

Jens, a German consultant, has been working for several years on a project for the German division of the Italian insurance company Generali Group. The project was to develop a visionary, product-driven insurance system that included bid, contract management, and claims management features. The subproject that Jens consulted on, and that used Alistair's Crystal Clear method, was the product definition system (PDFS). According to Jens, with all the twists and turns that occurred as the product evolved, the project would have been cancelled using the company's standard methodology. Crystal provided the practices and the philosophy that enabled the project team and management to weather several major shifts in direction.

Normally, implementing a new insurance product takes two to three years. Market research, domain-level development, training, actuarial calculations, and inefficient organizational practices take two-thirds of that time. The remaining time is due to IT implementation of the new product—and the more innovative the product is, the greater the percentage of time IT implementation requires. The vision was to shorten the insurance product introduction period to six to eight months, a goal sought after but not yet achieved by Generali's competitors. The new product needed to support the entire insurance value chain from offer and contract management to claims management, statistics, and reinsurance, and to provide an application that insurance domain experts could use to quickly define and test new product ideas. Once the idea was verified, the new system would automatically update all the affected IT systems.

The first company to have a such a system would have enormous competitive advantage, since it could implement innovative products fast and react to competitive challenges. At this point (mid-2001), Generali appears to be the closest, in Europe, to implementing such a system. Its first system will be deployed shortly in the Netherlands.

"The whole concept of product-driven insurance applications is new, so nobody really knew what it was," said Jens. "We started with a vision of 'a contract is an instance of a product' and tried to solve the domain questions from that. If you read the original requirements document today, it only scratched the surface—and went wrong in several aspects. On the other hand, the vision we started with proved to be quite stable; it only had to be corrected in some details."

The project (PDFS) began as a single-user system developed in Visual Age Smalltalk. The system was stand-alone, but there was an optional repository database on a server to synchronize users. The team started in January 1998 as a group of five (one project manager, one domain expert, and three developers, with one concentrating on domain analysis) plus Jens as a coach. Currently (mid-2001), the team has ten

members (one project manager, two domain experts and testers, one domain architect, five programmers, and the coach).

As with any innovative product, the project has had a checkered history. Asked if the project was successful and how he and his team measured success, Jens replied, "There was a simple measurement: survival. The project survived several severe political stands, and we finally convinced even strong opponents. We have managed to meet one of the major challenges in insurance IT today and starting production (under way) is the final step for success. Several other companies in the group have shown interest, so I think we are pretty successful."

Part of the problem with determining success on projects like this is that *conformance to plan* has little relevance. "The current system has only vague similarities with the original plan," said Jens. "We changed the technical platform and the complete architecture of the runtime components. There are some themes you find all through the project, but most things have changed. This flexibility was part of our success. A manager told me that the project, as planned originally, would have been canceled after eight months using their standard approach to projects."

The checkered history involved two major shifts in the project's goals. The first occurred when the team dropped the idea of developing a new object-oriented contract system. The original goal was to develop a contract system in a fat-client architecture using Smalltalk. This was part of an overall initiative to switch the complete back-office software to an object-oriented Smalltalk system. After one year of work with the PDFS, the group decided not to make the transition to Smalltalk but to enhance the existing mainframe systems. This change in direction did not affect short-term work too much but caused a major shift in the long-term schedule.

The second major shift occurred when a second project team in the insurance group started to develop another product system. Although the PDFS group thought its product concepts offered the better business opportunities, there were strong political pressures to integrate with the competitive system. Reacting to this challenge, the team changed its strategy and revamped its internal product.

On the technical side, there were two significant changes: The team changed the target operating system from OS/2 to Windows, and the persistency mechanism changed twice. The team discarded the original object database—after the database vendor proved unwilling or unable to fix a high-priority bug that caused regular system crashes—and went back to an ODBC repository.

The PDFS team used Alistair's Crystal Clear method as a framework but also included elements of XP. Jens, as the OO coach on the team, proposed an incremental approach. He began with a two-day process workshop and, based on the stringent timing for the first deliverables, had the team answer two questions: "If you think of your previous projects, what enabled you to go faster? What slowed you down?" The team members took the answers from this brainstorming session and designed a process that facilitated the accelerators and avoided the slow-down practices. They repeated a "process analysis" workshop after each increment to further customize the process.

"I had proposed to work with a lightweight approach because in several extensive exchanges, Alistair had convinced me that this way might work," said Jens. "I had bad experiences with heavyweight processes before and was quite sure that we would not succeed with a formal process. After the first increment, we called in Alistair to review the process. This approach had good support from middle and upper management." The company had a standard, well-defined process—which called for 211 reviews of different artifacts before coding could start. "The team all agreed that using this standard process would lead to disaster."

The team used increments of three to four months and prioritized requirements as a team at the beginning of each increment. The team wrote requirements documents for every feature but didn't maintain them after the feature was implemented. The design was done by a small team made up of the analyst, one or two programmers, and Jens. Their design tool was primarily a whiteboard, and a typical design session lasted one or two hours. The team used automated testing (using Kent Beck's testing framework and a

commercial tool named TestMentor) and also did regular refactoring. In the beginning, the team used interaction diagrams in design, but as the architecture stabilized, the key question changed from "What does the design look like?" to "How should we refactor our system?"

"However, all my efforts to implement pair programming failed," said Jens. "The team and the project manager just were not convinced that it provided enough benefit. Still, for difficult parts, sometimes a couple of programmers would pair."

The team did regular code reviews but abandoned them at crunch times. Jens said, "Everyone thought the code reviews helped, but time was a factor. Most of the developers have a good programming style, so I didn't consider this as a threat to the project." The team also used a configuration management tool, which Jens considers a vital ingredient to lightweight processes.

Changes to the "methodology" over time included the approach to analysis, design, and testing. The members of the team started with use case–driven analysis but over two or three increments found that they did much better by using features. The team experimented with UML, but did not find it very helpful, so the documents were text and GUI screen shots. These documents and the design meetings were used to brief the programmers, who worked in close collaboration with the analysts. As Jens observed, "When the programming is done, the solution usually is much closer to the original *ideas* of the analyst than the analysis documents."

Automated testing was introduced during the first increment for a small part of the system. "I showed it to other team members, and by the end of the increment, everyone had written test cases," said Jens. "Now we have more than a thousand test cases for the complete system. Changes rarely lead to new bugs, so the team heavily relies on the test cases—with appropriate feedback when someone forgets to secure his or her code with test cases."

With a light, Agile process that included aspects of both Crystal Clear and XP, what factors did Jens think had the greatest impact on success of the project? "We had one of the best teams I've had the pleasure to work with so far. The combination of a flexible process and early and fast results saved our life several times. The architecture proved to be stable over all of these changes, and the technical platform allowed us to implement the architecture without worrying too much about the technical details."

And what values and principles influenced the team? "The most important principle was always to leave the decision to the team," Jens replied. "The process was designed by the team, and we didn't adopt any practice that the team didn't find useful. I only watched that practices the team found useful were not abandoned just because of laziness. If someone stopped using a certain practice, he or she had to justify it in a team meeting. The second vital practice were the time-boxes. The team found increments of three to four months most useful, though I would have preferred shorter increments. Anyhow, these short deliveries gave us a strong position with management. They prevented us from going off on tangents, especially in the beginning, when there were ideas that would have been extremely expensive to implement. This literally saved millions."

This insurance project provides insights into exploratory projects. First, the business vision remained stable throughout the project, while everything else—specific features, architecture, and the product scope—changed. In these types of situations, teams need to be measured not on their conformance to plan, but on their ability to adapt to changing conditions in order to fulfill the product vision.

A second lesson from the Generali project is that methodology needs to adapt over time, while the team's ecosystem—its values, interactions, and conversations—has a tendency to stabilize. Third, Jens's experience on this project shows clearly that documentation and understanding are not the same thing and, furthermore, that conversation, not documentation, is key to understanding in a complex knowledge situation. A fourth lesson—which rigorous methodologies often try to ignore—is that stuff happens. Who, for example, would ever have anticipated mother-in-law features? More important, when stuff does happen, how do we respond? Should the mother-in-law factor—now clearly identified—be calculated into future

projects? In today's turbulent world, any methodology that ignores the "stuff happens" factor is doomed to failure.

Agile Is Not Ad Hoc

One of the standard criticisms of ASDEs—the same criticism leveled against RAD approaches in the early to mid-1990s—is that they are just excuses for ad hoc, undisciplined coding. But refactoring, design patterns, customer focus groups, test-first development, and pair programming are not the tools of cowboy or cowgirl coders. These are the tools of developers who are exploring new ways of meeting the difficult goals of rapid product delivery, low defect levels, and flexibility. For example, writing about quality, Kent says, "The only possible values are 'excellent' and 'insanely excellent,' depending on whether lives are at stake or not" ([Beck 2000](#)).

Part of the problem in our industry is the worn-out mantra, "If you do it differently than I do, you're not quality conscious." I've been around this industry more years than I like to admit to in public and worked with different types of software developers—from software vendors to internal IT shops and many of the leaders of the Agile community. A sense of what drives these individuals can be garnered from the interviews in this book, but I can summarize their passion in two phrases—customer value and high quality.

Technical excellence is critical to high-quality software. We can debate whether or not the various approaches achieve those goals or if they achieve them in the most expeditious manner or under what set of conditions one or another approach is preferable. "You're approach doesn't create high-quality code" is a reasonable complaint. "You don't believe in high quality" isn't. One of the Agile Manifesto principles— "Continuous attention to technical excellence and good design enhances agility"—speaks directly to the goal of technical excellence and quality. So what are the technical issues that influence the debate between Agile and rigorous approaches?

- Removal of defects
- Focus on code
- Simple design
- Big bang versus increments
- Modeling and abstraction
- Domain recognition
- Documentation versus conversation
- Specialists versus generalists
- Quality versus speed
- Establishment versus anti-establishment
- Values and principles

Although there are overlaps in a few of these issues, each deserves our attention.

Removal of Defects

Delivering low-defect code is critical to both delivery speed and ease of change. A quick example illustrates the point. Two programmers each deliver code modules (several modules each) consisting of 2,500 lines of Java code. Each module is entered on the project schedule as "done." However, one has a defect density of 8 defects per thousand lines of code (KLOC) and the other 27 defects per KLOC. Is the degree of "doneness" the same for these two modules? One software company I worked with estimates each defect takes about one day of testing time and one day for repair. (The numbers sound high, but the company's metrics are reasonably accurate.) In this case, the difference in defect levels translates to *95 additional days of effort* for the second module. Done isn't done.

Testing and defect repair time are geometrically related to defect density. Double the number of defects in a module, and testing time more than doubles. As many developers have discovered, once the defect density rises to a certain point, stabilization of the code becomes nearly impossible—it becomes an endless

cycle of testing, fixing, and then fixing bad fixes. Serial development, in which coding is followed at some later date by testing, creates a myriad of problems.

Software engineering practices emphasize two strategies for defect removal: multi-level testing and code inspections. All ASDEs call for frequent testing in order to deliver working software each iteration. XP's frequency is near instantaneous in its test-first mode of operation. Agile approaches call for constant integration testing and frequent builds. Agile approaches call for acceptance testing with customers, both with test cases (XP) and customer focus groups (ASD). Pair programming can be viewed as the equivalent of instantaneous code inspections. Code inspections are a key part of FDD and are advocated by other Agile approaches.

Agile developers may use less formal testing documentation and they may depend more heavily on the development team for testing than on a separate testing group, but they are no less dedicated to removing defects than any rigorous development group.

Focus on Code

Traditional rigorous methodologies, and software engineering in general, focus on architecture, requirements, and design; coding and testing are considered low-value "construction" activities. In these methodologies, the high-leverage activities are the up-front activities. Agilists reverse this emphasis. This is a real difference, and one that individuals and companies will have to address when they consider ASDEs.

The equivalent of a civil engineering blueprint is a UML diagram, say those vying to make software engineering the equivalent of civil, electrical, or mechanical engineering. Another equivalent would be an electrical engineering circuit diagram. I contend that these analogies are wrong. The equivalent of circuit diagrams or blueprints is code, not UML diagrams.[1] (Note: In the case of executable UML, the model and the blueprint are equivalent forms.)

[1] Jens Coldewey comments: "The idea of diagrams and blueprints is the idea of plans with growing detail that is deeply rooted in our Western culture. Agile development builds on what Christopher Alexander calls "piecemeal growth," you call "emergent behavior," and I'd call a bottom-up approach. Top-down plans are not of any use for a bottom-up approach. Discussing whether circuit diagrams are more equivalent to code than to UML may be the wrong front line. Taken to their extreme, UML diagrams are nearly code (although I don't think it's of any value to express something in complex boxes and unintuitive symbols rather than in an expressive programming language, such as Smalltalk or Java).

"Yikes," critics will say, "nobody but programmers can read code." My undergraduate degree is in electrical engineering. Developing a circuit diagram requires electrical engineering knowledge, and even reading one requires a fair amount of training and experience. An electrical engineer analyzes the problem, roughs out a design strategy, and constructs a circuit diagram. That circuit diagram then gets translated into silicon. Although there may be senior and junior engineers, there aren't electrical analysts, electrical designers, and electrical circuit diagrammers. There are, however, technicians (or software) that "translate" the diagram for manufacturing.

To Agile software developers, code is the equivalent of an electrical engineer's circuit diagram. How many programmers actually use anything other than code to understand a piece of software? A high-level blueprint may help them understand context, but they generally depend on reading the code. For detail-level work, programmers don't trust documentation to be accurate, they trust code. By devaluating coding, software engineering has fostered the creation of architect, analyst, and design roles in which the qualifications for the job do not include the ability to code or even to understand code. This is the equivalent of an electrical engineer not knowing how to read a circuit diagram!

The problem with the emphasis on blueprints and nonexecutable models has been twofold: form rather than substance and the disconnect between models and code. I've been in companies in which the methodology police run rampant. They don't understand the content of a diagram, but they worry if the

little arrow thingies are inconsistent. "I've talked to many users of modeling tools," says Jon Kern of TogetherSoft. "There is often a huge disconnect between models and code. Modelers, specifically those in organizations in which model builders have little coding experience or skill, build illegal models—things that can't be converted into code." So, if analysts and designers can't code, they need intermediate documentation that they *can* read—software engineering specialization run amuck.[2]

> [2] Jens again: "I wonder how many projects got into trouble because the 'architects' and 'designers' didn't know how to code. I wonder how many billions of dollars these misconceptions (and arrogance) have cost all over the world by now."

Agilists believe in the restoration of programming—the actual production of working software—to a place of critical importance in the development process.

Simple Design

All the Agile approaches advocate high quality through the use of short iterations, frequent builds, and continuous testing, but XP goes further. XP promotes three practices, rules in one sense, that if followed, generate complex behavior whose goal is technical excellence. These practices are simple design, automated test-first development, and refactoring.

Simple designs require thought. Simple, within the XP context, takes on two characteristics: no anticipation of future requirements and easy to change. First, "no anticipation" means to develop code for the story at hand. Don't think ahead; develop a specific solution, not a generalized one. Second, a good design is an easy-to-change design—one in which the intention of the code is clear, one in which elements are minimized, and one in which there is no duplication.

If we focus on coding the features at hand and not generalizing in anticipation of the future, then once the future arrives in the form of new features or stories, we can't just add them to the code base. First, we examine the code and ask, "If the code looked like that instead of this, could we implement the new feature more easily?" To keep the cost of change low, we periodically redesign, or refactor.

Kent refers to this entire design process as an organic process, one that is generative. This is a *system of practices*. Pick one practice—say, simple design without refactoring—and the generative nature of the system disappears. Pick all three—automated testing, simple design, and refactoring—and emergent results, those larger than the sum of the individual practices, occur. Simple designs reduce testing work. Constant testing reduces delivery time. Interleaving testing and coding gives developers, and testers, a better understanding of the code. Having an automated test suite enables developers to refactor with less apprehension about "breaking" the system. Refactoring allows the design to emerge from the code, not be dictated by metamodels. Refactoring maintains simple designs yet enables those simple designs to evolve as functionality is added.

Does this mean no anticipatory design, ever? No, but it does mean balancing anticipatory design and refactoring. Traditional approaches place little, if any, emphasis on refactoring. Project teams may spend months and months doing anticipatory design and none doing refactoring—because, of course, they surely got it right the first time. Maybe the database and user interface designs require a slightly different balance of anticipation and adaptation. By using these three practices, though, we can learn the right balance. Are we redoing a lot of up-front design work? If so, maybe we spent too much time on it. Are we refactoring a lot of code? Maybe a little more design would have been advantageous.

Refactoring may be the single most important technical factor in achieving agility. Mechanical devices degrade with time, requiring more and more effort to maintain. Software degrades with change, so without refactoring, software atrophies. As software atrophies, changes become more difficult. As change becomes more difficult and time consuming, we are less Agile, less responsive. Our ability to move from code refactoring to data refactoring to architecture refactoring may ultimately determine our ability to respond to customer demands in a turbulent environment. At the level of technical excellence, the emphasis on refactoring may be XP's greatest contribution.[3]

On the downside, refactoring—carried to its illogical extreme—leads to unfettered tinkering. However, coding itself can become unfettered tinkering. Other practices, including close collaboration with peers and short-delivery cycles to customers, balance against this potential problem.

Every technical group I've ever encountered wants to refactor—they just use different words for it. They want to recode some module or restructure a class hierarchy or refine a database schema in order to improve maintainability and enhanceability of the system. The problem in most organizations is that they wait (often because of an organization's short-term focus) until the system becomes archaic, at which point the refactoring job becomes a major rewrite that is expensive and risky. During a recent "methodology" consulting assignment for a software vendor, I made the comment, "No methodology is going to solve the problem of your antiquated code base. A major redesign effort is necessary to bring down the cost of maintenance."

Everyone "refactors"—it's just the increment size and time frames that are different. Agile developers are saying, "Refactor more often and on smaller increments in order to avoid refactoring later on huge increments with lengthy time frames." Furthermore, different components should be refactored on different timescales. Refactoring goes hand in hand with simple design.

Big Bang versus Incremental

Although incremental and iterative development has been advocated for a number of years, there is still a bias toward the big-bang approach in many organizations. For example, even though the SEI is supposedly neutral when it comes to incremental versus waterfall life cycles, I've talked to IT managers who have had difficulty convincing CMM assessors that their incremental approach passed muster.

To some, incremental and iterative development implies suboptimization, and in fact, this may be true in lower-change environments. To these individuals, suboptimization implies low technical quality and costly rework. Their argument boils down to, "IF you could do everything right the first time, up front, THEN it would be cheaper, better, faster." It's the "if" that runs afoul of reality. The upside of increments is constant feedback and quick wins. The downside is suboptimization and costly rework.

Information engineering (IE), from the 1980s, was clearly the biggest of the big-bang approaches (at least within IT). It foundered on limited feedback and excessively long delivery cycles. Based on the fundamental belief that data was more stable than process, IE proponents slipped into the mistaken conclusion that data was stable—a dangerous extension. IE began with a strategic study of the business. First came an information strategy plan (ISP), which was followed by a series of business area analyses (BAAs). Teams then built databases from the detail plans and finally developed application code. This highly front-end-loaded process could take 18 to 36 months before useful applications would emerge.

The big-bang excesses of the 1980s, particularly those attributed to overzealous IE practitioners, helped spawn the early 1990s interest in RAD. Some RAD projects, in part reacting to IE, garnered quick wins but suffered from longer-term suboptimization. RAD methods and client/server technology blossomed at about the same time and gave operating departments the technology and a fast-track approach to bypass IT departments. Many applications went from an overabundance of front-end planning to severe ad hoc development with no front-end planning.

To the inevitable question of how to achieve quick wins and reasonable optimization, the Agile response is: skeletal modeling up front (FDD, ASD, Crystal, LD) combined with refactoring (XP). On a scale of zero to ten, with zero representing the extreme of no anticipatory architecture or modeling at all and ten representing the extreme of IE, I think most current Agilists would place themselves around two to four on the scale—just enough, but not too much.

In the final analysis, there are two questions. First, how big is the bang, or, how small is the increment? In deciding the increment size, how do we avoid the problems of poor feedback or poor optimization? Agilists think that it is easier to solve the suboptimization problem—that the path to technical excellence and customer satisfaction lies in short increments, refactoring, and "just enough" skeletal modeling.

Modeling and Abstraction

The concept of "levels of abstraction" has been part of software engineering for many years; it is part of all engineering work. Abstractions, high-level views of a problem space that hide certain details, might be documented in diagramming models with successive refinements adding details to the model. Similarly, an outline is an abstraction of a book, and, through successive addition of detail, the abstract outline becomes a concrete book. The ability to "abstract" away from the details in order to create a good structure has been viewed as a path to technical excellence.

I was having dinner with a noted programming guru not long ago. He relayed a story about the most incredible programmer he had ever known. This programmer could sit down at a terminal and create (in a couple of weeks) a well-designed, 100,000-line program that contained virtually no bugs. He created the program—abstract to concrete—as fast as he could type. Would this programmer benefit from turning his abstract thinking into a diagram?

This illustrates the difference between abstraction, which is a recognizable talent of great developers, and diagrammatic models, which record the result of abstract thought. I like Scott Ambler's comment: "There are two fundamental reasons why you create models. Either you *model to understand* an issue, or you *model to communicate* what your team is doing." If we want to do abstract thinking, either as an individual or in a group, jotting rough diagrams on a flip chart may enhance the process. Still, we shouldn't conclude that lack of formal modeling equates to a lack of abstract thinking.

Architecture is another form of abstraction. Trying to define what "architecture" means is far beyond the scope of this book, but I would at least like to place architecture into two broad categories—environmental and system. Environmental architecture involves decisions about the environment in which the project team must operate—from database and language to the middleware and Web server. Many of the decisions on environmental architecture are outside the scope of individual project teams. Existing systems, overall product architectures, or cross-application and intercompany integration plans may constrain individual project teams.

This leaves system architecture decisions—data schemas, program modules, class hierarchies, method placements, stored procedure usage, and other decisions—to the development team. No doubt environmental architectures change, but at different time intervals than system architecture.

FDD, for example, has five process steps, the first of which is "Develop an overall model." Scrum defines one of the inputs to a Sprint (a development iteration) as systems architecture and plan. ASD specifies a zero iteration (a matter of weeks) in which an initial architecture is developed. In LD, Bob Charette advocates a front-end architectural component. ASDEs advocate "light" architecture, with the degree of lightness varying with project size and the degree of uncertainty about the features.

There are three key factors in doing "Agile" architecture or modeling. First is recognizing and constantly reminding ourselves that plans are just plans. Plans change and models change—constantly throughout a project. Second is to have as a constant mantra the need to keep things simple and understand that refactoring will help recovery from oversights. Third is that the best way to manage up-front architecture work is to time-box the effort—short time-boxes. I place architecture in iteration zero and define it as an iteration in which nothing useful is produced for the customer.

In today's fast-paced world, we might justify spending a couple of weeks (out of a six- to nine-month project) on working out a class structure model or a data model, but not much more. Guidelines are just that, but my experience has been that an initial time-box for iteration zero of one to two weeks per six

months of project is a reasonable starting place. In the FDD chapter of *Java Modeling in Color with UML* (Coad et al. 1999), Jeff De Luca recommends spending ten percent on up-front model development (about 2.5 weeks on a six-month project) and five percent on ongoing model development (in reality the ongoing modeling effort should decrease in the latter iterations). In *Surviving Object-Oriented Projects,* Alistair discusses his guidelines for modeling: "First, create a high-level model of your business. This should take no longer than two to four weeks.... [Then] create models of the business with not more than a dozen key classes and several dozen supporting classes" (Cockburn 1998).

Scott Ambler has been working on an Agile approach to modeling.[4] Agile Modeling (AM) practices include:

[4] For an in-depth coverage of "light" or Agile modeling, see Scott's *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process* (Ambler 2002) and his Web site, www.agilemodeling.com.

- Creating several models in parallel, applying the right artifact(s) for the situation
- Modeling in small increments
- Using the simplest tools
- Active stakeholder participation

Two of the 17 Agile Manifesto signatories—Jon Kern from TogetherSoft and Steve Mellor from Project Technology—advocate modeling, when in fact the models are easily translatable into code such that the "executable" models can be considered "working software."

Models as a communications or thinking mechanism serve a useful purpose. However, just as when coding and testing become un coupled and linearly related (coding is done by one group, testing by another, with a time delay between), modeling (as a component of design) and coding can become un coupled and linearly related. When modelers know nothing about coding, there is a problem. When programmers don't test, there is a problem. There needs to be an integration and a balance. Don't model very long before coding and don't code very long before testing.

Agile approaches are not anti-architecture or anti-modeling. A few weeks of up-front architectural planning, a few days each iterative cycle devoted to architectural planning, or a few hours sketching out blueprints can be effective in improving the goal of technical excellence of our software products.

Domain Recognition

One of the major difficulties in understanding various development perspectives is lack of problem domain recognition. For example, an email forum participant commented on XP, "I'm starting on a two million line-of-code system—XP wouldn't work." Well, one of the things that attracted me to XP was that Kent and others defined a domain of small, colocated teams and an involved user. There are ongoing efforts to extend XP, but the bulk of practical experience has been with smaller teams of ten or so people. Using Capers Jones's numbers, a two million LOC program would, on average, require about 110 people and, best case, around 70 people (Jones 2000). I doubt that any ardent proponent of XP would tackle a 70- to 100-person project without extending or tailoring the basic XP practices. The comment about XP indicated a clear lack of domain recognition. Of course the other view is that using an Agile approach might reduce "bloat" and enable the project to be done with far fewer people.

In his Crystal Methods series, Alistair is clear about the differences between a small team (Crystal Clear) and an 80-person project (Crystal Red). He recommends additional ceremony, documentation, practices, and roles for larger projects, but always keeping in mind the primacy of human interaction. Criticality also helps define a domain. The software needed to run medical CAT scan equipment and the software that runs the latest video game fall into very different domains. Similarly, high-risk, high-change projects define a different domain from those in which requirements are anticipated to remain reasonably stable.

The failure to recognize domains, and the applicability of a particular set of technical practices for particular domains, will ensure that many debates over traditional versus Agile practices will stay firmly focused on the wrong issues.

Documentation versus Conversation

"Zero-based documentation" is a term I remember Tom DeMarco writing or talking about, but I couldn't come up with the reference, so I queried Tom. "I do remember throwing the term around, but I don't think I ever wrote on the subject. The idea seems obvious (by extension from zero-based budgeting): It means that each document you impose on a project has to be justified as meaningful for that project. The opposite is today's norm: The set of documents required is the complete set that we required from past projects—relevant or not—plus whatever special stuff is needed for this new project."[5]

[5] Personal correspondence.

This is the state of much documentation today—documentation for documentation's sake. What's more, it is often justified by the sentiment, "Well, you couldn't possibly deliver technically competent software without these 47 document artifacts." Documentation doesn't, in and of itself, convey either understanding or a quality design, requirement, or the like. For example, Chapter 20 on FDD relates a case story in which the pre-FDD team produced 3,500 pages of use cases, or as the FDD project manager quipped, "useless cases." Should this be used as an indictment of either use cases or documentation? Of course not, but then minimal documentation shouldn't be written off either. It's the quality of the results that counts.

It's an interesting phenomenon: Technical staffs, by and large, hate to write documentation, and managers, although they talk about the need for documentation, hate paying for it. Why, then, has there been so much discussion about documentation? I logged in one Monday morning and had 60 emails from one discussion group—all about documentation, or the lack thereof, in XP. As Tom says in the first paragraph of this section, let's not do documentation that has little payback. Let's justify every piece of documentation.

If documentation were really so important, managers would be willing to schedule appropriate time into projects to produce documentation, and they would be willing to fund technical writer positions to assist in developing documentation. Neither of these happens in any reasonable proportion to all the "talking" about the importance of documentation done by managers.

Specialists versus Generalists

In *The Machine that Changed the World: The Story of Lean Production,* the authors discuss the issue of generalists and specialists within the automotive industry.

> As time went on and engineering branched into more and more subspecialities, these engineering professionals found they had more and more to say to their subspecialists and less and less to say to engineers with other expertise. As cars and trucks became ever more complicated, this minute division of labor within engineering would result in massive dysfunctions *[emphasis added].*

> Lean production calls for learning far more professional skills and applying these creatively in a team setting rather than in a rigid hierarchy (Womack et al. 1990).

A tremendous impetus to software engineering comes from the defense community, which engages in enormous projects. Large projects tend to foster specialization—process specialists, measurement specialists, database design specialists, user interface designers, algorithm analysts, quality assurance specialists. The bigger the project, the more specialists, the more specialists the less interaction, the less interaction the more documentation to counteract the lack of interaction—and, the lower and lower ratio of programmers to the total project staff. Programming work dwindles to less than ten percent of these large projects, and technical excellence becomes associated with technical specialization.

In contrast, the heart of "lean manufacturing" is the dynamic work team. What fast-moving, complex knowledge integration–oriented project teams need is constant interaction and conversation—"team" knowledge sharing and problem solving. The delivery team doesn't need a manual full of database design procedures to follow, it needs a team member who is knowledgeable about database design. The delivery team doesn't need a project office reviewing and approving its plans, it needs a knowledgeable project manager. Product excellence requires an integration of knowledge, often better accomplished by generalists than by specialists. Specialists are necessary, but as an integrated part of the development team, even if they are part time. Technical excellence requires people working closely together, not groups lobbing documents and sign-off forms back and forth.

The purpose of this section is not so much to debate the issues surrounding generalists and specialists as to point out that at least a part of the debate over technical excellence arises from this topic. Typically speaking, Agilists are generalists. Rigorous software engineers tend to advocate specialization. Although Agile developers realize that additional specialties and roles may be required, particularly as project sizes increase, Agile projects will tend to have fewer roles and specialties than rigorous projects.

Quality versus Speed

Herein lies the technologist's biggest fear and greatest complaint: "We never have time to do it right." This mantra is frequently repeated by rote as a spell against the evils perpetuated by anyone who isn't a technologist—particularly vilified are management and marketing. On the other hand, technologists use it on each other all the time—mostly to ward off new practices that they don't want to do. Three things strike me about the quality versus speed argument:

1. Most people talk in broad generalities about the need for quality, but few project teams spend any time defining what they actually *mean* by quality.
2. Speed and quality (the defect aspect of quality at least) are related, but not in ways many people assume.
3. Following in the footsteps of lean manufacturing, Agilists' goals are to increase speed *and* improve quality.

Quality may be important, but it is far from simple. Is quality synonymous with customer value or just defect levels? What does "good enough" software imply? Does all software need to be as defect free as the Space Shuttle software? As Pete McBreen (2001) reports, the Space Shuttle software has order-of-magnitude fewer defects than most commercial software, but its cost is also in orbit. For example, maintaining Microsoft Windows at a level of defects equivalent to the Space Shuttle software might cost upwards of $5 billion per year.[6] Are we willing to pay $5,000 to $10,000 for a copy of virtually bug-free Windows?

[6] Actually, given that Windows has upwards of 50 million lines of code and the Shuttle software has only 500,000 lines of code, I doubt that Windows could attain the same level of reliability for any amount of money.

While there is certainly a relationship between speed and defects, it is not an inverse linear one. I've been involved with some very fast, very low-defect projects—and some very long, very high-defect ones. In the end, the most important thing may not be speed, or quality, or low cost, or maintainability, or whatever—the most important thing is what the customer wants. If the customer says, "I need it by Friday," and we say, "OK, you can have it by Friday, but you'll only get a few screens, limited testing, and only 3 of the 24 business rules," and the customer says, "OK"—then we've been successful if we deliver that very limited function, somewhat buggy, working piece of software on Friday. Delivering buggy code may not be very satisfying to the developers, but it could be acceptable to the customer. We may laugh, but perhaps the customer is someone in marketing who needs to show a major prospect something tangible, whether it works well or not.

There is a story about a CEO who called down to the PC help desk for a loaner laptop. The technician had one, but it didn't work. "Fine, just bring it up," said the CEO. The technician scratched his head and took it to the executive. Two days later, following the meeting, the CEO's assistant returned the laptop. Curious, the technician asked, "What good was a broken laptop?" "It was for show," the assistant replied. "The boss just wanted to appear 'with it'; he really didn't need it to run." Projecting our own ideas about quality—"a broken laptop is useless"—onto our customers can be a dangerous proposition.

This should not be interpreted as a pitch for "anything goes" but as a reminder that quality can be a tricky thing to nail down. Developers, for example, could turn out a piece of software in which they have fixed all the known bugs and performed reasonable tests on the hardware and software configurations available to them. How long can they afford to look for unfound bugs? There's always another one somewhere. How many hardware configurations can they afford to buy and test? At what point have they fulfilled their role as software craftsmen and professionals?

Finally, the traditional wisdom is that if we reduce defects to a minimum level, other characteristics— speed, customer satisfaction, cost—will all improve. In the early 1990s, Capers Jones reported that good things happen at cumulative defect removal rates of 95 percent and higher. Agile developers believe in the advantages of low-defect software also—witness the intensive testing and inspection processes mentioned earlier. Agile developers just do it with smaller increments.

Establishment versus Anti-establishment

Being newer, or at least more recently recognized, the Agile community has a definite anti-establishment feel to it, and therefore part of the "technical" debate doesn't have to do with technical issues at all.

Reading through SEI's CMM material or the Software Engineering Body of Knowledge (SEBOK) manuals, one is struck by how many of the skills are related to process, not to delivering software. Furthermore, the SEBOK is organized in classic waterfall fashion—requirements, design, construction, testing. The layout of the document, ostensibly about skill, contributes to a particular vision of process. Furthermore, the SEBOK reads like an academic journal—formal and littered with hundreds of references. The newest version of the CMM, the *CMMI for Systems Engineering/Software Engineering Version 1.02,* is a mere 606 pages of federal Department of Defense–speak. To those in the Agile community, these "heavy" documents epitomize establishment bureaucracy. It may be as unfair to label the CMM bureaucratic and document heavy as it is to label Agile development "ad hoc," but the establishment versus anti-establishment biases will continue to influence the debate. Even in these labels are grains of real issues.

Agilists wish to avoid projects like the following (Goranson 1999). In 1985, U.S. intelligence agencies discovered a new Soviet air-to-air missile that was superior to any in the U.S. arsenal and presented a severe threat to U.S. planes in air-to-air combat situations. The Israelis, with primarily U.S.-supplied aircraft, responded to the threat with a new missile design and deployment process that took 6 years. The Soviet development effort was estimated to have taken 5 years. The U.S. Department of Defense "plan" is for a 17-year development cycle, which, given the nature of large projects, could stretch to 24 years!

Lean manufacturing and lean thinking admonish us to think about those activities that deliver value to the customer and those that don't. As I look at the CMM and SEBOK publications and remember that James Womack indicates that 80 to 90 percent of manufacturing steps are a waste of time from the customer's perspective,[7] my reaction is that many of the CMM and SEBOK activities just don't add any customer value. If we divide activities into those that make direct contributions to customer value and those that don't, it seems to me that the institutions many people look to for guidance are focused on internal processes more than external results, on the illusion of control more than reality, on non-value-adding practices more than value-adding ones. Furthermore, when we let these institutions define "technical excellence" as conformance to *their* practices and standards, we fight a losing battle.

---

[7] See Chapter 13's section on simplicity as minimalism.

Technical excellence should be measured by attributes of the product—does it work, does it break, does it process the volumes required, does it integrate with other systems as specified, does it perform. The next question is, "What are the skills we need to deliver that product?" Everything else detracts from agility.

So part of the debate about Agile versus rigorous is about old versus new, or traditional versus nontraditional, or about comments like "This isn't even new anyway"—essentially, establishment versus anti-establishment. This happened during the early years of object development and during the introduction of the PC. It is an inevitable part of integrating the best of the new with the best of the established.

Values and Principles

Some of the most vitriolic attacks on XP have been focused on pair programming. Even though, to me, pair programming seems an extension of, and an improvement on, the collaborative practice of code reviews, others feel that pairing is an outlandish, subversive, and thoroughly dangerous threat to their individuality. Many of the debates about Agile versus rigorous practices have no basis in fact—they are purely emotional and based on one's culture, one's values and beliefs. Now, emotion-based reactions are at least as valid as fact-based ones, but they do tend to create high-volume rhetoric.

The point here, to reiterate one of the major points of this book, is that values and principles are absolutely critical to one's point of view about methodology and software development ecosystems. They also greatly influence "technical" debates.

Reflections

Many issues spark debate about ASDEs. In general, the debates are healthy because they promote deeper understanding. To get the most out of these debates, to extract real differences from hyperbole and misunderstanding, the individual issues need to be teased apart and analyzed. ASDEs utilize traditional technical practices, but they have fostered new ideas, new rearrangements of traditional practices, and a different perspective on how to achieve technical excellence. Separating these issues will help everyone decide what parts are valuable to them and what parts are not.

Some people may be concerned that a "barely sufficient" approach to methodology will result in lower technical quality. However, Agilists' barely sufficient approach is based on the belief that technical excellence depends mainly upon talent and skill, not methodology. Methodology forms a framework within which people can work effectively together, but it cannot, and does not, substitute for talent and skill. If a methodology provides a framework and practices that encourage skill building, then it will be far more effective than one that merely prescribes activities and documents.

Chapter 12. Ward Cunningham

"I could experience Xerox PARC by reading their listings," Ward Cunningham remarked during our lengthy morning conversation. "I could see what they were thinking, and I wanted to emulate that—to make our thinking clear in code."

This focus on the aesthetics of code, the pure joy and satisfaction of writing code such that it is crystal clear what the author intended, was a hallmark of my conversations with each of the "three extremoes"— Kent Beck, Ron Jeffries, and Ward Cunningham. The quality of code inherent in test-first development, the aesthetics of clear expression of the code's intent, the focus on simple solutions to complex problems, and refactoring to reduce the cost of change are concepts that litter their conversations.

Because Ward's name isn't on one of the XP books, many don't realize his contributions to the concepts and practices that eventually became XP. But from the initial fascination with Smalltalk that he and Kent shared, to the development of CRC cards for expressing Smalltalk designs, to the pattern movement, Ward's contributions have been significant. If Ron is the "attack dog" of XP and Kent is the chief evangelist, then Ward is the consummate behind-the-scenes contributor.

As Ward and I talked, we discovered a strange twist of irony: During the time Ward and Kent were planting the early seeds of XP in the Tektronix lab (early 1980s), I was helping the IT group implement one of the rigorous methodologies of the time—Ken Orr's Data Structured Systems Development.

JIM: *When and where did you begin working with Kent?*

WARD: I was working in the research lab at Tektronix (in Portland, Oregon) in the early 1980s when Kent came on board. We were in the systems engineering group that was looking five years out—imagining the future in terms of computers, compilers, and operating systems.

JIM: *What was the work like?*

WARD: We had a great job. We would sit in the cafeteria on Monday mornings drinking coffee and talking about where the industry was going. Then we would start on something we eventually, jokingly, called the Beck-cycle (the short propose-analyze-code cycle). We would pose a problem to ourselves, think about it for a while, and then go downstairs and try to solve the problem by writing some code.

We wouldn't have a clear plan, but before lunch we would know if it was going to work, and in a couple of days we would have something to show. We were probing the boundaries of our own knowledge. This short cycle of problem proposing and then building something to see if we could solve the problem eventually became the "spike" technique in XP.

At one point, Kent's boss said he'd had this month-long assignment that had to be finished by the end of the month. Anyway, Kent hadn't started working on it, so he came in one day and said, "Ward, I can't play today, I've got to go do my real job." And I said, "Kent, why wouldn't I do that with you?" He said, "Really!" We reviewed the job and weren't quite sure why anyone would want the thing, but we said, "Sure, let's make it." So we did it in a day. It was amazing—a month's worth of work in a day. Part of it was that we were developing in Smalltalk, and we were pretty good at cutting corners.

JIM: *Much of your work during this period was in Smalltalk?*

WARD: We started using Smalltalk early on and made a couple of trips to Xerox PARC. One of our goals was to write large programs that could be picked up and read easily. I had visited MIT at some point and was shown this listing for their Design Procedure Language. This listing was three to four inches thick, and it was weird—I could open it up to any page, start reading, and I could understand what was going on—and I didn't even know LISP at the time. This property of start anywhere and convey understanding became part of our goal.

JIM: *What happened after Tektronix?*

WARD: Well, Kent moved on to Apple, and eventually I went to work for Wyatt Software, which did financial software for complex bond trading. I had to make the transition from laboratory work, in which we worked on our own problems, to the business world, where we worked on customer problems.

In this financial work, I knew I had to find some way to control complexity. In scientific computing, complexity came from the laws of nature, whereas in the financial computing, it came from the market— the market created complexity. People were making bonds left and right. They'd make a new bond somewhat like an old one and then try to sell it. So I told my colleagues I was going off to do field research. I do that long before I realized why they didn't pay much attention to research. And the reason for that was in research we could take a month to think about an idea—we savored interesting problems. In product development, you've got 100 uninteresting problems a day, and you just have to get those behind you to think about the more interesting ones.

I would not have made it without my boss. This guy knew that Smalltalkers, actually anybody, would go off on these tool bents and make tools instead of products. He was afraid of that, and the way he dealt with it was to walk into the office three or four times a day and ask what I was working on and why was I working on it. Most of the other people started locking their doors, but I developed a habit of stopping and asking myself, "What am I doing?" and "Why am I doing this?" I knew the right answer was going to help me deliver software for this customer.

His simple questions provided all the discipline I needed learn to prioritize my work, to break out of my Beck-cycle habits—which was a great way to work—and focus on industrial-grade software for external customers. Learning to focus on the customer's problems for major portions of the day was difficult, but this was a good transition from the lab. This was the start of maybe three to four years of process design. I realized that there were enough stupid questions that I faced every day that I couldn't afford to make decisions that just created more decisions later. The questions had to be about my customer, which I couldn't control, and they couldn't be about my computer program, which I could control. I had to grapple with programming problems that I couldn't explain to my customer, but at the end of the day I could say, "This is what I've been working on today," and he would say, "Boy, I'm glad you are working on that because we really need it."

Everything related to computers is related to a bunch of stupid work, like backups. Lots of stuff is like that, but we maintained—over three and one-half years of development of this one program—a focus on the customer to the point that any of our developers would spend at least one-half of their time using creativity on some customer problem, not technical busywork. That was the achievement. As I think of it, we had a very flexible environment, and we flexed it in ways that it didn't create problems for us, it solved problems for us. We focused on making decisions on behalf of our customer and turned those decisions into programs. It was great, but it wasn't easy.

JIM: *It sounds like an externalization of the Beck-cycle that the two of you used for solving research problems.*

WARD: The Beck-cycle was really a series of spikes. You don't know if you can do something, so you just try doing it to see. At the end, you'll know if you can do it or not. If you know you can't do it, you'd better not estimate it at two weeks! Coming out of the lab, I had to realize I wasn't writing software for myself but for my customer, and my manager at Wyatt really taught me that.

JIM: *Did you stay in contact with Kent during this period?*

WARD: We did. When he was down at Apple, we talked a lot, designed some papers together. But as time went on, we had less common knowledge. We launched the pattern stuff, and we would see each other several times a year related to that. My MO [modus operandi] is to do things that are a stretch for me, reflect on that, and then share that with people who are quick to understand. I don't care for teaching. I can do a course, but by the third time I hate it, because I don't like doing it over.

I've found that to get an insight, to be with a sharp group of people, to pass along one little thing that someone responds to—that's how I get my kicks. And people generally say nice things about me, so I think I have some pretty deep credibility in the industry. Kent and I have shared a lot, and he's a great promoter. Ideas are cheap—insight wouldn't go anywhere without promotion.

JIM: *John Holland, one of the complexity science gurus, talks about building blocks and that we all have building blocks of things—ideas, practices, insights. Sometimes it's not so much that you come up with a new building block as that you figure out a unique new way to organize existing ones—and that's being just as creative as discovering new building blocks. That seems like what XP has done to some extent. Kent says in his book that the building blocks of XP aren't new, but he cast them within a new framework, a new arrangement—a set of values and a specific system of practices.*

WARD: Yes, I think so. I was in college when some teacher—of course they didn't know much about testing back then—was offhandedly mentioning something about testing and that it should be done a certain way, and then he says, "You know, it's actually easier if you do the tests first. Code is really easy to write if you've got a test to test it." I can't even remember what class it was or what professor, it was an offhand remark that wasn't considered very important. But there is was—test first.

JIM: *In Jerry Weinberg's classic,* The Psychology of Computer Programming, *he talks about working together and the concept of egoless programming (Weinberg 1971). I once kidded Jerry that I didn't know any egoless programmers. In the 25th anniversary edition of his book (Weinberg 1998), Jerry comments that his "egoless programming" was probably the most widely quoted, and most misunderstood, thing in the book.*

WARD: When Kent and I were working together, we would reflect on what we really did, and we used those terms. The two of us have tremendous ego. But when we are working with each other, we set that aside—egoless between us, but around us there is tremendous ego.

JIM: *Where did you go after Wyatt?*

WARD: I left in 1991–92 and became an independent consultant. By that time lots of people knew CRC cards, and they were learning patterns. It was pretty clear that I had a reputation, and Smalltalk was still going. Because of my experience with Wyatt, I ended up consulting to East Coast banks and insurance companies. I could take a dysfunctional team and make it functional—partly by reminding people what their responsibilities were and partly by simplifying their goals to the point they could believe they could achieve them. Most people had bookcases full of requirements and just didn't know where to start. "Well, how about we start here?" We just asked the customers if it was a good starting place. "Wow, if we had that by the middle of next summer, it would be just great!"

JIM: *Did you try to write about your experiences?*

WARD: I tried to start a book, but the ideas were just too diffuse at the time. One time on an airplane, I was so upset at how things were going at a particular company, I sat down and wrote out 25 or 30 pattern names of things we just *did* at Wyatt. It was around the notion of problems, that you have to make progress on the problem before you set it down, and when you set it down, you leave it in such a way that when you pick it up tomorrow, you pick up where you left off. How do you get things broken into little bites that are human scaled? I dubbed them episodes. I equated it to climbing a hill; it's getting harder, and you don't know if you are going to make it.

JIM: *Is it worth getting to the top after all?*

WARD: That period is actually the most leveraged development time. Most people don't like it; they try to avoid it. Most methodologies try to keep people from being in that situation where you're not sure what the answer is. But you're just waiting for an idea to come, and then you get one, and you're not sure it will work. So you try out a little code, and it starts to look good. It's not until you tie the ribbon around it that you look back and say, "That was a great idea." That's an episode. I named it after a soap opera episode—you get into it, there is a climax, and then you wind it up. I think that's the heartbeat of development. Some of these patterns were published in *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects* ([Schmidt et al. 2000](#)).

JIM: *Did you have any involvement with the famous C3 project?*

WARD: Not really, except that I knew Ron and Kent. Kent took a demoralized team—most people would have thrown away the team and kept the code. He was smart enough to keep the team and throw away the code. Kent's focus was solving the morale problem. He said, "We only have to do a few things well." He specified the practices, and Ron was the coach who explained them to the team and filled in the little things that came up.

But I wasn't there. In fact, Alistair is the only person I know who even bothered to go there. I tell you, here is a guy who can call himself a methodologist, because he *studied* the method. Alistair, I love you!

JIM: *In fact, I'm having lunch with Alistair later today.*

WARD: Oh, don't tell him I love him. I don't want his head to swell.

JIM: *What do you see as the future of XP?*

WARD: Sometimes when I'm interviewed, people ask, "Do you do XP?" And I reply, "Not always." I like to discover things, and if all I do is XP, then it will be all I know. So I program lots of different ways. But I think it's important that XP be adhered to dogmatically. Many people who think they are doing XP don't have a clue what it is. You have to do it dogmatically, then you get "clued" and are free to explore.

We need to work on the weak link—marketing, the quality of information we get from the customer side so we don't go off on wild goose chases. We need XP-like revisions to communications processes all the way out to the real customer. We need to get the right insight from our real customers to our developers so they can make intuitive decisions. If we don't solve this other side that feeds requirements to development, we will begin to see more failures on XP projects.

Reflections

One of the things about collaboration, and particularly creative collaboration, is that it arises from interactions and long embedded knowledge that suddenly gets connected. We've all had the experience of having a piece of information hanging around in our brains for a long time, and suddenly somebody says something that sparks a connection, and we experience that "Aha!" Sam Bayer and I worked together over several years to perfect a set of rapid development practices. Some of the building blocks clearly came from Sam's experience and background, others from mine. However, the insights we developed, the

melding of building blocks to form a system of practices, the deep discussions about why certain things worked that then led to other ideas—this was clearly an interactive creativity and innovation at work.

A similar interaction obviously went on between Kent and Ward. For example, what possible set of seemingly random activities emerged as pair programming? According to Ward, he and Kent began working together in a traditional mode—static review of each other's work. As their relationship evolved, they began working jointly on projects, relying on instantaneous and dynamic development and feedback. At some point, this way of working together became deeply engrained and acquired the name pair programming. Much the same happened with refactoring. The seeds of refactoring came from their work with Xerox PARC and their obsession with clear, simple code. Their ongoing "rework" of code evolved into the principles and practices of refactoring. The important lesson is that the practices of pair programming and refactoring *emerged* from a collaboration—a collaboration based on discovering building blocks; placing a high value on clean, understandable code; and valuing each other's experiences. No one could have predicted that pair programming or refactoring, per se, were going to emerge from this collaboration, but we could have predicted that putting these two people together in an environment conducive to innovation would produce a valuable result.

Chapter 13. Do the Simplest Thing Possible

When the business landscape was simple, companies could afford to have complex strategies. But now that business is so complex, they need to simplify.

- *Kathleen Eisenhardt and Donald Sull, "Strategy as Simple Rules"*

The Survey Controller Team at Trimble Navigation

"Never do anything that is a waste of time and be prepared to wage long, tedious wars over this principle," declared Michael O'Connor, project manger at Trimble Navigation in Christchurch, New Zealand. A number of principles pepper Michael's rather irreverent approach to software delivery. Being an "old guy" myself, another of Michael's principles was of great comfort: "Hire only good people. And I mean 'good.' Good is not the same as being smart, clever, young, energetic, cheap, obedient, or able to use the latest COM+ thingy. Software development and project management are matters of the human spirit. They require courage, humility, and devotion. Old people are much better at them than the young."

Trimble Navigation uses global positioning system (GPS) technology in a wide variety of products, including equipment for the land survey and construction markets. The core product for this group is a Survey Controller running on a hand-held computer and Trimble's homegrown operating system. The controller is a core piece of several products that are continually evolving. Development, historically in C, has been migrating to C++.

One fascinating aspect of this product group is its approach to teams. "We don't organize a new team around each project," said Michael, "we build up teams and then throw work at them." Wow! What a concept—keeping well-working teams together rather continuously reorganizing. This approach has recently evolved into a group of 40 people who splinter and reform around projects and features on a timescale measured in weeks. The word "team" itself becomes confused in this setting.

The Survey Controller product team doesn't use a particular ASDE, and there was no conscious decision to pick one. "We started with 'code-and-fix' with a lot of input from users and slowly adapted according to various ideas of good engineering, but with a fairly severe eye to 'what actually works,'" said Michael. "The process at the moment is basically to invent a process for the current situation and apply it." Trimble's process includes feature-driven, time-boxed delivery in which the dominant customer value is delivery schedule. Features are traded off as needed, and costs are of less importance.

Michael defines the team's process as "extremely" lightweight. Lightweight means no written design documentation. "We tend to have written requirements and specifications, but we are currently looking to reduce the level of written material at this level, too. Lightweight means accepting that we cannot identify and control every little task. Lightweight means that we don't submit much in the way of status reports. We tend to just try things rather than analyze them to death. Lightweight means we spend very little time on estimates."

Although the team formalized the idea of a "change control committee" to negotiate the feature set, it found the need has declined to the point where the committee is a mere formality—feature selection just happens naturally, and the "process" shrank over time. Asked about process improvement, Michael remarked, "I expect that there are a zillion little methods from other places that we are using unconsciously. Mostly though, we find we have to expend time to fend off other people's ideas about good practices." The team's improvement process is one of continuous small changes, mostly simplifications. For example, programmers are now encouraged to break down planned features into bite-sized pieces of about a week's duration.

One fascinating aspect of Trimble's Survey Controller team is that its members have developed a set of principles and values that frame what they consider important to their software "culture." In addition to the "Never do anything that is a waste of time" principle cited above, these are:

- The orthodox is almost always wrong—and when it's right, it must be refitted for local conditions.
- Think hard about the underlying systems—technical and political. (A devotion to "never waste time" becomes mere belligerence without this step.)
- Think hard about the code. There is nothing other than the code (except, maybe, the test cases). The most important attribute of the code is that it be readable.
- Hire only good people.
- The processes must be ordered around the people, not vice versa.
- There should be somebody who does not care about the product and its design but is able to think strategically and tactically—and he or she should be the boss.
- Plan continuously but don't write plans down. (Actually, we use whiteboards a lot).
- Simple is good. It takes great skill to keep things simple.

Has the team's approach been successful? "Projects have been very successful, in that they are pretty much on time with the required functionality," said Michael. "The product is relatively bug free, and the product owner is very pleased with us."

However, the framework of this team's light, Agile process hasn't transitioned well into other Trimble development groups. "We are too radical for other groups," Michael lamented. Other groups view the Survey Controller team as having a total lack of methods, rather than having a set of light, Agile methods that focus on the essentials. "We are committed to people-centered development (per Tom DeMarco and Tim Lister), continuous improvement of processes, and fighting anything that is orthodox but stupid," he said. "There have been some great punch-ups."

One thing that stands out as I travel around the world talking to companies (I conducted a workshop for Trimble in August 2000) is how many projects take on similar characteristics. Michael and the staff of the Survey Controller group didn't follow one of the "noted" ASDEs—they improvised. But their improvised process is similar to the essence of XP, ASD, or Scrum.

"Simple is good" states one of Trimble's principles. That doesn't mean, however, that the same " simple" is good for every eventuality. For example, the Survey Controller team members instituted milestones, but as their experience grew, they just worked until they got features implemented and the bug count down—making their deliverable schedule but not worrying much about interim milestones. However, they are now planning to reinstitute milestones because they must integrate closely with another department for the next-generation product—milestones become a mechanism for synchronization. At one scale of effort, milestones became a learning tool, then they were discarded as unnecessary. At the next scale of effort, reimplementing them was a " simple" solution to an integration need. The Survey Controller Group seems to add, and subtract, practices and processes as appropriate for variations in its environment.

Musashi

Alistair Cockburn's approach to software development mirrors the practices of Miyamoto Musashi, a Japanese samurai from the early 1600s who was never defeated in a duel. Musashi's sword-fighting techniques focus on doing something in deadly earnest—nothing minutely superfluous allowed. While other samurai teachers might focus on style, grace, parries, or thrusts, Musashi's intensity was trained on winning.

As we move from the old economy to the Information Age economy, and as we try to manage the projects that drive this transition, it has never been more important to focus on critical issues. "Distinguish the superficial from the substantial," writes Musashi. "The field of martial arts is particularly rife with

flamboyant showmanship, with commercial popularization and profiteering" (Musashi 1993). It sounds as much twenty-first century as seventeenth century.

Musashi writes about weapons (particularly short swords versus long swords), guard positions, and five formal techniques. Regardless of sword length (i.e., tool), position, or technique, however, he never loses track of his single-minded focus on winning. Musashi used whatever tool was readily available. He used whatever technique seemed appropriate to the situation. He never said, "The long sword is the best, therefore I'll only use the long sword." In close quarters, sometimes short-sword strokes are more effective than long-sword ones.

The analogy to draw from Musashi is to focus on results rather than on process. Results may be expressed in a variety of ways—speed, scope, flexibility, low cost—depending upon the situation, but results remain the focal point. Customers care about results: working software that delivers business value. We can argue about rigorous versus Agile practices, about Java versus C++, about pair programming versus code reviews, but in the final analysis, success or failure depends on whether we deliver results.

As the quote at the beginning of this chapter admonishes us, we need to focus on simplicity. XP exhorts us to assume simplicity. LD principles state that minimalism is essential and an 80 percent solution today is better than a 100 percent solution tomorrow. ASD's guideline for rigor is "a little bit less than just enough." Crystal urges us to consider a barely sufficient process. But what does " simple" really mean? Is it really so simple to be simple? As we shall see, the real advantage to the right kind of simplicity is its ability to generate complexity. Musashi wasn't a simple sword fighter; his tools, techniques, and forms were complex and varied. But his simple rules focused that complexity.

The Three Faces of Simplicity

Simplicity should be simple—shouldn't it? Unfortunately, simplicity isn't always simple, and sometimes it's downright difficult. There are at least three dimensions, or faces, to simplicity in software development: minimalism, simple design, and generative rules. The last of these—the need for simple, generative rules to create complex behavior—is often overlooked and underappreciated by countless numbers of people within and outside the Agile community. So simplicity extends far beyond the oft-maligned practice of reducing documentation.

Simplicity as Minimalism

One of the Agile Manifesto principles is directed at simplicity as minimalism: "Simplicity—the art of maximizing the amount of work *not* done—is essential." Simplicity affects our ability to change, to be adaptable or responsive. Responsiveness relates to inertia. Heavy things have more inertia and are therefore harder to change.

James Womack, the coauthor of *Lean Thinking* (Womack and Jones 1996), gives a presentation in which he points out that 80 to 90 percent of all the steps in manufacturing are a waste of time from the customer's perspective (no value added), and 99.9 percent of the throughput time is wasted.[1] He gives examples in the automotive industry in which weeks of time in parts delivery and assembly include only 15 minutes of valued-added time. What might the same figures be for software delivery in large organizations? *Maximizing the amount of work not done* doesn't mean sacrificing quality, it means concentrating on those activities that deliver customer value and minimizing those that do not.

[1] Womack's presentation containing these numbers can be found at www.lean.org.

Minimalism means to do less: do fewer activities, produce fewer documents, reduce management reports. Alistair discusses methodology embellishments, those activities that creep into work processes because someone says, "This activity should help us deliver software of higher quality." Note the operative word is "should," because these are often activities that individuals read about in a book, hear about in a presentation, or learn in a workshop—not things they have actual experience *doing*. Embellishments are

insidious. Processes and documents get instituted in big organizations because some random error occurred, and even though it will probably never happen again, someone feels they have to do "something" to keep it from ever happening again. Thus procedures and processes get added regularly, but are never removed. Many, if not most, of these problems would be effectively resolved by a "fix on occurrence" strategy rather than adding overhead to every project. Once we begin viewing the software development process through our Agile glasses, it's amazing how much work can simply be cut out.

Ron Jeffries poses this question to people, "If you eliminated one-half of your processes, would you miss them?" As an experiment, try eliminating activities and documentation to the point at which things begin to unravel, then stop reducing. Or on your next project, cut the number of documents in half and reduce each document's size by half and see what happens. If the project doesn't appear to have any adverse reaction to less documentation, on the next project cut the number and size in half again. Continue until something negative happens, then add a little back. As a final experiment, a radical one, cut out all the documentation and then slowly add documents as an absolute need arises. By the way, don't try any of these experiments without working on improving conversations and collaboration at the same time.

Simplicity as Good Design

"Most problems are simpler than we can see," explained Bob Martin at one of his company's XP Immersion Workshops. Bob and James Grenning worked through a programming problem, after having the workshop participants develop a rough class model. The two then set the model aside and worked through code development—using pair programming and test-first development. The solution that emerged was much simpler than the class model indicated, and what's more, it was difficult to see how anyone starting from the class model would have arrived at such a simple, elegant solution. The moral to the story is that our modeling heritage, from data and process models to "UML" models, can lead us into elaborate and unnecessarily complex solutions. (Now, I must also concede that I've developed models that have evolved from complex to simple over time.)

Whether we approach working software from a modeling or a programming perspective, complex designs are easy; it's the simple designs that are difficult. Show me a clean, simple design, and I know that a lot of thought went into it. Several years ago I participated in a data architecture project for a worldwide consumer goods company. A previous project had produced architecture diagrams that unfortunately looked much like the proverbial rat's nest. Although complete, the model was not usable. Over a several-month project, the new team accumulated user stories related to the data and constructed, deconstructed, and reconstructed the data architecture. Slowly but surely, the architecture evolved into one that was readable, understandable, and simple. In the end, the team began using the words "aesthetic" and "streamlined" to describe it. Furthermore, as the model became simple and straightforward, the more confident the team became that it was correct.

Kent Beck's criteria for a simple design are: runs all tests, no duplication, expresses every idea, and minimizes elements (classes, methods). In this scenario, simple design (do things better) may take considerable work—simple design often takes more time, not less. In the data model example, we produced a model, but it took several months to turn "a" model into "a simple" model. Code designs evolve by an ongoing process of coding, testing, and refactoring (periodic redesign). Simple, aesthetic designs emerge from concentrated effort, including revisions. Simple designs don't mean less work, they mean more thinking.

If simple designs take additional work, why take the time? Simple—they are easier to develop, test, and enhance. A prime rationale for Agile practices is flexibility and adaptability to change, not necessarily pure delivery speed. Failure to work hard on simplifying and thereby improving code yields difficult-to-change code, which then slows subsequent iterations or releases.

XP proponents have their own simplicity acronym—YAGNI, for "You aren't going to need it." YAGNI means to include only functionality that has been specified; don't worry about the future (in terms of abstracting beyond what is currently needed). Sometimes people misinterpret this catch phrase, confusing the *do less* with the *do better* aspects of simplicity. YAGNI does not mean do less at the expense of doing

better. In fact, doing better—continuous refactoring and test-first development—may actually involve doing more. Interpreting *simple* rules (such as YAGNI) as *simplistic* rules can be dangerous.

"Technical skill is mastery of complexity while creativity is mastery of simplicity," says British mathematician E.C. Zeeman (Swainson 2000). Keep things simple, but remember that mastering simplicity isn't itself simple.

Simplicity as Generative Rules

Here we have the most misunderstood concept of Agile Software Development—generative rules. The concept of generative rules comes from complex adaptive systems theory and is new enough that many people just gloss over the concepts. Complex systems—and human organizations are clearly complex—consist of individuals (or groups of individuals) that interact with each other to create emergent results. So what are emergent results? "Emergence is a property of complex adaptive systems that creates some greater property of the whole (system behavior) from the interactions of the parts (self-organizing agent behavior). Emergence is similar to innovation, problem solving, and creativity; we have a difficult time understanding how they happen, but the results are clear" (Highsmith 2000).

Creativity and innovation are not accidental. Emergent properties are not accidental. They just don't conform to linear, cause-and-effect rules. Emergent properties come from creating the right environment and applying simple, generative rules. Generative rules give rise to what some have called "swarm intelligence." Jim Donehey, former CIO of Capital One, used four rules to help ensure everyone in his organization was working toward the same shared goals: Always align IT activities with the business, use good economic judgment, be flexible, and have empathy for others in the organization (Bonabeau and Meyer 2001). Do these four rules constitute everything that Donehey's department needed to do? Of course not, but would a 400-page activity description get the job done? What Donehey wanted was bounded innovation, a department that thought for itself in a very volatile business environment but also understood boundaries.

Probably the best, most concise statement of the power of simple rules comes from Dee Hock:[2]

[2] Dee Hock is the former CEO of Visa International, who presided over Visa's growth to $7.2 billion transactions and $650 billion annually.

Simple, clear purpose and principles give rise to complex, intelligent behavior.

Complex rules and regulations give rise to simple, stupid behavior (Hock 1994).

The biological world teems with examples of simple rules generating complex behaviors—the behaviors of ants and termites are often cited. African termites build massive nests that are extremely complex, but the roles and rules for individual insects are extremely simple—simple rules, complex behavior.[3]

[3] For comprehensive, scientific coverage of emergence, see Holland (1998).

Herein lies the force behind Agilists' drive to keep their processes simple. When the problems are exploratory, every problem becomes unique. When every problem is unique, only complex, intelligent behavior can deliver effective solutions to those problems. When we want complex, intelligent behavior, complex rules and regulations are not the answer.

Surely, the 12 practices of XP, the 9 principles of DSDM, or the 12 principles of LD do not constitute everything a development group needs to do. However, they do constitute a minimum set of rules that bind innovative and productive work. They create an environment of innovation and creativity by specifying a simple set of rules (practices are rules in this case) that generate complex behavior. XP, for example, doesn't prescribe configuration management as a practice; it assumes that when a team needs it, its members will figure out a minimal configuration management practice and use it. Most methodologies

contain inclusive rules—all the things you could possibly do under all situations. They attempt to describe "everything" that any development effort might need in thousands of pages of documentation.

In contrast, ASDEs contain generative rules—a minimum set of things you must do under all situations. By doing these things, teams will "generate" additional practices or rules for special situations. In the inclusive case, we depend on someone else to name in advance the practices and conditions for every situation. This approach obviously breaks down quickly. In the generative case, we depend on individuals and their creativity to find ways to solve problems as they arise. Creativity, not voluminous written rules, is the only way to manage complex problems and diverse situations.

However, we cannot forget that emergent results live at the edge of chaos, at the balance point between structure and flexibility. The *structure,* which protects projects from chaos, rests on skills and professionalism. A generative organization provides teams with a simple set of rules and an environment conducive to constant interaction in order to generate innovative results. Generative rules have three distinct advantages.

- Flexibility: The group can quickly adapt to a changing environment.
- Robustness: Even when one or more individuals fail, the group can still perform its tasks.
- Self-organization: The group needs relatively little supervision or top-down control (Bonabeau and Meyer 2001).

The idea that we don't have to specify everything is a radical one for many people. They don't give others credit for their ability to figure out what they need to get a job done. However, as Eric Bonabeau and Christopher Meyer point out, developing a useful set of rules is not a trivial exercise. The rules can often be counterintuitive, and seemingly minor changes can have unforeseen results. This is one reason proponents of XP may seem to be overly concerned about altering XP's practices. They know from long experience that the 12 practices reinforce each other in direct and subtle ways. Changing one practice, without understanding the interactions and the concepts of swarm intelligence, can cause the XP "system" to react in unexpected ways.

In the interview with Kent in Chapter 4, you may recall his statement that he would like to reduce the number of rules (practices) in XP. For those who are familiar with rigorous, process-centric methodologies, XP and Scrum and ASD seem almost naïve in their simplicity. Doubters want to know about configuration management or database design or user documentation or implementation planning or 100 other tasks that they know from experience need to be in a complete "methodology." There is a difference between core rules that influence behavior and generate creativity and checklist tasks that need to be done but aren't generative in nature.

The 12 practices of XP are simple—right? Why then does the XP forum on Yahoo-Groups have 2,500 members who exchange 2,000 messages a month? Why are there a half-dozen or more XP books if it is so simple? The reason may be that simple, clear purpose and principles give rise to complex, intelligent behavior. A wealth of information flows out of intelligent people trying to apply 12 simple (well, maybe not so simple after all) practices to the complex problem of developing computer software.

Adapting Simple Rules

Teams employing any set of rules require the capability to "self-adapt" those rules to specific situations. However, there are three kinds of adaptation to consider. First, generative rules create complex, unanticipated behaviors that go beyond the rules themselves. So while a configuration management practice isn't explicit within XP, it could be added as the result of complex behavior generated by XP rules. The generative rules ensure the *new* practices will be simple and people oriented; that is, they will conform to the *existing* practices and principles. So one kind of adaptation consists of new practices that arise from the application of the generative rules. Generative rules make Agile development dynamic. The danger with generative rules is their potential for always generating and never subtracting, so it's safer to start projects from the base set and let the team "generate" what they need.

A second type of adaptation substitutes one generative rule for another, or deletes one of them. Agility means being responsive or flexible within a defined context, such as the "context" provided by XP's values and practices. Although the context or generative rules can be changed, the changes must be made very, very carefully so as not to disrupt behaviors in negative ways. The 12 XP practices didn't just pop out; they arose from considerable experimentation. Equal experimentation would be necessary to change them. For example, it might seem reasonable—since pair programming has been compared with very short-cycle code reviews—that code reviews could be substituted for pair programming without harm. While this may be true, there is a significant possibility that this substitution would have a negative impact on group collaboration, simplicity of design, and more.

A third type of adaptation would be to interpret and extend a practice within a new problem domain without significantly altering the underlying principle that the rule was intended to instantiate. For example, if we see XP's planning game and on-site customer practices as creating a customer-developer interface of vision, stories, priorities, conversations about the stories, and acceptance tests, then what would be the extensions for a problem domain of six user departments with a total of 24 distinct business functions and therefore 24 "users"? I can think of several practices that might be appropriate to implement within the context of the defined customer interface "rules" in order to maintain similar dynamics.

If we extract the underlying principle from, say, the on-site customer practice, then it can be applied to other practices for extended problem domains. However, generalizing from practices to principles is also dangerous if we don't completely understand the practice. The on-site customer practice has several components: The on-site customer is physically there, interacts constantly with the development team, has decision-making authority, has domain knowledge, and understands the customer-developer partnership role. Some of these components are fairly evident and some are more subtle—particularly as the practices interact. Substitutions need to honor the subtleties.

So in adapting Agile systems of practices, it is important to keep the type of adaptation in mind (although there may be much debate over which category a specific change falls into). Are we changing a fundamental generative rule, using a practice that has been "generated" from the rules, or extending a practice within the context of an existing generative rule? We need to be very careful about adapting generative rules. Some people want to change them before they understand them. Others want to change them without understanding the nature of generative rules in complex systems.

A Final Note on Simplicity

I want to end this chapter as it began, with Trimble Navigation, and a further caution about misinterpreting simplicity. The 40-person Survey Controller team uses a methodology that is as light as Alistair's Crystal Clear, which is designated for teams of 6 or fewer people. Does this mean that other 40-person teams could work with this ultra-light methodology? Only if the other conditions were the same as at Trimble. The Survey Controller group is very experienced, many people have worked on the product for years, they work in close proximity, they have a good product definition group, and they get along well. Without similar conditions, using such an ultra-light approach with a 40-person team would be dangerous.

I recently asked the manager of a CMM Level 5 group the question, "What makes your group successful?" His answer was nearly identical to the success factors for Trimble's team: experienced software engineers, plenty of experience with the product, proximity, and an amicable group. If skilled people who work well together are the core elements for success, then everything else should be as absolutely simple as possible. If you have unskilled people who work poorly together, no amount of process will save your projects.

Chapter 14. Jim Highsmith

Jim Highsmith[1] has been working with systems development and project management issues for more than 30 years, in the roles of consultant, programmer, project manager, development manager, sales manager, and methodologist. As a result, he looks at development situations from many different perspectives. He has helped build software in the pre-methodology (similar to prehistoric), structured development, information engineering, RAD, and Agile eras, so he also looks at development from multiple historical perspectives.

[1] This chapter was written by Alistair Cockburn, with the author's thanks.

Jim directs the Cutter Consortium's Agile Project Management Service and is author of *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems* (Highsmith 2000). He has authored dozens of articles, including several "Agile" articles with Martin Fowler and myself. Jim has worked with IT organizations and software companies in the U.S., Europe, Canada, South Africa, Australia, India, and New Zealand to help them adapt to the accelerated pace of development in complex, uncertain environments.

I was very interested in what he has seen along the way, but I was even more interested in what caused the "sudden" appearance of such a radical book as *Adaptive Software Development*.

ALISTAIR: *I have a sense that when a person finds a way of working that is deeply satisfying, it carries a relationship to experiences from long ago or else has a resonance with some deeper or personal principles. I was wondering where that resonance is for you.*

JIM: My first job after getting my bachelor's degree in electrical engineering was for an Air Force contractor, testing software and equipment that tracked the returning Apollo spacecraft. Probably the only thing that still stays with me today from those days is how hard it is to achieve high reliability in systems built from many components. Even if each component has 99.9 percent reliability, putting 200 of those components together reduces reliability to 82 percent, and putting 1,000 of them together reduces it to about 37 percent. The multiplier effect makes high-reliability equipment or software very difficult to achieve. I never wondered why testing was so difficult after this stint in reliability engineering.

After getting my master's degree in management, I managed the development of an accounting system implementation that was going to replace both the software application and the entire accounting system (not the software). We were going to turn the old system off and the new one on—there was no going back. In that project, we were building and integrating lots of different subsystems. The one thing I did as a project manager that saved us was to build a test program that

compared the results of the existing subsystems (payroll, accounts payable, cost allocation) with the new or modified ones. So I had a comparison program that was very complex, since it involved all the systems, all the different codes, and the translations from old to new. As different subsystems would come up, we would find tons of errors. It was my first insight into how critical software testing was.

ALISTAIR: *I am wondering, "The project manager wrote this program? How many project managers these days have either the time or know-how to write such a program?" You were the project manager; why didn't you have one of the programmers write this program?*

JIM: Because they had plenty to do. This was the early '70s, so it was very much the case that everybody did everything—analysis, design, coding, testing, documentation. It was closer to what Agile developers do today. Most of the developers on this project were in their mid-50s; they developed the very first business applications for the company. As I remember, it never occurred to me to assign the testing program to someone else—I just did it. I wanted to *know* that the systems would work.

But my main goal, and everyone's main goal, was to have this new accounting system work. It did, and I got promoted to accounting manager at 27 years of age. The youngest employee I had was 47; everybody else was over 50, and most of them were unionized. It was a very interesting assignment.

One of the things that struck me about moving from a software to accounting manager job was that systems people work on projects. Although there's pressure, it's not like you've got to get things done *today*. There's always tomorrow. Whereas, in a lot of operations jobs, there is no tomorrow; you've got to get it done today. You've got to get the paychecks out, and so on. So when I went back to systems development, I could really appreciate how difficult it was to get the time commitments from the users that we always ask for.

In development, particularly when people were doing waterfall development and the end of the project was six, nine, or twelve months away, and you were doing a requirements document, you just didn't have the pressure that you had in a operational job. I gained a lot of respect for "users" in that job.

ALISTAIR: *I jumped when you described this. How different this situation was compared to modern XP and Scrum projects, where running code has to be delivered every two to four weeks! It's also no wonder that people experience more immediacy in their work in Agile projects compared with the older style of work. What came next?*

JIM: In the last half of the 1970s, we began to see some separation of roles. I was managing systems development at an electric utility company at the time. I began reading Yourdon, DeMarco, Orr, and others, picking up methodology and modeling techniques.

I realized that there were underlying principles to all of these methods. What was really important was whether I knew how to gather requirements and knew how to design, not whether I knew how to draw the diagrams. So even when leading a project that ostensibly used a particular modeling style, I never got too worried about the details of the drawings, preferring to focus on the information we needed. The clients told me that they liked what I did because I was always focused on the project first and the methodology second. We would do whatever we needed to do to get the project out.

One of the things I remember from that time is that a large number of people wanted a cookbook—to be assured that if they followed these 17 steps, the requirements document that they wrote would be correct, accurate, and reflect everything that the system needed to do.

ALISTAIR: *That hasn't changed any.*

JIM: Right. I would say, "Requirements is a nondeterministic process," and they would then ask again for the exact steps to ensure success.

ALISTAIR: *Did you use those words back then, or are those words that you developed a decade later?*

JIM: No, I used the words "nondeterministic process" even in the mid-1980s to describe requirements gathering.

I remained practical about development even as I got more involved in methodologies. In the 1980s, I was working with Warner-Orr techniques, CASE tools, structured analysis, structured design, entity-relationship data modeling, and those sorts of things. And although I used all of those, I was not religious about their use. I was mostly interested in getting the project out the door, so I would only use a little bit of the methodology as we needed it. I worked on the skills of the people, group facilitation, and even working without diagrams. I was always lighter than the methodologies were calling for at the time, and pragmatically focused.

ALISTAIR: *What caused the dramatic shift to the way of working described in* Adaptive Software Development*?*

JIM: The shift came with two project experiences in the early 1990s. The first was working with Microsoft over a period of several years, and the second was putting out a rapid application development methodology with Sam Bayer, then at Amdahl. I consulted at Microsoft for several years in the early 1990s, teaching a software quality dynamics workshop developed originally by Jerry Weinberg, as well as a project management workshop.

Given my IT background, my first impression of the teams at Microsoft was that they were not developing "properly" according to the rigorous methods I'd been using. But I got to thinking about how profitable Microsoft was, and I decided to watch what they were doing, what they were doing right, and how they got the software out. I learned a lot. In particular, they focused on the *practices* and on the *end* of the life cycle: coding and testing, code inspections, getting products out the door. Where the heavier methodologies focused on front-end architecture, modeling, and detailed requirements, Microsoft used conversational design: talk together at a whiteboard, make a decision, and then go and code it. They had rigorous procedures for activities like build management. They used retrospectives to think about what they had been doing.

That experience caused me to see things I hadn't spotted before. One thing kept niggling at me all the way through the '80s. I kept asking myself, "If I sat down to build a system, is this how I would actually work?" I admitted to myself that I didn't work in a nice linear fashion. It's much more integrated than that: a little requirements, a little design, a little coding. The one confirms the other, validates the other. I kept worrying about this idea of linear, sequential processes. So when I started working with Microsoft, I thought, "Here are some people who are doing it a different way and are very successful at it."

And then I started doing projects with Sam Bayer. Amdahl had developed a mainframe RAD tool, but the sales cycle was quite long. Sam was in marketing and needed a RAD methodology to shrink the *selling cycle,* so we put together a style of working that involved a one-month project with one-week iterative cycles. The methodology hung mostly on the fact that we put a small team together in a room; had short, iterative development; and used end-of-iteration customer focus group reviews. It was wildly successful, and we delivered lots of software. We published an article on our experiences in 1994 ([Bayer and Highsmith 1994](#)), and this work became the foundation for ASD.

These two things—thinking about how Microsoft worked and the RAD experiences with Sam—changed my whole thought process about software development. I liked our RAD way of working, and I could see from my Microsoft experience that it would scale, using incremental milestones,

feature-driven development, closely communicating people, and the like. I had seen Microsoft doing similar things with really big projects.

The final part of the shift came when I was working with Nike in the mid-1990s. A large team was restructuring the company's shoe design process, everything from concept to manufacturing engineering, and needed a new way of managing the creative process. Wayne Collier, from DH Brown & Associates, introduced the concept of "phase and gate development." In this process, shoe prototypes and other key work products were produced and evaluated at key milestones. Between milestones, very little was said about *how* to do things. It was results driven, not process driven. There was less control exerted during the creative activities. The "gates" were decision control points. It seemed to me like a way to scale that didn't involve more process and documentation. Not step-by-step control, which doesn't scale well, but sharing information and knowledge.

ALISTAIR: *How does that help scaling?*

JIM: By saying, "Here's *what* you need to produce for us to go forward," not "Here's *how* you need to produce those things. " It is managing the information about results, about completion states, so that people have access quickly and effectively. It's about focusing on getting team members the information they need to do their job—quickly and efficiently—and then trusting them to figure out how to get the job done.

ALISTAIR: *How did you start writing the Adaptive development book?*

JIM: Writing a book was one of those things on my "life list." It started out as a RAD book, but I was looking for some conceptual background to explain what I knew from experience worked. I had this blank spot that was Chapter 3—I knew something went in there, but I didn't know what. I started reading about complex adaptive systems and realized that it provided exactly the conceptual framework I was looking for. I'd read about chaos theory, but complexity was different—it was more encompassing. M. Mitchell Waldrop's book *Complexity: The Emerging Science at the Edge of Order and Chaos* (1992) got me hooked. Over the next several years, I read more about biology and physics than software development. The concepts really excited me, and still do. I think biological software, modeled on how living organisms respond within their ecosystems, is one of our next frontiers.

I used the concepts from complex adaptive systems to try to understand multiple perspectives and experiences. I wanted to handle scaling to larger projects from both technical and organizational perspectives. I wanted to understand how to get these techniques to work for whole organizations, because it would always surprise me that Sam and I would install one, two, or three very successful projects in an organization, but the process wouldn't catch hold. I wanted to deal with that.

So now I had the RAD methodology, the conceptual base, and evidence that it would scale up.

ALISTAIR: *Now that it is several years after you started using complexity concepts, are you still happy with that conceptual base and vocabulary?*

JIM: The science has been a nice background for understanding. Maybe it's only metaphorical, but I think it provides a model of how organizations work, a sense of how the world works. It has continued to be a powerful and useful model for me.

A lot of people misinterpret "adaptive" development as having no planning, but that's not true. To do good development, you need to have articulated the overall goal and mission and have some boundaries. I plan from a chaordic perspective. I plan, but I don't view the plan as a prediction but rather as a hypothesis—I don't really expect the actual to conform to the plan.

This point of view has freed me from worrying about a lot of things that people say you "should" do. Even when I was being very practical about getting a project done, there was always this niggling feeling that we "should" be doing something else—more rigor, more documents, more planning, more requirements. Now I don't feel guilty about not doing them. I have a theoretical foundation backed up by a growing body of work in both the sciences and management.

ALISTAIR: *I'm wondering whether your feeling at home with this way of working is related to some other personal principles you follow in the rest of your life.*

JIM: Yes, absolutely. The main thing is my belief in the fundamental worth and dignity of every person. I believe strongly in the unique contribution that every individual is capable of making, in championing diversity in all its forms as contributing to success, and our mutual interdependence—that no single individual has the range of abilities required for most software projects, and we therefore need to enhance our collaboration skills. This way of developing software, focusing on the individuals and their diversity, definitely has a resonance for me. I don't believe in "normal" curves with people who are above, below, and average. I want to focus on every individual person—the way that every individual person works is important. If you believe this, then the idea that there is one methodology for every project or every person just doesn't fit.

I want to approach a project with the thought that people want to do the best job they are capable of doing. It's my job as a manager to help them use their unique talents and strengths to be the best they can be. This means that I want to staff a project with different sorts of people, people with different personalities than me. I need to be aware of where my thought patterns need to interact with someone else's different thought patterns. Oftentimes this happens with people that I don't get along with, but I know I need them. For example, I prefer to set end goals, outcomes, and then let people work in their own way. Some people are not comfortable with that; some really want a list of steps. I help them work through that. I give them a list of steps if they insist, because I know that's how they work. That helps them feel good about what they're doing. It helps them work best, even though it doesn't do anything for me. But I don't actually ever think that things are going to work out as the list implies. They might come back to me later and say, "Well, it didn't work out like that," and I say, "Fine." As long as we're moving toward our end goal.

Chapter 15. Be Adaptable

No doubt, machines and hierarchies provide easier metaphors to use than markets and gene pools. So it is no wonder that most people are still more comfortable thinking about organizations in fixed, mechanical terms rather than in adaptive, decentralized terms.

- *Robert Axelrod and Michael Cohen,* Harnessing Complexity

The Mustang Team at Cellular, Inc.

Jeremy and his product team at Cellular, Inc. [a fictitious name] in Toronto, Canada, epitomize the headaches, heartaches, and tensions of working in highly uncertain business environments. The product team—code-named "Mustang"—faced major business, organizational, and technical hurdles in its quest to make up a five-year competitive deficit. The project's goal (generally, as the ground shifted often) was to deliver the software for the company's next-generation CDMA cellular telephone chip. In late 1999, it became apparent that the vendor that had been working on the software for the product was far from meeting expectations, so the people at Cellular decided to bring the product development back into the company. However, with nearly 300,000 lines of embedded C already developed, their first job was to figure out just what the vendor had accomplished.

Jeremy started building a team and "infiltrating" the vendor, trying to learn as much as possible before firing the vendor—a tricky proposition. Once the code base was brought under control of the Cellular team, the staff realized what truly foul shape it was in—300,000 lines of code in which the requirements were poorly implemented (interestingly, the design documents were reasonable, but the transition from design to code was atrocious) and very buggy. The team's first job was to try and stabilize the code by developing an adequate testing environment. As testing proceeded, the team was constantly finding that specified features, which had supposedly been implemented, were just not there.

The team faced other problems also. The company was now several years behind competitors— which already had their fifth-generation product in customers' hands—so the business viability of the team's work was constantly under review, and elimination of the product was a constant threat. Because of the precariousness of the product itself, special requirements from prospective customers caused the team to shift priorities again and again. In the end, each of these frantic refocusing efforts ended in vain as the prospects fizzled out.

Product requirements were in a constant state of flux. The team's initial goal was to deliver software that met a customer consortium's set of standards. These three large customers (major cellular phone suppliers) had established an interoperability lab to which suppliers submitted their products for certification. Unfortunately, the consortium's "requirements document" left many questions, and Cellular's product marketing group seemed unwilling—or unable (I only got one side of the story)—to help clear up requirements ambiguities. Sometimes the requirements from prospects were virtually "make it do what *this* phone does." When the team finally submitted its product to the interoperability lab, the staff there were very willing to answer questions—but the Mustang team had not been allowed earlier access to this "customer" when it would have helped substantially.

By the time December 2000 rolled around, Jeremy's team had grown to nearly 40 people (with another 30 in two other locations working on other aspects of the product that had to be coordinated), and nearly 100 person-years of effort were in the product. The team met its goal of passing the interoperability lab tests at the end of 2000.

The team continued to move forward toward full product release (as of spring 2001) in an uneasy fashion. My information on this team came from three perspectives: Jeremy's as development manager, Luke's as software process manager, and John's as a key developer. Their perspectives reflected similarities and, naturally, some differences. Each of the three considered the project to be a success—at least from the perspective of meeting the interoperability goal. Luke, however, was not entirely convinced that this goal should be considered a "success," considering the actual product release was, as of April 2001, still several months away. So, in general, if these team members thought the project was successful, to what did they attribute their success?

Jeremy mentioned three factors, in roughly this order: people, approach, and training. All three—Jeremy, Luke, and John—concurred that the team itself was the primary contributor to success. The individuals in the group were very capable and worked together well. Luke thought one of the reasons they were so willing to help each other was that they had a common "enemy," the firm that was responsible for burdening them with such a horrible mess. They could always point the finger of blame toward a common enemy rather than each other.

The second success factor was approach, although there were some differences of opinion here. Prior to discussing the project with me, Jeremy conducted a brief project "process" retrospective with a half-dozen team members. "It turns out," Jeremy said, "that many people listed some things both as key success factors and the things they least liked about our process," including:

1. Using our software release plan as the primary planning process.
2. Having a rigorous time-boxed release process.
3. Using our integrated problem-tracking system, code control system, and automated Web merge process to integrate all code.

Mustang's software planning and time-boxed release processes worked as follows. At the end of February 2000, senior management set the year-end goal of having a CDMA cellular phone with the chip and software protocol stack certified with the Industry CDMA Cellular Phone Interoperability Standard. This goal was part of the team's year-end bonus program.

During March 2000, the Mustang team planned all goals by quarter until the year's end. These quarterly goals contained major functionality, but the team members had some latitude with features. For the next three months, they set very ambitious, but possible, end-of-month goals that would get them to the quarterly goal. "We designated the end-of-month releases as 'major external releases,'" said Jeremy.

After completing a major external release, the team members had a replanning "ritual" to set weekly commitment goals for the next four weeks that would get them to the next month's major release. They also set targets for each week in the month after the major release. These last targets became committed goals at the following month's replanning session. While Jeremy used the terms "committed" and "target" to differentiate between firm goals that the team had a hand in estimating and general goals that were not commitments, he also admitted that the differences between these two types of goals were not always adequately communicated to the team. At the end of each week, the team completed an internal release on Friday and then ran automated regression tests all weekend.

The Mustang staff maintained this "scrolling planning window with telescoping granularity" through the end of the year. Thus they always had: a one-month plan with one-week granularity, a three-month plan with one-month granularity, and a one-year plan with one-quarter granularity. They referred to this methodology as "time-boxing," because the release dates were all fixed and the only variable was features assigned to the release.

Several team members, including John, were uncomfortable with this "adaptive" planning, as Jeremy referred to it. John felt that too much time was wasted getting ready for the monthly releases. Also, at times he and Luke thought the time pressures of the short iterations forced the

team into concentrating too much on getting features done rather than trying to deliver better quality. Although the team members attempted to do some redesign (refactoring) work, they were in a different position than a team starting with a clean slate—they were constantly battling with the existing real-time code, and even their expensive testing equipment contained unruly bugs.

Being adaptable isn't easy; the tensions caused by constant rescheduling can be hard on a team. The last item on the team's list of success factors—training—addresses this issue. With all the other pressures on the team, money was relatively easy to get, and adequate training was beneficial to the team. Jeremy mentioned the skills training the team received from Software Productivity Center in Vancouver—workshops in risk management, software engineering culture (conducted by Karl Wiegers), and ASD (conducted by myself).

"Your workshop helped reduce the level of anxiety in the group," Jeremy said, "by giving everyone an outsider's view that the turbulence and tension on this type of project are normal." One of the other aspects of my visit was to talk to a number of supervisors and team leads. When people are unaware of the "normalcy" of life at the edge of chaos (and even sometimes when they are), they tend to react negatively. Unfortunately, some of Mustang's team leaders were magnifying the problem. To a team member's lament that "everything is screwed up and getting worse daily," some of the leads would respond, "It sure is," or words to that effect.

One of the jobs of people in leadership positions—whether project managers or team leaders—is to absorb uncertainty and ambiguity, not reflect it back and thereby amplify it. Just having someone admit that things are somewhat chaotic and that this is normal, and having them realize there will be some apparent inefficiencies because of the situation, can help relieve the tension. Being Agile means responding to change, and responding to change means rework and revisiting priorities. However, both process and attitude can contribute to easing the tension. In Scrum, for example, once the feature list for a 30-day Sprint is set, it doesn't vary except in unusual situations. This provides teams with a degree of stability within an overall turbulent environment.

The Great Divide: Predictability or Adaptability

To repeat a quote from Virginia Postrel, "Do we search for *stasis*—a regulated, engineering world? Or do we embrace *dynamism*—a world of constant creation, discovery, and competition?" (Postrel 1998). Do we yearn for yesteryear, when predictability reigned (or at least that's our nostalgic belief)? Or do we accept the inherent unpredictability of our environment? The future of your organization may depend on this fundamental divide. The importance of understanding this philosophical difference between ASDEs and RSMs cannot be underestimated—it affects every aspect of development, from planning to gathering requirements to balancing design and refactoring.

"The underlying insight behind competing on the edge is that strategy is the result of a firm's organizing to change constantly and letting a semicoherent strategic direction emerge from that organization," write Shona Brown and Kathleen Eisenhardt (1998). Balancing on the edge means understanding, within the context of a particular organization or project, the right mix of anticipation (planning) and adapting (responding). There is no simple solution, but individuals and organizations tend to divide into one camp or the other. While too much planning creates mental straightjackets and too much adaptation can result in oscillation without result, anticipators and adapters approach projects in fundamentally and acutely different ways. How individuals and organizations approach this dichotomy of anticipating versus adapting is a critical piece of a chaordic perspective—one of the three components of ASDEs.

Ron Jeffries epitomizes a dynamist. He has an unquenchable faith that his ability to respond to change far surpasses his ability to predict it. In a way, whether Ron is right or wrong is immaterial (although I think he is more right than wrong). His belief in adaptability—bolstered by years of experimentation in what works and what doesn't—is profound. Ron approaches everything in software development from this dynamic, adaptive perspective. Where an anticipator would say,

"If a little design is good, more design must be better," Ron might say, "If a little design is good, then no (anticipatory) design must be better." Similarly, Ron might say, "A lot of documentation is bad, less documentation is better, no documentation is best."[1] Documentation presents a static picture of a dynamic world, like a snapshot photo at an instant of time. The higher the rate of change, the greater the unpredictability, the greater the chance that documentation is a snapshot of last month's reality—it may misinform rather than inform.

[1] Ron might *think* "no useless documentation," but he would probably *say* "no documentation."

Ron's belief in the ascendancy of adaptability over predictability comes from years of thought, experimentation, feedback, rework, rethinking, and reexperimentation. Others have an equally profound belief in stasis, in their ability to predict and control the world around them. There are many, probably a majority, who believe deeply in a deliberate, front-to-back, linear progression of software development. Similarly, there are those who believe that engineering and technology can overcome the impact of global warming on the earth's ecosystem, while there are others who think the biosphere won't survive the perturbations caused by burning fossil fuel and destroying rain forests. Adaptability and predictability are two fundamentally different belief models that individuals and organizations use to approach problems, manage projects, and filter data.

In talks, I've heard Kent Beck range from Taylorism to the ecology of a tree. Whether or not people agree with Kent, there is no doubt that his beliefs about people, the complexity of organizations, and how people should treat each other in an organizational context are deeply held. In the interviews with leaders in the Agile movement—Kent Beck, Alistair Cockburn, Ken Schwaber, Ward Cunningham, Martin Fowler, myself, and Bob Charette—it is clear that we take a dynamist's perspective.

This divide between adaptability and predictability forms the single most difficult barrier for organizations in adopting ASDEs. Some people gravitate instantly to Agile ideas because they match their fundamental dynamism. Others are put off by Agile ideas because they offend their basic anticipatory philosophy. Arguments between the two groups take on a surreal nature—much communication but virtually no understanding. Agility isn't defined by a set of practices, it is defined by a set of cultural beliefs. Understanding your organization's culture is, therefore, the first step in transitioning to an Agile ecosystem.

Two principles from the Agile Manifesto address the issues of predictability versus adaptability.

> *Welcome changing requirements, even late in development. Agile processes harness change for customers' competitive advantage.*

> *The best architectures, requirements, and designs emerge from self-organizing teams.*

Software process traditionalists view changes as defects and regard the dreaded "scope creep" as one of the demons of development. Granted, projects with higher change rates have a greater chance of failure, but many of those failures are caused because we view change as negative, not as inevitable. Agilists welcome change as an opportunity to help customers respond to marketplace turbulence. We turn out products while the anticipators are still drafting their detail plans—as several case stories in this book illustrate. "As for the future," wrote Antoine de Saint-Exupery, "your task is not to foresee it, but to enable it."

The concept of adaptation extends into the entire business enterprise. Stephan Haeckel, director of strategic studies at IBM's Advanced Business Institute, points out the difference between "make-and-sell" versus "sense-and-respond" organizations (Haeckel 1999). Automobile companies, for example, exemplify make-and-sell. They build cars and send them to dealers, and the dealers sell them to customers—a process fraught with customer dissatisfaction. Delivering a customer-configured car takes weeks, if not months (at least in the U.S.). On the other hand, Dell allows customers to configure their own computer system (within some limits). GM makes and sells. Dell

senses and responds. Haeckel doesn't advocate waiting until the customer calls to figure out what to do, but he does advocate building a flexible infrastructure that enables quick responses to customer needs.

In times of uncertainty and turbulence, relying on predictions encourages two types of errors: presumptive planning and excessive caution. Planners "presume" they know more about the future than they actually do, and they are at the same time overly cautious and fail to act quickly enough. There is a story about a bank that was so obsessed about releasing a defect-free financial product that it took 15 years to deliver a home banking system. Organizations become so mired in designing the ultimate, they fail to deliver "partial solutions" in reasonable time frames. Even sadder, in a rapidly changing environment, much of their anticipatory work will be wasted anyway, as events careen off in directions no one anticipated. Finally, because of the "presumption" of correctness, companies stick to a plan long after change has rendered it obsolete.

Great results—architectures, designs, and working software—emerge over time as competent individuals interact, create, learn, and rework. Anticipators view rework as the result of a broken process rather than the natural order of things in an ever-changing world. To be sure, there is a balancing act, but just as surely, there is a fundamental perspective gulf between anticipators and adapters. "Planning is not a solution to any problems," states Aaron Wildavsky. "It is just a way of restating in other languages the problems we do not know how to solve" (Petzinger 1999).

Our Changing Business Ecosystem

Change is changing. We can no longer think of change in the traditional vein of stuff-happens-and-today-it-happens-faster. In the Information Age, there are at least two new specific types of change: disruptive change and punctuated equilibrium. Furthermore, each type of change requires a completely different strategy.

Clayton Christensen's best-seller, *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail,* describes what happens to firms caught in disruptive technological change. When a new technology undermines a particular market, such as when personal computers disrupted the mini-computer market, established firms whose managers are successful by every traditional measure often fail. Christensen's study of the 20-year history of the disk drive market between 1976 and 1996 shows that in a disrupted market, speed to market becomes absolutely critical. When traditional "stuff happens" change occurs, efficient change management is a reasonable strategy, but as Christensen shows, when companies face disruptive change, speed becomes the preferred strategy. This is not an easy transition for many companies, as the demise of mini-computer companies indicates (Christensen 1997).

However, there is another type of change. While disruptive technologies might impact an industry, or a couple of industries, there are even more catastrophic changes that impact entire economies.

Punctuated equilibrium is an evolutionary biology term that has crept into the business vernacular. For many years, biologists' sole theory of evolution was that of continual, gradual change driven by mutation and natural selection. In the early 1970s, however, paleontologist Stephen Jay Gould and others began finding fossil evidence that periods of gradual change were interrupted by short but prolific bursts of new species creation, preceded by a time of rapid extinction (for example, the dinosaur extinction). The name given to this new theory was "punctuated equilibrium," or "punk-eek."

The Internet-fueled explosion of dotcom species, and its turbulent aftermath as companies struggle with eBusiness and eCommerce implications, qualifies as a punctuated equilibrium in many markets. In such a situation, a company's ability to *adapt* becomes more critical than speed. With so many companies frantically searching for the combination of strategy and capabilities that will lead to success, there will be more time for individual companies to explore possibilities. In a disruptive situation, time is the critical criterion. In a punctuated equilibrium, time takes second

place to exploring and evaluating possibilities. Amazon may have been the first online bookstore, but it maintains its lead because of its ability to adapt again and again and again.

The rise of the dotcom economy was a big change in one direction, and the fall of the dotcom economy was another big change—just in a different direction. Just as companies tried to adapt to sizzling growth in 1999 and 2000, they had to adapt to plunging revenues and profits in 2001. But it is all change, turbulence, risk, and uncertainty.

For many companies wondering how to respond to the opportunities and threats of an Information Age economy, agility, innovation, and adaptability are ultimately more important than speed. To survive punk-eek turbulence, adaptability skills must run throughout the organization, from strategies to tactics to day-to-day operations. To thrive, organizations must be attuned to types of change—traditional, disruptive, and punk-eek—and to the strategy implications of each—efficiency, speed, and agility. To execute on these change strategies, organizations need to adopt a chaordic perspective, embrace change, and promote collaborative values and principles.

Embracing Change

Change control boards (CCBs), a recommended traditional project management practice, should be labeled "change resistance boards." Forms are filled out, approvals are obtained, and then periodically the CCBs are convened to review a long list of changes. Some changes are approved while others (most) are rejected, but the impacts of the CCBs are to slow projects down and discourage people from requesting changes because the process becomes so onerous.[2]

> [2] Characterizing CCBs this way may be unfair. Any practice can be ponderous or Agile depending on how it is implemented. However, CCBs are particularly vulnerable to heavyweight implementation.

Now, I would agree that change practices such as a CCBs don't start out to be ponderous, but they invariably end up that way. In a turbulent, high-change environment, this approach to change is doomed to fail. In order to succeed in high-change situations, we have to approach them from a new perspective. For example, everyone jumps on the bandwagon that "feedback" is important, but they somehow miss the conclusion that the result of accepted feedback is change. How many project plans have some kind of feedback practice—code reviews, management reviews, prototype reviews—but no allocated project time for implementing the changes that are generated by those reviews!

On the one hand, we seem to acknowledge that change is pervasive, but our practices don't support this view of change. Our "change control" practices operate on the assumption that an acceptable level of change is 10 to 15 percent, but even that degree of change is viewed as a problem rather than an opportunity.

Exploratory projects are basically unpredictable. Plans are speculative, architectural components can vary over time, requirements are volatile, and designs must focus on change as the highest priority. Until we embrace this fundamental shift in perspective, managing these projects will continue to be a terrifying prospect. Once we accept unpredictability, we can begin looking at practices from a different viewpoint.

- Facilitating change will be more important than controlling change.
- Getting better at rework becomes a virtue.
- Change "control" is best focused on the final product components.
- Feedback must be built into every level of development.
- Facilitating change requires multi-level processes.

Facilitate Change

First, we want our practices to encourage changes, not resist them. (Naturally, there are reasonable limits to this.) We want to encourage customers to revise their requirements to accommodate new knowledge, and we want developers to alter designs to incorporate new information and make them easier to change in the future. Contrast the behavior of adaptive and anticipating cultures when faced with changes during a project. Anticipating cultures (which, by the way, are in control of the predominant software development and project management professional organizations) offer something like the following when discussing project change.

> *Scope creep in software projects is a serious problem that causes many projects to miss their planned targets and a significant number to fail completely. The solutions to this problem are rigorous front-end requirements and tight change-control procedures.*

Of course, anticipating managers recognize that some level of change is inevitable; however, their fundamental posture is change avoidance. Contrast this with an adaptive viewpoint.

> *Scope creep in projects is a natural outcome of a turbulent business and technology environment; therefore we must implement processes that encourage change (in order to respond to the reality of that turbulence) and reduce its cost. We need to facilitate change so that we are always moving toward our business goals.*

Scope creep can adversely impact Agile projects, of course, but practices such as freezing feature priorities during an iteration are used to mitigate adverse affects. However, at the same time, Agilists encourage change in order to deliver business value. Agilists use CCBs, but their focus is on informing those who need to be involved in the change decision or made aware of a change, not on blocking change. It is about attitude more than specific practices.

This doesn't mean that Agile projects accept every idiotic change request, any more than rigorous projects reject every change request. However, the development change process must be geared to the true change rate in the external market. Accepting change for change's sake is no better than rejecting change just because it is a change. Nevertheless, we have to get away from the mindset that changes are fundamentally "the result of errors" and "too expensive." We need to adopt the view that changes are "the result of ongoing improvement in our information" and focus on reducing the cost of responding to them.

View Rework as a Virtue

We must get better at rework. From a traditional perspective, rework is the result of a broken process. But there are two sources of rework: broken processes and new information. There are more and more changes in today's projects that can't be "fixed" by improving a linear delivery process. Instead they must be accommodated by improving the process for incorporating changes; that is, we need to become better at rework rather than eliminate rework. Refactoring (continuous redesign) is a key practice in this quest to get better at rework.

In thinking about rework, we are left with a critical question: How much *anticipatory* design work should we do? None? A little? Simple design produces only the functionality specified. Anticipatory design builds in extra facilities, anticipating future requirements. Anticipatory design trades current time for future time, under the assumption that a little time now will save more time later. Yet under what conditions is that assumption true? Might it not be faster to redesign later, when we know exactly what the changes are, rather than guessing?

This is where refactoring enters the equation. "Refactoring," according to Martin Fowler, "is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" (Fowler 1999). XP proponents practice refactoring as a way of incorporating change. If changes are continuous, then we'll never get an up-front design completed. Furthermore, as changes become unpredictable—a great likelihood today—then much anticipatory design will likely be wasted.

The left-side diagram in Figure 15.1 depicts the situation prior to the rapid-paced change of the Internet era. Since the rate of change (illustrated by the positioning of the balance point in the figure) was lower, a higher level of anticipatory design versus adaptive refactoring was effective. However, the right-side diagram shows that as the rate of change increases, the viability of anticipatory design loses out to refactoring—a situation that defines exploratory software projects.

Figure 15.1. Balancing Anticipation and Adaptation
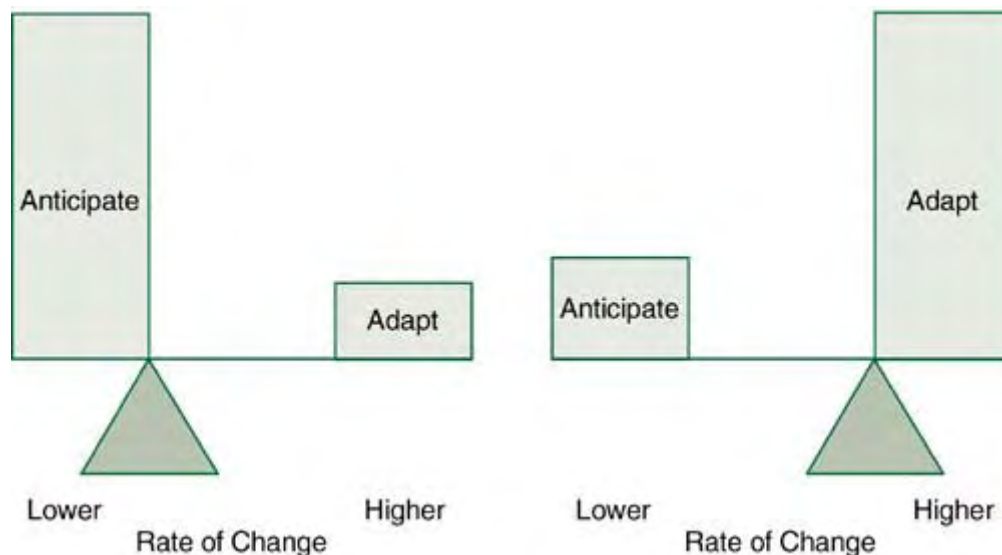
In the long run, the only way to test whether a design is flexible is to make changes and measure how easy they are to implement. One of the biggest problems with an "anticipatory design and then maintain" strategy has been that software systems exhibit tremendous entropy—they degrade over time as maintainers rush fixes, patches, and enhancements into production. Current refactoring approaches aren't the first to address the problem. Back in the dark ages—circa 1986—Dave Higgins wrote a book titled *Data Structured Software Maintenance,* which addressed the high cost of maintenance caused by the cumulative effects of changes to software over time. While Dave advocated a particular program-design approach (Warnier/Orr), one of his primary themes was to slow the degradation of systems over time by systematically redesigning programs during maintenance (Higgins 1986). Eventually every system will be replaced, but many business-critical applications written in the early 1980s are still in use, and their maintenance costs have soared, in some part due to lack of systematic redesign.

Dave's approach to program maintenance was first to develop a pattern (although the term "pattern" was not used then) for how the program "should be" designed, then to create a map from the "good" pattern to the "spaghetti" code. Programmers would use the map to help understand the program and, further, to revise the program over time to look more like the pattern. Using Dave's techniques, program maintenance counteracted the natural tendency of software to degrade over time. "The objective was not to rewrite the entire application," Dave told me, "but to rewrite those portions for which enhancements had been requested."

While this older-style "refactoring" was not widely practiced, the ideas are the same as they are today—the need is just greater. Two things enable, or drive, increased levels of refactoring—one is better languages and tools, and the other is rapid change.

How do we determine where the balance point should be between anticipation and adaptation? One possibility is to push the envelope until something breaks and then back off a little. Ron Jeffries, for example, is very clear about his intentions. His avowed goal is to "break" XP, to anticipate as little as possible to see if XP breaks. But Ron and others are experimenting—and these experiments have taken place within certain project domains. Several other Agilists

115

recommend a little anticipation up front—maybe a week or two for a six-month project. Compared to XP, the skeletal up-front modeling of FDD may seem "heavy," but compared to traditional, rigorous methodologies, both XP and FDD are on the ultra-light end of the spectrum.

Control Final Components

Rather than trying to control all intermediate deliverables at a minute level, developers should focus first on controlling the final product results. In software, this means focusing on controlling the code (and test cases) first, rather than tracking every requirement alteration.

Constant Feedback at Multiple Levels

Project teams must devise an entire series of feedback practices, from the near instantaneous feedback of pair programming to the longer-cycle practice of customer focus groups at the end of each project milestone (iteration). Feedback practices must involve every stakeholder group. "Early and often wins" is the Agile mantra.

Predictive mindsets can lead to blindness through lack of adequate feedback. If we approach projects from the perspective of "I'm the boss and I'm in charge and we *will* meet our plans," any attempt by team members to raise information that contradicts the plan will be written off as negativism or with the charge that "they're not team players." It's a Pollyannaish view of the world. Fortunately, the marketplace is in charge. The more adamant one is about the "correctness" of a plan, the more projects will suffer. Most large project failures are the result of managers failing to listen to reality, not developers failing to perform.

Agile approaches use a number of feedback practices, geared to different time frames. Pair programming in XP offers almost instantaneous feedback at the coding level, as does test-first development. With other ASDEs, frequent unit testing provides feedback in the tens-of-minutes range. Frequent interaction with users provides hourly to daily feedback on requirements. Daily team meetings (Sprint meetings in Scrum) provide daily project progress information. Daily (or more frequent) builds indicate the state of the code base.

On a slightly longer time scale, iteration plans (weeks) provide checkpoints on progress, while customer focus groups (ASD) provide in-depth customer reviews of features. After several months, releases (including beta releases) provide feedback on live products in actual use. There is so much feedback in Agile development that some may wonder, "Aren't you overdoing it?" But thinking back to Ken Schwaber's description of defined versus empirical processes offers comfort and justification. Empirical processes, those that are characterized by significant variations in inputs and heuristic transformation rules, can only be controlled hour to hour, day to day. Feedback, and the constant learning that attention to feedback implies, controls Agile development projects.

Multiple Process Levels

Finally, facilitating change requires multiple levels of process. For example, within a four-week development iteration, we may give the project team (which includes on-site user involvement) the authority to make requirements changes—without external approval. Only changes that would impact the delivery goal would require external approval. Project teams may need to identify four or five types (including size) of changes, with varying levels of disposition and approval authority. In volatile environments with tight delivery schedules, every change decision that has to be made by some person or some group outside the immediate project team will slow the project down. We have to give the members of the project team more authority if we want them to accept the responsibility for timely delivery.

While change is pervasive in today's projects, many people have not made the mental switch needed to facilitate and embrace change as opposed to controlling it. In my experience, it is not an easy transition, but it is a necessary one. Adaptive managers recognize that some level of stability is required, but their fundamental posture is change tolerance.

Balancing Adaptation with Anticipation

depicts a generic model for making anticipatory versus adaptive decisions—up-front requirements gathering versus incremental requirements gathering; up-front architecture versus emerging architecture; front-end project planning versus continuous end-of-iteration planning. Anticipating cultures often brand adaptive cultures "ad hoc" or "immature," but there is another perspective on balancing adaptation with anticipation.

"Improvisation is popularly thought of as 'winging it,'" write Brown and Eisenhardt (1998). "But true improvisation is distinguished by two key properties. First, performers intensely communicate in real time with one another. Second, they rely on a few very specific rules." The authors go on to outline common traits of companies with too little, or too much, structure. First, they point out the traits of companies with too little structure.

- A rule-breaking culture in which freedom and individual decision making are carried to extremes.
- A loose structure in which roles, responsibilities, and goals are unclear. People don't really know what they are supposed to do.
- Random communications, rather than focused and relevant communications. While rich information flows are characteristics of complex adaptive systems, random, unfocused information flows tip the system into chaos.

The flip side of too little structure is too much—namely, bureaucracy.

- A rule-following culture in which procedures and standards are seen as signs of effective discipline. Changes are viewed as annoyances that upset the rules.
- A rigid structure of "tightly choreographed processes, elaborate job descriptions, carefully crafted organization charts, and rules for every occasion."
- Channeled communications that follow very formal procedures.

The correct strategy, according to Brown and Eisenhardt, is navigating the edge of chaos—just enough, but not too much. The "improvisational" businesses that manage to do this also have three traits:

- An adaptive culture in which managers "anticipate the need to iterate, backtrack, and adjust"
- Semi-structures in which a small set of key structures, such as a small set of generative rules, are never violated (two of the structures mentioned, setting priorities and deadlines, are key pieces of Agile approaches)
- Real-time communications that focus on the tasks at hand

The point about semi-structures reminds us to be careful when thinking about generative rules. These "rules," whether they are XP's twelve practices or a Scrum meeting's rules, should not be modified without careful thought and experimentation, because careful thought and years of experimentation went into their creation. If you have forty-nine development practices, altering one or two may not be a problem (unless, of course, you inadvertently alter a critical one), but if you have four or twelve, altering one or two may negate the semi-structure.

Balancing anticipation and adaptation is critical; however, organizations—depending upon their cultures—approach balancing from very different perspectives. Some managers and developers

retreat from making balancing (tradeoff) decisions, while others recognize that balance is critical in high-change situations.

Putting Lipstick on a Bulldog

> *When outcomes are uncertain, answers hard to devise,*
>
> *That's the time to form a team, tap dreams, and improvise....*
>
> *Putting lipstick on a bulldog won't transform enough,*
>
> *Makeup can't hide everything; change takes deeper stuff.*
>
> *—(Kanter 2001)*

Is there a culture of change—an Agile or adaptive culture? In her book, *e-volve*!, Rosabeth Moss Kanter, a world-renowned expert on change, sends two clear messages: There is a new digital culture of the future that we have to understand, and a little makeup here and there won't be enough to meet the challenge. A little less documentation, sprinkling an "extreme" practice here and there, revamping a couple of procedures, or pseudo-empowering teams won't be enough to build the team ecology required to deliver value as markets change again and again over the next decade.

The new digital frontier, unlike the frontier West, is not one of place. "It is a frontier of technologies, ideas, and values," says Tom Petzinger, 20-year columnist ("The Front Line") for the *Wall Street Journal* and author of *The New Pioneers*. In his first chapter, "The Age of Adaptation," Petzinger describes these new pioneers.

> *The new pioneers celebrate individuality over conformity among their employees and customers alike. They deploy technology to distribute rather than consolidate authority and creativity. They compete through resilience instead of resistance, through adaptation instead of control.*
>
> *Indeed when you look deeply enough into a growing organization or spend enough time with thriving entrepreneurs, business begins to look very much like pure biological evolution, propelled into real time on the fuel of human intelligence (Petzinger 1999).*

The central question related to turbulence, change, and our changing social systems is: Do our traditional, fundamental models of the world and how to manage make *sense* anymore? For a growing cadre of scientists, business executives, and Agile development leaders, the answer is *no*.

Since the mid-1980s, scientists have introduced the concepts of complex adaptive systems (CAS) into the scientific debate. Similarly, since the mid-1990s, a growing group of managers have begun applying CAS concepts to make sense of how organizations operate in an increasingly complex world. Many, but not all, of the leaders in the Agile Software Development movement have based their ideas about management on the sense-making ideas of complex adaptive systems.[3] Not every leader in the Agile community would articulate his or her perspective in CAS terminology—Ken Schwaber, Kent Beck, and myself are probably the most ardent about a complexity worldview—but nearly all articulate a compatible perspective. When Kent talks about "organic design" or "generative rules," his conceptual foundation is an understanding of complexity, as are Ken's ideas when he designs daily Scrum meetings.

[3] For an extensive discussion of complex adaptive systems, see Adaptive Software Development (Highsmith 2000). Also, a number of organizations, notably the Center for Business Innovation at

There are five key ideas about complex systems that are helpful in understanding organizations in new ways. These ideas reinforce ideas that managers have found to work in actual practice.

1. Complex systems, be they biological or human, are composed of independent, decentralized agents.
2. These independent agents, in the absence of centralized control, will self-organize.
3. Self-organization will create complex behavior and emergent results that are not evident from studying the agents themselves.
4. Rich information flows in an ecosystem balanced at the *edge of chaos* define the most effective pattern for generating emergent results.
5. Simple, generative rules guide the creation of complex behaviors (as described in Chapter 11, "Technical Excellence").

The Agile Manifesto principle that individuals and interactions are more important than process and tools is a reflection of these complexity principles. Individuals ("agents" in CAS terminology) are independent, and it is the *interaction* between those individuals that creates innovative (emergent) results. If these agents are guided by a simple set of rules rather than volumes of processes and procedures, they have a much better chance of responding to the complexity of the world around them. Why? In a turbulent, oft-changing environment, a set of comprehensive procedures will never be comprehensive enough to address every situation, and the more comprehensive they become, the more difficult they are to apply. Simple rules guide innovative, intelligent responses. Comprehensive rules guide rote, routine responses.

In complex environments, such as those found in exploratory projects, "Adaptation is significantly more important than optimization" (Highsmith 2000). Optimization (anticipation) practices are based on assumptions of predictability and control. If the environment is relatively predictable, and the problems at hand are similar to ones we have encountered before, then optimizing practices—getting better and better at what we already know how to do—are useful. However, in unpredictable environments, those in which we have little experience, maintaining the flexibility to try different approaches, making mistakes and learning from them, and embracing change rather than fighting it become more useful strategies. It is part of the old dichotomy "doing the right thing versus doing the thing right." Adapting helps us explore the unknown—with only a vague idea of where the trails may be. Optimizing helps us traverse known trails at lower cost and with greater efficiency. Information Age projects require more of the former (adaptive practices) and less of the latter (optimizing practices).

Self-organizing teams are not the same as self-managing or self-directed teams, a concept that flourished in the early 1990s. In complex systems, the absence of centralized control doesn't mean lack of local leadership. Self-direction and empowerment, two closely related concepts, are about letting different leaders emerge at different times and lowest-level decision making. Empowerment is a concept leftover from hierarchical management structures in which decision making is pushed down (theoretically at least) to the lowest possible level.

Self-organizing within a Leadership-Collaboration management model has a different flavor entirely. Setting direction, establishing boundaries, assigning staff with certain talents to roles, building a multi-tiered decision-making process (in which managers have the responsibility and authority to make certain decisions) are all necessary to establish a leader-directed, self-organizing team. Management and leader roles in a Leadership-Collaboration culture are different from those in a Command-Control culture, but they are still critical roles for achieving performance. Command-Control cultures place control within the hierarchy of organizational levels. Self-managing cultures appear to abdicate management completely. Effective Leadership-Collaboration cultures, those that understand how to create effective self-organizing teams, balance the two.

Decision-making processes are rarely mentioned in traditional methodologies, yet making decisions is often the most critical "process" a project team encounters. The intention of "empowerment" was to delegate decision-making authority to lower levels of organizations. For the most part, empowering involved a lot of talk and little action, because the fundamental hierarchical, Command-Control management structure remained firmly in place. Empowerment focuses on who "makes" decisions. Agile approaches practice collaborative decision making, which focuses on who gets "involved" in decisions. Managers who make decisions without input from subordinates and peers make poor decisions. Subordinates who make decisions without input from managers and peers make poor decisions. Who makes the decision is less important than getting the right "whos" involved in the decision. Leadership-Collaboration managers understand collaborative decision making.

Rich information flows in an ecosystem balanced at the edge of chaos define the most effective pattern for generating emergent results. Emergent results, creativity, and innovation require a rich stew of information, of conversations and interactions. The "edge of chaos" defines a narrow band in which there is enough order to stave off randomness, yet enough disorder to generate new ideas. Roger Lewin and Birute Regine (2000) refer to this balance point as the zone of creative adaptability. Dee Hock (1999) refers to "chaordic" management, balancing between chaos and order. Creating an adaptive, Agile project team or organization involves balancing on this precipice—a very uncomfortable position for those raised within a management culture whose worldview is one of Newtonian certainty.

The Cost of Change

Early in his book *Extreme Programming Explained* (Beck 2000), Kent challenges one of the oldest assumptions in software engineering: the pervasive "cost-of-change curve" that goes something like, "If a defect is found in the definition phase, it costs $1 to fix; if found in the design phase, it costs $10 to fix; if found in the coding phase, it costs $100 to fix; and if found after implementation, it costs $1,000 to fix." Structured methods and then comprehensive methodologies were sold based on these "facts." I know—I developed, taught, sold, and installed several of these methodologies during the 1980s using these figures. These numbers have been around for so long, I decided to track down their origin. The earliest reference I came up with was to an article that Barry Boehm wrote in the December 1976 issue of *IEEE Transactions on Computers* (Boehm 1976).

Not knowing if this was, in fact, the earliest reference, I sent an email inquiry to Barry, to which he graciously replied at some length. "On the cost-to-fix trend numbers, my December 1976 article summarized data I'd collected at TRW, plus corroborative data I'd picked up from sources at GTE, IBM, and the Bell Labs Safeguard program—all covering experience on large projects. This data and its source references are also in my *Software Engineering Economics* book (Boehm 1981), along with data I'd collected on small student projects. At TRW in the 1980s, we were able to identify the high-risk 20 percent of our defects that were causing 80 percent of our rework and to factor this into our risk management and architecture practices. On our best early '90s projects, this led to fairly flat cost-to-fix curves."

Commenting on another study in which he and Vic Basili compared the impact of ten defect reduction phenomena on the cost-to-fix curve, Barry wrote, "We didn't find a lot of definitive recent data, but what we found indicated that 100-to-1 cost growth was still happening on large, unenlightened projects, while 20-40-to-1 was more characteristic of large projects starting to pay more attention to software architecture and risk management."

So in general, the idea that early detection of defects reduces costs appears to still be valid; however, the data also indicates that good risk management and development practices mitigate the cost deferential. This historical cost data has been instrumental in convincing a generation of software engineers to utilize front-end, concept-driven, sequential development methodologies. However, Barry introduced his spiral, risk-driven development model in 1987 and has been

refining it ever since. So even the originator of the cost-of-change numbers works to mitigate costs over the development life cycle rather than trying to eliminate all defects in a front-to-back linear fashion. The spiral model was (and is) a change-tolerant model.

So, are the historical cost-of-change numbers in conflict with Kent and other Agilists' assumptions? I think the apparent conflict can be resolved by altering the traditional perspective of change as resulting from defects. Traditional process management—by continuous measurement, identification of errors, and process refinements—strove to drive variations out of processes. If we consider change to be the result of a defect, then early detection will ameliorate the cost to fix. However, as the rising number of changes are the result not of a defect in process but of an external change from the environment—from customers, competitors, and governments—then whether it is cheaper to catch "defects" up front is irrelevant. If it is impossible to anticipate requirements (or design), then the fact that it's cheaper to do so is a moot point. The right strategy, according to Agilists, is to work on reducing the cost of change throughout the project by developing simple solutions (less to change, easier to change), refactoring (making the design easier to change), and constant testing (early, less expensive, defect reduction).

Our practices should prevent defects, but in addition, they must be geared to reducing the cost of continuous change. As Alistair Cockburn points out, the high cost of removing defects provides an economic justification for practices like pair programming. Change is expensive, no question about it. However, consider the alternative—stagnation.

Conform to Actual: Measuring Success

"If the date is missed, the schedule was wrong," says Tom DeMarco (2001). Lewis Carroll's oft-quoted maxim "If you don't know where you are going, any road will do," from *Alice In Wonderland,* might seem to apply to Agile development, particularly to Agile planning. Critics might go so far as to label Agile planning "Wonderland Planning" or "Alice's Folly." Traditionalists, comfortable with very detailed 12- to 18-month plans, don't understand. Colleague Ken Orr once quipped about PERT charts, "Everything outside a three-month window is the area of well-documented ignorance." Scrum developers plan. XP developers plan. Adaptive developers plan. Lean developers plan. All these approaches advocate long-cycle release or project plans combined with short-cycle (weeks) iteration plans—all driven by features rather than tasks. However—and this is what stymies critics—we don't believe our own plans! In fact, the only thing we are sure of—our only 100 percent accurate prediction—is that our plans will change, often dramatically.

Plans should not be thought of as predictions, but as hypotheses to be tested. Plans are static documents, but planning is the dynamic interaction of individuals discussing their uncertainty and anxiety about the future. A plan is important, not as a plan itself, but as a vehicle for intense conversations, thinking, and collaboration. As General Dwight Eisenhower said, "Plans are nothing. Planning is everything." When plans are considered predictions, team members and managers feel locked in, and adapting to change becomes difficult. When plans are considered to be hypotheses, team members and mangers open themselves up to the flexibility required for exploratory projects.

"When the Standish Group bemoans that over two-thirds of all projects fail or vastly overrun," said Ken Schwaber, "they mean that the projects fail to live up to the predictions and definitions provided by the methodology. The people don't fail; the definition of the project fails. The work doesn't fail; the expectation set by the methodology fails" (Highsmith 2001). Often the "methodology" is used by executives to impose unrealistic goals on projects.

When major studies, year after year, decry that software development is in the dark ages because our conformance to planned schedules (a traditional measure of success) remains poor, maybe it's the measurements that are faulty. If plans are predictions and we expect projects to meet those predictions, then we are ignoring the reality of risk and uncertainty. With exploratory projects,

conforming to a predetermined plan is an accident. Year after year after year, we try to mold the world into our linear, predictable model, and when it doesn't comply, we force it by fudging the actual numbers to meet the plan.

Granted, projects fail for a wide range of reasons—many of which have to do with execution rather than planning or conforming to plans. However, we need to change our ideas about what success and planning mean in the face of volatile environments.

One Agile Manifesto value states that it is more important to respond to change than it is to follow a plan. Taken to its most radical form, this means project managers should abandon "conformance to plan" and embrace "conformance to actual" for project control purposes. In other words, measure what was actually accomplished, compare the results to a product or project vision, ask the customers if the working software has value, and respond accordingly.

If the business world is as turbulent as we have told ourselves over and over again, and agility is the core competency that enables us to navigate through these waters, then one of project management's most cherished control mechanisms is outdated—The Plan. Conformance to plan has been the mantra of project managers for decades—it forms the foundation for project management "certification" and is at the core of how managers think. It's time, however, to alter that thinking. Rather than conformance to plan, we need to think about conformance to vision and value. We need to think of plans as guides, not straightjackets. We need to focus on customer collaboration and determining what the customer wants, rather than conforming to plans made months previously.

I was recently talking to a project manager for a large systems integration firm. On a major project, the actual expenditures were slightly over the planned budget. "Fix the numbers so we're within plan," he was told by a senior manager. The chances of this organization being Agile are zip, nil, nada.

Let's review the traditional sequence of project planning. Some business initiative is expressed as a vision statement and a series of business objectives, followed by a period of minimal requirements gathering and then development of a project plan—scope, schedule, and resources. Whether the project plan is dictated—"You *will* be through by December 1"—or the delivery team has significant input into the final document, The Plan, as the saying goes, gets set in concrete. From this point forward, The Plan becomes a control mechanism. Variations between plan and actual require reams of data and significant project management time to explain. Customers who dare make suggestions are asked to traverse elaborate change control procedures. Colleague Laurie Williams reminds me that "dates" per se aren't bad, or even inappropriate. The problem with dates comes about when they are arbitrary (sketchy requirements and no planning) and combined with a reluctance to make necessary tradeoffs in other dimensions—cost and scope. In the hands of authoritarian Command-Control managers, plans become bludgeons rather than guides.

In a turbulent environment, plans are more analogous to powder snow than concrete. Business objectives change as competitors and customers provide new information; technology decisions must be revisited as technology choices expand; features change as further analysis brings deeper understanding; and new features arise as business objectives change. In a Harvard Business School case study of the development of My Yahoo!, Marco Iansiti (1998) notes that half of the code changed in the last four weeks of the project, culminating in almost three entire rebuilds of the product over the development cycle.

Projects don't have to be nearly as volatile as My Yahoo! was in order to make plans obsolete as control mechanisms. I've been very careful in this last statement. I haven't said plans are obsolete, just that they are obsolete as control mechanisms. Plans, particularly those that have regular feedback and adaptation mechanisms, are still very useful as guides, as directions to move toward, as long as those plans don't prevent project teams from changing.

So the next question becomes, "OK, so what *do* we use to control projects? We can't just let project teams wander around wherever the mood takes them." Certainly not. However, if plans are constantly being altered to reflect a new reality, then the control mechanism must be "actual results" rather than plans. Rather than conformance to plan, we compare actual results to vision and value. We have to continuously interact with the customers (both buyer and user) to determine, one, if the software we have actually delivered conforms to their evolving vision of what they want, and two, if the software application, as demonstrated to the customers, delivers value. We want to determine whether the project is still moving in the right direction and whether the customers feel they are getting value for the money they spent for one delivery iteration.

These checkpoints need to occur every four to six weeks. If the answer to the question about vision and value is "yes," the project proceeds to the next iteration (in which the first task is to update the plan). If the answer is "no," then the project is terminated or corrective action is initiated. At the end of an iteration, a customer may say, "I thought I understood what I wanted four weeks ago, but now that I see the results, and based on my recent business input, we need to change direction." Or let's say the development group delivers a set of features, but for some reason it missed what the customer intended. In either scenario, there is little blaming (let's face it, *no* blaming is just not consistent with human nature), and the response is, "Now that we know what we now know, how do we best move forward?" To slightly alter Tom's quote, "If the plan was missed, it's because the plan was wrong." If pressure to conform to plan causes a project team to fail to respond to a customer's changing business needs, we have a dysfunctional control mechanism.

There is no doubt that a vision-and-value control mechanism is more difficult to implement than a conformance-to-plan mechanism. Measuring conformance to plan can be so easy, we lose track of the fact that the information it provides may be useless. Conforming to vision and value is fuzzier and requires a real partnership between customers and deliverers. However, in turbulent environments, we need to find other mechanisms besides monolithic plans with which to measure success.

This isn't to say that plans are not important; they are. Budgets need to be made, implementation and marketing plans need to go forward, management needs to understand when project results may be ready. However, the difference here is that the plan is a guide and not a control point—a guide to assist in making hard tradeoff decisions as changes occur.

Would you rather have a team work to achieve a plan or work to its potential? If plan and potential are reasonably aligned, then either answer may suffice. However, if plans are constantly changing, requirements are changing, and feature priorities are changing, maybe the answer has a great impact. What happens with most aggressive plans, those that a project team feels have been imposed upon it, with little input? Team members grudgingly plod along, doing only what is specified.

In contrast, think of the motivation for a project team when its effort is naturally assumed to be the best it can be and actual results are assumed to be acceptable. Consider the charge to such a team from management.

- Here is our product vision and why it is important to our company.
- Every four weeks we expect the team to deliver as many features, which the customer has prioritized, as you can. Our customers will tell us at the end of every iteration if they think what we are delivering meets their product vision and is valuable to them.
- During each iteration, we trust you to do the best you can do. If we miss the planned number of features, we will assume that the plan was wrong.
- At the end of each iteration, the team should examine its own performance to see what it might improve during subsequent iterations.

- We will always base plans for the next iteration on the last one. If your performance improvement ideas are successful in delivering additional features in the current iteration, planned deliveries for the subsequent one will then be increased.
- Management (project and executive) will assist the team in any way possible to remove roadblocks (impediments) and provide the necessary resources.

Agile developers and our customers know where we are going—sort of—we just know that there will be multiple paths, and they will change along the way. "Yikes!" say critics. "If the plan changes, how can you possibly control projects?" Easy, our control variable is results, not plans. The real measure of success is whether or not business value was delivered to our customers, at an acceptable level of quality, within a reasonable time frame, at an acceptable cost. Results are measured every iteration. If the delivered results fail our customer's expectations, we get fired. Plans are guides. Results are our control.

Adaptability Is a Mindset

An exploration drilling project executive, deep in the darkest jungles of Indonesia, hits a 25-billion-barrel oil reserve. After a nervous $250 million investment, the find will be worth tens of billions of dollars in long-term profit. Oh, and the project was over budget by $50 million. Was the project successful? By most traditional project management measures, this project was a failure. If we really value our ability to create and respond to change, then executives and managers must attune themselves to different measurements of success.

Colleague Doug DeCarlo uses the term "Newtonian neurosis—an affliction born out of the mindset that projects should be stable and predicable." Since many projects are in reality moderately to severely "unpredictable," those afflicted with Newtonian neurosis attempt to "control" the uncontrollable by imposing processes, controls, ceremony, and better estimating—all of it futile.

If you read the last paragraph and your blood began to boil, Agile development probably won't work for you. But be careful what kinds of projects you attempt with your traditional software development and project management methodologies—don't go drilling on the frontier.

Adaptability and agility involve profound cultural changes, as readily seen in Dee Hock's use of the word "chaordic" to describe these adaptable organizations that are balanced on the edge between order and chaos. The Leadership-Collaboration model embraces Hock's chaordic idea. In this model, leadership replaces command, and collaboration replaces control. Managers who embrace the Leadership-Collaboration model understand their primary role is to set direction, to provide guidance, to facilitate the decision-making process, to expedite the connection of people and teams, and to figure out through thought and experimentation a simple set of generative rules.

Project managers who believe in predictability will control projects by conformance to plan. Project managers who believe in adaptability will control projects by demonstrating results and measuring business value along the way. Agile developers and managers have more confidence in their ability to adapt than in their ability to plan. Being Agile and adaptable isn't about practices, it's about deep-down fundamental philosophy. Predict or adapt—you can't have it both ways.

Chapter 16. Bob Charette

My favorite anecdote about Bob Charette, which indicates the depth to which Bob thinks about things, involved a discussion we had several years ago when he was constructing a new office. Bob was telling me about his new bookshelves. "Yes," Bob mused, "I had to reduce my collection to only 8,000 books." Bob is my favorite reference librarian—whenever I need to know about a subject, I fire off an email asking for references. Whether it's about the Austrian school of economic theory or decision making, Bob usually provides a thought-provoking list.

Of all the ASDEs, Bob's Lean Development has the most strategic focus. LD is derived from the principles of lean production, which restructured the automobile manufacturing industry in the early 1980s. In LD, Bob extends the traditional view of change as a risk of loss to be controlled with restrictive management practices to a view of change as producing "opportunities" to be pursued using "risk entrepreneurship." He views LD as an implementation vehicle for building change-tolerant organizations—a genesis that starts at senior executive levels.

Bob serves as a senior risk advisor to Global 100 CEOs, CFOs, and government officials worldwide on the effectiveness, impacts, rewards, and risks of their information and telecommunications systems. Bob chaired the IEEE committee for the "IEEE Standard for Software Life Cycle Processes—Risk Management," which was released early in 2001. I interviewed Bob in the lobby of Cambridge's University Park hotel during the 2001 Cutter Consortium Summit conference.

JIM: *What were the threads—the event history—of how LD evolved?*

BOB: LD has a funny story. I came at it from parallel paths. Part of it was the work I did in software engineering environments trying to understand how to better create software systems. One of the things that I found out—and this has been a hallmark of my whole career—is that I always thought that if I could get control of this one bit, life would be better. But there was always one additional bit that I needed to get control of.

LD came about as a direct relationship of my risk management work and saying, "Well, I'm interested in innovation, I'm interested in entrepreneurship." In the late 1980s, I was doing the research for my risk books and came across the Austrian School of Economics and the ideas of creative destructiveness, innovation, and uncertainty and their role in profitability.

JIM: *Coming from a software engineering background in Washington government circles, how did you get to innovation and entrepreneurship?*

BOB: In 1986, I was recruited by the U.K. government, moved to London, and got heavily involved in telecommunications. This was in the mid- to late 1980s, when the Bell breakup occurred, and the ideas of privatization, free market, and Reagan economics caused a lot of rethinking about economic models. My involvement with innovation came about in trying to discover a better way of doing things in the telecom business.

JIM: *What were some of your projects?*

BOB: I worked on the Department of Health and Social Security Operational Strategy, which at that time was the largest computerization project in the world. The U.K. government was trying to push the state of the art in computing.

I then went to work for a couple of international telecom companies that were going through a myriad of organizational changes. They had a lot of "start-up" money and basically had to compete for the first time. There was this ferment of innovation and strategy in IT because the telcos had to revamp all their systems simultaneously. These guys were making multi-billion-dollar bets—and you can have a lot of fun stuff with that type of cash being thrown around.

Part of LD came out of the idea of trying to build things for these telcos. Around this time, a company called Template Software [now Level 8 Software] was started. It developed an engine to create templates for workflow and process control. Template technology became another piece of my puzzle.

Telecommunications, Austrian economics, innovation, and then came the idea of lean production popularized by James Womack. I had been looking at the idea of discontinuities: How did social, business, or technology discontinuities happen? How do you challenge assumptions? I had been collecting stories on why people couldn't recognize the obvious, at least the obvious after you recognized it. From this came a core element of innovation: setting "stretch and leverage" targets by challenging assumptions. I was enamored with how the Japanese lean manufacturers challenged their assumptions, set up these outrageous goals, and worked hard to make them happen.

To be honest, I can't tell you what finally crystallized LD. Part of it was that I wrote several articles for a magazine in London called *Software Management*. I wrote about application templates and application strategies and the idea that, "What we ought to do is marry the ideas of innovation from lean manufacturing with software development." In the early '90s, I was reading the same books you were in terms of chaos and complexity theory and looking and saying, "Well, maybe there is something here. What can we do to create a process that synthesizes all these different things together?"

So all these ideas were floating around, and I pulled them together into LD. The Template Software guys were able to show that their templates were able to speed up system development by 20 to 40 percent, but they had no process or methodology. I thought the combination of templates and LD would be powerful.

JIM: *How would you characterize templates as being similar to or different from components?*

BOB: They are at a higher level. Templates are domain-oriented functions: process control, workflow, eCommerce, customer care, or billing. One of the biggest problems is being able to describe templates.

A spreadsheet is a template. It allows you to create an infinite number of systems out of high-level parts. You use the template like a spreadsheet and then add application-specific code to it. So a

template can be considered an 80 percent complete application like a spreadsheet, which might be 80 to 90 percent of a complete financial analysis system.

JIM: *Much of your emphasis is at the strategic, top-down level. Why?*

BOB: I took this top-down viewpoint because I soured on trying to get things done bottom-up. It was part of the emergent principle that you needed to have a commitment across the organization. You couldn't just pick and choose elements of LD but had to have the whole thing to get the benefits. I never felt you were going to make significant change—one-third the defect rate, one-third the cost, one-third the schedule—without top-down support. I take a dogmatic approach in saying that LD is about getting all three of these simultaneously. Not one of the three, not two of the three—LD says, "How do you get all three of them?"

JIM: *So have you had any better luck with top-down than with bottom-up?*

BOB: I think the results are about the same—they both stink. We've had a lot of interest over the years from people who are at a project level who want to know about LD. Unfortunately, when we talk to them, they say things like, "Well, we can't do this. We can't do that. We can't talk to the customer." And after a couple of "We can'ts," you say, "Well, don't try LD." I think the cynic in me tries the top-down approach. Maybe it is just a matter of getting older.

I am also interested in the economic side. Let's see if we can build a system that has very little investment cost—very little in terms of what I call the "anchor effect," where people feel that they have to keep an IT system around for a long time because they have sunk all this money into it. If I am creating systems that have to be around, then I start putting constraints on managers. All of a sudden, those things start making me change intolerant. So from a corporate standpoint I say, "Well, let me build the things I need to do to satisfy my customers extremely well, but don't lock me into a corner where IT becomes this gigantic anchor because I've spent a million dollars on this bloody thing."

As an example, I think about those companies that are saddled with their ERP systems for the next 20 years. They will never recover the half-billion dollars that the systems cost to install, and they can't even consider doing anything else.

JIM: *The people at Dell spent $200 million and then decided their ERP system wasn't going to be flexible enough for them and canned it and started their own.*

BOB: Here's a company that shows management and risk savvy, recognizing that there is a sunk cost and you aren't going to recover it. Wisely, it moved on to something else.

JIM: *Tell me about EuroTel [a fictitious name; the EuroTel case story begins* Chapter 21*].*

BOB: I really underestimated the disincentives for LD. To be honest, I probably could have thought about it for a long time and still not figured out the sources of resistance. For instance, the marketing people did not want us to get close to the customer. They "owned" the customer, and they thought that the software guys had no business talking to the customer.

JIM: *Did you have difficulty with domain specifications?*

BOB: Probably the hardest thing was that people weren't used to thinking in terms of domains. We had two divisions, communications and transport, that were doing nearly identical bus systems. You would think that one was building a bus system and the other a people mover. Neither group saw the commonalities. This is a weak point of LD, getting people to recognize domains.

We had a hard time getting people to think, "Well, if you can use this template on the bus system in Milan, you could probably use the same template in Berlin." Sadly, they believed the Berlin buses were unique—no similarities between the two bloody projects. They weren't bus systems. To the guys in Milan, it was the Milan bus system, and to the guys in Berlin, it was a Berlin bus system. I said, "Well, you both use buses, right?" Their response, "Well, you don't understand; our buses use electricity. Theirs don't." "Yeah, but they still go from point A to point B, right?" "Yeah, but you don't understand. There is Italian writing on our side; theirs is in German."

I am exaggerating a bit, but people always look for uniqueness. One of the weaknesses of our field is that we keep looking for uniqueness. Our requirements and our specification processes are geared to look for unique bits, the areas of potential conflict. When we ask people to look for common stuff, they don't know how to do it.

JIM: *What problems did you have after the first group of successful projects at EuroTel?*

BOB: They tried to kill us under the guise of verifying that we were actually getting what we said we were getting. We were highly criticized for meeting with the customer and having the customer as part of our team. Basically they [the central software research group] said, "A lot of your results are because you get all this feedback from the customer and you get it quickly, and they are involved and they help you with requirements. That's not fair. Most of the projects don't get that advantage." They wanted us to be able to create systems that worked well for the customer without ever working with the customer, because that was their baseline.

The baseline we were measured against was to create great systems, get productivity improvements, without the bother of having to talk to the customer. They wanted us to be mind readers. They had this model, "Well, our organization doesn't talk to customers, so you have to come up with a process that can give us all the benefits you claim, but you can't talk to the customer." And you start thinking, "Well, I'm not Harry Houdini. I don't have the ability to escape out of every trunk." But that is what they wanted. They wanted a Houdini.

JIM: *I worked with one project team that was very successful in the face of terrible constraints and fluctuating requirements. It was a successful project, but they demoted the project leader. His boss said, "You succeeded. This was a very successful project, but you didn't play nice according to the organization norms, and that's more important than success."*

BOB: Right. And I think that is the issue that you find in all the stuff that we are doing. I find it in risk management all the time. People want, for some unknown reason, for us to come up with the benefits based on all their constraints. Give me some magic bullet that will cure my ills without me changing my approach. It is not so much silver bullets, it's the magic pill.

JIM: *So this group was saying that the fact that you were able to talk to your customers was an unfair advantage and you shouldn't be given credit for what you did?*

BOB: Yeah, and that we used templates. "Using templates, that's not fair either! You've got 80 percent of your solution up front." They said, "Well if we did that too, we would be able to get those numbers too." And we responded, "Yeah, that's the point." I felt like I was in this land of the absurd.

JIM: *The comparison I make sometimes is with Southwest Airlines. Other airlines say, "If we had only one kind of plane, we could deliver low-cost service too." Duh. It's not like Southwest didn't actually plan it that way.*

BOB: We got beat up unmercifully for talking to the customer, for using domain analysis, for having templates, for having short build cycles, for not having to worry about reams and reams of tests. That was another thing, "Well, we want you to test out the template." We would respond, "Why? It already works." "Well, we want you to test out the template, because it is part of the

software. Our procedures say you've got to have tests for all the software." It took us forever to convince them we didn't need to do that. We would ask them, "OK, well, every time you use an Excel spreadsheet, you test it, right? How can you crunch your numbers without testing out that Excel spreadsheet every single time you use it?" The idiocy we encountered!

JIM: *So, where is LD now, and where would you like it to go?*

BOB: Well, thanks to you, LD is coming out of its stealth mode. I talk about LD to certain clients, but I am very careful because (this may sound cynical) probably only ten or twenty companies in the U.S. could make use of it, given all the constraints I put on it. I believe that it is a very viable approach for the right company. I think that there is just a tremendous ability to create very high-quality systems, very quickly, and very inexpensively.

Now, what would I like to see? I'd like to see people using LD more. LD-2 is going to be a lot better than LD-1 in terms of pulling in your adaptive and Kent's XP practices at the development level. One of the things about LD is that you can build fairly big systems. It is not just for a five- or ten-person project—it can be scaled up to a large size.

JIM: *What gives it the ability to scale?*

BOB: Part of it is the template capability, being able to work at a high level of granularity. I believe that SNAP, the code engine that Template Software created, had close to a million lines of code, and the generic templates were large as well. Really robust systems could be created from these templates.

JIM: *What are the selling points for LD?*

BOB: All the economic arguments are extremely solid, but there is something holding it back. When I coined the term "LD," I did it on purpose, because senior executives knew about lean manufacturing. I figured that the next step was doing the same with software. Now it may be that people are five years behind where I expected. I think for LD, and even the Agile stuff, that we are ahead of most people's idea curve.

JIM: *When I look at the technology adoption curve, I think we are still in the early adopters stage, and we're still approaching Geoffrey Moore's chasm.*

BOB: I think you are right, and I should have known that when I was doing LD. Unfortunately, I have this bad habit of being out there in the wilderness saying, "Hey, this is really obvious. Hey, everybody, this is obvious. Isn't it obvious?"

JIM: *In some ways, LD is a tough concept.*

BOB: I agree. I think there is, thanks to Kent and yourself, likely to be more interest in LD as people start to say, "How do I scale up? How do I go to the next steps?" The notion of emergent properties, the idea of organizations as organisms—those things ring true with a lot of individuals within companies.

I think the risk entrepreneur question comes down to, "How are you going to innovate?" My question to the senior managers is, "Hey, if you really claim that you have to be quick and you can't waste money, then how are you going to streamline the process so you can do this?" You [Agile leaders] are saying, "Here is a process that you can streamline that makes business sense." But you are coming bottom-up. I am coming top-down, and I think maybe the merging of the two is key.

JIM: *Having spent a lot of years in the defense business, how do Agile approaches fit, or not, within frameworks like the CMM?*

BOB: LD would never fit into the CMM as most people perceive it. There are a number of people who believe that you have to have a single process in your organization to meet the CMM. In an innovative organization, you might have Lean, Extreme, Iterative, Adaptive—six or seven of these buggers. A good organizational model ought to have a whole palate of these different approaches to use depending on the circumstance. Instead, most organizations say, "Ah, we want to standardize on one, because that is the only way we are going to meet the CMM." Again, we got into this discussion/argument/idiocy at EuroTel, where we went in and said, "Hey, for 60 percent of the software you do across the company, LD is a terrific fit." It wasn't good enough. They wanted a total solution. They wanted a software process that could go across the board—from developing telephone real-time switch software to manufacturing software. They were willing to wait for a final solution rather than to make incremental changes that could help them.

JIM: *So, it was the software R&D group that was the problem?*

BOB: It was R&D and business-unit people. There was this pervasive idea that they didn't want to manage a diverse set of tools and techniques. Both the R&D and business-unit people felt that the easiest way to get CMM certification was to reduce tool sets, reduce processes, reduce, reduce, reduce. Moreover, they told senior management that the CMM could save money by consolidating all their processes and tools.

So we come along saying, "Hey, you know all that work that you did? It is really good, but what you really need to do is start expanding your tool set again." So I think the CMM has been abused, because to meet the CMM, you consolidate. It is extremely hard to get out of that box, because people are worried about losing accreditation. I don't think that the SEI lets them know that they are not going to lose it.

JIM: *It is interesting what they are saying, "Well, we can accommodate XP or Agile development within the CMM." As opposed to saying, "Yeah, CMM is good for* this *stuff, and Agile is really good for* that *stuff, but they are separate. They've got some different philosophies, and we don't need to have the CMM subsume them."*

BOB: Yeah, but I think some in the SEI are worried that the Agile approaches will make the CMM largely irrelevant.

Reflections

For those who think ASDEs are only about programming, a few hours with Bob will disabuse them of the notion. Bob's ideas about risk, economics, lean production, risk entrepreneurship, and change tolerance are both Agile and strategic for large, multi-national enterprises.

Part III: Agile Software Development Ecosystems

Chapter 17. Scrum

Scrum relies on self-commitment, self-organization, and emergence rather than authoritarian measures.

—Ken Schwaber

In 1996, Ken Schwaber wrote an article for the *Cutter IT Journal* (then *American Programmer*) titled "Controlled Chaos: Living on the Edge." Even at this early date, Ken brought an understanding of complexity concepts to software development and project management (Schwaber 1996).

Ken, an expert in rigorous, process-centric methodologies in the late 1980s and early 1990s, began to realize that the increasingly detailed and specific methodologies—overburdened with phases, steps, tasks, and activities (with documents to support each)—had a fundamental flaw. "The core of the Scrum approach is the belief that most systems development has the wrong philosophical basis," Ken says. He asserts that software development is not a "defined process," as rigorous methodologies assume, but an "empirical process."

In his investigation of industrial process control, Ken discovered that the differences between defined and empirical processes were not only profound, they required a completely different management style. A defined process draws heavily on fundamental physical and chemical laws that "define" the transformation of inputs to outputs. Defined processes can be reliably repeated time after time with little variation. An empirical process doesn't conform to first principles (scientific laws), and the input-to-output transformation is complex and variable. Empirical processes cannot be consistently "repeated," and therefore require constant monitoring and adaptation. "Developers and project managers are forced to live a lie—they have to pretend that they can plan, predict, and deliver," says Ken. People build PERT charts in Microsoft Project knowing that's not how the work will actually be done, and then are constantly asked to explain deviations.

I heard a similar refrain working with a systems integration firm in New York. As one of the company's sales managers said, "We've fallen into the linear, process-centric approach with a heavy emphasis on up-front requirements documentation. We're not happy with the results, the customers aren't happy with the results, but we can't seem to get out of the rut because this is the approach that's professed to be industry best practice."

Scrum starts with the premise that we live in a complicated world and, therefore, "You can't predict or definitely plan what you will deliver, when you will deliver it, and what the quality and cost will be," says Ken. You can, however, bound the empirical process with explicit monitoring criteria and manage the process itself with constant feedback mechanisms.

Whereas XP has a definite programming flavor (pair programming, coding standards, refactoring), Scrum has a project management emphasis. Scrum has been applied mainly to software projects, but a number of nonsoftware projects have also been managed with Scrum—the principles are applicable to any project. A "scrum" in Rugby occurs when players from each team huddle closely together and clash with the players from the opposite team in an attempt to advance down the playing field. For those who have visited New Zealand, where Rugby is a national passion, or have otherwise had the opportunity to witness a Rugby game, controlled chaos seems an appropriate term. "Both are adaptive, quick, self-organizing, and have few rests," says Ken.

Controlling a chaordic (empirical) process and controlling a predictable (defined) process require different techniques and changing one's view of "control." Traditional project management practices—which have evolved from a hierarchical, Command-Control perspective on

management—assume that projects are predictable (because some manager declared them to be) and that variations from the "plan" are caused by people shortfalls such as lack of motivation or poor execution. Scrum (and other ASDEs) views the work to be accomplished as unpredictable and operates on the assumption that the people are doing the best they can under the circumstances. Therefore, the project management emphasis is on improving the "circumstances" to the greatest degree possible, monitoring the features being delivered, and constantly adjusting. The circumstances, in a Scrum project, involve facilitating the interaction of the team members based on the belief that communication, collaboration, coordination, and knowledge sharing are key to delivery.

The Scrum Process

Scrum defines a project management framework in which development activities—requirements gathering, design, programming—take place. In Scrum, the development period is a 30-day iteration called a Sprint, as shown in Figure 17.1. The Scrum framework has three components: Pre-Sprint, Sprint, and Post-Sprint. The focal point is the 30-day Sprint, in which working software gets developed.

Figure 17.1. The Overall Scrum Process (Courtesy of Ken Schwaber)
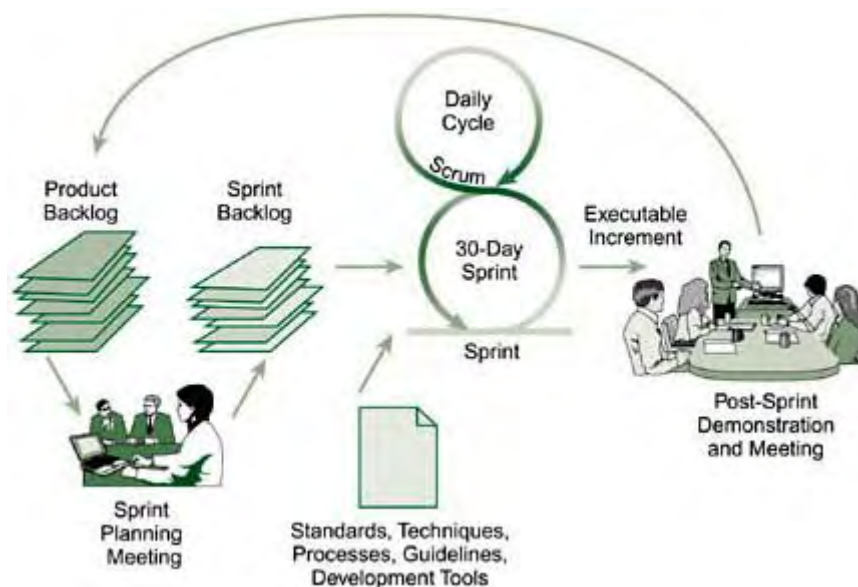
Pre-Sprint Planning

Scrum teams use a series of "backlogs" that identify product features and are used to manage the project. The Product Backlog contains a list of all business and technology features envisioned for the product. Scrum, like ASD but unlike XP, places customer functionality *and* technology features into the backlog. For example, a feature to implement transaction processing middleware could be part of the Product Backlog. The Product Owner is the person (or team of customers) who creates and prioritizes the Product Backlog. (In a product company, this would probably be a product manager.) Technical staff and managers also contribute potential "features," and technical management may be part of the Product Owner team. The Product Backlog is the starting point for two other "backlogs," the Release Backlog and the Sprint Backlog. At the beginning of the project, the Product Owner selects the features that should be in the next product release—the Release Backlog—which is a subset of the Product Backlog.

The Sprint Backlog identifies and defines the work to be accomplished by the development team during a 30-day Sprint—it is a subset of the Release Backlog. At one level, the Sprint Backlog identifies the features, while at another level, it defines the tasks required to implement those features. The Sprint Backlog is developed in a planning meeting that includes the development team, management, and the Product Owner. (Depending on the project size, there may be two meetings: one to select features and another to plan the detail tasks.) In the meeting, the Product Owner decides which features are required for the next Sprint, and the development team figures out the tasks required to deliver those features. The development team determines the task list and then estimates how much can be accomplished during the Sprint. The development team and the Product Owner then cycle through the process until the planned features fit with the available resources for the Sprint.

One final piece of the planning process is to develop a Sprint Goal, a business purpose for the Sprint. A Sprint Goal can be achieved even if some of the detail features or tasks are delayed or dropped. "The reason for a Sprint Goal is to give the team some wiggle room regarding the functionality," say Ken and his coauthor Mike Beedle.[1] Similar to a "cycle objective" in ASD, a Sprint Goal constantly reminds the team what the detail tasks should achieve. Without this goal, the team can become overly focused on tasks and lose track of *why* the tasks are being performed. In addition, keeping the goal in mind encourages the team to adapt to conditions that may arise over the course of the Sprint.

[1] For an in-depth look at Scrum, see *Agile Software Development with Scrum* (Schwaber and Beedle 2002).

Sprint

The rules for a 30-day Sprint are simple: team members sign up for tasks, everyone works hard to accomplish the Sprint Goal, and everyone participates in a daily Scrum meeting. "Teams let themselves be guided by their knowledge and experience, rather than by formally defined project plans," says Ken. There aren't any elaborate plans during a Sprint—team members are expected to use their talents to deliver results. As plans are used as guides, Scrum epitomizes a conformance-to-actual approach to project management. One thing that makes Scrum a powerful approach is that it explicitly reduces time fragmentation and constantly shifting priorities by "locking" features for the 30-day Sprint. Except in dire situations, priorities don't get changed during a Sprint—by the Product Owner, by management, by anyone. At the end of a Sprint, the Product Owner could choose to throw away all the developed features or reorient the project, but within a Sprint, priorities remain constant. This is the "control" part of "controlled chaos."

Why is this so critical? In an environment of constant change, there has to be a point of stability. *Everything* can't change all the time. There needs to be a zone of stability so that the members of the development team can feel that they can accomplish something. Hence, other than in exceptional situations, no changes to the Sprint Backlog are allowed during the 30 days. A key function of the "Scrum Master" (who may be a consultant, coach, or project manager) involves fending off changes during a Sprint to protect the team from getting sidetracked. This no-change rule may be difficult to implement in organizations in which managers, customers, or marketing staff are conditioned to making changes on a daily basis.

The daily Scrum meeting energizes a Sprint. Principles that say "improve communications" or "get customers involved" are as old as projects. Project managers can't control these interactions, but they can influence them and create an environment that encourages working together. Unfortunately, many of our experiences with communications meetings are less than stellar. Long meetings and poor results are too often the norm. One reason for poor results is the failure to differentiate between information meetings (status checks, issue raising, etc.) and development meetings (two or more people get together to solve a problem, develop code, create a test case, etc.). Daily Scrum meetings are informational (they both inform and coordinate) and are conducted accordingly.

Because empirical processes are defined by their uncertainty and change, it is important that checks be exercised daily. Defined processes (for which the outcome is relatively predictable) can be managed with infrequent feedback, but empirical processes require closer attention. The daily Scrum meeting enables the team to monitor status, focus on the work to be done, and raise problems and issues. Scrum meetings:

- Are held at the same time and place every day
- Last less than 30 minutes (the target should be 15 minutes)
- Are facilitated by the Scrum Master
- Are attended by all team members (developers, users, testers, etc.)
- Are attended by managers to keep track of status but not to participate
- Are used to raise issues and obstacles but not to pursue solutions
- Allow each participant to address three questions:

  - What did you do since the last Scrum?

  - What will you do before the next Scrum?

  - What impediments got in the way of your work?

Daily software builds are used to raise the visibility of development work and ensure that code modules integrate. Daily Scrum meetings serve the same purpose for people—raising the visibility of each person's work (to facilitate knowledge sharing and reduce overlapping tasks) and ensuring that their work is integrated. If daily builds are good for code, then daily "builds" should be even better for people.

A Scrum meeting is the type of practice that generates ripples in an organization. If the Product Owner and management accept the idea of no changes during a Sprint, they are then encouraged to put a little extra thought into Sprint planning. Yet they know things can change again in 30 days, so they aren't trying to predict or plan for the entire project, just the next 30 days. Everyone begins to get into a certain rhythm, a certain pace, that becomes very productive. People understand both the impact of their decisions and the fact that they can't escape from them.

When a team begins using them, daily Scrum meetings may feel forced, but very quickly people respond positively because they find these short meetings efficient and effective. The meetings eliminate the need for other meetings and help the right people to team up to resolve issues. The Scrum Master's main responsibility becomes removing the impediments to getting the work done that are raised during the meeting. Impediments can be logistic, such as lacking equipment or supplies, or organizational, such as slow approvals or inadequate user participation. Ken uses Scrum to break down an organization's established norms and attune it to higher speed and throughput. This "interference" can often make the Scrum Master unpopular.

Some XP proponents, Ken Auer for one, advocate a daily stand-up meeting that is similar to the daily Scrum meeting. Auer goes so far as to say that the daily meeting should be the thirteenth practice of XP (Auer and Miller 2002). While several of the Agile approaches recommend daily meetings, Scrum defines in great detail how to conduct them effectively.

Post-Sprint Meeting

At the end of a Sprint iteration, a Post-Sprint meeting is held to review progress, demonstrate features to the customers (similar to the customer focus group sessions recommend in ASD), and review the project from a technical perspective. At the conclusion of this meeting, the Scrum process is repeated—starting with a planning meeting for the next Sprint.

Monitoring Progress

Jeff Sutherland, who codeveloped Scrum with Ken Schwaber and has been applying it to real projects for years, has refined Scrum monitoring to a reasonable daily administrative overhead—one minute per day for developers and ten minutes per day for the project manager!

Scrum uses a very simple but powerful tool to monitor project progress: a Sprint Backlog Graph. The graph plots days on the horizontal axis and work remaining in hours on the vertical axis; at the end of 30 days, the work remaining should be zero. So at day zero, the work remaining would be the total number of hours estimated for the Sprint Backlog features. Each day, in Jeff's system, developers record the days (hours) invested in a task and its percent completion. His automated backlog tool calculates work completed and work remaining. If a developer has a two-day task that, once started, becomes a three-day task, the work remaining would go up the next day. By watching the Sprint Backlog Graph, the project leader has daily feedback on progress (or lack thereof) and any estimates that have proved to be inaccurate.

However, Mike and Ken point out, "This should not be confused with time reporting, which is not part of Scrum. Teams are measured by meeting goals, not by how many hours they take to meet the goal" (Schwaber and Beedle 2002). Managers are looking at the overall trends on the Sprint Backlog Graph, not micro-managing task estimates and completions.[2]

[2] In this respect, monitoring in Scrum projects is similar to that used in Eliyahu Goldratt's Critical Chain project management.

Scrum's Contributions

In the July 1, 2001, issue of *CIO* magazine, there was a article about Agile Software Development that unfortunately included the terrible subtitle "Now Agile Development is promising nothing less than 100 percent success" (Berinato 2001). In contrast to this hype, Ken provides a great perspective on Scrum, which extends to all ASDEs. Figure 17.2 provides his philosophical view of the advantages of Scrum. It is one of my favorite charts, because it candidly admits that projects of medium to high complexity have a significant possibility of failure—*no matter what approach is used*. What the figure shows is that at lower levels of complexity, the probability of success using an ASDE may not be any greater than with a rigorous methodology. However, as the complexity increases, Agile approaches have a higher probability of success. ASDEs can be used with many types of projects, but they shine brightest when confronting complexity.

Figure 17.2. Complexity/Success Graph (Courtesy of Ken Schwaber)

Complexity/Success Graph—Scrum
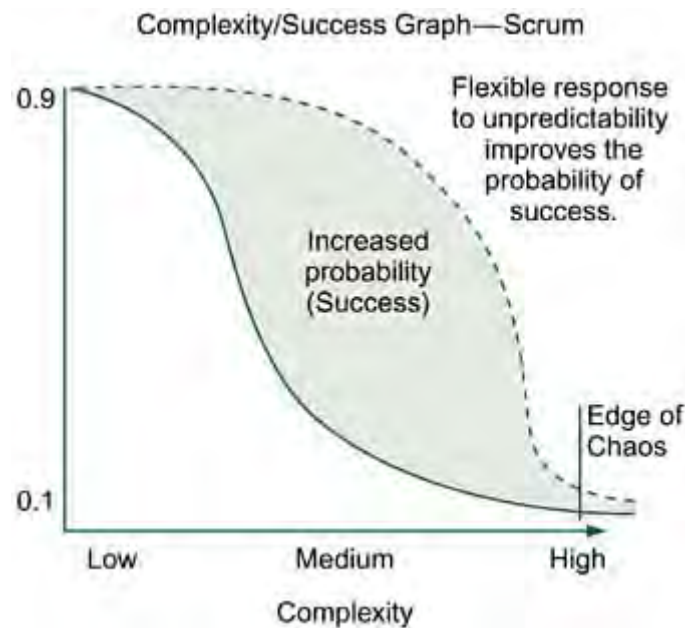
From a conceptual perspective, Ken's ideas about empirical and defined processes contribute a great deal to understanding why rigorous, heavy processes don't fulfill their anticipated performance objectives. Defined processes depend on repeatability, an impossibility for processes besieged by noise on the one hand (constant changes, diverse people, knowledge overload) and a lack of formulaic transformation (no formula for translation from inputs to outputs) on the other. Empirical processes can spin out of control quickly, so the key to success is constant monitoring via the daily Scrum meeting and Sprint Backlog Graph, while at the same time facilitating the "chaordic" creativity required to solve complex problems. The concept of a daily meeting as a coordination and collaboration activity is a critical element that practitioners of any Agile approach should strongly consider. I recommend daily Scrum-type meetings to all my own clients.

Can project management be this easy? No, it's not easy, but it can be simple—and hard. It's easy to identify hundreds of product features—but hard to prioritize those that contribute to a product vision. It's easy to spend days generating page after page of PERT charts, task descriptions, resource allocations, and task dependencies—all built upon the quicksand of the underlying uncertainty—but hard to admit what we don't know and devise simple daily controls to determine progress toward a short-term goal. It's easy to fragment staff time, multi-tasking during the day to address the hot issue of the moment—but hard to focus on a single set of features and not become distracted. It's easy to succumb to organizational procedures, policies, standards, and expectations—but hard to challenge those same things when they become impediments to success.

Scrum says, "Do these few things well, and projects will succeed," and furthermore, "If you don't do these few things well, it doesn't matter how many other hundreds of things you do well, you won't succeed." Scrum helps us focus on the key project management activities that we must do well, the really difficult things, the really politically difficult things, including hard decisions. If we can't focus on these difficult issues, then spending enormous amounts of time on the other issues won't matter.

This is why, if you want to try Scrum, don't start out with an initial "learning" project that is of marginal importance. Start on a project that is absolutely critical to your company; otherwise it will be too difficult to implement all the hard things Scrum will ask of you. If you have an absolutely critical project—one that reeks of uncertainty and complexity—try Scrum. It will increase the odds of success, or at least give you the information you need to cancel the project early. If you find that your organization can't make the hard decisions that Scrum demands, then high-risk, uncertain projects have very little probability of success in your organization.

Chapter 18. Dynamic Systems Development Method

Dynamic Systems Development Method is a formalization of RAD practices that arose in the early to mid-1990s. DSDM originated in England, has become popular in Europe, and has a number of sites using the approach in the U.S. DSDM has proven to be a viable ASDE for many companies.

DSDM, like all Agile approaches, has gained from experience over the years. While the acronym remains the same, the words and meaning have changed, according to Dane Falkner.[1] The first "D" in the DSDM acronym, for "dynamic," reflects the ability to adapt to on-the-fly changes. Today, the "S" reflects a focus on business "solutions" rather than "systems." DSDM is focused on customer solutions and business value. The second "D" in DSDM reflects "delivery," a broader concept than "development," and indicates the importance of product deliverables (features or stories) rather than traditional tasks. Finally, the "M" may be more appropriately represented by the word "Model" than "Method," reflecting a business perspective on projects. So today's DSDM would be a "Dynamic Solutions Delivery Model," at least from Dane's point of view.

[1] Dane Falkner is the chair of the North American DSDM Consortium (in 2001) and president of Surgeworks, a U.S. company that provides DSDM training and consulting services.

Arie van Bennekum, of Solvision in Holland and a member of the DSDM Consortium board of directors, made a quick two-day trip from Holland to Salt Lake City for the initial Agile Alliance meeting. Prior to the meeting, several of the alliance participants knew about DSDM, but few really understood how closely DSDM and Agile principles aligned.

Arie described a project his company implemented for the governmental organization RDW (RijksDienst voor het Wegverkeer en centrum voor voertuiginformatie), which is responsible for the administration and issuing of license plate documentation. RDW has an 80-person help desk operation to answer questions about lost licenses, new vehicles, import of vehicles, and more. The help desk operators receive about 2,400 calls per day. The calls took excessive time because of an inefficient set of systems providing information from more than ten back-end and seven front-end applications.

The use of the DSDM practice of facilitated workshops (referred to by others as JAD sessions) turned the project around immediately. "The major event in this project was the first workshop," said Arie. "Here the developers showed 'their' solution, which was totally rejected by the involved users. The workshop was put on hold and, after a 30-minute break, we restarted from scratch and developed a model that totally satisfied the need and expectations of the end-user organization." DSDM emphasizes the use of facilitated workshops in both the Business Study and Functional Model phases to involve the key customers and users in getting the project initiated properly.

Arie van Bennekum

I met Arie at the Agile Alliance meeting at Snowbird and then corresponded with him about his work with DSDM and how he gravitated to DSDM from rigorous methodologies.

JIM: *What were the historical threads in your work that motivated you switch to DSDM?*

ARIE: It was not as much a thread but more a way of work that did not seem logical at the time. Companies were using large, top-down designs that were not understood and accepted by the end users and, above all, did not really benefit the actual business needs. Questions were raised mainly

about the "why" in the approach. Seven years ago, I got in touch with James Martin Associates and was trained on rapid application development. This was an eye-opener to me.

JIM: *What is the vision or purpose that drives your work?*

ARIE: My primary objective is to be worth the money I charge. This means that I have to deliver added value to my customers. DSDM is focused on business benefits and avoids doing redundant work. This means that I minimize costs and try to achieve a maximum business benefit. It also creates a lot of joy for the people in the teams. This is something that is often not recognized by organizations. Employee satisfaction guarantees commitment, continuity, and quality.

JIM: *What are the principles in DSDM that you think are the most important?*

ARIE: I especially value the self-steering aspect of the team within high-level baselined requirements. People get committed by having responsibilities. Everybody should do what he or she is good at. This creates maximum speed, commitment, and synergy.

JIM: *What do you think differentiates Agile approaches from rigorous ones?*

ARIE: The main differentiators are also the opportunities for Agile approaches as a shared holistic approach. As a whole, they are different from traditional approaches in their flexibility to deliver the *right* software instead of delivering software that was once specified somewhere by people who are not really into the day-to-day business where it is going to be used.

JIM: *What do you think are the biggest challenges in organizational acceptance of Agile approaches?*

ARIE: I think the idea that up front it is not clear what functionality is going to be delivered. Empowering end users within a high-level baselined business objective is a second key idea.

DSDM Principles

DSDM clearly evokes the ideas of an exploratory development approach. Early in the DSDM manual, it stresses that "nothing is built perfectly the first time," reiterates the old 80–20 rule (the last 20 percent can be very time-consuming), stresses that systems users cannot foresee all their requirements in the beginning, and recommends an iterative approach in which "the current step need be completed only enough to move to the next step." The nine principles of DSDM reflect concordance with principles of the Agile Manifesto.

1. Active user involvement is imperative.
2. DSDM teams must be empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are baselined at a high level.
8. Testing is integrated throughout the life cycle.
9. A collaborative and cooperative approach between all stakeholders is essential.

Most of these principles are close enough to the Agile Manifesto principles that further explanation shouldn't be required, except possibly for number 6. Principle 6 calls for consistent and pervasive configuration management such that, at any point, changes made in documents or

code could be rolled back to a prior state. "It is almost like I have given the team (users and developers) a potion of courage," says Dane in an email note. "In fact, I call principle 6 the courage principle. People are not afraid to make mistakes [as they can be quickly undone], so they make decisions more rapidly."

The DSDM Process

Figure 18.1 shows an overview of the DSDM development process. Each of the major phases— Functional Model iteration, Design and Build iteration, and Implementation—are themselves iterative processes. DSDM's use of three interacting iterative models and time-boxes (for both iterations and releases) can be confusing initially, but once the definitions are sorted out, they can be used to construct very flexible project plans.

Figure 18.1. The DSDM Development Process (Courtesy of Surgeworks)
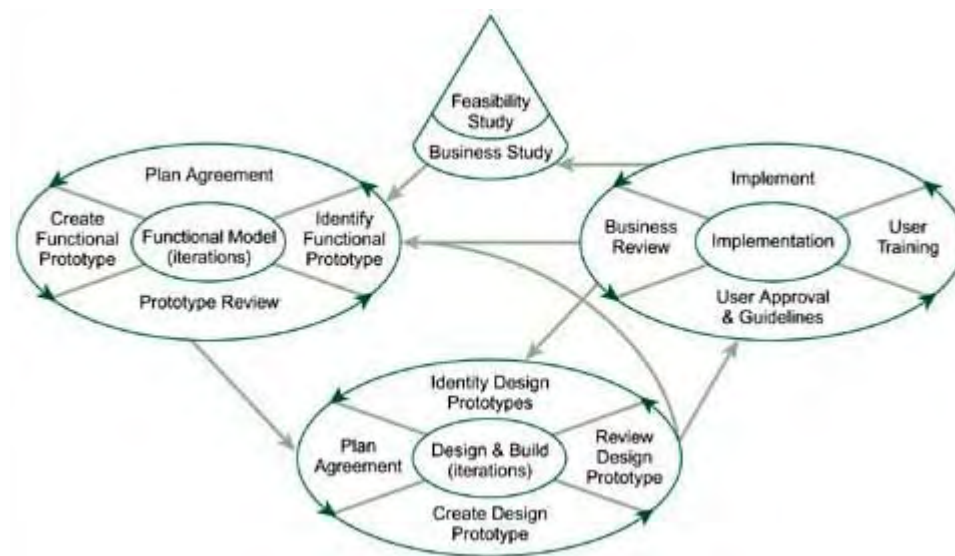
The Functional Model iteration is a process of gathering and prototyping functional requirements based on an initial list of prioritized requirements. Nonfunctional requirements are also specified during this phase. Most requirements are documented in prototypes rather than in text. The Design and Build iteration refines the prototypes to meet all requirements (functional and nonfunctional) and engineers the software to meet those requirements. One set of business functions (features) may go through both Functional Model and Design and Build iterations during a time-box, and then another set of features goes through the same processes in a second time-box. Implementation deploys the system into a user's environment.

Although in Figure 18.1 the Functional Model iterations may appear to be complete prior to the Design and Build, in practice they can overlap considerably. These iterations of Functional Model and Design and Build can also fit within a lower-level time-box (two to six weeks in length), such that the differences between the DSDM life cycle and other ASDE life cycles is less than initial observations might suggest.

DSDM does use a variety of prototypes: business, usability, performance, and capacity. While instances of these prototypes could be compared to XP's use of "spikes"—quick code development to investigate an area of uncertainty—in general DSDM relies more on prototypes than the other ASDEs. But here again, the actual current practices of DSDM vary from its historic roots. Today, while the label "prototype" remains from previous usage, in actual projects, the prototype is more

likely to be code that will be refined into a finished product than traditional prototype code that is thrown away. So in practice, DSDM prototypes are closer to the Agile definition of working code than the word "prototype" might indicate. DSDM's performance and capacity prototypes would be comparable to an XP spike—potentially throw-away code designed to quickly test some aspect of the system's performance.

DSDM also addresses other issues common to ASDEs. First, it explicitly states the difference between DSDM and traditional methodologies with respect to flexible requirements. The traditional view, according to the DSDM manual, is that functionality stays relatively fixed (after it is established in the original requirements specifications), while time and resources are allowed to vary. DSDM reverses this viewpoint, allowing the functionality to vary over the life of the project as new things are learned. However, while functionality is allowed to vary, control is maintained by using time-boxes.

DSDM also addresses the issues of documentation, or lack thereof—a constant criticism of ASDEs. Because one of the principles of DSDM relates to the importance of collaboration, it uses prototypes rather than lengthy documents to capture information (that is, it uses the Functional Model rather than a functional specification). DSDM recommends, however, that "local practices should be followed but perhaps scaled down to minimize unnecessary barriers to rapid movement through the project." Consequently, DSDM recommends only 15 work products from its five major development phases, and several of these are prototypes. There is an interesting comment in the DSDM white paper on contracting—"The mere presence of a detailed specification may act to the detriment of cooperation between the parties, encouraging both parties to hide behind the specification rather than seeking mutual beneficial solutions"—that mirrors the observation in Chapter 9 that documentation may create barriers to conversation. After all, once a detailed requirements specification has been completed, customers can view their participation as no longer needed, while developers can always point to the specification and say, "But the spec said that this was how we should do it." Not only can face-to-face conversation be the most powerful method of gaining and maintaining a common understanding of the requirements, but as the above statement cautions, too much documentation can give both parties an excuse to discontinue dialogue.

In keeping with its minimalist approach, DSDM identifies only 11 roles within a project (although there are also additional specialist roles possible). While many of these roles are the typical ones—project manager, team leader, and developer—there are a couple of interesting subroles to the usual "user" role. DSDM identifies user roles as "visionary," "ambassador," and "advisor." The ambassador user represents the whole user community, bringing in people with specific skills as needed. While the ambassador user should understand the business process and goals of the business process being automated, the visionary user makes sure that the high-level intent and vision for the product are not lost. The advisor user role brings day-to-day knowledge of business details to the development team. This advisor user would be similar to XP's on-site user, provided he or she had the appropriate decision-making powers.

With respect to work products, DSDM, unlike rigorous methodologies, doesn't offer detailed documentation formats for its 15 defined work products. Instead, the DSDM work product guidelines offer a brief description, a listing of the purposes, and a half-dozen or so quality criteria questions for each work product. Since exact document contents are customized by organizations and project teams, this approach seems very appropriate. Organizations can supplement the DSDM work product guidelines with more useful sample documents.

Another area that DSDM focuses on is establishing and managing the proper culture for a project. The manual describes, for example, the different emphasis of project managers and points out how difficult the transition can be for project managers steeped in traditional approaches. A couple of passages from the DSDM manual illustrate this point.

A traditional project manager will normally focus on agreeing to a detailed contract with customers about the totality of the system to be delivered along with the costs and time scales. In a DSDM project, the project manager is focused on setting up a collaborative relationship with the customers.

In the traditional project, the manager is concerned with understanding the requirements in complete detail. In the DSDM project, the manager is concerned with agreeing with the users on the process by which the business requirements will be met.

Also in a traditional project, the plan is created in great detail and is ideally executed with minimal change. In a DSDM project, the initial plans are created in sufficient detail to establish the main parameters of the project with the firm expectation that the customers will change the plan during the course of the project (DSDM Consortium 1997).

The focal point for a DSDM project manager shifts from the traditional emphasis on tasks and schedules to sustaining progress, getting agreement on requirement priorities, managing customer relationships, and supporting the team culture and motivation.

In looking at the DSDM diagram in Figure 18.1, one might wonder where the "testing" phase resides. Again, in keeping with an Agile philosophy, testing isn't shown as a separate, distinct activity because it is an integral part of both the Functional Model and Design and Build iterations.

Collaborative values and principles are another key piece of DSDM. In Dane's interpretation of DSDM:

- Teams are empowered to make decisions
- People are 100 percent dedicated to the success of the project
- Work groups are organized across multiple specializations aligned to a single goal
- Time is the criterion for everyone's success
- Performers are quickly identified and easily rewarded
- Collaboration and cooperation are encouraged between all individuals and work groups

DSDM's Contributions

Two DSDM contributions are the support and service it offers and its principles. First, DSDM has been developed and maintained by a consortium consisting of member companies. The basic DSDM manuals and support materials (white papers) are made available to consortium members for a nominal annual cost. Individuals and companies are certified to teach DSDM seminars, so there is a significant base of service and support, particularly in Europe. So of all the ASDEs, DSDM has, at this point, probably the best organized support and service.

Second, DSDM, like other ASDEs, is principles driven. The DSDM manual, for example, is a 250-page document that reads like a book rather than a stuffy methodology manual. The nine DSDM principles map very closely into the twelve principles of the Agile Manifesto. In addition, in several areas such as testing and modeling, the nine basic principles are used to develop additional detailed principles. The DSDM materials are oriented toward building a framework of what development groups should be doing and why these things are important, but they leave the detailed "how to do it" step to skill-building training and other source material.

For an initiation fee of $1,000 to $3,000 (depending on organization size) and an annual fee of $250 to $650, belonging to the DSDM Consortium is a bargain. Also, for those organizations for which certification is important, DSDM has been accredited as being ISO 9001 compliant.

In terms of methodology, values, and principles, DSDM fits well within the Agile framework, although if we line up the Agile methodologies from more radical to less radical, DSDM falls toward the less radical end of the spectrum. In addition, while there is no explicit linkage of DSDM concepts to a chaordic perspective, its principles reflect a compatible view. However, without the concept of simple, generative rules, DSDM has the potential to become weighted down with inclusive practices and rules (trying to elaborate on all desirable practices). As the Agile movement matures and begins adding elements to make it more attractive to mainstream organizations, this delicate balance will be difficult for all ASDEs to maintain.

DSDM continues to evolve. In the methodology world at least, DSDM is analogous to open source software in that development of the manuals and white papers and the organization of the consortium are the results of significant volunteer efforts. The consortium's latest effort, released in mid-2001, is e-DSDM, a version of the method tailored for eBusiness and eCommerce projects.

DSDM's primary domain target has been small- to medium-sized projects that have a significant user interface component, hence the emphasis on prototyping. For larger infrastructure (middleware, business-to-business, and enterprise application integration) projects, it would need additional tailoring, as would other ASDEs.

Chapter 19. Crystal Methods

Focusing on skills, communications, and community allows the project to be more effective and more agile than focusing on processes and plans.

—Alistair Cockburn

Crystal Methods, Alistair Cockburn's contribution to the Agile community, articulate the primacy of communication and conversation. Software development is "a cooperative game of invention and communication," says Alistair (Cockburn 2002). He focuses on people, interaction, community, skills, talents, and communications as first-order effects on performance. Process remains important, but secondary.

If the first-order effects on performance relate to people and community and the second-order effects relate to process and tools, then how do these observations impact process design? First, they mean that because each person and each team of people have unique talents and skills, then every team should utilize a process uniquely tailored to itself. Second, they mean that since process is secondary, it should be minimized—"barely sufficient," in Alistair's view.

Ask yourself this question: "If I had an extremely talented group of developers, individuals who consistently delivered results, and they refused to implement a new company 'process' standard, what would I do?" (Forget for a moment that most groups practice malicious compliance anyway, complying in form but not substance.) What if such a group just refused? If process rather than people were in fact the first-order effect on performance, then firing the first group and finding a compliant second one would be just the ticket. If you believe that people and their talents are key, then you will find some way to tailor the process to the team, or better yet, you will let the team members tailor the process to themselves.

However, having every team start from scratch and develop its own unique process may not be productive, so Alistair proposes a "set" of methodologies from which teams can select a starting point and then further tailor it to their needs. The word "Crystal"refers to the various facets of a gemstone—each a different face on an underlying core. The underlying *core* represents values and principles, while each *facet* represents a specific set of elements: techniques, roles, tools, and standards. Alistair also differentiates between methodology, techniques, and policies. A methodology is a set of elements (practices, tools); techniques are skill areas such as developing use cases; and policies dictate organizational "musts." For example, in developing potentially life-threatening medical equipment software, policies would be more numerous—the team would have less tolerance on some practices. For a life-threatening software product, code inspections might be mandated by policy, whereas in other problem domains, they would be done at the discretion of the team.

Methodology Design Principles

Alistair articulates seven methodology design principles that can be summarized by two general statements.

The team can reduce intermediate work products as it produces running code more frequently, as it uses richer communication channels between people.

Every project is slightly different and evolves over time, so the methodology, the set of conventions the team adopts, must be tuned and evolve.

There are only two absolute rules in Crystal: the use of incremental cycles not to exceed four months and the use of reflection workshops in order that the methodology be self-adapting. Just as XP is characterized by its 12 practices, Crystal is characterized by increments and self-adaptation.

In designing a methodology, Alistair identifies 13 elements: roles, skills, teams, techniques, activities, process, milestones, work products, standards, tools, personality, quality, and team values. With these elements, it's easy to see how a methodology's size (the number of items in each element plus their interactions) can mushroom quickly. This is a prime reason why designers need to keep in mind the essence of any methodology—that it is a description of the conventions of how people work together. If it helps people work together, keep it. If it doesn't, discard it. A methodology's weight is a product of its size and ceremony (the formality and detail of documents, for example). A methodology with 35 work products, each of which is very formal and detailed, would be heavier than one with 8 informal work products (story and CRC cards, for example). A methodology with 23 roles is heavier than one with 6. Crystal Orange, one of the Crystal family methodologies, is heavier than Crystal Clear, but they are both Agile because of their principles and values.

Alistair's principles for methodology design and evaluation are very instructive.[1] Most methodologies spend 90 percent of their time on non-people-related issues, such as work products, processes, activities, and tools, and 10 percent on communications, conversations, and other people-to-people issues. Crystal and other ASDEs encourage us to reverse the emphasis, maybe not 10–90, but possibly 30–70, with the people issues dominating. According to Alistair, time spent thinking about people, skills, communications, and interactions will lessen the need for expensive, heavy methodology elements. However, costly methodology weight is sometimes necessary when the risk of failures is very high.

[1] See Chapter 25 for an explanation of these principles.

"If your project only needs 3 to 6 people, just put them into a room together," says Alistair. "But if you have 45 or 100 people, that won't work. If you have to pass Food and Drug Administration [FDA] process scrutiny, you can't get away with this. If you are going to shoot me to Mars in a rocket, I'll ask you not to try it."

The Crystal Framework

Figure 19.1 shows the framework for Crystal Methods and the three factors that influence the methodology design: communications load (as indicated by staff size), system criticality, and project priorities. Moving to the right on the staff size X-axis moves teams away from face-to-face communications. Colocate a 4-person team in room, and all you have to do is stand back. However, an 80-person team, even in the same building or on the same floor, requires additional communication and interaction practices and tools. Crystal reviews the people and interaction issues first, and then addresses process, work product, ceremony, and other issues.

Figure 19.1. The Family of Crystal Methods (Courtesy of Alistair Cockburn)
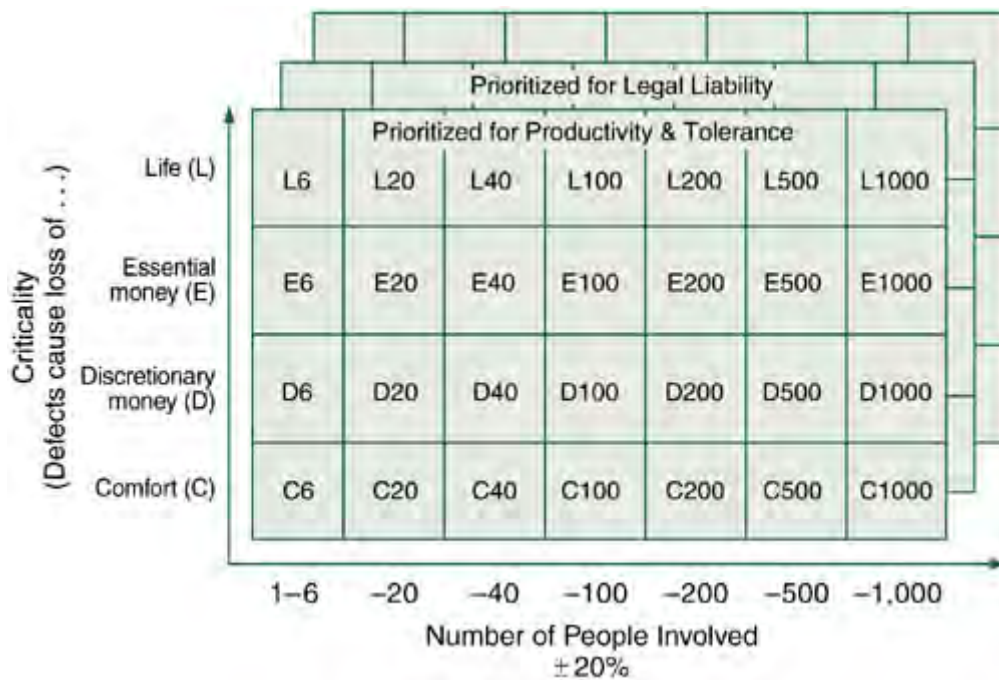
The Y-axis in Figure 19.1 addresses the system's potential for causing damage: loss of comfort, loss of "discretionary" money, loss of "essential" money (e.g., going bankrupt), or loss of life. Alistair points out that most people don't make the finer distinctions between the first three but do differentiate between life-threatening and non-life-threatening applications. The outside world usually dictates methodology elements in the latter case. Whether it is the FDA's regulations regarding medical software, the Federal Aviation Administration's regulations for aircraft engine testing software, or the Transportation Department's regulations for embedded systems in automobiles, software that can potentially threaten life usually has external demands placed upon the methodology used to build it.

However, these external demands should not unduly influence methodology design. They are constraints, not design goals. For example, several years ago I worked with a company in Canada that produced software for monitoring and evaluating clinical drug trials. The company staff had interpreted the FDA requirement for detail requirements documentation and traceability as dictating a waterfall life cycle. I showed them how they could fulfill the FDA requirements with an iterative development approach and a back-end-heavy documentation and traceability process. They began using an Agile approach for development, then supplemented it at the end. They met the constraint but didn't let the constraint drive their software development process. They ended up with a better product and continued to meet regulatory requirements.

The Z-axis plane in Figure 19.1 reflects different project priorities: time to market, cost reduction, exploration, or legal liability. The boxes in the grid indicate a class of projects in which size, criticality, and objective intersect. Alistair then groups individual grid boxes into sets that identify reasonable methodology frameworks: Clear, Yellow, Orange, Red. This approach makes apparent, for example, that when a project team's size increases from 8 to 24 people, the conventions of how they work together need to change.

"Having worked with the Crystal methods for several years now, I've found a few surprises," Alistair told me. "The first surprise is just how little process and control a team actually needs to thrive. It seems that most people are interested in being good citizens and in producing a quality product, and they use their native cognitive and communication abilities to accomplish this. This matches what you write about in *Adaptive Software Development* (Highsmith 2000). You *need* one notch less control than you expect, and less is better when it comes to delivering quickly. More specifically, when you and I traded notes on project management, we found we had both observed a critical success element of project management: that team members understand and

communicate their work dependencies. They can do this in lots of simple, low-tech, and low-overhead ways. It is often not necessary to introduce tool-intensive work products to manage it."

With Crystal, Alistair focuses on the key issues of trust and communication. A project that is short on trust is in trouble in more substantial ways than just the weight of the methodology. To the extent we can enhance trust and communication, we reap the benefits of Crystal Clear, ASD, and the other ASDEs.

Crystal Method Example: Crystal Clear

In Chapter 16, Bob Charette related the story of EuroTel, in which the centralized "methodology police" didn't like Lean Development because it would handle only 60 percent of the company's projects. Other large methodology vendors promote the fact that they have a "framework" that can be tailored to projects of any size. In a sizzling email on the XP discussion group, Larry Constantine responded to one vendor's concept of "tailoring."

> *However, doing XP by way of RUP strikes me as a little like buying an expensive 18-wheel moving van, then throwing away the trailer, chopping down the cab, swapping the diesel for an economical 4-banger, and adding extra seats in order to have a sprightly little runabout for getting quickly around town with the kids and the groceries. If you must pay the price, perhaps this approach is better than driving the bloody truck through alleys and into the Wal-Mart lot, but it seems that doing XP simply by doing XP (or agile usage-centered design or whatever) would be cheaper and more efficient.*

Starting with an RSM, it would be very difficult to arrive at a methodology as "light" as Crystal (or another ASDE). One thing I like about the Crystal Series is that Alistair defines the domain for each Crystal "facet." So Crystal Clear is targeted at a project domain identified by cell D6 (which would also include C6 and could be extended to E6 and possibly D10) in Figure 19.1 The domain of Crystal Clear is: six people, colocated in a room, on a non-life-critical project. The methodology elements of Crystal Clear are

- *Roles:* sponsor, senior designer-programmer, designer-programmer, and user
- *Work products:* release plan, schedule of reviews, low-ceremony use cases (or equivalent), design sketches, running code, common object model, test cases, and user manual
- *Policy standards:* incremental delivery every two to three months, some automated testing, direct user involvement, two user reviews per release, and methodology tuning workshops

That's it. Crystal Clear also provides for equivalent substitutions, so a release plan could utilize an XP story practice or a Scrum backlog approach. Crystal Clear epitomizes the philosophy that if a small team is close together, then almost anything its members do—as long as they maintain communications and community—has a good chance for success. Crystal Clear is light (few methodology elements), tolerant (many variations are acceptable), low ceremony (documentation is informal), and barely sufficient.

Alistair compares Crystal Clear and XP, both light, simple, low-ceremony approaches.

XP pursues greater productivity through increased discipline, but it is harder for a team to follow.

Crystal Clear permits greater individuality within the team and more relaxed work habits (for some loss in productivity).

Crystal Clear may be easier for a team to adopt, but XP produces better results if the team can follow it.

A team can start with Crystal Clear and move itself to XP. A team that falls off XP can back up to Crystal Clear.

Crystal's Contributions

Through his Crystal Methods, Alistair has contributed to the Agile movement in five key ways:

1. Archaeology
2. People and communications
3. Design principles
4. Domains
5. Bare sufficiency

First, many methodologists operate by thinking about how software development *should* be done and then write about their shoulds, often after "some" experimentation. Alistair practices software archaeology, interviewing scores of project teams to uncover what works and what doesn't, or more precisely, what works within certain domains. I like Alistair's definition of process success: It works, and the team would do it again.

Second, from all his archaeological excavations, Alistair has extracted the key issues of people and communications based on trust. Digging down past the surface dirt, even on rigorous projects, he finds over and over again the same explanation for project success: "It was the people. They were great, and we worked well together."

The third aspect of Crystal is its delineation of domains. All too often, arguments over practices take on the proverbial "apples" versus "oranges" character because one arguer is talking about a 500-person military project and the other is talking about an 8-person Web content project. Crystal's domain definitions help focus discussions, and methodologies, on appropriate problem domains. While some practices may be widely usable, the domain definition will dictate the particulars of how they are used.

Many methodologists articulate the need to tailor methodologies to an organization or a project. However, in my experience, this task has been difficult for many project managers. Even when commercial methodologies have articulated different "paths" for different types of projects, the missing ingredient has been "why." Why design a methodology one way versus another? Alistair's focus on people and communications and his seven methodology design principles articulate a "why," a significant contribution to this often overlooked problem area.

Last, as with all the ASDEs, Crystal is designed for bare sufficiency. While Crystal Red may have more elements than Crystal Clear, the elements are still the barely sufficient ones that a team of 80 people, operating with a good sense of community, need to frame how they work together.

No matter which ASDE you select, reading Alistair's material on methodology design will enhance your understanding of how to select and tailor a framework to specific projects.

Chapter 20. Feature-Driven Development

We think most process initiatives are silly. Well-intentioned managers and teams get so wrapped up in executing processes that they forget that they are being paid for results, not process execution.

—*Peter Coad, Eric Lefebvre, and Jeff De Luca,* Java Modeling in Color with UML

For those who wonder if Agile approaches scale, Jeff De Luca, project director of Nebulon, an IT consulting firm in Melbourne, Australia, offers a couple of examples of Feature-Driven Development project success. The first, a complex commercial lending application for a large Singapore bank, utilized 50 people for 15 months (after a short initialization period). The second project, for another large bank, employed 250 people over an 18-month period. "But I'd never do one of these [the 250-person project] again," Jeff vowed. "It was way too big. A very large bank was trying to completely reengineer its technology platform."

FDD arose, in name at least, in the 1997–98 time frame during the Singapore project. Jeff had been using a streamlined, light-process framework for many years. Peter Coad, brought in to develop the object model for the project, had been advocating very granular, feature-oriented development but hadn't embedded it in any particular process model. These two threads—one from Jeff and the other from Peter—came together on this project to fashion what was dubbed Feature-Driven Development. "Peter had been talking about features for years but didn't have a process," said Jeff. "I adapted my process, which had evolved over 20 years, with Peter's ideas about features. My approach wasn't as granular."

Jeff, who seems more interested in managing projects than writing about methodology, was urged to write about FDD by John Gage from Sun Microsystems. Intrigued with the Singapore project, John wanted to know who the project "anthropologist" was—who would write down the experience? Egged on, Jeff wrote the chapter on FDD in *Java Modeling in Color with UML* (Coad et al. 1999).

Peter's company, TogetherSoft, specializes in software development tools that integrate object modeling and implementation. (Those who like to work in code can do so and the models reflect the code, while others can work on models first.) An early version of the tool was used on the Singapore project. The bulk of the development effort for this "TogetherSoft Control Center" product is done in Russia using FDD, said Jon Kern, director of product development for TogetherSoft. Jon, one of the Agile Manifesto coauthors, told me that while FDD is not fully implemented at the site, its emphasis on customer-related features has helped significantly in focusing the Russian team's development effort.

"I find that pictures and models are often a more concrete way to augment textual requirements," said Jon. "They help in communication and eliminate misunderstanding." Because the models generate code and vice versa, Jon stressed, TogetherSoft's tool mitigates disconnects between models and code.

The presence of Jon (along with Steve Mellor) in the Agile Alliance serves notice that Agile approaches are not anti-modeling. The Agile community's concerns are not modeling per se but product development efforts that remain at a high level of abstraction too far into the project. Even then, the real problem is not the abstractions, because abstractions are beneficial to the thinking process, but the fact that product teams using rigorous approaches (and their inherent specialization) seriously underestimate the chasm between their abstractions and working software. Because of Jon's conviction that TogetherSoft's product reduces the gap between abstraction and concrete code, he felt comfortable signing the Manifesto principle about working software.

The Singapore Project

The Singapore lending project had been a colossal failure. Prior to Jeff's involvement in the project, a large, well-known systems integration firm (which obviously can't be identified here) had spent two years working on the project and finally declared it undoable. Its deliverables: 3,500 pages of use cases, an object model with hundreds of classes, thousands of attributes (but no methods), and, of course, no code.

The project—an extensive commercial, corporate, and consumer lending system—incorporated a broad range of lending instruments (from credit cards to large multi-bank corporate loans) and a breadth of lending functions (from prospecting to implementation to back-office monitoring). "The scope was really too big," said Jeff.

Less than two months into the "new" project, Jeff's team was producing demonstrable features for the client. The team spent about a month working on the overall object model (the original model and what Jeff refers to as the previous team's "useless cases" were trashed), and then spent another couple of weeks working on the feature decomposition and planning. Although the artifacts from the previous two-year effort were not valuable in themselves, Jeff stated that the underlying information was very helpful in getting his team off to a faster start than would have been possible otherwise. Finally, to demonstrate the project's viability to a once-burned and skeptical client, Jeff and his team built a portion of the relationship management application as a proof of concept. From this point on, with about four months elapsed, they staffed to 50 people and delivered approximately 2,000 features in 15 months. The project was completed significantly under budget, and the client, the CEO of the bank, wrote a glowing letter about the success of the project.

As Jeff and I talked, a couple of things struck me about this project. Certainly the FDD process contributed to the project's success. When I asked him what made FDD successful, though, his first response was that the overriding assumption behind FDD is that it embraces and accepts software development as a decidedly human activity. The key, Jeff said, is having good people— good domain experts, good developers, good chief programmers (CPs). Jeff concurred with my hypothesis that a project with talented people and a streamlined process has a high probability of success, while a project staffed with under-talented[1] people—no matter how good their process— will probably fail. As discussed in Chapter 7, process isn't a substitute for talent or skill.

> [1] I use "under-talented" here in the sense that individuals don't have the necessary skills or knowledge for the problem domain.

Based on his successful delivery of several complex applications, I'd say that Jeff is a very talented and skilled project manager. For his part, Jeff believes "that Peter Coad is the best object modeler in the world. Peter says that this was the most complex domain he'd ever modeled." It appears that this lending system was a world-class problem and that the ultimate critical success factor was assembling world-class talent. We can assume that the original vendor assigned reasonably talented and skilled people to the project and that it employed a "proven" process (all systems integration companies have standard development processes)—but it didn't have world-class talent leading the effort. Jeff recruited a few highly talented people for key architect and CP roles and then used these people to train and mentor others.

My guess is that even if the first vendor's staff had used FDD as a process model, they would not have been successful because they just did not have the appropriate level of technical and project management talent. However, had they been using a FDD-like process, their inability to complete the project might have surfaced in less than two years. This is a clear example of why working code is the ultimate arbiter of real progress. In the end, thousands of use cases and hundreds of object model elements didn't prove real progress.

The FDD Process Model

FDD[2] was developed by Jeff and modified while he was working with Peter on the Singapore project. FDD addresses the problem of response time to shorter and shorter business cycles. The authors' experience from the 1980s mirrors that of many other Agilists: "A decade ago, one of us wrote a 110-page process for a large development team. No matter how hard he tried to defend every word of his process as something of great value, the team members always (always) looked at the 4-page summary in the back and ignored the rest" (Coad et al. 1999).

[2] For an in-depth look at FDD, see *A Practical Guide to Feature-Driven Development* (Palmer and Felsing 2002).

"Yes, developers *and* managers *and* clients like FDD," said Jeff. Developers get closure on something delivered within two-week intervals (or less). Managers have a way to plan that includes meaningful milestones and risk reduction due to frequent, tangible results. Clients see plans with milestones that they can understand. While TogetherSoft's primary product is its software not the FDD methodology, the company mission statement—"Improving the ways people work together"—illustrates the philosophy of figuring out how people work *together* and then building technology to support those findings.

The FDD process reflects the learning from earlier, process-centric methodologies. Jon describes FDD as client-centric, architecture-centric, and pragmatic. Peter characterizes FDD as having "just enough process to ensure scalability and repeatability and encourage creativity and innovation all along the way."[3]

[3] Peter Coad, in a presentation to the JavaOne conference, 2000.

FDD asserts that:

- A system for building systems is necessary in order to scale to larger projects
- A simple, well-defined process works best
- Process steps should be logical *and their worth immediately obvious to each team member*
- "Process pride" can keep the real work from happening
- Good processes move to the background so team members can focus on results
- Short, iterative, feature-driven life cycles are best

At a high level, the FDD process epitomizes simplicity—it has only five processes,[4] as illustrated in Figure 20.1. The amount of project time spent in each process breaks down like this:

[4] Jeff De Luca maintains an updated version of FDD processes at his Web site, www.nebulon.com. The descriptions in this chapter are taken from Jeff's updated version of FDD.

1. Develop an overall model (10 percent initial, 4 percent ongoing)
2. Build a features list (4 percent, 1 percent ongoing)
3. Plan by feature (2 percent, 2 percent ongoing)
4. Design by feature
5. Build by feature (77 percent for design and build combined)

The hand-drawn nature of the boxes and script labels in Figure 20.1 conveys that FDD is a set of guidelines rather than a prescriptive process. Jeff also mentioned that the ten percent time figure for process 1 was the most valuable and that the other percentage figures varied.

Figure 20.1. FDD's Development Process (Courtesy of TogetherSoft)
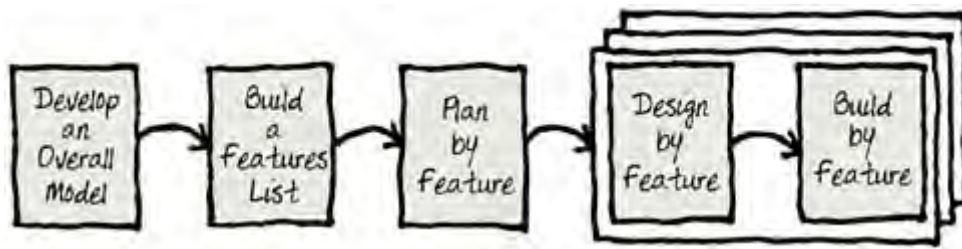
Process 1: Develop an Overall Model

"This is where we model the entire breadth of the domain, but only at the level of 'shape' modeling—which contains the classes and how they connect," said Jeff. The overall model in the first phase is an object model that focuses on the high-level "shape" or skeleton of the product rather than the details, which come into play in the design by feature phase. This process is led by a chief architect. For larger projects, domain teams consisting of modelers and domain experts develop shape models for subject areas. These domain-specific models are then combined into an overall model during an "integration" modeling meeting. The domain experts provide overall requirements, which are used to develop the shape model and are then refined into features in process 2.

"The integration is usually done along the way during the modeling," said Paul Szego, who has been a CP for Nebulon on many FDD projects. "We talk about 'component' models, which come out of a single modeling session in support of one domain walkthrough (that is, one topic that the domain expert has presented in 20 minutes or less). Depending on complexity, we usually do a model merge into the overall model every other day or so, sometimes with the entire group, but often just at the end of the day with the chief programmer and chief architect being the only ones involved."

While this might appear to be an overly front-end-loaded approach, FDD also provides some rough planning guidelines for how much time should be spent in each process. For a six-month project, developing an overall model would take in the range of two weeks, conforming to what I would consider an "Agile" approach to modeling.

Each of the five processes in FDD is depicted on a one-page process description using the ETVX pattern (Entry criteria, Task, Verification, eXit criteria), which includes a brief one-paragraph overview of the process itself. As an example, the "develop an overall model" process contains only seven tasks:

- Form the modeling team
- Domain walkthrough
- Study documents
- Develop the model
- Refine the overall object model
- Write model notes
- Internal and external assessment

I particularly like the "write model notes" task, during which the chief architect and chief programmers meet and write down notes about the model's shape and why some alternatives were selected and others rejected. This activity is both informal and directed toward effective knowledge transfer—conversations backed up by minimal documentation for reference.

"We all get together at some point, after having done the other steps, to remember all this information," says Paul. "Hopefully you made a *lot* of notes along the way, in particular during the

modeling. We ensure that someone is the 'scribe' for each session. The way we do modeling is to split into teams, model, get back together with each team presenting, then merge the models. Many variations may be considered until we get consensus on a group model. So it's a little of both—taking a lot of notes as you go and getting together to write up notes afterwards as well."

Process 2: Build a Features List

From the modeling effort in the prior step, process 2 builds a complete list of all features in the product (very similar to XP's story cards or ASD's features). This process is accomplished primarily by the CPs who participated in process 1. The process is one of functional decomposition—from subject area, to business activity, to business activity step. The most fine-grained piece—the business activity step—is the feature, but the business-oriented terminology is maintained for its clarity to the clients. Feedback to customers during the project is usually at the level of business activity, not features. A business activity might comprise 10 to 20 features, although this can vary widely. Paul also indicated an issue that has caused some trouble in explaining FDD to others, which is that "the 'functional decomposition' is totally orthogonal to the way the object model is composed." One is business focused, the other technology focused.

FDD shares with other ASDEs a focus on features, which FDD defines as "small, useful in the eyes of the client, results." A feature, using this definition, would be something like "check the customer's credit rating," as opposed to a technology-centric feature such as "develop the objects to access the SQL Server database." FDD further defines a feature as "a client-valued function that can be implemented in two weeks or less" and also specifies a naming convention for features (e.g., calculate the total of a sale); business activities (e.g., making a product sale to a customer), which are composed of features; and subject areas (e.g., product-sales management).

Features can be one to ten days of work, with ten days being the maximum. If a feature appears to be greater than ten days' worth of work, it is decomposed further. However, unlike some other Agile approaches, the two-week limit on features does not indicate a two-week development cycle. No weighting or prioritization of features is made at this point.

Process 3: Plan by Feature

A planning team, consisting of the project manager, development manager, and chief programmers, gets together and plans the order in which features will be developed. At the level of a business activity, completion dates (month and year) are scheduled. Planning is based on a number of factors—dependencies, risk, complexity, work-load balancing, client-required milestones, and checkpoints. Responsibility for each business activity is assigned to a CP, and every class is assigned to a specific developer.

Once the feature list has been generated and reviewed with the client, the majority of the feature scheduling for development is done by the technical team, in particular the project manager and the development manager and then the CPs. Paul discussed how scheduling happens at a number of levels. This multi-layered scheduling is one practice that has enabled Nebulon to apply FDD on larger projects.

"We often talk about 'self-organized' construction within planned assembly," Paul observed. "The on-the-ground workflow is something that wasn't discussed in the book [Coad et al. 1999], but it's important to how we go about processes 4 and 5. There are a couple of levels of delegation. At the top, the project manager and/or development manager look at what features they want started next. Often this is done in whole business activity chunks, but sometimes they select individual features to schedule. Each feature is assigned to the CP responsible for that business activity.

"Each CP has this 'virtual in-box' of features,"[5] Paul continued. "The CP knows what features are currently being developed, what's planned to happen over the next weeks, and which programmers are tied up doing what tasks. They decide what's to happen next and formulate work packages.

[5] This practice is much like Scrum's backlog.

"In contrast, the project manager is more focused on the overall project plan—what things we do in what sequence, at a macro level. The project manager brings risk forward by tackling more complex tasks earlier, satisfies any external constraints like interim deliverables or demos, and of course manages the planned dates (MM/YY) for each business activity. The development manager balances the load across the CPs."

Process 4: Design by Feature

The output of the design by feature process is design packages and CP work packages. In process 3, features were assigned to CPs, usually based on business activity. The CP now looks at his or her in-box of features and, based on a class-feature analysis and other information, bundles sets of features together into work packages. Whereas business activities comprise sets of features within a business framework, work packages comprise sets of features within a technical construction framework.

The CP then develops a design package—consisting of sequence diagrams and class and method design information—and refines the object models with attributes and other details. "The CP leads the development, and the entire team participates," said Paul. "The sequences are usually done as a group activity: this is the design part. The class and method refinements are done by the class owners."

Process 5: Build by Feature

The feature teams then take each design package through the steps:

- Implement classes and methods
- Code inspection
- Unit test
- Promote to the build

Beyond the FDD Process Description

When I first read the FDD description in *Java Modeling in Color* and the updated version on Jeff's Web site, two things struck me: Where was the customer interaction? How were changes handled? I expected that both of these issues had to be covered in FDD, and my conversations with Jeff confirmed my expectation.

"Requirements is not a box at the start of the life cycle that is never touched again. Requirements are something that have to be managed across the life cycle," he said. "There are two key places where the domain experts interact, and that is in process 1 [develop an overall model] and process 4 [design by feature]." During process 1, the classes and their connections are modeled by the technical team and appropriate domain experts. While a few key attributes and methods are identified, most of those details are left until process 4. For example, in the lending domain, there may be an "interest rate formula calculation" feature. The feature would be identified in process 1 and the classes and connections are modeled—for example, a "TermAndConditionDescription" class—but the interest rate calculation is not defined in detail until the feature is designed. In

process 4, two optional tasks—domain walk through and study referenced documents—may be necessary when the feature contains either algorithmic or data complexity that require domain expert clarification.

However, Jeff cautioned, "Common sense always prevails." At any point, domain expert involvement may be needed. In practice, the FDD processes are similar to XP, ASD, and other Agile feature- or story-based planning approaches. While pure XP project teams may not develop a skeletal model, they produce a release plan that identifies stories for which the requirements details are fleshed out during the iteration in which they are implemented.

FDD's approach to changes offers some unique ideas. While the intent of the overall modeling is to reduce subsequent changes, the need for incorporating new features or modifying existing features is inevitable. When new features are added, they are placed within a business activity labeled "new features." The project manager then monitors the feature addition using an old Ed Yourdon rule that Jeff has used for many years: "If any project dimension gets off by more than ten percent, then you can't recover without something else giving by ten percent. So when the number of new features [or changes to existing features that are assigned to "new features"] exceeds ten percent, we expose them to top management. It's a real simple equation—either they cut the scope or adjust the schedule. You demonstrate great control as a project manager here, as you have the quantitative and qualitative data behind you that they are already used to and comfortable with."

From an organizational perspective, FDD utilizes three roles: chief programmers, class owners, and feature teams. The CP, a role inspired by Harlan Mills' ideas on surgical teams, is a more experienced developer who acts as a team lead, mentor, and developer. Responsibility for each class is assigned to a particular developer, who is responsible for updating or modifying it. Feature teams are temporary groups of developers formed around the classes with which the feature will be implemented. The class owner list indicates who owns those classes, and thus a feature team dynamically forms to implement the feature.

Paul commented on class ownership and feature team formation: "Of course I'm not saying that you can't get decent results without doing this; it's just that we've found that more often than not, we get better classes using class ownership. If you don't, you tend to lose the 'Zen' of the class—what it's all about. Very rarely a class owner will delegate some changes to someone else, usually if the change is trivial. But they're still responsible for the result." The class ownership practice contrasts with XP's collective ownership practice.

"You usually have multiple teams, each at a different stage in processes 4 and 5," said Paul. "We often talk about a developer's 'pipeline,' of what tasks are ahead of them. The CP's goal is to keep this pipeline filled to at least two weeks ahead, so the developer can see what's coming and plan accordingly. If you look at processes 4 and 5, at the milestones," he continued, "it's really only the design and code tasks that take significant chunks of time. The other tasks are basically short meetings (sometimes only 15 minutes, never more than one hour) that need to be slotted in around the other tasks that are measured in days. So a developer might attend the design meeting for feature team 'A' first thing in the morning for an hour, then code for team 'B' for the rest of the day. Next day after lunch, [he or she] might attend the kick-off for team 'C' and finish the coding for team 'B.' Feature teams never last more than a two-week period."

From his years of managing projects, Jeff also has an interesting idea about the typical end-of-project crunch time. "I run projects so that we coast the last part of it and give ourselves the maximum time for optimizations and so on. I know the idea of the big crunch toward the end of a project where people are working nearly round the clock with stale pizza and loads of empty Jolt cola cans is a very romantic notion, but I don't think that's much fun at all. I strive to break the back of a project as early as I can and then coast in to the finish line. I don't like people working overtime."

155

Conceptual Similarities and Differences

"For me, I'm a project manager," says Jeff. "The system I build is the project—an org chart, some roles, people to fill the roles, a set of well-bounded processes, and some technology decisions. That system is going to build the 'whatever' application system."

The core of FDD seems to be: iterative cycles of fine-grained features, a very streamlined process that utilizes a small set of proven traditional practices, a bias toward group interaction and knowledge sharing, and skeletal front-end modeling to provide the context for all development. However, FDD accomplishes these core goals in ways that are somewhat different than other ASDEs.

"FDD and XP are really diametrically opposed in some of their principles," said Jeff. "The secret to FDD, unlike other iterative or incremental approaches that tend to look only at thin slices through the life cycle, is that FDD has a large blobby thing at the front that we call 'process 1.' This allows us to build the feature list and implement without doing a lot of rework. We are biased toward getting it right the first time. It's not that we're anti-refactoring, but that we'd prefer not to do much refactoring."

At their most "extreme" position, XP proponents say that the overall architecture (an object model in this case) should emerge from a process of incremental story (feature) implementation and continuous refactoring. FDD largely eschews refactoring and advocates front-end skeletal modeling. But are these positions really so far apart? Jeff's guideline is to perform about two weeks of up-front "shape" modeling per six months of project time. So on a six-month project, Ron Jeffries would start coding in week one and Jeff De Luca would start in week three. By the end of the first month, both would be continuously delivering features to the client every week or two. Differences? Yes. Still, I would offer that, in this case at least, the differences are secondary, particularly when compared to heavier methodologies in which abstract modeling can go on for months.

Two other FDD practices remind us that we need to take intent and principle into account in order to understand how a particular practice impacts the culture of a project. FDD advocates use of chief programmers and formal inspections. An initial judgment might, therefore, place FDD and XP at significant odds—and in some ways, they are. FDD uses a CP for each feature team to provide a breadth of knowledge about the skeletal model to the feature team, to develop work packages and assign them to developers, to schedule inspections and arbitrate technical decisions, and to be a mentor for team members. FDD also uses inspections, which Jeff thinks are more effective than pair programming.

However, when I asked Jeff about the intent of both the CP and inspection practices, he responded that their primary purpose was mentoring and knowledge sharing on a person-to-person basis. To the extent that FDD's CPs act in this mentoring and collaborative knowledge-sharing role, the CP would be analogous to XP's coach. To the extent the CPs act in a hierarchical, technical decision-making mode, FDD would contradict XP. Jeff's implementation of the CP concept is collaborative in intent. The CP-to-developer interactions are intended to convey overall architecture and context information, while inspections convey detail information from developer to developer. "The CP-to-CP collaborations are key also," Jeff noted.

"A very valuable benefit of inspections is the team aspect," said Jeff. "That is, propagating and promulgating syntactic and semantic standards, team culture, and so on. However, it is also the most effective defect removal technique. In explaining inspections, I first talk about defect removal effectiveness and cost of defects, then mention the team benefits of inspections."

Jeff also observed that within the context of FDD, with its emphasis on small features, inspections were reasonable to implement. The monolithic nature of many approaches leads to inspecting large blocks of code, or models, and reduces both the effectiveness of inspections and the incentives to do them.

Another area in which FDD differs from XP, Scrum, and ASD in particular is the scheduling of features to be delivered. Within the boundaries of technical reasonableness, XP, Scrum, and ASD insist on the customer's setting development priorities at the beginning of each iteration. The customer determines which features to develop next based on their current business value assessment. FDD assumes that the overall value of the features is determined early in the project and that scheduling those features should be primarily a technical decision. Either way, developed features provide demonstrable evidence of progress.

There are advantages and disadvantages to customer scheduling versus technical scheduling. Letting customers schedule features on a short-cycle basis could increase development costs by causing additional rework because of technical dependencies. Conversely, technical scheduling can allow the project to drift off course from the customer's perspective. FDD actually offers a potential path for a "best-of-both" approach—customer scheduling at the business activity level and technical scheduling at the feature level. In fact, as I suggested to Jeff, customers may not be thrilled with scheduling features, especially at the two- to five-day level of granularity.

"Correct," said Jeff. "We only put a date at the level of the business activity and that date is a month and year only. Clients are OK with this because it is expressed in client value terms. Thus, as a project manager, I get a lot of management flexibility. If I said, for example, they would get the calculation total of a sale feature on Friday the 24th, then I have a problem with the client if it is anything other than that exact date and time. This is stupid."

"You create a huge potential for problems and having to explain things that really don't matter," he went on. "But once you promise this way, it does matter to the client. Bottom-up detailed planning and estimating approaches (like Gantt charts) are stupid. How could you possibly estimate at the level of features across a project with accuracy? A feature like 'generate the unique number for an order' might be only hours of work. The cumulative error in estimating at this level is enormous. If you do a Gantt properly and resource-level them at such granularity, someone having an extra-long toilet visit would break your plan. If the client knows they will get loan registration and customer demographics in February, and then lines-of-credit in March and then collaterals in April, they are satisfied with that. Remember that at any time, weekly, they can see in great detail where every single feature is and what the completion rates are across the project."

When I asked Jeff about what contributes to FDD's success, there was a pause in the conversation. "The number one thing," he said, "is that there isn't one thing, but a series of things." He views FDD as a collection of practices that work well together—a system of practices. "If you want to slice and dice, take one practice out and put another in, I don't know what the results might be," he said. Quoting his friend Brian Coombs (of IBM GSA), Jeff commented, "'The optimal solution always contains suboptimal parts.' I view FDD as a collection of suboptimal parts—that in total provide an optimal solution."

The second thing that contributes to FDD's success, in Jeff's opinion, is the acceptance that software development is a human activity. Third, Jeff thought that process 1—the development of a "shape" object model—was critical, as was the use of inspections to validate work products and share information. What do customers like best about FDD? "The domain modeling up front and the results-oriented reporting," said Steve Palmer, senior mentor at TogetherSoft

FDD's Contributions

157

While there are similarities in values and principles among the major ASDEs, there are also differences. For example, FDD advertises that it is repeatable, whereas other Agilists (myself included) think that the word "repeatable" conjures up the wrong image of detailed processes and precision measurements that are incompatible with a highly turbulent environment. FDD also emphasizes "doing as much right the first time" as possible, recommending up-front modeling. In contrast to XP, FDD recommends that each class have an "owner" (XP practices "collective ownership") and formal design and code inspections (XP practices pair programming). FDD recommends a CP; XP tries to eliminate role distinctions.

I think these differences illustrate that ASDEs are not identical but instead contain variations in practices and ideas. There is a span of social and management cultures, from virtually nonhierarchical, self-directed teams to rigid, top-down, Command-Control-driven teams. None of the ASDEs advocate the latter, but they are not in lockstep on cultural issues either. They all emphasize collaboration and knowledge sharing, but the practices for achieving those ends are different. Nevertheless, at the values and principles level, when thinking about the importance of people and their inter actions, when thinking about lighter process and documentation, when thinking about incremental development providing small wins every few weeks, when thinking about the contributions of talented individuals—ASDEs are very consistent.

Several things about FDD appeal to me: the light but defined process, the light front-end modeling, the emphasis on collaboration and knowledge sharing, and the acknowledgement that talent and ability—the human factors—are critical to success. FDD's success on large projects indicates that the additional "structure" of the chief programmer role (provided you can find capable CPs), skeletal modeling, and defined but light processes can be effective. This additional structure is compatible with Alistair Cockburn's and my own ideas about scaling to larger projects. Paul had an interesting comment about FDD scaling: "We (Jeff and I) obviously know it scales up. The more interesting question we've been asked is, 'How small can it scale down?'"

While DSDM can be characterized by its emphasis on prototyping, FDD can be characterized by its emphasis on modeling—specifically object modeling. However, the linkage between models and "working software" is very close, given the use of TogetherSoft's software tool. Finally, I particularly like the format of the FDD processes. The simple, one-page, ETVX documentation pattern combines a lot of key information in a usable format.

Chapter 21. Lean Development

Businesses are finding too often that their software systems act as brakes on their competitiveness, rather than as accelerators. Businesses are discovering that while their markets change rapidly, their software systems do not.

—*Bob Charette,* Foundations of Lean Development

One-third the time, one-third the budget, one-third the defect rate. These are the goals of Lean Development, a software management approach originated by Bob Charette. While most other ASDEs are tactical in nature, Bob thinks that the major changes required to become Agile must be initiated from the top of the organization. Organizational strategy becomes the context within which Agile processes can operate effectively. Without this strategic piece, Agile development— as all those who have tried to implement ASDEs in organizations can testify—are shunted aside by the organizational forces that seek equilibrium.

EuroTel

An LD client provides one of the definitive studies on ASDE performance improvements. An early proponent of LD, this European telecommunications firm, EuroTel [a fictitious name], has over 100,000 employees operating in more than 100 countries. EuroTel's interest in LD arose from discussions with VPs of various business divisions. They identified the need for dramatic reductions in cycle time, costs, and defects, while their analysis showed "that our CMM process was seriously insufficient," according to a EuroTel IT executive. They found that the CMM improved quality and cost, but not schedule—and they needed all three. In looking for alternatives, EuroTel turned to Bob and his LD ideas.

The initial results at EuroTel were dramatic. LD was used on six projects, ranging from "analysis and control of telecom networks" to a "traffic control system for a large city." These projects ranged between 10 and 50 person-years of effort (up to several hundred thousand lines of code). According to the IT executive, "Our initial measurements show that we reduced cost and time by at least 40 percent. A further reduction could be potentially achieved by basing our developments even more on domain solutions." Responding to my question of how the improvement had been measured, he continued, "The reduction was measured in two ways: an external audit by a German software research institute, which analyzed our processes and technology in principle, comparing them to traditional practices as described before; and, actual results in projects (time and effort) against estimations of times and effort on the basis of estimated LOC [lines of code] or feature points and historical performance."

According to informal surveys, customer satisfaction improved as well. Even in the absence of formal measurements, the IT executive's view was that "the most significant practice of LD is the early customer involvement in the product definition process and the demonstration of evolving features. Note that this is not prototyping but the fast evolution of a running system."

EuroTel's goals for the future were even more impressive. "LD is merely a stepping-stone toward mass customization, which we set as a target cutting again by 50 percent. That is, building change-tolerant systems in one-sixth the time, cost, and defect levels." Unfortunately, as the interview with Bob in Chapter 16 pointed out, the organizational "antibodies" within EuroTel doomed subsequent initiatives.

The Strategic Foundation of Lean Development

LD is the "operational" piece in a three-tiered approach that leads to change-tolerant businesses, as shown in Figure 21.1. LD provides a delivery mechanism for implementing "risk entrepreneurship," which itself is part of risk leadership. The key in business, according to Bob, is that the opportunity for competitive advantage comes from being more Agile than competitors in one's market. The difficulty for many companies, however, lies in new competitors that enter the market from unforeseen directions. These companies, often significantly more change tolerant, or Agile, than existing firms, can cause major disruptions to a company's plans.

Figure 21.1. A Change-Tolerant Organization (Courtesy of Bob Charette)
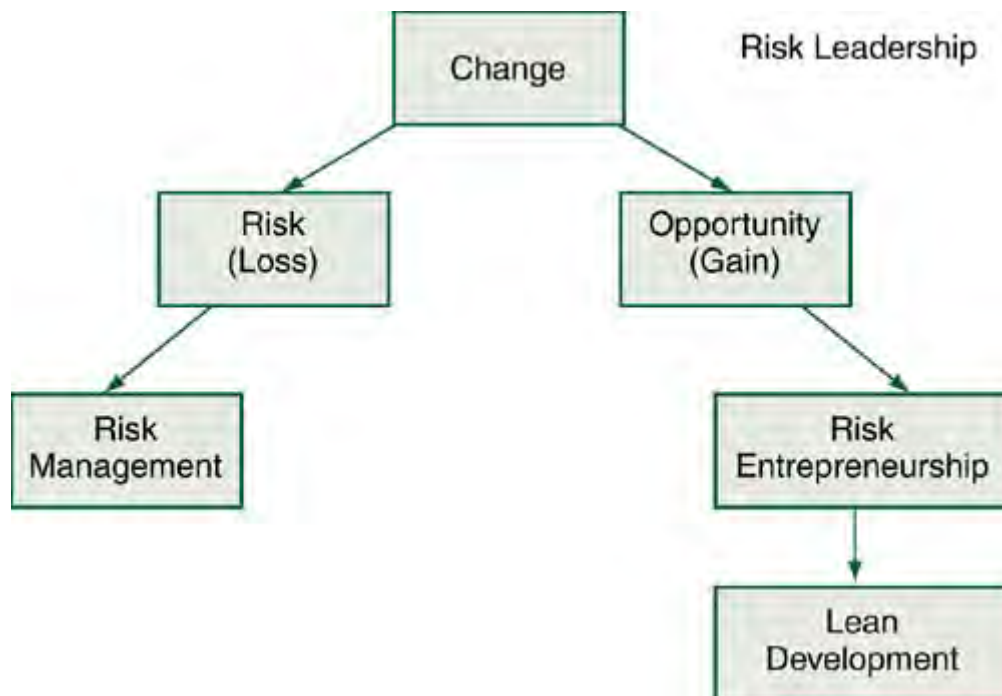
LD's risk entrepreneurship enables companies to turn risk into opportunity. Bob defines change tolerance as "the ability of an organization to continue to operate effectively in the face of high market turbulence." A change-tolerant business not only responds to changes in the marketplace, but also *causes* changes that keep competitors off balance. "Most software systems are not Agile, but fragile," Bob says. "Furthermore, they act as brakes on competitiveness." Every business must deal with change, building a change-tolerant organization that can impose change on competitors. For example, a manufacturer whose product release cycle is 9 months generally has an advantage over one whose release cycle is 18 months. Often, when the second manufacturer tries to "catch up," it flounders, because it takes on more risk and more change than it has the skills and processes to handle.

As shown in Figure 21.1, risk leadership involves managing risk (the possibility of loss) and opportunity (the possibility of gain). Fundamentally sound risk management is the first responsibility of risk leadership. It provides the foundation—the true understanding of risk—upon which companies can venture forth into risk entrepreneurship. As the stock market of 2000–2001 clearly demonstrated to investors, venturing forth into risky areas—be they stocks or business opportunities—first requires a good understanding of the risks. Risk entrepreneurship is the process of moving from a defensive stance on risk—primarily protecting against loss—to an offensive stance on risk—striving to turn the opportunities uncovered by change into profits. In the software world, risk entrepreneurship requires the ability to explore opportunities rapidly and cost effectively. It is the ability to weigh the potential for loss against the opportunity for gain, based on a realistic assessment of what the organization can actually do.

Bob's work sends several key messages to Agile developers and business stakeholders in IT. First, wide adoption of ASDEs will require strategic selling at senior levels within organizations. Second, the strategic message that will sell ASDEs is the ability to pluck opportunity from fast-moving, high-risk "exploration" situations. And third, we proponents of ASDEs must understand, and communicate to our customers, the risks associated with Agile approaches and, therefore, the situations in which they are and are not appropriate. We must apply risk management to our own practices.

LD is as much a management challenge as a set of practices. Bob comments, "You have to set the bar high enough to force rethinking traditional practices. LD initiatives focus on accelerating the speed of delivering software applications, but not at the expense of higher cost or defect rates. These three objectives need to be achieved concurrently, or it isn't LD."

Risk is the probability of failure or ending up worse off than when we started. Fast development cycles create greater risk of failure; however, if your company can increase development speed at lower risk than a competitor, the advantage can be substantial. Actually, the benefits can be staggering. Steven Wheelwright and Kim Clark (1992) discuss how a faster cycle time compounds competitive advantage over several cycles: "While the initial advantage of the fast-cycle competitor is relatively small, the ability to move quickly to market eventually creates a significant performance gap."

Lean Development's Origins

LD is a term that emerged from the manufacturing realm of lean production in the 1980s. In *The Machine that Changed the World: The Story of Lean Production,* authors James Womack, Daniel Jones, and Daniel Roos (1990) are "convinced that the principles of lean production can be applied equally in every industry across the globe and that the conversion to lean production will have a profound effect on human society." They define lean production by contrasting it with early 1900s craft production methods in automobile manufacture and the subsequent mass production techniques initiated by Henry Ford. Lean production leads to the next phase, which is mass customization.

Lean manufacturers strive to better mass producers in every area—half the effort, half the floor space, half the tool investment, half the product development time, and half the defects. However, as an automotive industry study showed, automation was not the answer. One European auto plant, the most automated in the world, required 70 percent more effort than another with much less automation. The study's conclusion: "Lean organization must come before high-tech process automation" (Womack et al. 1990).

In software development, the era of craft techniques probably ended in the mid-1970s with the advent of interest in structured methods. Software development strategies corresponding to mass production (and its incumbent specialization) blossomed during the 1980s, culminating in voluminous methodologies, CASE tools, and the Software Engineering Institute's (SEI) Capability Maturity Model (CMM). According to Bob, the early 2000s will be an era of Agile approaches such as LD.

Another defining characteristic of lean production that has parallels in software development is the change in the use of specialists. Craft producers use "highly skilled workers and simple but flexible tools to make exactly what the consumer asks for," while mass producers use "narrowly skilled professionals to design products made by unskilled workers tending expensive, single-purpose machines" (Womack et al. 1990). LD, although it utilizes specialists, places emphasis on general skills.

On the issue of documentation, Bob makes a telling case against the "abundance" of documentation in the average project.

> *For LD projects, the creation of the minimum amount of documentation is an assumption. Excessive documentation does not add value but only eats up resources and time. Studies of documentation produced during conventional software development indicated that for an average 1,000-function point software project (about 125,000 lines of code):*

- Requirements documents average 300 pages
- Plans average 100 pages
- Design documents average more than 1,500 pages
- User manuals average more than 600 pages
- Test reports average more than 5,000 pages (Charette 2002)

Few users read 600-page manuals. And if the software is changing rapidly, most of the other documentation will be out of date in a short period. ASDEs like LD don't advocate eliminating documentation, but they surely advocate reducing the above averages significantly

What Is Lean Development?

LD embodies the concept of *dynamic stability* (similar to Scrum's "controlled chaos" or ASD's "balancing at the edge of chaos"), the ability to adapt quickly and effectively to a wide range of customer demands (the dynamic part), combined with the ability to build stable, continuously improved internal processes that are general purpose and flexible across a wide range of products. LD is the creation of change-tolerant software with one-third the human effort, one-third the development hours, one-third the time, one-third the investment in tools and methods, and one-third the effort to adapt to a new market environment.

These aggressive goals are established to challenge the status quo, to get managers to think about issues in entirely different ways. The first question someone might ask about these goals is, "What is the baseline?" Bob responds that the baseline for measuring improvement is an SEI CMM Level 3 organization. For example, the goal would be one-third of the development time of a Level 3 organization.

The principles of LD were developed in conjunction with several clients, EuroTel in particular. While none of the principles are new, their accumulation under the banner of LD provides an emergent, powerful synergy. Bob's ideas are similar to Kent Beck's for XP—it is the *system* of practices that breeds synergistic effects. Picking and choosing, particularly when a team is new to a methodology, can adversely impact its performance.

Bob stresses four critical success factors for LD: (1) creating visible customer value rapidly, (2) building change-tolerant software, (3) creating only necessary functionality and no more, and (4) aggressiveness, stubbornness, and belief in meeting LD's stretch goals. Expanding from these factors, we get the 12 principles of LD.

1. *Satisfying the customer is the highest priority*.

   The development team must have practices to determine customer priority and others to listen to their responses. The goal is maximizing customer satisfaction. Not meeting customer expectations is viewed as a failure.

2. *Always provide the best value for the money*.

Software should help solve a customer's problem or provide them a new opportunity at a reasonable cost. Value is the goal, not perfection. Value is a combination of product features that meets a customer's needs at a specific time for a specific price.

3. *Success depends on active customer participation.*

   Active participation is a collaborative, joint effort. Customer participation is more than just "buy-in"; active participation is essential to adapting to change and making real-time tradeoff decisions. As Bob comments, "No customer collaboration, no LD."

4. *Every LD project is a team effort.*

   Multi-disciplinary teams rather than isolated individuals are needed because diversity is so key to innovative, fast-cycle time development. However, the more diverse a team, the more difficult it is to create an environment in which the team can jell.

5. *Everything is changeable.*

   Rigorous development practices are based first on eliminating changes by defining requirements in the beginning and then by controlling any changes that happen to sneak in. LD assumes that requirements changes will be continuous and that learning to adapt to changes is a better strategy than trying to control them. The constant questions asked in a LD effort are, What kinds of changes could occur? What would we do if they occurred? How can we build the application to be more tolerant of these types of change?

6. Domain, *not point, solutions.*

   One-of-a-kind software is too expensive in most cases. Software that is applicable across multiple domains—companies, markets, products—spreads the cost and thereby contributes to the value equation. This idea of domain solutions differentiates LD from other ASDEs.

7. *Complete, don't construct.*

   Buy rather than build has long been a viable strategy for most application development groups. Recent implementations of enterprise resource planning (ERP) systems attest to this trend. Components and templates are another dimension of this principle.

8. *An 80 percent solution today instead of 100 percent solution tomorrow.*

   Markets are moving too fast to provide 100 percent solutions. It has been my experience with dozens of accelerated projects that the customers are almost always willing to give up functionality for speed. Furthermore, studies have shown that more than 45 percent of the functionality of software applications is never used.

9. *Minimalism is essential.*

   LD seeks to eliminate waste by minimizing paperwork, keeping teams small and colocated, and keeping the product scope focused.

10. *Needs determine technology.*

    Choose the objectives of the development effort, and then the technology to support it, not vice versa. The technology options today are so vast, it is easy to spend more time changing technologies than delivering business applications.

11. *Product growth is feature growth, not size growth.*

   The critical factor in LD is delivering change-tolerant features. When evaluating new features, the team should always consider how business practices are impacted. Size of the product itself is not the issue.

12. *Never push LD beyond its limits.*

   This last one might be labeled, "Understand the category of problems that LD is designed to handle." LD is not a silver bullet, it is a framework driven by a certain class of business problems.

The Lean Development Environment

The three elements of a LD environment are "the policies and guidelines used to manage the effort; the processes, methods, and tools utilized; and a software/system architectural foundation" (Charette 2002). LD's policies and guidelines span a wide number of issues, from a project entry criterion that ensures that the value of the software to the customer has been determined to a self-learning guideline that says project lessons should be updated continuously.

One facet of LD is a focus on business "domains." Software is built for certain domains that make high-level reusability possible. Bob offers the example of the flight-control software for the Airbus, which is change tolerant and domain specific. The same basic flight control system is used on all airplane models, providing cost savings, flexibility, and lower pilot retraining costs.

The process side of LD, similar in part to Scrum and DSDM, has a distinct project management flavor, with three high-level phases: start-up, steady-state, and transition and renewal. The start-up phase focuses on a customer value analysis, a business study, and project feasibility. Based on Bob's focus and expertise in risk management, much of the start-up phase aims at reducing risks and increasing the capability of the team to assume additional risk.

The steady-state phase consists of an iterative, time-boxed "whirlpool"—a "build a little, test a little, build a little" process carried out by a design and build team. Bob depicts development activities—analysis, design, test, integrate, implement—whirling around an evaluate and reiterate core. Iterations should take no more than 90 days and preferably no more than 60 days.

Once the LD product has been delivered and accepted by the customer, the transition and renewal phase governs how the product passes into a renewal, or constant evolution, state. Transition includes knowledge transfer activities such as documentation and training.

Figure 21.2 shows why a process such as LD is necessary. The graph plots functionality versus time for both customer requirements and development capability. The horizontal distance between the two lines indicates the lateness in delivering products, and the vertical distance indicates the shortfall in features delivered. The area between the lines shows the inappropriateness of (or dissatisfaction with) the delivery process. The slope of the customer requirements line indicates the speed of business change—a greater slope indicates a more rapidly changing environment. The slope of the development capability line indicates the speed and adaptability of the delivery process.

Figure 21.2. Lean Development Drivers (Courtesy of Bob Charette)
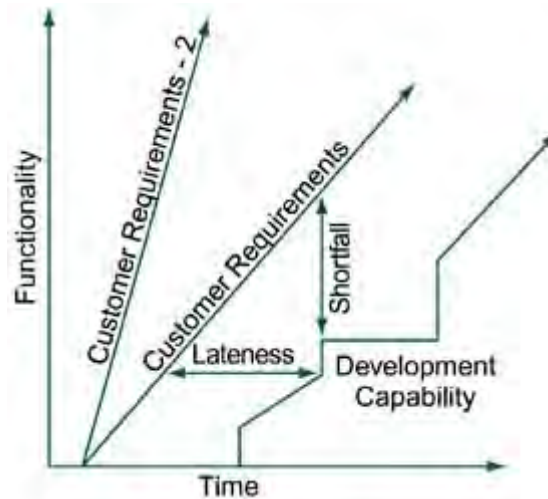
The changing slope of the customer requirements line reminds us of the old adage, "The faster we go, the behinder we get." The relevant question for many application development groups is not their rate of improvement, but whether or not their rate of capability improvement is keeping pace with the rate of customer requirements change.

Information Age business changes drive the slope of the customer requirements line in Figure 21.2 ever steeper, as shown by the second requirements line. RSMs will not be able to keep pace; the slope of the customer requirements line is getting steeper faster than the capability line.

In an environment of complexity and turbulence, LD creates a "virtuous circle," Bob says. Self-learning and self-adapting are encouraged through the use of short delivery cycles, which keep the effort close to the customer; lower costs, which make "throw-away" possible; and the use of templates, which encourages "reusable knowledge" in a transferable form.

Lean Development's Contributions

Bob's contribution to the Agile Software Development movement has four parts: strategic focus, a linkage to lean production, risk entrepreneurship, and stretch goals. First, of all the ASDE authors, Bob probably has the most presence in CEOs' offices. XP focuses at the developer-customer level, while Scrum and DSDM focus on project management. LD sells to top executive levels. Bob starts with the message that incremental improvements are not enough, that software delivery should learn the lessons of lean production in manufacturing—that major improvements require radical rethinking of the entire problem space.

Bob's stretch challenges require radical rethinking of how we go about software development. The challenge is a management and executive one of rethinking from top to bottom—not just imposing a set of new techniques on development teams, but changing how management approaches software development. The overarching issue remains change tolerance, the ability to create and respond to change as a business enterprise. Embodied in this goal is the need for management, at the highest levels, to better understand risk management and risk entrepreneurship. The risk of exploration must be accepted at that level and not driven down to the project teams as impossible goals ("Implement this highly risky project, and, by the way, there's no allowance for failure"). If management attempts to hide from risks, project teams will also.

Chapter 22. Extreme Programming

Always be doing the most important thing you can be working on.

—Kent Beck (XP Immersion workshop)

Extreme Programming[1]—or XP—exploded onto the software development scene in 2000–2001. Several of the other ASDE proponents toiled for years in relative obscurity before XP came on the scene. But XP hit a nerve and has helped vault the entire "Agile" category into the spotlight. Why? I think there are several reasons. First, its audience is developers, and there are legions of them in the world that are tired of "methodologies" getting in their way. Second, with the advent of Internet and Web-based development, there was a perceived need for a development approach geared to speed, flexibility, and quality. Third, Kent has been an effective promoter of XP. Fourth, Kent picked a great name—Extreme Programming—that targeted the developer audience and told the world that there was something new, something "extreme," inside.

[1] Portions of this chapter were first published in Cutter Consortium's *e-Business Application Delivery,* Vol. 12, No. 2, February 2000.

There is now an entire series of XP books by Kent Beck, Martin Fowler, Ron Jeffries, Chet Hendrickson, Ann Anderson, Bob Martin, James Newkirk, Ken Auer, Giancarlo Succi, and Michele Marchesi, with additional books on the way. So this chapter hits the highlights of XP and shows how it succeeds at software development by being both new and not-new at the same time.

Although Kent doesn't claim that practices like pair programming and iterative planning originated with XP, there are important concepts articulated by XP that are new. There's a lot of talk today about change, but Kent has good ideas about how to actually manage change, hence the subtitle to his book—*Embrace Change*. XP is derived from good practices that have been around for a long time. As Kent says, "None of the ideas in XP are new. Most are as old as programming." I might differ with Kent in one respect: While the practices XP uses aren't new, the conceptual foundation and how they are melded together are new and greatly enhance these "older" practices.

XP: The Basics

XP proponents are careful to articulate where they think XP is appropriate and where it is not. Even though practitioners like Kent and Ron may envision that XP has wider applicability, they are generally circumspect about their claims. For example, both are clear about XP's applicability to small (fewer than 10 people), colocated teams, a situation with which they have direct experience. They don't try to convince people that the practices will work for teams of 200.

The most prominent XP project reported on to date is the Chrysler Comprehensive Compensation system (the C3 project), which was initiated in the mid-1990s and converted to an XP project in 1997. Kent was brought in to improve the performance of the Smalltalk system and ended up revamping the entire approach to development. Kent brought Ron in to assist on a full-time basis, and he worked with the C3 team until spring 1999. The initial project scope was the payroll application for 10,000 salaried employees. The system consists of approximately 2,000 classes and 30,000 methods and was ready within a reasonable tolerance period of the planned schedule. Since work on C3 acted as a catalyst to bring the 12 practices of XP into focus, hundreds of projects and thousands of developers have used XP practices in various combinations and permutations.[2]

XP Practices

XP practices were originally intended for use with small, colocated teams, although people like Martin Fowler and others at ThoughtWorks have been working to extend XP to larger and distributed projects. XP's minimalist approach is legendary, at least as far as artifacts other than code and test cases are concerned. The way XP's practices are presented has both positive and negative aspects. At one level, they sound like rules—do this, don't do that. Kent explains that the practices are more like guidelines than rules, guidelines that are somewhat pliable depending on the situation. The XP practices are truly a system of practices designed to interact, counterbalance, and reinforce each other, such that picking and choosing which to use and which to discard can be tricky.

The planning game. XP's planning typifies that of other Agile approaches. Short, three-week iterations, frequent plan updates, and assigning "stories" (a story defines a particular feature requirement and is displayed on a simple 3x5 card) are all part of XP's approach to planning. The planning game defines the boundary—or better, the point of interaction—between customers and developers, specifying the responsibilities of both parties. Developers can, and should, offer stories for consideration, but customers make priority decisions. The planning game includes a release plan, which indicates the stories assigned to each iteration until the product's release date. Although this provides an indication of the entire project's scope, cost, and schedule, all parties assume the plan is really a speculation about the future. Both customers and developers collaborate in the planning game. Martin described a ThoughtWorks project in which 40 people participated in the triweekly planning game. Although unwieldy at times, joint participation helped everyone understand the plan in ways that reading a document could not.

Small releases. "Every release should be as small as possible, containing the most valuable business requirements," states Kent (Beck 2000). This echoes two of Tom Gilb's principles of evolutionary delivery: "All large projects are capable of being divided into many useful partial result steps" and "Evolutionary steps should be delivered on the principle of the juiciest one next" (Gilb 1988).

Small releases provide a sense of accomplishment that is often missing in long projects and frequent, relevant feedback. However, a development team needs to consider the difference between "release" and "releasable." The cost of each release—installation, training, conversions—needs to be factored into whether or not the features delivered at the end of an iteration are actually released to the end user or simply declared to be in a releasable state.

Metaphor. XP's use of the terms "metaphor" and "story" take a little wearing to become comfortable. However, both terms help make technology understandable, especially to customers. At one level, metaphor and architecture are synonyms—they both are intended to provide a broad view of the project's goal. But architectures often get bogged down in symbols and connections. XP utilizes the term metaphor in an attempt to define an overall coherent theme that developers and, to a lesser extent, customers can relate to. The metaphor describes the broad sweep of the project, while stories are used to describe individual features.

Some critics have equated metaphor to architecture and have then proceeded to criticize XP for being woefully lacking in architecture. To my mind, metaphor wasn't intended to replace architecture but to act as a focusing point for design and programming activities—a high-level vision of architecture. Having reviewed many "architectural" diagrams that were

incomprehensibly voluminous and detailed, I concluded that a simple focusing metaphor would have been welcome.

Simple design. Simple design has two parts: (1) Design for the functionality that has been defined, not for potential future functionality, and (2) Create the best—as in simplest—design that can deliver that functionality. In other words, don't guess about the future—create the best, simple design you can today. "If you believe that the future is uncertain, and you believe that you can cheaply change your mind, then putting in functionality on speculation is crazy," says Kent. "Put in what you need when you need it" (Beck 2000).

In the early 1980s, I published an article in *Datamation* magazine titled "Synchronizing Data with Reality" (Highsmith 1981). The gist of the article was that data quality is a function of use, not capture and storage. Furthermore, it postulated that data that was not systematically used would rapidly deteriorate as the world changes and the data doesn't. Data quality is a function of systematic usage, not anticipatory design. Trying to anticipate what data we will need in the future only ensures that we will probably design for data that we will never use, and even if we guess correctly about what data we will need, the data itself won't be correct by the time we get there. XP's simple design approach embraces the same concepts. This doesn't mean that no anticipatory design ever happens; however, it does mean that the viability of anticipatory design has changed dramatically in our volatile business environment.

Refactoring. If I had to pick one thing that sets XP apart from other approaches, it would be refactoring—the ongoing redesign of software to improve its responsiveness to change. While RAD approaches have often been associated with little or no design, XP should be thought of as continuous design. In times of rapid, constant change, much more attention needs to be focused on refactoring.

Refactoring is closely related to factoring, or what is referred to today as using design patterns. Design patterns serve modern-day object-oriented development much as Larry Constantine and Ed Yourdon's structured design served a previous generation—they provide guidelines for program structures that are more effective than other program structures. Design patterns provide the means for improving the quality of initial designs by offering models that have proven effective in the past.

You might think, "Why have a separate refactoring practice? Can't we just utilize the design patterns in redesign?" As all developers, and their managers, understand, altering existing code can be a ticklish proposition. The cliché "If it ain't broke, don't fix it" lives on in the annals of development folklore. However, as Martin comments in his book *Refactoring: Improving the Design of Existing Code,* "The program may not be broken, but it does hurt." Fear of breaking some part of the code base that's "working" actually hastens the degradation of that code base. However, Martin is well aware of the concern: "So before I do the refactoring I need to figure out how to do it safely" (Fowler 1999). Martin's book catalogs not only the "before" (poor code) and "after" (better code based on patterns), but also the steps required to migrate safely from one to the other. These migration steps reduce the chances of introducing defects during the refactoring effort.

Kent describes his "two hat" approach to refactoring, in which adding new functionality and refactoring are two different activities. Refactoring, per se, doesn't change the observable behavior of the software, it enhances the internal structure. When new functionality needs to be added, the first step is often to refactor in order to simplify the adding of new functionality. The proposed new functionality, in fact, provides one impetus to refactor.

Refactoring might be thought of as incremental, as opposed to monumental, redesign. "Without refactoring, the design of the program will decay," says Martin. "Loss of structure has a cumulative effect" (Fowler 1999). Historically, our approach to maintenance has been "quick and dirty," so even in those cases in which good initial design work was done, it degraded over time.

Testing. XP is full of interesting twists that encourage one to think. How about "test, and then code?" I've worked with software companies, and a few IT organizations, in which programmer performance was measured by lines of code delivered, while testing performance was measured by defects found—neither group was motivated to reduce the number of defects prior to testing. XP utilizes two types of testing: unit and functional. However, the practice for unit testing involves developing the test for the feature prior to writing the code. Furthermore, the tests should be automated. Once the code is written, it is immediately subjected to the test suite—instant feedback. Thinking back to my own programming days, test-first sounded like a good idea, but watching Bob Martin and James Grenning walk through a test-first example at their XP Immersion workshop convinced me.

A development life cycle emerges from XP: listen (requirements)—test—code—design. Listen closely to customers while gathering their require ments. Develop test cases. Code the objects (using pair programming). Design (or refactor) as more objects are added to the system. This seemingly convoluted life cycle begins to make real sense in an environment in which change dominates.

Pair programming. One of the few software engineering practices that enjoys near-universal acceptance (in theory at least) and has been well measured is software inspections. At their best, inspections are collaborative interactions that speed learning as much as they uncover defects. At their worst, inspections degenerate into unproductive personal attack sessions accompanied by reams of forms and metrics. One of the lesser-known facts about inspections is that while they are very cost effective in uncovering defects, they are even more effective at preventing defects in the first place through the team's ongoing learning and incorporation of better practices.

Pair programming takes inspections to the next step—rather than the incremental learning using inspections, why not continuous learning using pair programming? "Pair programming is a dialogue between two people trying to simultaneously program and understand how to program better," says Kent (Beck 2000). Having two people sitting in front of the same terminal, one entering code or test cases, one reviewing and thinking, creates a continuous, dynamic interchange. Research conducted by Laurie Williams for her doctoral dissertation at the University of Utah confirms that pair programming benefits aren't just wishful thinking (Williams et al. 2000, Williams and Kessler, manuscript in preparation).

Collective ownership. Collective ownership gives anyone on the project team "permission" to change any of the code at any time. For many programmers, and certainly for many managers, the prospect of collective ownership raises concerns, from "I don't want those bozos changing my code" to "Who do I blame when problems arise?"

Collective ownership provides another level to the collaboration begun by pair programming. Pair programming encourages two people to work closely together—each drives the other a little harder to excel. Collective ownership encourages the entire team to work collaboratively—each individual, each pair strives a little harder to produce high-quality designs, code, and test cases. Granted, all this forced "togetherness" may not work for every project team.

For all the hoopla about teams and teamwork over the past decade or two, teams still resist joint ownership of results. "Of course I'm a team player," everyone indignantly responds when questioned about their "teaming" skills. "But hey, don't think you're going to get me to do this collective ownership or pair programming stuff—I don't want anyone else mucking around with *my* code." To these individuals, I offer the question, "OK, so in what kinds of team practices *are* you going to participate?"

Continuous integration. Daily builds have become the norm in many software companies, mimicking the published material on the "Microsoft" process (Cusumano and Shelby 1995). While many companies practice daily builds, XP practitioners say that daily integration is wimpy, opting instead for frequent builds every couple of hours. XP's feedback cycles are quick—develop

the test case, code, integrate (build), test. The perils of integration defects have been understood for many years, but the discipline of XP and newer tools have helped put that knowledge to good use.

40-hour week. "Sustainable development," the Agile Manifesto phrase, corresponds to the XP 40-hour practice. Hours are not the entire issue, but the 40-hour rule establishes a philosophy. Ron once told me, "What we say is that overtime is defined as time in the office when you don't want to be there, and that you should work no more than one week of overtime. If you go beyond that, there's something wrong—and you're tiring out and probably doing worse than if you were on a normal schedule." In his book *Slack,* Tom DeMarco agrees: "Extended overtime is a productivity-reducing technique."

I mentioned to Ron that there are situations in which working 40 hours is pure drudgery and others in which the team has to be pried away from a 60-hour work-week. "I'm with you on the 60-hour weeks when we were young and eager," Ron replied. "They were probably OK. It's the dragging weeks to watch for."

For me, "sustainable" indicates volunteered commitment. Do people want to come to work? Do they anticipate each day with great relish? People have to come to work, but they perform great feats by being committed to the project, and commitment arises only from a sense of purpose. Practitioners talk about the intensity of pair programming and note that three or four hours at this intense level is exhilarating, draining, and very productive.

On-site customer. This practice corresponds to one of the oldest cries in software development—user involvement. XP, like every other ASDE, calls for ongoing, on-site user involvement with the project team.

Coding standards. XP practices are supportive of each other. For example, if you do pair programming and let anyone modify collective code, then coding standards are a necessity.

Values and Principles

On *Saturday,* January 1, 2000, the *Wall Street Journal* (you know, the Monday through Friday newspaper) published a special 58-page millennial edition. The introduction to the Industry & Economics section was written by Tom Petzinger, who stated, "The bottom line [is that] creativity is overtaking capital as the principal elixir of growth."

Petzinger wasn't talking about a handful of creative geniuses but the creativity of groups—from teams to departments to companies. Once we leave the realm of the single creative genius, creativity becomes a function of the environment and how people interact and collaborate to produce results. If your company's fundamental principles point to software development as a statistically repeatable, rigorous, engineering process, then XP is probably not for you. Although XP contains certain disciplined practices, its intent is to foster creativity and communication.

Environments are driven by values and principles. XP (or other ASDEs) may or may not work in your organization, but ultimately success won't depend on 40-hour work-weeks or pair programming—it will depend on whether the values and principles of XP align with those of your organization. Kent identifies four values, five fundamental principles, and ten secondary principles, but I'll mention five that are key.

Communication. So what's new here? It depends on one's perspective. XP focuses on building person-to-person mutual understanding of the problem environment through minimal formal documentation and maximum face-to-face interaction. "Problems with projects can invariably be

traced back to somebody not talking to somebody else about something important," says Kent (Beck 2000). XP's practices are designed to encourage interaction—developer to developer, developer to customer.

Simplicity. XP asks of each team member, "What is the simplest thing that could possibly work?" Make it simple today, and create an environment in which the cost of change tomorrow is low.

Feedback. "Optimism is an occupational hazard of programming," says Kent. "Feedback is the treatment" (Beck 2000). Whether it's hourly builds or frequent functionality testing with customers, XP embraces change by constant feedback. While every approach to software development advocates feedback—even the much-maligned waterfall model—the difference is that XP practitioners understand that feedback is more important than feedforward. Whether it's fixing an object that failed a test case or refactoring a design that is resisting change, high-change environments require a keen understanding of feedback.

Courage. Whether it's a CMM practice or an XP practice that defines one's discipline, discipline requires courage. Many might define courage as doing what's right, even when pressured to do something else. Developers often cite the pressure to ship a buggy product and the courage to resist. However, the deeper issues can concern legitimate differences of opinion as to what is right. Oftentimes, people don't lack courage but conviction, which puts us right back to other values. If a team's values aren't aligned, its members won't be convinced that some practice is "right," and without conviction, courage doesn't seem so important. It's hard to work up the energy to fight for something you don't believe in.

"Courage isn't just about having the discipline," Ron told me. "It is also a resultant value. If you do the practices that are based on communication, simplicity, and feedback, you are given courage, the confidence to go ahead in a lightweight manner."

Quality work. OK, so how many of you advocate poor-quality work? Whether you are a proponent of the Rational Unified Process, the CMM, or XP, the real issues are "How do you define quality?" and "What actions do you think deliver high quality?" Defining quality as "no defects" provides one perspective on the question. Jerry Weinberg's definition, "Quality is value to some person," provides another. The first question to ask of any methodology should be, "Do the designers of the methodology believe in the fundamental principle that individuals *want* to do a good, high-quality job?"

This is a fundamental tenet of XP proponents and other Agilists—we fundamentally believe in people and their desire to do high-quality work. If this is not your fundamental belief, then thousands and thousands of pages of methodology manuals are needed to manage and control every aspect of people's behavior. There will always be a condition or situation that isn't covered, so another 100 pages need to be added to cover it. Then another. Then another. If you fundamentally trust people, you provide a crisp definition of the desired outcome, a light framework within which to work, a feedback mechanism to ensure communications haven't been snarled, and you then let people perform. Maybe this is one reason XP has attracted such a large following—the practices are based on trust and quality work, two characteristics that appeal to many developers.

XP's Contributions

Of all the Agile approaches, XP has generated the most interest to date and is the most concrete in terms of specific practices for a well-defined problem domain—small, colocated teams. Kent and the other codevelopers of XP have wrapped the conceptual ideas of embracing change, collaborative teamwork, altering the often dysfunctional relationships between customers and

developers, and simple, generative rules into a set of 12 practices guided by set of well-articulated principles. A number of XP practitioners are exploring ways to expand the use of XP into other problem domains, particularly those involving larger projects and distributed teams.

The popularity of XP—including publicity from both proponents and critics—has fostered lively and valuable debate within the software development community. Whatever your particular thoughts or feelings about XP (or other Agile approaches, for that matter), the debate and discussion fostered by the XP community have greatly benefited our profession.

Chapter 23. Adaptive Software Development

Collaboration is difficult, especially when it involves other people.

—Ken Orr (Cutter Consortium Summit 2001)

In 1992, I started working on a short-interval, iterative, RAD process that evolved into Adaptive Software Development. The original process, developed in conjunction with colleague Sam Bayer, was used to assist in marketing a mainframe RAD tool. Sam and I worked with prospects on pilot projects—one-month projects with one-week iterations—in companies from Wall Street brokerage houses to airlines to telecommunications firms. Over the next several years, Sam and I (together and separately) successfully delivered more than 100 projects using these practices, and in June 1994, we published an article on our experiences (Bayer and Highsmith 1994). During the early to mid-1990s, I also worked with software companies that were using similar techniques on very large projects, while Sam continued to evolve the practices in his work.

In the mid-1990s, my interest in complex adaptive systems began to add a conceptual background to the team aspects of the practices and was the catalyst for the name change from RADical Software Development to Adaptive Software Development (Highsmith 1997). ASD has been used by companies from New Zealand to Canada for a wide range of project and product types. Interestingly, I became aware of XP just a month prior to the publication of *Adaptive Software Development* (Highsmith 2000), when Kent and I exchanged emails. *Extreme Programming Explained* (Beck 2000) and *Adaptive Software Development* were published within a couple of months of each other.

Complexity theory helps us understand unpredictability and that our inability to predict doesn't imply an inability to make progress. ASD works *with* change rather than fighting against it. In order to thrive in turbulent environments, we must have practices that embrace and respond to change—practices that are adaptable. Even more important, we need people, teams, and organizations that are adaptable and Agile. Agile practices alone are not nearly enough; they depend on competent individuals who are nimble and thoughtful. We have become so enamored of precise planning, we forget that products evolve from a little planning and a lot of learning as we proceed.

For all the many books on requirements engineering, the best way to determine how a product should evolve is to use it. Iteration—building, trying, succeeding, failing, rebuilding—governs successful product development, particularly in extremely competitive environments. "Good enough" requirements need to be followed by quick delivery and use, and then followed with evolutionary changes to the requirements and the product based on that use. Although a number of modern software development life cycles have adopted an iterative approach, they still miss the mark in dealing with the messiness of complex environments. Despite the fact that development is iterative, many people's fundamental assumptions are still deterministic—they think of short waterfall life cycles strung together.

The practices of ASD are driven by a belief in continuous adaptation—a different philosophy and a different life cycle—geared to accepting continuous change as the norm. In ASD, the static plan-design-build life cycle is replaced by a dynamic Speculate-Collaborate-Learn life cycle (see Figure 23.1). It is a life cycle dedicated to continuous learning and oriented to change, reevaluation, peering into an uncertain future, and intense collaboration among developers, management, and customers.

Figure 23.1. The Speculate-Collaborate-Learn Life Cycle
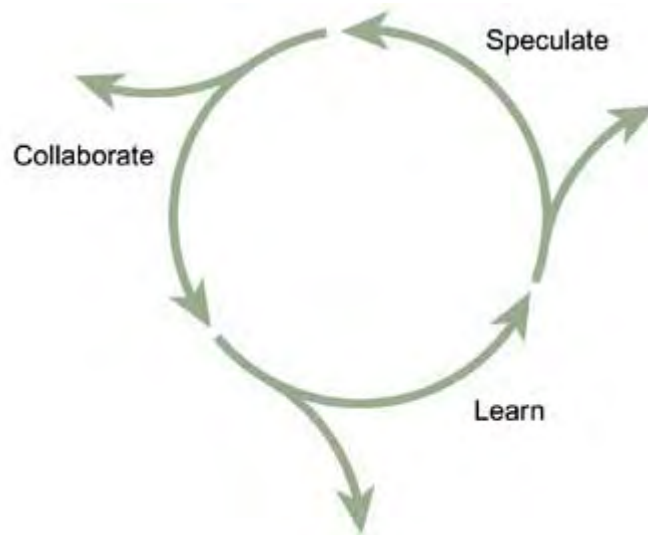
A Change-Oriented Life Cycle[1]

A waterfall development life cycle, based on an assumption of a relatively stable business environment, becomes overwhelmed by high change. Planning is one of the most difficult concepts for engineers and managers to reexamine. For those raised on the science of reductionism (reducing everything to its component parts) and the near-religious belief that careful planning followed by rigorous engineering execution produces the desired results (we are in control), the idea that there is no way to "do it right the first time" remains foreign. The word "plan," when used in most organizations, indicates a reasonably high degree of certainty about the desired result. The implicit and explicit goal of "conformance to plan" restricts a manager's ability to steer the project in innovative directions.

"Speculate" gives us room to explore, to make clear the realization that we are unsure, to deviate from plans without fear. It doesn't mean that planning is obsolete, just that planning is acknowledgeably tenuous. It means we have to keep delivery iterations short and encourage iteration. A team that "speculates" doesn't abandon planning, it acknowledges the reality of uncertainty. Speculation recognizes the uncertain nature of complex problems and encourages exploration and experimentation. We can finally admit that we don't know everything.

The second conceptual component of ASD is collaboration. Complex applications are not built, they evolve. Complex applications require that a large volume of information be collected, analyzed, and applied to the problem—a much larger volume than any individual can handle by him- or herself. Although there is always room for improvement, most software developers are reasonably proficient in analysis, programming, testing, and similar skills. But turbulent environments are defined in part by high rates of information flow and diverse knowledge requirements. Building an eCommerce site requires greater diversity of both technology and business knowledge than the typical project of five to ten years ago. In this high-information-flow environment, in which one person or small group can't possibly "know it all," collaboration skills (the ability to work jointly to produce results, share knowledge, or make decisions) are paramount.

Once we admit to ourselves that we are fallible, then learning practices—the "Learn" part of the life cycle—become vital for success. We have to test our knowledge constantly, using practices

like project retrospectives and customer focus groups. Furthermore, reviews should be done after each iteration rather than waiting until the end of the project.

An ASD life cycle has six basic characteristics:

1. Mission focused
2. Feature based
3. Iterative
4. Time-boxed
5. Risk driven
6. Change tolerant

For many projects, the requirements may be fuzzy in the beginning, but the overall mission that guides the team is well articulated. (Jens Coldeway's insurance project discussed in Chapter 11 is a good example of this.) Mission statements act as guides that encourage exploration in the beginning but narrow in focus over the course of a project. A mission provides boundaries rather than a fixed destination. Without a good mission and a constant mission refinement practice, iterative life cycles become oscillating life cycles—swinging back and forth with no progress. Mission statements (and the discussions leading to those statements) provide direction and criteria for making critical project tradeoff decisions.

The ASD life cycle focuses on results, not tasks, and the results are identified as application features. Features are the customer functionality that is to be developed during an iteration. While documents (for example, a data model) may be defined as deliverables, they are always secondary to a software feature that provides direct results to a customer. (A customer-oriented document such as a user manual is also a feature.) Features may evolve over several iterations as customers provide feedback.

The practice of time-boxing, or setting fixed delivery times for iterations and projects, has been abused by many who use time deadlines incorrectly. Time deadlines used to bludgeon staff into long hours or cutting corners on quality are a form of tyranny; they undermine a collaborative environment. It took several years of managing ASD projects before I realized that time-boxing was minimally about time—it was really about focusing and forcing hard tradeoff decisions. In an uncertain environment in which change rates are high, there needs to be a periodic forcing function to get work finished.

As in Barry Boehm's spiral development model, the plans for adaptive iterations are driven by analyzing the critical risks. ASD is also change tolerant, not viewing change as a "problem" but seeing the ability to incorporate change as a competitive advantage.

The Basic ASD Life Cycle

Figure 23.2 shows an expansion of the ASD life cycle phases into specific practices.

Figure 23.2. The Adaptive Life Cycle Phases

Adaptive Lifecycle
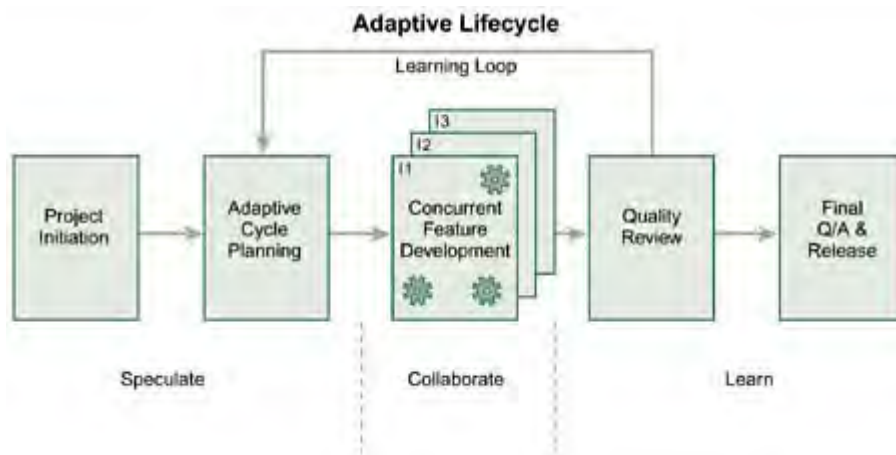
Speculate: Initiation and Planning

There are five general steps in "speculating," although the word "steps" is somewhat of a misnomer, as each step may be revised several times during the initiation and planning phase. First, project initiation involves setting the project's mission and objectives, understanding constraints, establishing the project organization, identifying and outlining requirements, making initial size and scope estimates, and identifying key project risks. Because speed is usually a major consideration in using ASD, much of the project initiation data should be gathered in a preliminary JAD session. Initiation can be completed in a concentrated two- to five-day effort for a small- to medium-sized project or take two or three weeks for larger projects. During the JAD sessions, requirements are gathered in enough detail to identify features and establish a skeletal object, data, or other architectural model.

Next, the time-box for the entire project is established based on the scope, feature set requirements, estimates, and resource availability that result from project initiation work. Speculating doesn't abandon estimating, it just means accepting that estimates are tenuous.

The third step is to decide on the number of iterations and assign a time-box to each one. For a small- to medium-sized application, iterations usually vary from four to eight weeks. Some projects work best with two-week iterations, while others might require more than eight weeks (although this is rare). The overall project size and the degree of uncertainty are two factors that determine individual iteration lengths.

After establishing the number of iterations and a schedule for each, the team members develop a theme or objective for each of the iterations. Just as it is important to establish an overall project objective, each iteration should have its own theme (this is similar to the Sprint Goal in Scrum). Each iteration delivers a demonstrable set of features to a customer review process, making the product visible to the customer. Within the iterations, "builds" deliver working features to a daily (or more frequent) integration process, making the product visible to the development team. Testing is an ongoing, integral part of feature development—not an activity tacked on at the end.

Developers and customers assign features to each iteration. The most important criterion for feature assignment is that every iteration must deliver a visible, tangible set of features to the customer. In the assignment process, customers decide on feature prioritization, using feature estimates, risks, and dependency information supplied by the development team. A spreadsheet is an effective tool for feature-based iteration planning. Experience has shown that this type of planning—done as a team rather than by the project manager—provides better understanding of the project than a traditional task-based approach. Feature-based planning reflects the uniqueness of each project.

Collaborate: Concurrent Feature Development

176

While the technical team delivers working software, project managers facilitate collaboration and concurrent development activities. For projects involving distributed teams, varying alliance partners, and broad-based knowledge, how people interact and how they manage interdependencies are vital issues. For smaller projects in which team members work in physical proximity, collaboration can consist of informal hallway chats and whiteboard scribbling. Larger projects, however, require additional practices, collaboration tools, and project manager interaction.

Collaboration, an act of shared creation, is fostered by trust and respect. Shared creation encompasses the development team, customers, outside consultants, and vendors. Teams must collaborate on technical problems, business requirements, and rapid decision making.

Learn: Quality Review

Learning becomes increasingly difficult in environments in which the "get it right the first time" mantra dominates and development progresses in a linear, waterfall fashion. If people are continually compelled to get it right, they won't experiment and learn. In waterfall development, each phase completion discourages backtracking because there shouldn't be any mistakes. Learning from mistakes and experimentation requires that team members share partially completed code and artifacts early, in order to find mistakes, learn from them, and reduce the total amount of rework by finding small problems before they become large ones. Teams must learn to differentiate between shoddy work and half-done work.

There are four general categories of things to learn about at the end of each development iteration:

- Result quality from the customer's perspective
- Result quality from a technical perspective
- The functioning of the delivery team and the practices team members are utilizing
- The project's status

Getting feedback from the customers is the first priority in Adaptive projects. ASD's recommended practice for this is a customer focus group. Derived from the concept of marketing focus groups, these sessions are designed to explore a working model of the application and record customer change requests. They are facilitated sessions, similar to JAD sessions, but rather than generating requirements or defining project plans, customer focus groups are designed to review the application itself. Customers relate best to working software, not documents or diagrams.

The second review area is technical quality. A standard practice for technical quality assessment is periodic technical reviews; pair programming accomplishes a similar result. Although code reviews or pair programming should be continuous, other reviews, such as an overall technical architecture review, may be conducted weekly or at the end of an iteration.

The third feedback area is for the team to monitor its own performance. This might be called the people and process review. End-of-iteration mini-retrospectives help determine what's not working, what the team needs to do more of, and what the team needs to do less of. Retrospectives encourage teams to learn about themselves and how they work together.

The fourth category of review is project status. This leads into a replanning effort for the next iteration. The basic status review questions are: Where is the project? Where is it versus the plans? Where should it be? Determining a project's status is different in a feature-based approach. In a waterfall life cycle, completed deliverables mark the end of each major phase (a complete requirements document, for example, marks the end of the specification phase). In a feature-based approach, completed features—working software—mark the end of each iteration.

The last of the status questions is particularly important: Where "should" the project be? Since the plans are understood to be speculative, measurement against them is insufficient to establish progress. The project team and customers need to continually ask, "What have we learned so far, and does it change our perspective on where we need to go?"

Leadership-Collaboration Management

Many companies are steeped in a tradition of optimization, efficiency, predictability, control, rigor, and process improvement. The emerging Information Age economy requires adaptability, speed, collaboration, improvisation, flexibility, innovation, and suppleness. "Successful firms in fiercely competitive and unpredictably shifting industries pursue a competing on the edge strategy," write Shona Brown and Kathleen Eisenhardt. "The goal of this strategy is not efficiency or optimality in the usual sense. Rather, the goal is flexibility—that is, adaptation to current change and evolution over time, resilience in the face of setbacks, and the ability to locate the constantly changing sources of advantage. Ultimately, it means engaging in continual reinvention" (Brown and Eisenhardt 1998).

What we name things is important. Traditional management has been characterized by the term Command-Control. It is reminiscent of the traditions of the military, Fredrick Taylor's scientific management, and William Whyte's "Organizational Man" of the 1950s. While new terms have spewed forth from the pages of the *Harvard Business Review* and management books—empowerment, participative management, learning organization, human-centered management, and the like—none have encompassed the breadth, or depth, necessary for managing modern organizations. One alternative to Command-Control management is Dee Hock's "chaordic" style of management (Hock 1999), which I've called Leadership-Collaboration in ASD. It applies to both project and organizational management.

Today, we need leaders more than commanders. "Commanders know the objective; leaders grasp the direction. Commanders dictate; leaders influence. Controllers demand; collaborators facilitate. Controllers micro-manage; collaborators macro-manage. Managers who embrace the Leadership-Collaboration model understand their primary role is to set direction, to provide guidance, and to facilitate connecting people and teams" (Highsmith 2000). Leaders understand that occasionally they need to command, but that's not their predominant style. Leaders provide direction and create environments where talented people can be innovative, creative, and make effective decisions. The Leadership-Collaboration model encompasses the basic philosophical assertion that in turbulent environments, "adaptation" is more important than "optimization."

However, becoming adaptive is not easy. The shift from optimization to adaptation—for an organization or a project team—requires a profound cultural shift. First, we must let go of our need to be tidy and orderly. Of course there are places where orderliness is necessary, but in general, the business world's pace of change precludes it. We need practices such as risk management and configuration control, but to assemble hundreds of practices under the roof of orderliness and statistically controlled rigor ignores the real world.

My guideline for "barely sufficient" rigor in complex projects is "employ slightly less than just enough." Why? Complex problems—those characterized by high speed, high change, and uncertainty—are solved by creativity, innovation, good problem solving, and effective decision making. All of these capabilities suffer from an emphasis on rigorous, optimizing processes. To stabilize a process (make it repeatable), we endeavor to restrict inputs, transfer only the necessary information between processes (for efficiency), and strive to make the transformation as algorithmic as possible. But complex problems in today's organizations require the interaction of many people, diverse information, out-of-the-box thinking, quick reaction, and, yes, rigorous activity at times. Balancing at the edge of chaos provides just enough rigor to keep from plunging

into chaos, but not enough to quell creative juices. Chaos is easy—just do it. Stability is easy—just follow the steps. Balancing is hard—it requires enormous managerial and leadership talent.

There is a subtle difference between adapting and changing. Adapting is outwardly focused; it depends on a free flow of information from the environment (one's market) and realistic decisions based upon information and organizational values (goals). Many changes in organizations are in response to internal politics rather than external information. Adaptive organizations learn from the outside in rather from the inside out.

Adaptation needs to happen at all organizational levels, which means free-flowing information and distributed decision making. It means that the traditional hierarchical management structure must share power with the horizontal, networked team structure. Hierarchical power is stabilizing; networked power is more adapting. Too much of one type of power breeds stagnation, while too much of the other breeds chaos. Sharing power is another fundamental characteristic of adap table organizations.

In organizations, power defines who gets to make decisions. Over time, the pattern of decisions an organization makes determines success or failure. Control-oriented managers hoard decision making. New-age managers empower others to make decisions, often diffusing power so much that little gets done. Adaptive managers understand that making good decisions is more important than prescribing activities; further more, they understand when to make decisions and when to distribute them. It is a constant, delicate balancing act that depends on understanding patterns that work rather than processes that prescribe.

Optimizing cultures tend to see the world as black or white. Adaptive cultures, on the other hand, recognize gray. They understand that planning is tenuous and control nearly impossible. Adaptive cultures understand that success evolves from trying a succession of different alternatives and learning from success and failure. Adaptive cultures understand that learning about what we don't know is often more important than doing things we already know how to do. The traditional, linear, Newtonian cause-and-effect perspective no longer adequately models the nature of our world. The concepts of complex systems, and the Leadership-Collaboration ideas about organizational and project management derived from them, provide a better sense of our economic environment.

ASD's Contributions

My hope for ASD is that it contributes to the discussion about the perspective, values, principles, and practices of Agile Software Development and, more critically, how project teams and organizations need to respond to our change-driven economy.

Part IV: Developing an ASDE

Chapter 24. Articulating Your Ecosystem

A methodology is the conventions that your group agrees to. "The conventions your group agrees to" is a social construction.

—*Alistair Cockburn,* Agile Software Development

How can we determine if an Agile Software Development Ecosystem fits an organization's particular culture, strategic opportunity, or problem domain? Although strategies for changing organizational culture are far beyond the scope of this book, articulating an organization's culture is a critical step in successfully implementing an ASDE—or any methodology, for that matter.

A software development methodology reflects the accumulated knowledge of the practices and processes that have demonstrated an ability to deliver results. An ecosystem reflects deeper cultural issues of how we think about people and their relationships to each other and organizations. Ecosystems reflect both *what* we practice and *why* we think those practices are effective. This raises three critical issues in selecting or designing a methodology:

1. What are the strategic opportunities and problem domains?
2. What is the culture of the organization?
3. Are the opportunity and culture congruent?

Traditional RSMs have undeniably contributed to solving key project management and software engineering problems, as have ASDEs. So rather than launch into a debate about whether one or another approach is right or wrong, it seems more appropriate to define relevant business problem domains, examine different fundamental cultures, and then match problem domain with culture. ASDEs can then be addressed within this problem domain and cultural context. For example, few would argue that an Internet start-up company and a hierarchical, Command-Control culture would be an effective match. Similarly, the methodical mining of copper ore in a huge open-pit mine might not be the best environment for a laid-back Silicon Valley culture.

Opportunity and Problem Domains

As in exploring for oil or building an investment portfolio, companies need to approach projects from a portfolio perspective—assessing risks and benefits. Unfortunately, most companies don't go beyond portfolio assessment to the critical execution aspects of portfolio management—selecting a methodology to match the problem type and then determining the appropriate success criteria that match the problem type. A Web-based eBusiness project requires a different methodology than a payroll system enhancement. Furthermore, the measures of success for these two projects may be completely different.

Opportunities and problems should be evaluated at both strategic and tactical levels. I've chosen Geoffrey Moore's (1995) technology adoption model to illustrate the strategic level, but other models could be substituted. Moore models how people react to discontinuous innovation. The adoption model graphs a bell-shaped curve that measures risk aversion, beginning with early innovators, who are the first to embrace a new technology, and ending with the conservative laggards, who grudgingly adopt a technology only when there seems to be no alternative. The four critical phases in the model are: Early Market, Bowling Alley, the Tornado, and Main Street.

The Early Market is driven by enthusiasts who enjoy tinkering with new technology. They have little money but are essential to early marketing efforts. The enthusiasts are interested in the technology itself, not necessarily in its usefulness to the business.

The Bowling Alley lies between sporadic acceptance in the Early Market and wide acceptance on Main Street. Innovative revolutionaries who are forging new business opportunities populate the Bowling Alley. They want to exploit new technology and make a mark, and they have money. But the Bowling Alley also lies in the chasm between the Early Market and Main Street—a gulf between early revenues and sustainable revenue flows that is much wider (in time) than companies forecast.

The Bowling Alley strategy involves picking a market segment and working to provide a total problem solution for that single segment. When one segment is satisfied, the next is targeted (as if it were the next bowling pin in a bowling alley). A Tornado forms (possibly) when enough "pins" have fallen and the market virtually explodes, as if a tornado were sweeping through the landscape.

Main Street is more sedate. During the Tornado, market share positions are established—usually for a product's entire life (hence the intense pressure to emerge in first place from the Tornado). The Tornado establishes a product as the "standard," and year after year of revenue and profits flow from that dominant market position. Microsoft, Oracle, and Cisco were all beneficiaries of Tornado markets.

Traditional firms—the bricks-and-mortar variety—operate on Main Street. Their management practices and cultures have evolved to sustain their position. Occasionally, they have had to spin off a subunit (such as IBM's PC group) to handle a disruptive technology, but these firms have less experience in the rough-and-tumble world of the Bowling Alley and the Tornado. Yet this is where Main Street companies have to learn to operate. eBusiness and eCommerce business initiatives, and the projects that support these initiatives, are flinging these companies back into the turbulence of early product phases—not all of the business surely, but enough that spinning off a separate unit here and there is not a viable long-term strategy.

Many IT organizations, effective in supporting Main Street operations, fail to consider the dynamics of early markets. The Early Market, the Bowling Alley, and the Tornado—these are the market domains in which agility is a critical skill.

Just as oil companies must do both exploration and production drilling, organizations have to deliver software to meet a variety of opportunities and problems. Although a company may be predominantly on Main Street, it may also have new products in the Early Market and others just entering the Bowling Alley. Projects supporting each of these phases will have different objectives, uncertainty, risk profiles, criticality, and constraints—and therefore different methodologies.

Cultural Domain

More methodology implementations flounder from failing to adequately account for an organization's cultural factors than for any other reason. A particular culture is not necessarily change tolerant or change resistant—but it may resist certain types of changes and embrace others. Culture provides both the strength to persist in the face of adversity and the weakness to persist in the face of folly. Any attempt to implement a methodology requires that an organization analyze its own fundamental culture—its core values. Many methodology failures are caused by a problem definition followed by a solution design, with little analysis of whether or not the solution design fits the company or the project team's culture.

According to Moore, there are four basic organizational cultures: cultivation, competence, collaboration, and control ([Moore 2000](#)).[1] Each is characterized by a particular motivation and illustrated by well-known organizations. A cultivation culture motivates by self-actualization (from Maslow's work) and can be illustrated by Silicon Valley start-up companies. A competence culture is driven by the need for achievement: Microsoft is an obvious example. Collaboration cultures, epitomized by Hewlett-Packard, are driven by a need for affiliation, whereas control cultures, like IBM and GE, are motivated by the need for power and security.

[1] The cultural models are based on work in Schneider (1994).

Cultures aren't right or wrong, they just are. And while cultures aren't right or wrong, in the extremes they can be dysfunctional, and they can certainly be appropriate or inappropriate for different parts of an organization or for different stages in a product's evolution from Early Market to Main Street.

Moore contends that from a business management perspective, the critical aspect of culture can be reduced to the basis on which organizations make tough decisions. Objective data, inspirational ideas, and subjective insight each form a basis for decision making (and, of course, no one uses one criterion to the exclusion of others). A culture that makes decisions based on inspirational ideas wouldn't fare well trying to implement a measurement-dominated CMM approach. Conversely, a culture that bases decisions on precise objective data would have a rough time implementing an Agile approach that stresses fuzzier business value data.

The Competence Culture

Competence cultures stress individual responsibility and accountability: They are "can do" organizations. "At their best … these organizations are marvelously adaptive in their strategies and awesome at execution," Moore states. Competence cultures exalt knowledge and expertise over job title. Managers are expected to be technically competent first and management aware second. Intensity, high speed, and long hours are characteristic of competence cultures: Working late into the night becomes a badge of honor, and peer pressure to do the same runs high. Sales organizations, with their emphasis on measurable revenue generation, are often competence cultures.

A competence culture can excel at Early Market, Bowling Alley, or Tornado stages but can fall into an increasingly internal focus, with serious consequences, on Main Street. "It devolves into a caste system ruled over by an aloof and increasingly cynical elite," says Moore. "The end result is a tightly controlled guild—law, medicine, and accounting all come to mind" ([Moore 2000](#)). The danger for competence cultures is that they begin ignoring the external marketplace and focus on internal measures for success. Highly process-centric software development and project management methodologies, with their focus on measurements and internal practices, can be the result of a competence culture driven too far.

The Control Culture

As Moore relates, control cultures rally around the cry "Stick with the Plan." Control cultures are power based and security oriented. Control cultures have created powerhouse companies that have achieved sustainable competitive advantages on Main Street. But these same companies, comfortable for years in established market positions, may be uncomfortable when thrown back into Bowling Alley and Tornado markets. As Moore says, "The weaknesses of control culture are being brought to light by the challenges of the new economy" ([Moore 2000](#)).

The Collaboration Culture

Cross-functional teams dominate collaboration cultures. Leadership tends to be role based rather than title based. For example, at Intel, decision making is geared to who has the knowledge instead of who has the power. Collaboration cultures, if led appropriately, have the ability to generate a constant stream of innovative products as knowledge and shared diversity power creativity. Carried too far, collaboration cultures degenerate into endless consensus building, delayed decisions, and cliques.

Control cultures thrive on strong management—plans, objectives, roles, activities, checklists. Collaboration cultures thrive on strong leadership—establishing a purpose, setting boundaries, encouraging interaction, knowing when to take decisive action. Collaboration cultures put teams in a prominent position.

The Cultivation Culture

"Like a collaboration culture, cultivation culture puts people first, but as individuals rather than as teams," Moore (2000) says. At their best—Xerox PARC, for example—cultivation cultures produce novel and creative products and ideas. At their worst, they produce arrogance and egotism. Cultivation cultures tend to be self-organizing; people drift off to work on whatever gets their creative juices flowing. One problem with cultivation cultures is a great reluctance to work on anything once it passes the "interesting" stage. Individuals tend to work on what they want to work on, not what someone else suggests they work on. Cultivation cultures are very difficult, if not , to scale up, since participants loathe nearly any form of structure, process, or documentation.

Cultural Relativism

Cultural relativism means that, again, cultures are not good or bad per se, they just are. However, some cultures may be better or worse at addressing a particular opportunity or problem type. They may also be better or worse at attracting talented developers. The critical issue for methodology selection is understanding one's own culture, the *ecosystem* in which a methodology must be implemented. Without that understanding—whether one uses Moore's model or some other— methodology implementation faces hurdles, ones that are present but often nearly invisible.

Matching Methodology to Opportunity and Culture

Methodologies are an expression of how people work and, therefore, an expression of underlying cultural characteristics.

- A particular methodology may be more or less appropriate for a particular market stage (problem domain type)
- A particular methodology may be more or less appropriate for a particular culture.
- There will be mismatches for which organizations may employ several strategies.

First, let's postulate what type of methodology seems to fit with each market phase and culture. To keep the analysis reasonably simple, three methodology categories will be used: rigorous (RM), Agile (AM), and ad hoc or none (NM).

Tables 24.1 and 24.2 indicate how methodology type matches with market phase and culture. These tables are oversimplified but are useful in gaining a broad perspective. Putting the results of Tables 24.1 and 24.2 together yields Table 24.3, which indicates the general compatibility of methodology types with culture and market phases. Table 24.3 prompts a number of observations.

- Although RSMs are shown to fit best within only one box, the control culture/Main Street intersection, this is by far the largest sector. For all the new economy hype, the preponderance of organizations are still solidly on Main Street.
- While both the Early Market phase and the cultivation culture are critical to new product innovations, they are smaller sectors. From both a cultural and market phase perspective, either no methodology or very light Agile methodologies seem to fit best in this sector.

Table 24.1. Matching Methodology Type to Market Phase		
Early Market	NM	No methods or very light methods would be appropriate for early research and development.
Bowling Alley	AM	High levels of customization and intense customer interaction could be enhanced by a collaboration-oriented Agile approach.
Tornado	AM	The high-speed reactions required during this phase would preclude a rigorous, document-centric methodology, although some rigor would be required for control of the rapidly scaling operation.
Main Street	RM	A rigorous method, with standards and procedures emphasizing operational efficiency, would be an appropriate approach.

Table 24.2. Matching Methodology Type to Culture		
Competence	AM	Focused on getting the job done and performance based on smarts and merit, competence cultures focus on skills over process. Therefore, they will see the benefits of proven (by use) Agile methods, while they may not tolerate rigorous documentation and process.
Collaboration	AM	Collaboration cultures are amenable to Agile methods that enhance interaction and group process, and they will be far less accepting of rigorous methodologies.
Control	RM	Control cultures want to impose order and predictability and therefore would lean toward a rigorous methodology.
Cultivation	NM	The entrepreneurial nature of cultivation cultures would tolerate only a minimal (very light) methodology, if any.

Table 24.3. Compatible Methodologies for Market Phase/Culture Intersections				
	Competence	Collaboration	Control	Cultivation
Early Market	NM-AM	NM-AM	CLASH	NM
Bowling Alley	AM	AM	CLASH	NM-AM
Tornado	AM	AM	CLASH	CLASH
Main Street	CLASH	CLASH	RM	CLASH

- The clash between a Main Street market phase and a cultivation culture is so great that most companies will never make it to Main Street if the culture is one of cultivation. This is one clash that may not be solvable except by spinning off part of the organization.
- An Agile methodology seems more appropriate for the intersection of either a Bowling Alley or Tornado market phase with a competence or collaboration culture. A large percentage of exploratory projects fall within these market phases.
- There are two major areas of conflict: first, when a competence or collaboration culture must work with Main Street products, and second, when a control culture tries to work in the Bowling Alley or Tornado.

Methodology Selection

I've worked with project management and software development methodologies for more than 20 years. In that time, I've developed, consulted on, trained on, and implemented various methodologies and participated in several methodology "debates." From rounded rectangles versus bubbles, to various data modeling schemes, to information engineering versus structured analysis and design, to object methods versus non-object methods, there have been endless debates about the effectiveness of various approaches, but they have all advanced the profession of software development and management.

All of these practices have succeeded, and all of them have failed. So on what basis should an organization manage complex software development projects? That basis should be threefold. First, decide what methodology appears most compatible with your unique organizational culture. Next, decide what methodology appears best suited for a particular type of project. Finally, when there is a clash between the first and second decisions, determine how to customize your predominant methodology. The order of these three decisions is important. There are several tasks associated with making these decisions.

- Articulate the predominant culture, in terms of values and principles, in your IT or software development organization (and determine its compatibility with the overall company's culture). If there is not a predominant culture or the culture varies from group to group depending on the day of the week, methodology selection will be the least of your worries.
- Determine how well your predominant culture matches with the amount of work forecasted in each market phase for the next few years. For example, if your predominantly control culture is faced with a high percentage of work in the Bowling Alley and Tornado market phases, you will need to consider either establishing a separate part of the organization that exhibits a competence or collaboration culture or altering the fundamental culture of the entire organization—a daunting task that can take years.
- Pick a methodology that best matches the predominant culture of the software development or IT organization and, one hopes, the problem characteristics of the projects planned for the near future. Don't try to use a software development approach as a cultural change mechanism (by itself at least); it won't work. It is easier to tailor a methodology based on a specific project's needs than it is to force fit an approach that doesn't correspond to the organization's culture.
- On a project-by-project basis, determine the project type and assess the applicability of your culture or ecosystem to the task at hand. For example, if the first three decision steps result in an RSM selection but a particular project falls in the Early Market category, a significant "lightening" of the rigorous methodology would be necessary.

Despite the need to match a software development ecosystem to culture first and project type second, there will still be a strong tendency in organizations to force fit projects into their predominant methodological mold. And finally, in large multi-national companies, there will be a collection of ecosystems. The variations in culture among locations within the same company will dictate variations in methodology also.

Articulate Values and Principles

"The first thing we did was articulate and document 13 principles," the head of a very large consulting organization (more than 10,000 IT consultants) told me about her firm's engagement to help a client with a methodology project. Articulating an organization's values and principles is key to understanding its culture and to successful methodology implementation.

But unfortunately, or maybe not, there is no ten-step procedure to articulating values and principles. In fact, how your organization approaches this articulation process may itself indicate what kind of culture it has. If the CIO and his or her direct reports meet for a few hours and then issue a "principles" edict, one can guess at the cultural implications.

This is not a book on organizational cultures, or even one on how to articulate culture in general, but one that tries to articulate the values, principles, and perspectives of an Agile culture. Geoffrey Moore's cultural categories or product adoption curves may not help your analysis, but without articulating cultural and problem domain issues in some fashion, project management and software development methodology implementations will suffer.

Chapter 25. Designing Your Agile Methodology

The most effective teams create themselves.

—*Tom Petzinger,* The New Pioneers

The word "methodology" has developed a bad connotation for many people—it raises the specter of bureaucracy, burdensome documentation, and micro-management—but methodologies assist in improving performance and providing solutions for customers. The *American Heritage Dictionary* offers two definitions of methodology. The first definition, "The system of principles, practices, and procedures applied to any specific branch of knowledge," fits the traditional rigorous methodology mold. However, the second definition, "The branch of logic dealing with the general principles of the formation of knowledge," fits more closely with ASDEs.

Some RSMs have been tagged with the label "one-size-fits-all," suggesting that every problem fits into a single methodological framework. The apt phrase should be "one-size-fits-one." Problems are different, people are different, teams are different. All too often in today's fast-paced, high-change business world, the products we are asked to build are new, and our approach to development needs to reflect that reality. However, while one size doesn't fit all projects, neither does every project team need to start from scratch and design its own methodology—we can reuse elements previously developed. Designing an Agile methodology means balancing the consistency needs of business enterprises with the flexibility required by project teams.

Every software development organization must build products that continuously adapt to the competitive environment molded by competitors and customers. Adapting to the external environment requires an Agile, adaptable development methodology. At the same time, the development methodology must itself adapt to the strengths, talents, skills, personalities, and knowledge of the individuals on the product team. Adaptation to the environment and self-adaptation are both applicable to methodology design.

Finally, there is a difference between Agile and rigorous methodologies—in their practices, perspective, and culture. To paraphrase Andrea Branca, "You can easily make a project *look* like an Agile project, but you can't make it *feel* like an Agile project."[1]

[1] Andrea Branca is a frequent contributor to the discussions on Agile Software Development at www.crystalmethodologies.org.

Agile methodology design and adaptation should be based on:

- Realistic expectations of a methodology
- A clear definition of the elements of a methodology, including its system of practices
- An effective set of design principles
- An overall framework, project templates, and development scenarios
- A series of collaborative design steps
- Processes for methodology customization and tailoring
- Guidelines for scaling to large projects

Methodology Expectations

Before an organization sets out to design a methodology, it must first address the question, What should we "expect" from a methodology? Most traditional RSMs begin and end with the word

"repeatable," but repeatability is a myth—at least in the statistical quality control sense of the word. Many methodologists and managers champion repeatable processes, but the reality of unpredictability means that they hold out false hope. Product development processes, software or otherwise, should strive for a degree of consistency, but repeatability would require that every project have the same input information, people, politics, constraints, and algorithmic design processes. Because all these things change from project to project, trying to construct a repeatable process will be impossible.

Consistency, however, is a reasonable goal. Given that all the project variables are reasonably consistent with prior experience, there should be an expectation that the results be reasonably consistent with prior projects. This expectation is much different from expecting results to be quantitatively repeatable. Reusable elements can improve consistency, but they should not imply repeatability.

There is a fundamental disconnect between business management and product development of any kind. Financially and operationally, businesses strive for a degree of predictability. Business managers need to know how many widgets to make and when to launch a new product. They need to meet a governmental regulatory deadline or schedule a new employee benefit. Whether it is appropriate or not, they are burdened by the quarterly expectations of Wall Street. Unfortunately, predictability has more to do with what lies *outside* (external changes and constraints) any process than what lies inside it. If the person who has maintained the corporate HR system for five years makes the employee benefits system changes, chances are she will get them in on time. If the new product launch involves a brand-new technology platform, chances are the product team's "predictions" will be off until a few months prior to that launch. Predictability can be improved in one of two ways: Do things that you have experienced before or shorten the prediction period.

To the degree that a methodology can assist in incorporating past experience into the development process, it can improve consistency. To the degree that a methodology can help delineate between the known and the unknown, it can help determine an appropriate prediction horizon. But asking a methodology to know the unknown or predict change asks too much.

Therefore, we constantly need to consider two questions when designing a methodology: What do we expect of a methodology? and Are those expectations reasonable? I've seen far too many companies whose expectations are totally unrealistic. They expect that somehow, some way, a methodology will protect them from change. They want flexibility *and* predictability, but they don't want to make the hard tradeoff decisions that balancing these opposing goals demands.

Methodology Elements and the System of Practices

Methodologies are made up of elements: standards, roles, tools, scenarios, policies, practices, phases, work products, and more.[2] Figure 25.1 shows the elements in a methodology and a second set of elements for a team ecology. The second set is usually given low priority in methodology development, so I've separated them for emphasis. However, for simplicity, I will include all the elements within the scope of "methodology" in this chapter.

[2] For an in-depth discussion of methodology components, characteristics, and design principles, see *Agile Software Development* (Cockburn 2002).

Figure 25.1. Sample Elements of Team Methodology and Ecology

Team Methodology | Team Ecology
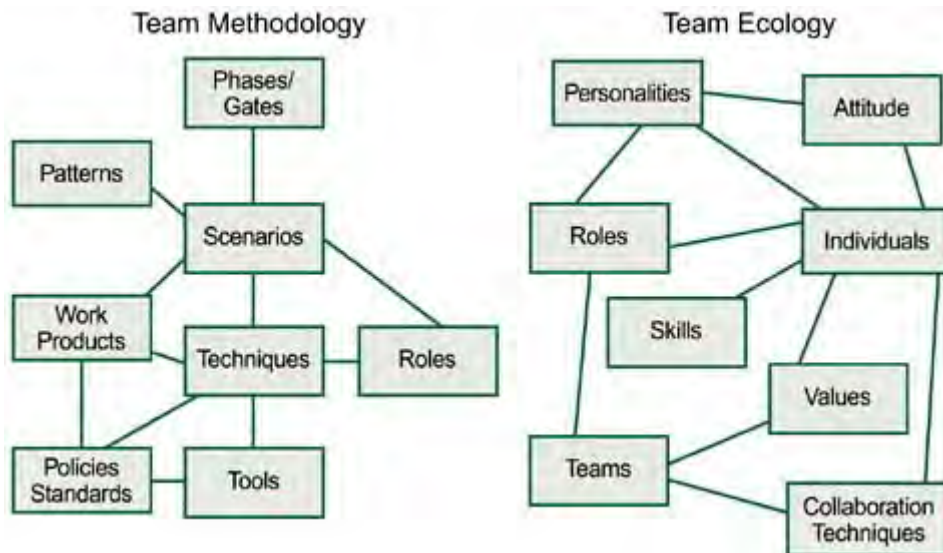
Key characteristics of methodologies are size (the number of elements), ceremony (detail and formality), weight (the product of size and ceremony), and tolerance (variations allowed to the project team). A methodology with ten practices, three roles, and four work products that used hand-written story cards, CRC cards, and project status reports on flip charts would be a low ceremony, small-sized, very lightweight methodology.

Within a methodology, individual practices and the way in which they form a system of practices are key focal points. We want to build a generative system of practices out of a set of good individual practices by understanding how the individual practices impact each other. For example, simple design without refactoring would not form a good system of practices, because it would lead to a constantly degrading design.

I've also found it useful to sort practices into collaboration, project management, and software development categories. Collaboration practices concern how team members interact with each other, customers, and management; project management practices concern how projects are managed and monitored; and software development practices concern how the software gets designed, built, and tested. Because of the critical nature of person-to-person interactions in ASDEs, I explicitly break out collaboration practices. When asked, "How should we scale our project from 10 to 30 people?" my first thoughts focus on collaboration, not process, tools, or documentation.

Using these categories is also a way of showcasing the strengths of different Agile approaches. For example, Scrum's practices are nearly all project management and collaboration related, whereas XP's are predominantly software development and collaboration related. XP's minimal project management practices are oriented toward a small colocated team and would need to be supplemented for larger projects. Some practices have both a primary and secondary impact (for example, the planning game can be considered part of both project management and collaboration); however, if each of the 12 XP practices had to be assigned to a single category, this would be my categorization:

- *Collaboration:* pair programming, collective ownership of code, on-site customer
- *Project management:* planning game, 40-hour week
- *Software development:* metaphor, simple design, refactoring, test-first coding, continuous integration, coding standards

Development efforts produce results, collaboration efforts define how people work together to produce results, and project management efforts define the vision and boundaries for work.

Keep It Simple

ASDEs are designed to envision and explore rather than plan and do. Agile approaches shine when speed, uncertainty, and change characterize the project environment. As such, Agile methodologies need, at a minimum, practices in six key areas.

- *Mission elaboration:* A good visioning practice helps ensure Agile projects remain focused on key business values. (Example: ASD's product visioning session)
- *Project initiation:* A short, succinct project initiation and scoping activity ensures that customers and developers have a baseline for project scope, cost, and schedule. (Examples: DSDM's feasibility study, ASD's project data sheet and project profile matrix)
- *Short, iterative, feature-driven, time-boxed development cycles:* Exploration should be done in definitive, customer-relevant chunks. (Example: FDD's feature planning)
- *Constant feedback:* Empirical processes require constant feedback to stay on track. (Examples: Scrum's short daily meetings, ASD's customer focus groups)
- *Customer intimacy:* Focusing on business value requires constant interaction between customers and developers. (Examples: DSDM's facilitated workshops, XP's on-site customer)
- *Technical excellence:* Creating and maintaining a technically excellent product makes a major contribution to creating business value today and in the future. (Examples: XP's refactoring, Crystal's emphasis on testing)

When we begin to consider all the elements that can go into methodology design, it is easy to slip into the traditional methodologist's role, in which the mantra seems to be, "If a little methodology is good, then a lot of methodology is better." So remember the three key reasons to keep methodology simple: (1) the primacy of people, (2) a focus on adding value, and (3) the creation of a generative environment. First, if we believe—and Agilists do—that staff competency is ultimately the arbiter of success, then process and methodology should focus on assisting and guiding people in their working together without getting in the way. Second, in the lean thinking sense of eliminating non-value-adding steps, methodologies should be as "light" as possible— barely sufficient to the task at hand. When a methodology becomes something to circumvent in order to get the job done, it impedes progress rather than assisting it.

Third, maintaining a team in a generative state, one where innovation and creativity can flourish, requires balancing at "the edge of chaos," or being chaordic. Too much order causes stasis, too much chaos degenerates into randomness—a little bit less than just enough is the right place to be. Scrum, for example, advocates a "controlled" chaos—letting chaos in when developing priorities for backlog items, and then disallowing feature changes during an iteration.

Practices and Principles

Principles are "instantiated" by practices; practices without principles are sterile and inflexible. Most methodology design processes skip the "principles" step and are, therefore, usually doomed from the start.

Practices by themselves don't dictate a particular culture, but some reinforce a particular culture better than others. For example, Agile approaches are characterized by a culture of collaboration that emphasizes knowledge sharing, team responsibility for product delivery, and collaborative decision making. Jeff De Luca's description of FDD emphasized interaction and collaboration via three practices: the use of chief programmers, inspections, and class ownership. XP utilizes three other practices: coaching, pair programming, and collective ownership. On the surface, the three FDD practices and the three XP practices appear diametrically opposed, but because both approaches use these practices guided by the principle of collaboration, the resultant working environments are very similar.

Having said this, however, I would argue that the XP practices reinforce the principle more directly. Chief programmers, inspections, and class ownership could also be utilized in a very control-oriented, noncollaborative work group. The chief programmers could make all the decisions and inform the staff. Inspections could be very confrontational, and class ownership could breed a sense of isolation rather than collaboration. In this case, the principles aren't embodied in the practices, they are externally imposed. Conversely, it is very difficult to imagine a "control" culture utilizing pair programming, coaches, and collective ownership of code. The practices, in a direct way, "force" a working culture. This may be one reason XP generates such heated debate—it doesn't offer much wiggle room.

Methodology Design Principles

In putting a methodology together or customizing one, there are several factors, or design principles, that should be kept in mind. Alistair Cockburn articulated six of these design principles for methodology, and I've added three more.

Principle 1: Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.

The more complex the information, the more this principle holds. Not everyone can interact face-to-face on larger projects, but the principle needs to be kept in mind when designing a collaborative framework. Documentation—either text or models—can supplement and enhance face-to-face communications, but the interaction is key to understanding.

Principle 2: Larger teams need heavier methodologies.

Heaviness is a combination of methodology size and ceremony. A methodology with five work products is *heavier* than one with two. A detailed, precise requirements specification is *heavier* than an XP story card. Although larger teams need heavier methodologies, the heaviness must be balanced with other factors.

Principle 3: Excess methodology weight is costly.

In Chapter 21, which is on Lean Development, Bob Charette referenced studies that show a 1,000–function point software project might have 300 pages of requirements documentation. By the time the document is written, reviewed, read, and modified by multiple project participants, its costs are dramatic. On very large projects, studies have shown as much as 35 percent of the total costs are documentation related. Excess weight can be very costly.

Principle 4: Greater ceremony is appropriate for projects with greater criticality.

The higher the criticality, the higher the ceremony needs to be in order to protect product users from adverse consequences. However, ceremony by itself may not reduce defects to acceptable levels. In life-critical systems, redundancy is also required. In any case, products with high criticality will be costly in both dollars and time.

Principle 5: Discipline is not formality, process is not skill, and documentation is not understanding.

Alistair borrowed this one from me, so now I'll borrow it back. The section on scaling covers this principle in depth.

Principle 6: Increasing feedback and communication reduces the need for intermediate deliverables.

In linear, waterfall life cycle methodologies, intermediate deliverables were often intended as the single source of information transfer. As collaboration and conversations increase among team members, particularly those with different roles, the "weight" of intermediate deliverables can be reduced.

Principle 7: Efficiency is expendable in non bottleneck activities.

Eliyahu Goldratt's (1984) theory of constraints has shown that inefficiency in some activities actually leads to increased throughput (faster delivery) for the overall product. This principle means that people performing non bottleneck activities must do everything possible to minimize the amount of work for those engaged in bottleneck activities, even if it means being somewhat inefficient themselves.

Principle 8: Think flow, not batch.

Making the value-creating steps *flow* is a key principle of lean thinking (Womack and Jones 1996). First, streamline and eliminate all non-value-producing steps. Then make sure that processes flow; that is, eliminate batch processing wherever possible. When a project team has to wait for an architecture group to create a class diagram, a data administration group to complete a physical design, a project office to approve a milestone completion, or a manager to approve a change request, batches are formed. Batches cause queues, queues cause time delays, time delays break the flow. Even occasional rework is a small price to pay for eliminating batches.

Principle 9: Greater methodology ceremony may be required to meet regulatory, security, and legal considerations.

There is another aspect of methodology that needs to be considered, particularly by those in the Agile community who tend toward less formal documentation. That is defensibility—can we defend the practices and processes we used in a regulatory environment or in a legal proceeding?

Ski resorts are in a legal quandary. In trying to protect skiers, they often post signs indicating hazardous terrain. However, if someone gets hurt in an area that wasn't posted, their lawsuits often plead negligence on the part of the ski area for the nonposted area. So unless resorts post every tree and gully, they may be better off posting nothing, from a legal perspective. In the twisted logic of liability law, skier safety takes second place to liability protection. (Most ski areas post dangerous areas anyway.) Similarly, better but less widely accepted development practices may not pass legal muster without adding back some of the inefficiencies that have been taken out.

Product liability potential often forces companies into using widely accepted existing practices rather than potentially better new ones. Product liability concerns can dominate issues like documentation—not only do companies have to use widely accepted practices, but they must also be able to back them up with documented proof.

Frameworks, Templates, and Scenarios

Methodology designers are in a quandary. While every product and project team is unique and therefore needs a customized methodology, organizations require a degree of consistency for monitoring, evaluation, and decision making. In addition, one of the basic definitions of agility involves balancing between structure and flexibility in order to balance at the edge of chaos— facilitating creativity while maintaining a degree of order. Thus the first part of the designer's

quandary is how to balance the organization's need for consistency with a team's need for flexibility. A second aspect of the methodology quandary is how to reak away from the linear, prescriptive, dependent phases, tasks, and activity model of development. We need a framework but not a straightjacket. We need guidelines but not prescriptions. We need to support concurrency and collaboration rather than linearity and endless artifacts. The solution: a *phase and gate life cycle framework* established at the enterprise level, a series of *problem domain templates* targeted to (and customizable by) operational units, and a series of development *scenarios* targeted to (and customizable by) project teams.

A phase and gate life cycle framework defines the high-level flow (phases) and critical decision milestones (gates) of projects from a senior management perspective. This type of framework is used to track progress and make project portfolio, funding level, and project continuation or cancellation decisions. The decision milestones are also used to synchronize projects whose components must be integrated, as in hardware products with embedded software or large multi-function software products. The life cycle framework embeds enterprise-level policies to which all, or at least most, projects must conform.

Problem domain templates are specific combinations of methodology elements organized to address a defined problem domain. They are constrained by the high-level phase and gate life cycle. Smaller companies may develop a single combined life cycle and domain template from which to customize, while larger companies may employ several project templates as project starting points. I know of one large company that uses six predefined templates from which project teams select and customize. I worked with one smaller firm on a combination Adaptive-XP template that the company used as a starting place for customization.

Phase and Gate Life Cycle Frameworks

One of the insidiously dangerous aspects of RSMs is that their mere presentation format creates an aura of linearity and task dependencies. When a requirements phase is followed by a design phase, which is then followed by a development phase, linearity is implied, if not explicit. When the requirements phase consists of 47 activities, each numbered and listed 1, 2, 3, 4, ..., 47, project managers trained by the Project Management Institute can't resist sticking those 47 activities on a PERT chart. Another problem with traditional process diagrams is that they show phases, activities, and milestones, but they frequently ignore the most critical aspect of a milestone as a decision-making point.

No matter how many times we say, "These activities are not linearly related," they are consistently thought of as linear because of the pictorial representations we use. Feature- or story-driven development therefore has difficulty depicting what goes on during a development phase. Managers are so attuned to process flow diagrams and fixed boundaries (whereas iterative projects have a variable number of boundaries, or iterations) that they want to retain something that looks familiar. Concurrent, iterative development has proven difficult to depict.

A phase, gate, and scenario approach to documenting methodology, as shown in Figure 25.2, is a way of mitigating many of these problems. In phase and gate life cycles, used extensively in manufacturing product development, phase descriptions concentrate on the products, product components, or key artifacts to be built, while the gates describe the decision points.[3] When managing large projects, particularly when there are extensive hardware and software component, manufacturing, and external supplier considerations, a phase and gate approach can be very helpful. It focuses project managers on the *what,* not the *how*.

[3] For example, Toyota controls uncertainty by managing process gates. "U.S. companies view design processes as networks of tasks and control them by timing information hand-offs between the tasks, as in the familiar PERT chart. But Toyota views its process as a continuous flow, with information exchanged as needed. Toyota manages this process through a series of gates, each tied to an integrating event that brings all the pieces together, e.g., a vehicle prototype. Toyota

controls the level of uncertainty at these gates, reducing it at each successive gate" ([Duward et al. 1999](#)).

Figure 25.2. An Overview of a Phase and Gate Life Cycle
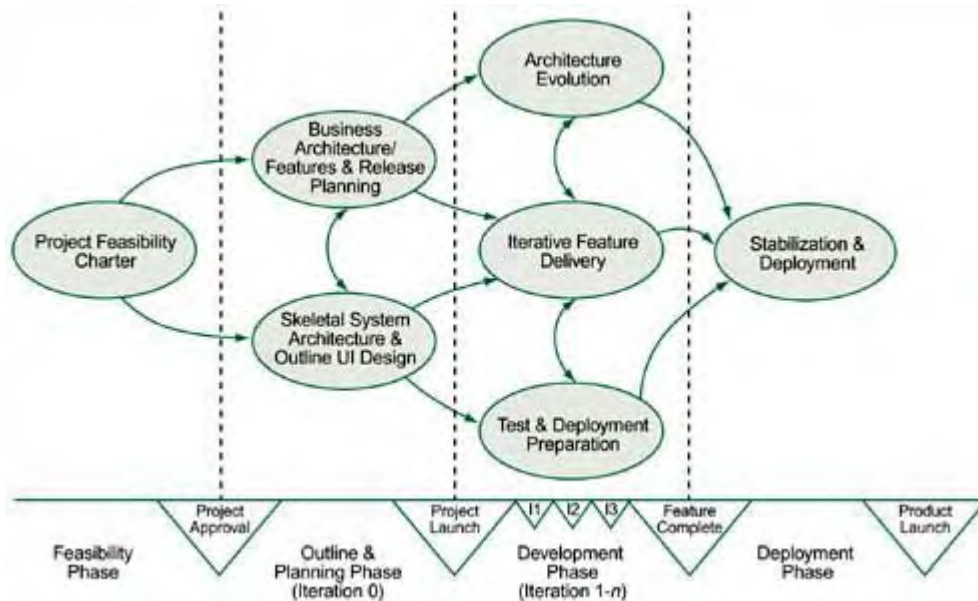
At a high level, most projects fit into a standard phase progression: product management, project feasibility, project planning, product development, stabilization, and deployment. This high-level sequence might apply to 6-, or 60-, or 600-person projects. The differences between organizations, or between life cycle frameworks within an organization, lie in the decision criteria at each gate. Life cycle frameworks are high level and should reflect the need for information to make decisions at the product or project portfolio level. Greater possibilities for variation are found at the next levels down—problem domain templates and scenarios.

Problem Domain Templates

Each organization has a portfolio of projects that reflect strategic initiatives and have patterns of project objectives and characteristics. Based on analysis of projects under way and planned, an organization can develop a set of problem domain templates, possibly using Alistair's Crystal series as a first-order guide. [Figure 25.2](#) shows not only the major phases and decision gates, but also an example of an iterative development domain template that utilizes "shape"-level modeling as part of the outline and planning phase. An XP template might leave these architectural components out. Each problem domain template should include an explicit domain definition.

A domain template brings together specific elements, particularly development scenarios to be applied in a particular problem or project domain. A template for a 50-person project would contain more elements, including scenarios, than one for a 6-person project. Templates should be brief, no more than one or two pages per life cycle phase. Crystal Clear is a domain template—a collection of methodology elements designed for a particular problem domain. Domain templates are used as a starting point for teams, which then customize the elements and scenarios to their particular situation.
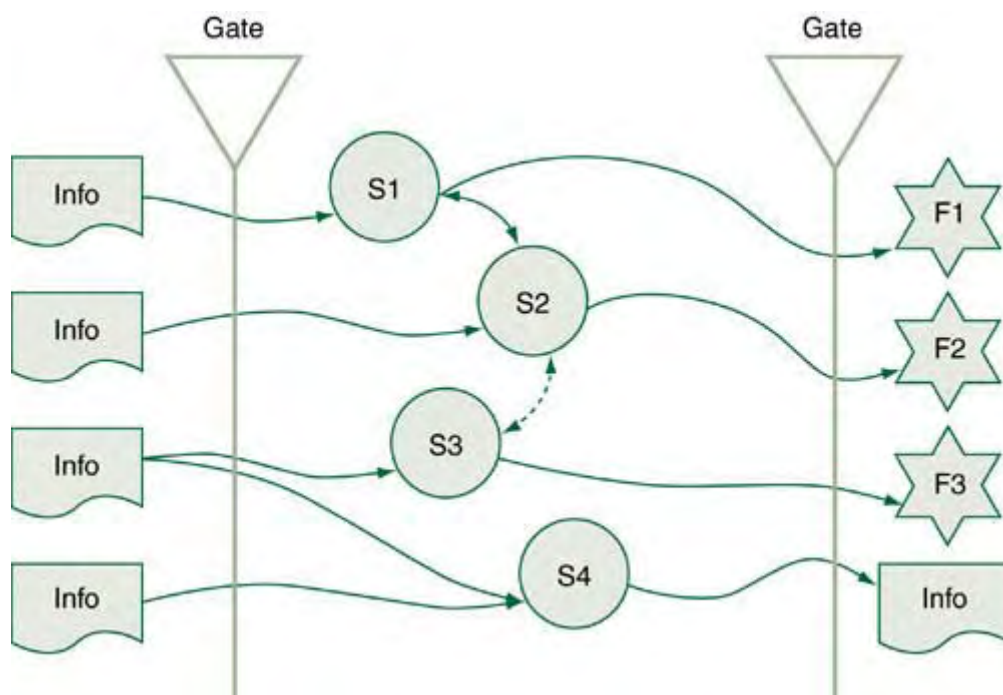
Scenarios

Scenarios replace static, sequential activity lists that are found in most methodologies or process descriptions. Scenarios relate short *stories* about how people work together to produce results. A scenario will include some combination of the roles that individuals take, how those individuals

195

interact (collaboration practices), what they produce (work products), how they produce them (development practices), and possibly scenario timing. Scenarios are stories within which activities are performed, and there are no implied sequential relationships between them. Scenarios should emphasize the collaboration and communication practices used by development teams in their internal interactions and those with customers and management.

Figure 25.3 shows how the "flow" of scenarios might interact during a development iteration. The diagram indicates information flow into a scenario, which scenarios share information back and forth, and the fact that scenarios deliver completed features. Note that information passes back and forth between scenarios—there are interdependencies of information, but not the linear dependencies reflected in PERT charts. The dotted line between Scenarios 2 and 3 indicates "light" collaboration between the groups.

Figure 25.3. Scenarios within a Phase

Following are some examples of scenarios that a project might use.

Scenario 1: Developers (role) work together using pair programming (collaboration practice), simple design, refactoring, and test-first development (development practices) to produce a working feature (work product) that fulfills a specified user story for an iteration.

Scenario 2: At 4 each afternoon, the designated developer and tester meet together to create an integrated product build and run it against the build-verification test suite.

Scenario 3: Each morning at 9:30, the feature team conducts a 15-minute daily team integration meeting to ensure that work across pairs and among all team members remains in sync and impediments to progress are elevated to the project manager.

Scenario 4: Each Friday morning, the project manager and feature team leads meet to ensure that all architecture, design, and integration issues between the three feature teams have been identified and resolved.

Scenario 5: An on-site customer representative works with the developers each day to refine requirements specifications and prepare acceptance tests.

Note that the scenarios are *not* sequentially related, although the work done in one scenario may have some relationship to the work done in another (features from three subteams are integrated via the build); people working in one scenario have interactions with those who work in another (one developer works on the daily integration scenario); or individuals may be involved in several scenarios.

A small XP or Crystal Clear project might be adequately described in three to five brief scenarios. A larger XP or Crystal project might use most of the same scenarios but alter the "customer interaction" scenario to include business analysts and multiple "users." Scenario 5 would be replaced with 5a.

Scenario 5a: The business analyst works daily with the development team to facilitate the flow of requirements information from the 6 user departments and 25 designated user specialists who are the business domain experts. The analyst will be responsible for resolving priority conflicts among the user specialists.

Although scenarios contain activities, they focus on the interaction rather than the activities. Within the text of a scenario, references can be made to roles, techniques to be used, work products to be produced, or patterns that might be employed. For example, a project management scenario might refer to a set of organizational patterns, or a development scenario might refer to a set of refactoring patterns. Scenarios may be reusable across both domain templates and life cycle frameworks.

Scenarios provide a vehicle for rich variation from project to project. One 20-person project may have a simple customer interface with one on-site customer for each 6-person feature team. Another 20-person project may have an extremely complex customer interface with hundreds of customers spread over three geographic locations. The "customer interface" scenario for these last two projects will be very different, while all the remaining development, testing, and architectural scenarios might be the same. Similarly, moving from a 6-person to a 20-person project might require the addition of an "architectural coordination" scenario, while all other development scenarios remain the same. Domain templates provide for high-level variations among projects, but scenarios provide a way to inject near-infinite variations while mitigating the problems associated with activity-based methodologies.

Collaborative Methodology Design Steps

There are three general steps in developing a methodology:

1. Articulate values and principles (covered in Chapter 24)
2. Evaluate project objectives and characteristics
3. Design a methodology framework, templates, and scenarios

Designing an Agile methodology is just like designing software—there is no linear, step-by-step, approach. It is an iterative, adaptive effort that takes an understanding of the problem domain, the culture of the organization, and the unique characteristics of the actual people on the project team. Having a methodologist, someone skilled in this area, can help, but ultimately if the development team has accountability for results, it should be the final arbiter of its own process.

Evaluate Project Objectives and Characteristics

A project's objectives identify business value and tradeoff priorities. Business value could be anything from creating a new product or business to adding an application enhancement to meet a new government regulation. Every project should have tradeoff priorities established. Is schedule more important than scope, or vice versa?[4] Is meeting the cost budget more important than delivering all the intended features? In Agile projects, business and project objectives are nearly synonymous: There is no such thing as meeting the project's objectives but failing to meet the business objectives.

[4] In Chapter 4 of *Adaptive Software Development* (Highsmith 2000), I describe a "product mission profile matrix" to capture tradeoff prioritization.

Business value might be categorized into *infrastructure* (lower value but required), *foundation* (such as ERP systems that deliver basic business functionality and provide the "foundation" for other initiatives), *transformational* (potential high returns that transform the business), and *frontier* (on the bleeding edge, whereas "transformational" would be on the leading edge). This categorization might be appropriate for an IT organization, but a software product company would probably want to devise categories related to new product development, mature product enhancements, and technology exploration. Business value categorizations are part of product portfolio management.

In addition to a project's business-value-oriented *objectives,* there are four project *characteristics* that drive methodology design:

1. Team size
2. Criticality
3. Risk and uncertainty
4. Activity scope

Team size is the number of people on the project. Although problem size (described in terms of some measure such as function points or lines of code) may also be useful, methodology design is a function of team size. Alistair contends that doubling team size increases the communications load enough that a 20-person team and a 40-person team are significantly different, hence his size categories of 6, 20, 40, 100, 200, and 500 people. But what about team distribution? As team size increases, distribution will occur; the only question is distance. Distance can take on multiple dimensions—miles, time zones, cultures, companies. The collaboration practices for a single colocated team of 10 will be different than those for a colocated team of 40, which will be different than those for a team of 40 distributed across four worldwide locations.

Project criticality measures the seriousness of the consequences if the software fails. Alistair identifies four categories—comfort, discretionary money, essential money, and life criticality— but says that, in reality, the key distinction is whether or not a software failure could threaten life. Often, software products that are potentially life threatening have methodology constraints imposed by government regulations. Another useful categorization might be: life critical, mission critical, mission support, and entertainment (games, for example). Criticality indicates that the methodology for a flight avionics system, a payroll system, and a computer game would be very different.

Risk and uncertainty relate to criticality, but they also indicate the volatility of the problem space. Criticality relates to the consequences themselves, whereas risk and uncertainty relate to all the various factors that might cause the adverse consequences. A project that faced high requirements uncertainty but "comfort" criticality is different than one with high requirements uncertainty and potential loss of life. A high-risk project might be a Web-based project to launch a new business initiative using .NET technology. A low-risk project might be enhancing a payroll system to accommodate a new union contract modification.

Activity scope identifies what parts of the development life cycle—from project planning to minor enhancements—are to be part of the methodology. For example, the activity scope of DSDM is greater than that of Scrum, as DSDM covers pre-project activities such as feasibility studies.

The following are examples of project domain definitions.

- An eCommerce project might typically occupy some combination of the medium-to-high risk, small-to-medium size, and transformational or frontier categories.
- An ERP system implementation probably occupies the high-risk, large-size, foundation intersection.
- A CRM implementation might occupy the high-risk, medium-size, transformational category.
- A payroll system enhancement project might occupy the low-risk, small-size, infrastructure category.

Design a Methodology Framework, Templates, and Scenarios

The simplest approach to this step would be to select XP, Scrum, or another Agile approach and proceed to customization for a project. Each of these ASDEs could be thought of as a domain template. However, as project objectives, project size, criticality, risk factors, or activities dictate, the selection and design process may take more work than just selecting an existing approach.

An overall methodology framework consists of defined principles, a brief description of the scope and intent of the methodology, a phase and gate life cycle description, the identification of specific domain templates, and a description of each methodology element (scenarios, roles, techniques). The framework should be brief—no more than 25 to 50 pages. It should act as an information portal rather than provide voluminous detail data. For example, roles should be described in a paragraph or two, while techniques might have a single sentence description with references. A methodology framework is *not* the place to find skill-building or technique information such as how to develop use cases or do refactoring.

One reason that traditional methodologies have filled multiple notebook binders or innumerable Web pages is again the failure to distinguish between methodology (how we work together), skill (how we accomplish something), and knowledge (what background material we need to know). In many traditional methodologies, the detailed "process" steps are actually a feeble attempt to convey skill, as if a listing of steps conveys the information needed to build skill. If we ripped out all the pseudo-skill material from traditional methodologies, they would be much smaller. Skill-building material—training workshops, computer-based training sessions, books, reference manuals—can be referenced by a methodology but should not be included as part of it. (Or, at least, it should be *clearly* separated from the process piece.) When skill building and knowledge management are separated from a software development methodology, each will be better served.

Template design should focus on defining scenarios—identifying key project management, collaboration, and software development practices and policies. Practices (or techniques) bundled into scenarios are the product delivery vehicles. Pair programming, 15-minute Scrum meetings, customer focus groups, refactoring, and feature-based planning are all specific practices that are packaged together into a system of practices and then into scenarios. Policies are those parts of the methodology that are mandated by the organization. A policy statement would be, "Development teams must use development iterations of no more than two months" or "Project status reports must be done monthly." There is a difference between having a set of practices that teams can draw on and having a set of mandated policies. Every organization has certain policies that it uses to maintain control over finances, resources, and project progression. One measure of methodology "tolerance" is the number of policy statements it contains; the lower the number of statements, the more tolerant the methodology.

Customize Templates to the Team

If methodology concerns how people work together, then every team member should be involved in methodology design and customization. Most individuals will not be involved in the high-level life cycle framework design, but every team member should be involved in customizing and tailoring domain templates and scenarios. Every team should conduct a methodology customization session during project initiation and periodic tailoring sessions at the end of project iterations.

As mentioned earlier in this chapter, larger organizations will tend to have domain templates that teams can use as starting points, while teams in smaller organizations may devise their own from scratch. The design principles and ideas from the previous sections are relevant to this customization step.

A Customizing Approach

Let's consider one approach to customization.

1. Select a domain template with an absolute minimum set of practices or scenarios.
2. Substitute scenarios or practices that match more closely with your project team.
3. Very carefully add scenarios or practices that address specific organizational or project situations.
4. Evaluate every prospective practice against the principles.

In step 1, for example, you might start with an XP domain template consisting of 12 practices that could probably be combined within three or four key scenarios. Resist the temptation to start instead with a maximum set of practices and remove some practices. My experience has been that it is far too easy to leave a practice in, as people (especially those in project offices and methodology groups) can always find good reasons for a particular practice. Leaving practices in often results in increased costs and lengthening schedules. People have a tendency to embellish a methodology with practices that they think "should" be done instead of those that actually work in practice.

Substitution comes into play in step 2. There is an important caveat here—remember to substitute practices, not principles. So, continuing our XP example, if a project team doesn't feel that pair programming is workable, another collaborative practice such as frequent peer reviews might be substituted. Maybe peer reviews aren't as productive as pair programming, but they are certainly a reasonable substitute that furthers the *principle* of collaborative development. Furthermore, because of the "simplicity" principle, peer reviews shouldn't degenerate into a heavyweight practice.

The third step is to carefully add scenarios or practices. In the scenario section above, the substitution of Scenario 5a for 5 when the user community expanded is an example of addition. Or, as part of the XP "planning" scenario, I might add a one-page project data sheet.[5] What emerges from this process may not be pure XP, but it could be a customized template that works for your particular situation.

[5] Project data sheets are described in Adaptive Software Development (Highsmith 2000).

Finally, evaluate every additional practice against the principles. This step serves two purposes—it helps validate both principles and practices and it guides practice implementation. While one can write about or discuss principles nearly ad nauseam, discussing them within the context of a specific practice—a demonstrated *use* of the principle—helps bring clarity. For example, without applying the principle of simplicity, one can take a practice and add enough ceremony to render it burdensome. A simple peer review can be laden with so many procedures, forms, and metrics that

the principle of supporting small group collaboration and shared learning gets lost. Keeping the principles of simplicity and minimalism in mind helps temper our natural tendency to try to reduce practices to a string of templates, forms, and activity lists.

There are a wide variety of ways in which you might further customize scenarios.

- Make adjustments for variations from the domain template. For example, if the generic template was defined for 40 colocated people and your team is distributed, some adjustments in scenarios should be made.
- Make adjustments for actual team staffing. Specific talents, skills, knowledge, and personalities will impact how the team works together. Acknowledge these and adjust accordingly.
- Determine the degree of ceremony for each work product. For example, Alistair suggests several use case formats, from "fully dressed" (high ceremony) to "casual" (low ceremony) ([Cockburn 2000](#)).
- Determine the degree of traceability and currency of each work product. Maintaining traceability from requirements to code has a major impact on cost (often for little benefit). Similarly, some documents should be viewed as working papers rather than permanent documents—they don't require updating. Some documents need to be current, others don't.
- As the size of project teams increase, focus customization on collaboration and interaction issues first and ceremony issues last. Remember, documentation may aid understanding, but conversations build effective understanding.
- Project objectives should drive development practices, although constraints (regulations, for example) may influence appropriate ceremony and weight.

Adapt the Template to Use

Once the project gets under way—stuff happens. Just as projects change to adjust to the external environment of customer and competitor changes, teams adjust as they work together and find out what works and what doesn't. Every iteration or two, the team should conduct a short retrospective to adapt its scenarios or practices. In any case, teams should conduct these short evaluations every two months.

In every methodology design and customization activity, one of the key questions should be, Do we have the right mix of talent, skill, and knowledge for this project? The tendency is to confuse methodology problems with competency problems, and in so doing, teams attempt to "fix" problems with the wrong solution: more methodology, more process, more documentation. Often, the right solution revolves around acquiring more talent, improving skills, and increasing interactions.

Part of the customization process involves tweaking specific practices to match the team. For example, there are many variations on the practice of pair programming—rotating pairs, pairing developers and testers, pairing time per day. Teams that are practicing pair programming need to experiment with variations to find those that work best for them.

It often takes three or four iterations for a team to get into a "groove" and understand just how each of its scenarios will work out in practice. Teams should therefore take care with any adaptations early in a project, particularly when using previously untried scenarios or practices.

Scaling

One of the most pervasive comments about Agile methodologies is that they don't scale. Many critics concede that Agile approaches work fine for a single six-to ten-person project team, maybe

even for a couple of teams of that size, but once we get to "real" project size, these approaches just won't scale. This criticism gives rise to two questions: Does scale matter? and Are the critics correct?

The simplest answer to the first question is that no, scaling doesn't matter. Here's why. Michael Mah, principal of metrics firm QSM Associates, told me that in the 1997–2000 time period, the average project size was nine people, up from six to seven people earlier in the decade.

David Garmus of the David Consulting Group, another metrics consulting firm, also talked with me about project size. "Outside of military projects," David said, "very few projects have more than 100 people." Furthermore, most large projects are broken down into relatively autonomous subprojects, each of which would have far fewer than 100 people. David looked through his company's database and came up with the following. For a large automobile manufacturer, primarily in sales and marketing applications, David estimated that 60 percent of all projects had 10 or fewer people, 35 percent had 11 to 50 people, and 5 percent (or less) had more than 50 people. In those same categories—10 or fewer, 11 to 50, and greater than 50—the numbers for a large telecommunications company were 40, 40, and 20 percent, and for his banking and insurance clients, the numbers were 67, 25, and 10 percent. These numbers include both new development and enhancement projects.

What this means is that *if* we constrained Agile methodologies to projects of 50 or fewer people, the probability is that they would *only* apply to around 75 to 80 percent of all projects in the world! Maybe we should just stop here.

However, ASDEs do scale. Although most of the projects reported on to date have been small to moderate in size (in the up to 50 people category), there is evidence from software companies such as Microsoft and Borland that Agile-like practices have been used successfully for very large projects. Jeff De Luca (Chapter 20) reported on two large FDD projects (of 50 and 250 people), and there are a number of practices described in my own *Adaptive Software Development* book that address scaling issues. Ken Schwaber and Mike Beedle (2002) have used the scaling techniques identified in their book on Scrum on larger projects. Bob Charette has employed LD on large projects, and, with its emphasis on business domains and templates, LD has the potential to scale to very large projects.

So the scaling issues might be summarized as follows. First, because at least 50 percent of all projects require fewer than 10 people and probably 75 percent or so require no more than 50 people, ASDEs are perfectly sized for the majority of projects. Second, there have been a number of successful Agile projects in the 50- to 250-person size range, but the data should be considered preliminary at this point. Finally, in the over 250-person size range, there is little data on Agile approaches being applied. Then again, on projects with more than 250 people, methodology will have almost no impact on success or failure—politics will dominate. Furthermore, I doubt many companies want to incur the risk of exploratory projects, which Agile approaches target, in a project of several hundred people.

Methodology Scaling: Balancing Optimizing and Adapting Elements

But how to scale methodology is still an issue. Historically, most scaling efforts have focused on process—adding formality, process steps, and documentation. Although the process approach to scaling has resulted in improved quality and reduced costs in some application domains, process-driven scaling tends to reduce innovation and adaptability.

In order to adequately deal with methodology scaling, three aspects of the Agile versus rigorous debate (outlined in Principle 5 above) need to be addressed.

- Documentation is not understanding.
- Formality is not discipline.
- Process is not skill.

First, as discussed in Chapter 9, documentation is not understanding. Knowledge management experts have alerted us to the difference between explicit (written down) and tacit (embodied in experience, skill, and judgement) knowledge. Learning what we need to know in order to manage and build a complex software application requires far more than explicit, documented information; it requires tacit knowledge—information in context.

Recent studies and experience indicate that a typical requirements document, for example, may provide only 15 to 20 percent of the understanding needed for development. Documents provide an essential starting point, but they fail to convey anywhere near the total understanding required to build a software application. Understanding arises from the interaction of people who have the tacit knowledge—customer to analyst, developer to developer, analyst to developer, developer to tester. Many people operate under the misconception that more documentation equals more understanding.

Second, formality is not discipline. While no one would question that high quality is related to certain types of discipline—disciplined testing, for example—we must question the usefulness of formality. For example, practitioners of XP are disciplined about testing, but the discipline may be conducted in an informal way. Formality deals with adherence to established forms or rules. Discipline relates to behavior. RSMs have produced document-heavy processes that are formal, but the discipline to actually implement those processes is often missing. Given the choice between a disciplined group of testers that are informal and an undisciplined group of testers that merely fill out a bunch of forms, I'll take the former group every time.

Finally, as discussed in Chapter 7, process is not talent or skill. Even though process can support and leverage skill, increasing process does not make up for skill deficiency. The essence of skill isn't what we record on paper; it's defined by proficiency, mastery, virtuosity, savvy, and artistry.

When designing and particularly when scaling a methodology, these three factors—process versus skill, formality versus discipline, and documentation versus understanding—need to be taken into account. Looking at Figure 25.4, the horizontal axis indicates "optimizing" practices—those most often associated with process, formality, and documentation. The vertical "adaptation" axis takes on the dimensions of collaboration (understanding), discipline, and skill. So, while larger projects indeed require additional formality and documentation, they first require practices to scale collaboration.

Figure 25.4. Balancing Optimizing and Adapting Dimensions

High ─

Typical
Agile
✗ Methodology

Adapting

Typical
Rigorous
✗ Methodology

Low ─

Light   Heavy

Optimizing
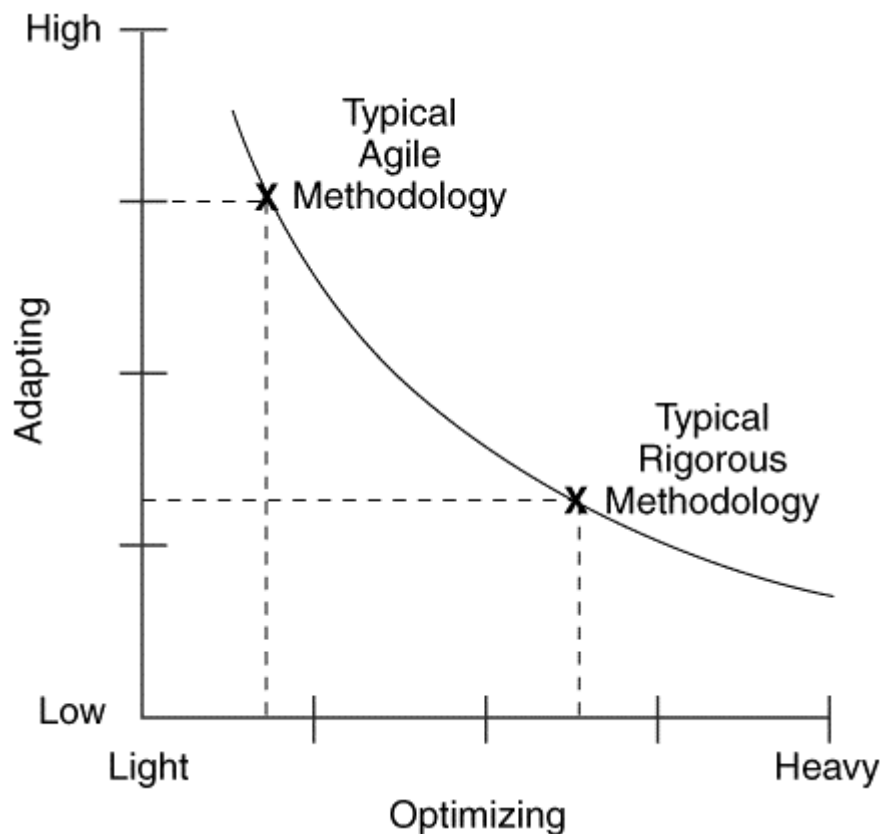
[Figure 25.4](#) helps reorient the debate about rigorous versus Agile methodologies. In most debates, the horizontal axis becomes the focal point. People argue endlessly about documentation, for example: none versus plenty, formal versus informal, diagrams versus text. Traditionalists argue that anyone tending toward the left on the axis is unprofessional or, even worse, immature! Agilists counter that less is better.

Management may be uncomfortable with the vertical axis because it's fuzzier. Innovation, creativity, skill, collaboration, and exploration are all harder to manage and measure than process and artifacts. Rather than argue about the level of documentation or process, I think the debate should be about the relative "mix" of the vertical and horizontal axes' characteristics. I have clients who must maintain their ISO 9000 certifications or conform to US Food and Drug Administration regulations. They must follow prescribed processes (which they should feel obligated to modify over time) and documentation, but they must also innovate. Just because an organization reduces process and documentation doesn't mean it automatically becomes innovative and skilled. Nor is the reverse true.

One of the reasons for the divide between process and practice is often the perception that an onerous process reduces the incentive to use *any* process. Reams of standards, policies, forms, and process descriptions sit in binders on shelves. (Well, there actually *has* been progress. They now reside on fancy Web sites—still unused, of course.) Promoting minimal process and maximal skill has the added attraction that the minimal processes might actually be used.

And so, the arguments shouldn't be about process versus no process or documentation versus no documentation. The focus should be on balancing adapting and optimizing practices, balancing innovation with high-quality production, supporting talented individuals and teams with light processes and tools, and separating the really important from the perception of importance. "People, and relationships, are the new bottom line of business, not simply for humanistic reasons, but as a way to promote adaptability and business success," write Roger Lewin and Birute Regine (2000). Focusing on skills and relationships doesn't mean abandoning tasks, tools, and artifacts; however, it does mean altering one's perspective on them.

Collaboration Scaling

When rigorous methodologists approach scaling issues, they typically look to increase methodology weight first—increasing the number of methodology elements and their ceremony. An Agilist looks first at how people work together—how they collaborate—and only then at what minimal increases in documentation or process are necessary.

For larger, distributed teams, different stages in a project may require different collaborative interactions. For example, formulating the initial architecture and interfaces may require constant interaction at the outset, but less interaction once each group understands the modules or classes it is to build and the workings of the common interfaces.

By reducing the essence of practices like customer-developer interfaces to their basics—vision, features, priorities, conversations, and acceptance (as described in Chapter 5)—teams can employ additional practices such as JAD sessions or customer focus groups as size increases. Peer-to-peer collaborative practices can also be extended. For example, two distributed feature teams could extend the practice of pair programming to one of *traveling pairs*. Every other iteration, a pair from one feature team could travel to the second site where they then pair with developers from the second team. Traveling pairs transfer knowledge about the team's features at a working level. No matter how good the architectural decomposition of a product, two distributed teams working on the same product need a certain amount of working-level conversation and collaboration.

Similarly, the practice of collective ownership of code could be extended. I've seen criticisms of collective ownership implying that while it may work for single feature teams, it surely wouldn't work across multiple feature teams. Rather than say collective ownership doesn't work across multiple teams, why not ask the question, "How could we extend the concept of collective ownership to work across multiple feature teams?"

What generally happens when one feature team needs another team to build or modify a piece of code? The first team writes a request and sends it to the team's lead, who approves it and sends it on to the project manager, and so on. A collective-ownership derivative, for reasonably close-distance teams, might be for one developer to walk over and request to "pair" with someone from the other team. Together, this pair made up from one person from each feature team would sit down, hash through the requirements and design, and then develop and test the new code. Since one programmer understands the requirements and the other, under the collective-ownership practice, could change his team's code, this practice gets the job done with little, if any, paperwork.

From a project management perspective, one might argue that this cross-team pairing could disrupt the work of both teams. However, if this happens, it would be a clear indication of architectural problems that should be addressed. As long as the practice takes a reasonable amount of time, it helps the collaboration between feature teams, streamlines interactions (little project management overhead), and provides a indicator of the quality of architectural modularization. The system of practices interact to focus efforts on valued-adding activities, not control procedures.

The range of collaboration practices and their use with increasingly larger teams is beyond the scope of this book. However, the point is that scaling collaborative practices is vital to designing methodologies that work on large-scale projects.

Architecture and Integration Scaling

The word "architecture" is nearly as abused as the word "quality"—everyone has his or her own definition. The Software Engineering Institute's Web site lists page after page of architecture definitions.[6] These definitions contain a number of common words—common goal, decisions,

components, structures, relationships, interfaces—in a bewildering array of combinations. Interestingly enough, none of them contain the word "integration."

[6] www.sei.cmu.edu/architecture/definitions.html

Deciding on iterations and what features go into each iteration is *project* planning. Architecture is *product* planning. Just as project planning involves some up-front "anticipation" and then continuous "adaptation," so does a product architecture need the same balancing of anticipation and adaptation. The project team has to begin work on some feature, and the iteration plan tells them which one. The project team has to make some technology decisions—database, middleware, hardware, operating system, development language, class structure—and the architecture plan tells them where to start. Some parts of the architecture will change more than others during development; for example, the database schema will change more often than the DBMS.

Software architecture for large projects usually entails defining modules so that subteams can work somewhat independently on different parts of the product. Modules with greater cohesion and less coupling are thought of as more independent of each other, and therefore they require less interaction between subteams working on them.

From a methodology perspective, the problem is that architecture has often been perceived, by architects especially, as a front-end activity with no discernable feedback. Once the "architecture" has been designed, the architect's work is over, and he or she go on to the next job. There is often little room in the project plan for feedback, rework, and *testing* of architectural decisions. One of the tenets of Agile development is that final results—working software—are the arbiter of success. As such, the architect's job should extend to both knowledge integration and product integration testing.

Part of the architect's job (the "architect" could be a few people from each subteam) should include making sure that subteams continually engage in the appropriate level of conversation to ensure that every member understands the architecture. Publishing interface specifications isn't enough. If the architects find that a two-hour total team meeting once every two weeks is sufficient to resolve architectural issues, that is a good indication that the product's modularity is working well. If the teams are meeting two hours per day to gain that same degree of understanding, the module boundaries are probably wrong and need to be changed.

The architecture team should also be responsible for integration testing and software migration. If one of the objectives of architecture is to ensure good product integration, then the people responsible for defining modularization and interfaces should test them. Poor integration should feed back into the architecture refactoring process. (Of course there's an architecture refactoring process!) Finally, architecture refactoring may lead to migration work. For example, if a database schema changes during product development, there may be a data migration task to accomplish. If architects never get feedback on what architecture decisions were good and which weren't, they are doomed to repeat mistakes. Having architects heavily involved in cross-team knowledge integration, integration testing, and software migration will keep the sometimes "pie-in-the-sky" architectural activities grounded in the "mud" of reality.

Do large Agile projects require an architecture? Of course. But an Agile approach to architecture doesn't focus on documents, process, and models (although some of each is important) but on knowledge sharing through interaction, viewing architecture itself as constantly adapting, and ensuring that the architecture team is grounded in the reality of producing working software.

Agile Methodologies for the Enterprise

Small organizations may select a single ASDE and customize it to their needs. This chapter was *not* for them. Over the past several years, I've worked with a number of large, international companies with diverse, far-flung software development groups. They want to be Agile—to try ASD, or Scrum, or XP—but they also want a degree of consistency across their enterprises. This chapter *is* for this group—it is about evolving an Agile framework that can incorporate the concepts of ASDEs within an enterprise.

Organizations with a greater range of project sizes and objectives may develop a suite of Agile (and rigorous) problem domain templates that can be selected, tailored, applied, and then adapted to individual projects. For example, if Scrum was the selected ASDE, then additions might be required to scale from a 6- to 8-person project to one of 25 to 30 people. Similarly, if Crystal Methods were selected, organizations would then have a set of templates: Crystal Clear, Orange, and Red. Finally, an enterprise might need a set of templates in which ASD, LD, XP, or FDD could be used independently on different projects—or in some combination. Several companies have "wrapped" XP with other Agile approaches. I've been involved in an ASD-XP combination effort and talked to others about Scrum-XP and LD-ASD combinations. Alistair and I are pursuing a Crystal-ASD combination.

There are differences between *practices* in various Agile methodologies and similarities, though not unanimity, in *principles* and *values*. There are differences in philosophy between rigorous, software engineering–oriented methodologies and Agile, software craftsmanship–oriented methodologies. RSMs tend to be prescriptive, high ceremony, change resistant, externally imposed (not by the project team), tightly controlled by hierarchy, with many policies (must-do elements). ASDEs tend to be heuristic, low ceremony, change tolerant, internally imposed (by the project team), loosely controlled by peers, with few policies. Every large organization needs a suite of methodologies, and every project team needs a customized version in order to match with its problem domain and culture. To develop these, teams need to understand the fundamental differences and similarities of each approach.

Designing an Agile Methodology doesn't have to be a time-consuming, elaborate process, but it does need to be thoughtful and incorporate both the practices and the principles of Agile development.

Chapter 26. The Agile Metamorphosis

This is the culmination of many faddish ideas for producing software more efficiently. But behind it lies a healthy emphasis on the virtues of teamwork in a business plagued with prima donnas.

—The Economist, *"Agility Counts"*

Metamorphosis, from the Greek, means to transform, literally to "form differently." *Metamorphoses,* a poem written by the Roman poet Ovid, chronicles a series of stories based on Greek myths in which the characters undergo personal transformations. Software people are in the middle of such a metamorphosis—we are caterpillars struggling to become butterflies.

With such a metamorphosis under way, we are left with a simple question: Can we differentiate Agile from non-Agile ecosystems? If we can't, then it will be difficult to say what problems might be best addressed by ASDEs or what cultural domains are conducive to Agile approaches. If we can't differentiate, then it will be difficult to understand strengths and weaknesses of Agile and rigorous approaches and when it might be risky to try one over the other. If we can't differentiate, then we won't know if we are caterpillars or butterflies.

Talented technical people will always engage in debate. Just as programmers argue over C++, Java, and Smalltalk, methodologists argue over practices, processes, and development philosophies. Sometimes the debates become heated, and periodically they become overheated. In the early 1980s, I participated in the debates over structured analysis and design (process-centric methods) versus data-structured methods. Ed Yourdon, Tom DeMarco, Tim Lister, and others were on one side of the debate. Ken Orr, J. D. Warnier, myself, and others were on the opposite side. There were some lively discussions, but in our differences, there was always respect and collegiality.

The point is that spirited debate and even some in-your-face jabs are good for our profession. Healthy, collegial debate helps everyone think through the issues clearly. Surely, if we are going to debate something as boring as methodology, at least we should have a little fun! Only by defining and debating will others be able to understand similarities and differences and be able to apply the right mix of methodology elements to match their own particular cultural perspective and problem domain.

At one level, ASD, FDD, Crystal, and RSMs are represented by a collection of artifacts and practices and therefore cannot be Agile or rigorous in and of themselves. Their "Agility" originates not from things but from the people that implement them. XP can be, and has been, implemented in very dogmatic, inflexible ways, and some RSMs have been implemented in very Agile ways. For example, the Modern Transport case story in Chapter 9 describes how an RSM organization in India applied a "lighter" approach to a specific project.

"There is no opposite to Agile Software Development," says Alistair Cockburn (2002a). "The word 'Agile' depicts where the people choose to focus their attention. Alternatives to Agile Software Development arise as soon as they focus their attention elsewhere." Some choose to focus on the predictability of a methodology, others on its repeatability. Some choose to focus on collaboration, others on the technical aspects of software engineering. Each focal point has advantages and disadvantages.

So the primary assertion of this last chapter is that, yes, there are meaningful differences in *emphasis* between Agile and rigorous ecosystems. Furthermore, three traits can assist us in analyzing the similarities and differences between them: a chaordic organizational perspective, a collaborative set of values and principles, and an emphasis on barely sufficient methodology. I hope that highlighting these differences and similarities, and thus establishing a framework for

collegial debate, will benefit the software development community. In this debate over perspective, it is also critical to keep problem domains in mind. The analyses in this chapter focus on complex, high-speed, high-change problem domains.

Chaordic Perspective

There are many attributes of organizational culture and management that flow from a chaordic viewpoint: Leadership-Collaboration management, the use of generative rather than inclusive rules, the fostering of emergent rather than cause-and-effect practices, and more. What particularly differentiates Agile from rigorous, though, is their respective views on predictability and repeatability. Traditional managers have viewed change negatively, seeing it as a destabilizing force to be eliminated if possible and tightly controlled otherwise. Agile managers understand that change can be destabilizing but also that the ability to manage destabilizing forces, and thereby harness change, is critical to delivering value in turbulent environments. If one believes in the primacy of prediction and planning, then change will be viewed as a necessary evil and conformance to plan as paramount.

Agilists perceive that predictions are tenuous, and therefore we must find ways to embrace change in the development process. Responsiveness to change takes precedence over conformance to plan. Plans are guides, and they help us make difficult tradeoff decisions. Business goals are achievable, but the specific path (the project plan) to those goals often remains murky. Traditional managers acknowledge that planned targets are often missed, but they attribute those misses to *immature* processes. "If we would just do everything our processes told us to do, we would achieve our plans," runs their thinking. Agilists, on the other hand, believe that the blend of chaos and order inherent in the external environment and in people themselves argues against the prevailing "wisdom" about predictability and planning. Furthermore, I think that failure to recognize the limitations of a planning-centric perspective often leads to dysfunctional management—managers' actions work against the very objectives that they are trying to achieve.

"I finally learned that I could write decent stuff if I let go of planning, control, and vigilance," says Peter Elbow. "I had to invite chaos and bad writing. Then, after I had written a lot and figured out a lot of thinking, I could go back and find order and reassert control and try to make it good" (Elbow 1998). Elbow writes in a chaordic fashion. By and large, schools teach methodical writing: develop a theme, create an outline, annotate the outline, write the text to fill in each topic, edit the text to correct grammar and spelling errors, review and revise the text. Some writers work well this way, and others—like myself—would appear from the outside to approach the writing task in a rather disorganized fashion. Developing a few scattered notes, writing down random paragraphs, sketching out some ideas, developing a brief outline, seemingly random reorganizations, shuffling through stacks of paper on my desk—the process appears chaotic.

Which approach appeals to you—chaordic or methodical? Are there distinct differences between the two? Might we use one approach to write a technical manual and the other to write a science fiction thriller? People work in different ways. Let them.

Nowhere is this dichotomy in approach more pronounced than in the realm of process repeatability. The word "repeatable" remains a mantra in corporate executive ranks. Some RSMs are based on exhaustive metrics that are used to refine processes until they are repeatable within minor tolerances. In fact, I believe that rigorous projects often conform to plan because the processes *aren't* repeatable. It's the adaptability of people using any processes, not the repeatability of the process itself, that enables project teams to "meet" goals.

Agilists believe that good practices and processes can improve consistency but that repeatability—in the statistically controlled feedback loop sense of the term—is a fantasy. Unfortunately, it is a

fantasy that many corporate executives believe in, and that belief exacerbates the dysfunctionality between product development and management. Executive management has been told that they *can* have it all, and they want to believe it. They are then disappointed when plans don't work out. Their solution: more processes and more standards. Are traditionalists willing to commit to this view of repeatability as a problem rather than a solution? Agilists believe that change must be adapted to, that it can't be planned away. You can have flexibility, or consistency, or some blend of both. But expecting a process or methodology to provide ultimate flexibility and complete predictability at the same stretches the limits of credulity.

In any project, there are two fundamental components—innovation and repetition. Innovation is inventing and problem solving—figuring out things that we don't know how to do already. Repetition is doing things we have done before—repeating things that we know how to do. In software development, the daily build is a repetitive process that we do over and over again during a project.[1]

> [1] Daily builds are closer to defined processes and are therefore repeatable to some degree. However, it's the empirical processes that are critical to success.

Innovation is unpredictable but not uncontrollable. One of the ironies of chaordic approaches to development is that control comes from letting go, not holding on—at least for the innovative parts. Whether it's ferreting out elusive user requirements, debating over a class structure, or agonizing over a coding problem, often the best approach involves circling around the problem rather than attacking it directly. Creativity comes from alternately focusing and defocusing, from trying alternatives, from setting the problem aside for a few days and letting it percolate in the subconscious mind.

People do not think in a straightforward, linear manner. Our thoughts are random, they change over time, they grow and mature, they get stuck for long periods and then burst with invention. Thinking is an organic, cyclic process that grows from variety, not certainty. As Henry Petroski, civil engineering professor and author of *The Evolution of Useful Things,* writes, "The form of made things is always subject to change in response to their real or perceived shortcomings, their failures to function properly. This principle governs all invention, innovation, and ingenuity; it is what drives all inventors, innovators, and engineers" (Petroski 1994). Or, from Peter Elbow (1998), "This paradox of increased overall control through letting go a bit seems paradoxical only because our normal way of thinking about control is mistakenly static." By trying too hard to force linearity into our thinking, we actually stunt the creative process. By letting go, we encourage the creative process, and therefore results happen more quickly and more productively.

An Agile view is a chaordic view—balanced between chaos and order, perched on the precipice at the edge of chaos, peering down into the maelstrom of chaos on one side and the monotony of order on the other. A chaordic perspective does not abandon predictability or control but views them from an alternative perspective. For example, to achieve some measure of predictable milestones, we understand that scope will vary—no matter how good we are at requirements gathering. As we increase our competence at requirements gathering, we reduce some of the variability but not all of it, and often not the causes of the largest variations.

ASDEs are another step in the 50 years of progress we have made in understanding how to development software. Agile approaches incorporate many of the traditional practices and tools that are part of improving productivity and automating highly repetitive steps. Nevertheless, Agilists take a different perspective on how to cope with uncertainty and approach process management differently.

In order to control the results, Agilists often "uncontrol" the steps. When we tell the members of a development team that the "price the customer order" feature has been assigned to them in the next iteration, they are then free, within a broad established context, to figure out how to achieve that end in an organic, nonlinear fashion. When we tell a development team that it has 16 hours to

complete the requirements, 8 hours to do design, 20 hours to code, and 12 hours to test a feature—in sequential order of course, so we can "control" progress—we short-circuit creativity.

"If we were to set out to design an efficient system for the methodical destruction of community," writes Dee Hock (1999), "we could do no better than our present efforts to monetize all value and reduce life to the tyranny of measurement." In contrast, Watts Humphrey writes, "The objectives of software process management are to produce products according to plan…. The basic principles are those of statistical process control" ([Humphrey 1989]). Hock, founder and CEO emeritus of Visa, and Humphrey, founder of the Software Engineering Institute, have irreconcilable differences. Both men have gained a great deal of respect and success in their fields, but their values related to organizations and how they should be managed are 180 degrees apart. We should affirm their differences, not try to homogenize them.

I know people who thrive in the defined, prescribed, measured, precise culture of high-level RSM organizations. I know others who would wither in this type of culture—myself being one of them. Still, I don't imagine that working for Dee Hock would be any less intense than working for Watts Humphrey—just very different. The issues of organizational perspective and values and principles have been missing from methodology debates in the past, yet they are the very foundation of why people work the way they do and what motivates them. If nothing else, the Agile Software Development movement has brought these issues to the fore.

Collaborative Values and Principles

National Basketball Association star John Stockton, all-time assist leader and holder of many other accolades, had a knee lockup prior to a game. The game, one of those preseason ones that don't mean a lot, was one he could have easily skipped, but he hobbled through warm-up and managed to play—dominating one of highly touted NBA rookies. Utah Jazz Coach Jerry Sloan said after the game, "You can coach all you want. But if you don't have guys like that, you aren't going to succeed much."[2]

[2] Sloan is quoted in Steve Luhm, "An Offer He Can't Refuse," *The Salt Lake Tribune,* Sept. 2, 2001, p. C9.

Contrast Jerry Sloan's quote about John Stockton with the CEO's quote (which I've paraphrased) from Chapter 1, "Software development is just very labor-intensive, mechanical work." Is software development labor intensive or competency intensive? There is a tendency to give a flip answer to this question, but the answer has deep philosophical connotations. "Labor-intensive" means "thinking un-intensive," as in one group does the thinking and another does the work. In the Taylorist manufacturing plants of the mid-twentieth century, typified by automobile assembly lines, the rule of micro-managing efficiency experts was the cause of much labor versus management strife.

If work is mechanical, then it can be routinized and standardized. It can be precisely measured, planned, and controlled. It requires the process designers to think and the processes implementers to follow. I'm obviously describing the extreme, but this is how people *feel* about these environments. It's just these kinds of assumptions about people and process that cause rampant dysfunctionality. An exmple is executive management who want an improved software development process—one that is productive, effective, and predictable. However, in order to be responsive to customers—the clarion call of executives, and not an inappropriate one—they constantly juggle development priorities and people. They want flexibility and adaptability but productivity and predictability as well. They want to change all the inputs but have the outputs remain predictable.

As the Agile Manifesto states, process isn't inherently bad or wrong; in fact, a dollop of process every now and again can be healthy. However, beginning with the heavy emphasis on software process engineering and business process reengineering methodologies in the late 1980s and early 1990s, we've sacrificed the very characteristic that creates great software—or any great product, for that matter—reliance on individual strengths.

Just as the business world has launched itself into an intense period of turbulence and change, brought about in part by software itself, we are shrinking away from change by trying to ignore it—by believing that performance is measured by repeatability and that it comes from standardization. *In our quest for repeatability, we have sacrificed individualism.*

Streamlined, minimal, customized methodologies flow from competency thinking. Competency-based thinking scales by first considering how to extend, leverage, and acquire competency. In rock climbing, competency has a distinct measurement system. Easy climbs are numbered 5.6 or 5.7 in difficulty, while extremely difficult climbs are rated 5.13 or 5.14. A competent 5.8 climber might attempt a 5.9 climb or struggle up a 5.10 climb, but would only dream about the day he might master a 5.12 route.

There are not, and should not be, numbered grades for programming or database design skills. However, there *are* competency levels that need to be ascertained for any project. There is nothing "wrong" with a 5.8 climber; he just may not have acquired enough skill or knowledge to attempt a 5.10 climb. There is nothing "wrong" with a Java developer who doesn't have enough skill or experience to design a complex class structure; she just needs to be assigned to a part of the project in which she can be effective or to work closely with someone more experienced. Methodology provides a framework that can help people organize and communicate, but it doesn't make up for lack of skill, experience, or knowledge.

Mechanical thinking views individuals as interchangeable and therefore attempts to improve performance by employing processes to standardize people to the organization. Competency thinking views individuals and teams as having unique and valuable competencies and therefore improves performance by customizing processes to capitalize on those unique qualities. As projects get larger and more complex, mechanical thinkers work to build processes and standards, while competency thinkers work to find or build the competencies required.

Basketball is a team sport filled with individual talent. Software development is similar. Collaboration—joint production of work products, collaborative decision making, and knowledge sharing—builds high-performance teams out of groups of individuals. Collaboration is multi-faceted: team member with team member, management with team, team with customer, team with team.

Few deny that people, collaboration, communication, knowledge management, innovation, teamwork, and decision making are vital to successful project completion. It's not that Agilists are "for" collaboration and others are not, but there is a different *emphasis* on collaboration. To get some feel for this emphasis, I picked out three books on the Unified Process (UP) (Jacobson et al. 1999, Kruchten 2000, Royce 1998) and three on Agile development (Beck 2000, Cockburn 2002, Highsmith 2000) and did a simple comparison. Using the hypothesis that items that are mentioned in a book's index are more important than those that aren't, I searched the indexes for seven terms: communication, collaboration, decision making, knowledge sharing or management, inspections or pair programming, innovation, learning, and teams or teamwork. I then tallied the number of index entries (there might be several subentries under communication, for example) and the number of page references.

For the three UP books, there were *5 index entries* on these topics, for a total of *19 pages*. For the three Agile books, there were *98 entries,* for a total of *more than 400 pages*. Although this is a quick and unscientific analysis, these key UP contributors, at least as reflected in their published writing, place less emphasis on collaboration than the three Agile writers.[3]

[3] In a series of discussions and emails, Ivar Jacoboson, vice president of process strategy for Rational Software, argues that RUP (Rational's commercial version of the Unified Process) is more collaborative than these numbers indicate. The RUP provides 112 references to communications, 76 on collaboration, 86 on decision making, and additional references on other topics.

I am also sure that, to ardent supporters of RSMs, Agile development appears to be an undisciplined, unmeasured, uncontrolled, ad hoc process that produces inferior, high-cost, difficult-to-maintain code. RSM supporters emphasize fundamentally different cultural values. I believe that it is more beneficial to celebrate the differences between ASDEs and RSMs than to try and mesh them into a single framework. RSMs describe a particular ecosystem that works well in certain organizations for a certain set of problems. ASDEs describe a different ecosystem that works best with a different, although sometimes overlapping, set of problems and in a very different organizational culture.

Barely Sufficient Methodology

A *barely sufficient* Agile methodology means more rather than fewer processes, briefer documents, and less formality. An Agile methodology focuses on results: It is simple, it is responsive and self-adapting, it stresses technical excellence, and it emphasizes collaborative practices. Using these broad categories, we can identify additional differences in emphasis between methodology camps.

A focus on results includes developing a clear mission and tradeoff priorities; planning frequent, feature-oriented iterations; and stressing working software over documentation and models. On a scale of 1 to 5, with 1 being less Agile and 5 being more Agile, I would rate all the ASDEs in the 4 to 5 range and RSMs in the 1 to 3 range (depending on the particular one and its degree of emphasis on modeling, extensive documentation, process, or metrics rather than working code).

A simple methodology stresses value-adding (versus administrative and overhead) activities, keeping the "weight" down by de-emphasizing ceremony and the number of elements, and letting additional processes emerge as teams need them. Again, using the 1 to 5 agility scale, XP, Crystal Clear, and FDD might rate 5s; DSDM a 4; and RSMs 1 to 3. These ratings are for similar-sized projects; so for a team of four to eight colocated people, for example, a streamlined version of an RSM is still likely to be heavier than XP or Crystal Clear.

An Agile methodology both responds to external project influences and self-adapts to the team based on use. All the ASDEs are very responsive to external influences. RSMs should be self-adapting, but in practice they often become rigid, particularly in organizations that take the one-size-fits-all approach to methodology. With several RSMs, the emphasis is not on responding or self-adapting but on measurement and refinement. These are not inherently wrong things to think about, they just are not Agile. RSMs often stress customizing to different project types, but stripping them down to something simple and constantly adjusting them can be difficult because of their overall bulk.

On the self-adapting scale, I'd probably put ASD and Crystal at 5 and XP at 3. In my opinion, XP *is* self-adapting, although it does receive some flack about its lack of internal agility. XP was clearly designed to handle external changes, the first type of agility, but is sometimes criticized for its perceived resistance to the second type. "If XP says you *must* do the 12 practices (pair programming, refactoring, test first, etc.), then it is not self-adapting and in fact is just as rigid as the rigorous methodologies," goes the complaint. "XP professes a belief in individuals and their competency, but then forces compliance to a set of practices."

To some extent, all methodologies—Agile or rigorous—face this dilemma of discipline in performing a set of practices versus self-adaptation. Crystal might have 4 "musts," XP 12, and an RSM 42, but at some point there has to be a line that divides the thing (the methodology) from not

the thing—otherwise there is no difference between XP and the UP, or FDD and the CMM, or Agile and rigorous development in general. Crystal and ASD have explicit practices for adjusting or adapting practices at regular intervals, but I don't therefore consider them self-adapting and XP and FDD not. XP may not be as quick to self-adapt as Crystal or ASD, but there are also good reasons for this, as discussed in the section on generative rules in Chapter 13. There can be such as thing as *too* adaptive.

The danger in being *too* adaptive or Agile is moving from one practice to another without adequate understanding of how the practice should be implemented or integrated with others. If we believe in a chaordic organizational model, then when we find a set of core development rules, we *should* be reluctant to alter them without serious consideration. Balancing structure (not changing) and flexibility (changing) is part of adopting a chaordic perspective.

As Chapter 11, which is on technical excellence, points out, ASDEs stress technical excellence as a key factor in building change-tolerant software. Poor design and poor testing lead to change-intolerant software that in turn causes the next set of changes to take longer, be more expensive, and further increase the degree of change intolerance. RSMs and ASDEs are all intended to create technically superior results—they just go about it in different ways. And, although FDD or DSDM might turn out as high quality a product as the CMM, for life-critical applications like control of medical equipment, the extra verification, review, and control practices of rigorous CMM-like methodologies are required to ensure safety (or at least to provide the appearance of safety, much like confiscating fingernail clippers from air travelers).

While all methodologies indicate the importance of collaboration, the Agile approaches place more emphasis on them and include a greater number of explicit peer-to-peer collaboration practices.

The Agility Ratings

Any rating of the relative "agility" of methodologies is fraught with subjectivity and personal bias. Nonetheless, given my own definition of an ASDE as a combination of chaordic perspective, collaborative values and principles, and barely sufficient methodology; my own understanding of each of the approaches; and my personal biases, I'll hazard the attempt. Table 26.1 reflects my view of the relative agility of each of the Agile ecosystems versus the CMM and the UP.[4] I tried to rate each of the traits independently. Although the various ASDEs may have strengths in different areas, they all rank high in the barely sufficient methodology category.

[4] While there are people who might implement ASD, for example, in a very non-Agile way, and others who might implement the UP or the CMM in very Agile ways, I've constructed this analysis based on my interpretation of the norms of each methodology's implementations. There are certainly more Agile implementations of both the UP and the CMM, but their normal usage tends to be more rigorous than Crystal, Scrum, or other Agile approaches.

Table 26.1. Agility Ratings (Note: Higher is more agile.)								
	CMM	UP	ASD	Crystal	DSDM	FDD	Lean	Scrum	XP
Chaordic Org View	1	2	5	4	3	3	4	5	5
Collaborative Values	2	3	5	5	4	4	4	5	5
Barely Sufficient Methods (BSM)									
Results	2	3	5	5	4	4	4	5	5
Simple	1	2	4	4	3	5	3	5	5
Responsive & Adaptive	2	3	5	5	3	3	4	4	3
Technical Excellence	4	4	3	3	4	4	4	3	4

Collaboration Practices	2	3	5	5	4	3	3	4	5
BSM average	2.2	3.0	4.4	4.4	3.6	3.8	3.6	4.2	4.4
Overall Agility Rating	1.7	2.7	4.8	4.5	3.6	3.6	3.9	4.7	4.8

I am sure many UP and CMM proponents will argue about these ratings, just as I'm equally sure some of the Agilists will disagree also. However, my primary objective in presenting these ratings is not to end the debate but to provide a framework for continuing it.

Final Thoughts

Much of the debate about ASDEs has centered around practices—a technical debate about pair programming versus inspections, say, or the push for minimal documentation. But the heart and soul of Agile development is not about practices but about a particular organizational perspective and specific values and principles.

In the preface to *Extreme Programming Explained,* Kent Beck discusses two societies—one of scarcity and one of plenty (from the anthropological works of Colin Turnbull)—and compares these cultures with organizational cultures. Making an analogy to a culture of plenty, Kent says, "Such a 'mentality of sufficiency' is humane, unlike the relentless drudgery of impossible, imposed deadlines that drives so much talent out of the business of programming" (Beck 2000).

In *Adaptive Software Development,* I make the point, "Self-organization, emergence, and collaboration are the central themes of Adaptive Software Development. Collaboration puts an emphasis on community, on trust and joint responsibility, on partnership with the customer rather than on an adversarial relationship…. I hope that the Adaptive Software Development approach makes obsolete the notion that linearity, predictability, arduous process improvement, and Command-Control are the only way to develop software" (Highsmith 2000).

In his description of the Agile Alliance revolution, Ken Schwaber talks in strong language: "They had become one with the people. And now, they were gathering to share their experiences and discuss the tyranny that had so damaged their profession and their people. The country is the country of systems development, the people are the software engineers that build systems, and the tyranny is the heavy-handed approach to building systems that has smothered our industry for the past 20 years" (Schwaber 2001).

"Considering software development as a game with moves is profitable," says Alistair Cockburn, "because doing so gives us a way to make meaningful and advantageous decisions on a project. In contrast, speaking of software development as *engineering* or *model building* does not help us make such advantageous decisions…. In my travels, people mostly use the word 'engineering' [software engineering] to create a sense of guilt for not having done enough of something, without being clear what that something is" (Cockburn 2002).

These comments from leaders of the Agile movement reflect people-oriented values, not process and technology. One cannot absorb the essence of Agile Software Development without understanding the depth and breadth of these values. Still, are we being naïve?

In reviewing this book, colleague Sam Bayer brought up an interesting question. In the early 1990s, cutbacks resulting from reengineering left many employees wondering about corporate "loyalty." As the early years of the twenty-first century unfold, corporations are again rife with layoffs. In this seemingly employee-unfriendly era, is it realistic to base a movement on collaborative values whose foundation is trust? Sam asked, "Is this the Achilles' heel of Agile?" Maybe. Maybe embracing trust, communications, and collaboration will ultimately prove too great

a barrier to the wide adoption of Agile approaches within corporations. Maybe we are destined to tilt ineffectually at the windmills of sign-offs, change control boards, and 400-page process descriptions.

Perhaps, but I believe that it is just these values that have generated the surge of interest in Agile ecosystems. Traditional methodologies have long harbored implicit values that treat developers like machine parts to be manipulated—they have stifled rather than uplifted. Knowledge work in the Information Age differs from factory work in the Industrial Age, and it is time to organize and manage work to reflect those differences. There are many bumps in the road to an Agile organization, but therein lies both its competitive advantage and the fun of tilting.

So, what about the future? To the extent that the future business environment continues to be turbulent, I think rigorous cultures face a difficult challenge. No amount of process thinning or document pruning will make them Agile—Agile is an attitude, a sense of how the world works in complex ways. However, to the extent that executives and managers still *want* the world to be predictable and plan-able, Agile cultures and ASDEs will be difficult to implement. In the final analysis, though, businesses gradually but inevitably gravitate to practices that make them successful, and ASDEs will increasingly contribute to successful software projects. RSMs will remain, but there will be many fewer companies using them five years from now.

Silver bullets don't exist, and Agile development isn't easy. Colleague Lou Russell expressed her thoughts about many of the case stories in this book when she said, "I really liked the fact that the case stories were full of the difficulties of actual projects. Things weren't perfect; they were messy and very hard—which reflects the real world."

"Reflects the real world" captures the essence of Agile Software Development—it reflects the practical and the practiced rather than the theoretical and the imposed. Time after time I (and other Agilists) have heard from people, "Your approach reflects how we *actually work*." In some ways, we Agilists are like Don Quixote and his sidekick, Sancho Panza, riding around the software development landscape, tilting at the windmills of traditionalism. We've made dents here and there, but the bulk of large corporate and governmental development remains skeptical. Even in organizations in which teams have successfully implemented Agile projects, their peers often remain unconvinced. This is always the case with something new that challenges the status quo.

In the end, ASDEs have two powerfully appealing characteristics. They offer an answer to the problem of developing better software faster in highly volatile, demanding situations, and they offer a cultural environment that appeals to many individuals. People and organizations have personalities—Ken Schwaber isn't Jeff De Luca; Sun Microsystems is not AT&T. So I have to agree with Martin Fowler that the essence of Agile Software Development boils down to a fundamental belief in the *unpredictability* of our turbulent business environment and the *predictability* of people's and teams' capability to successfully deliver software in the face of that unpredictability. In the end, it's skilled individuals and their interactions among themselves, with customers, and with management that drive success in this cooperative game of invention and communication called software development. We are striving to be butterflies in a world of caterpillars.

Bibliography

"Agility Counts." The Economist, Technology Quarterly section, 22 September 2001, 11.

Ambler, Scott. *Agile Modeling:Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons, 2002.

Anthes, Gary H. *"Charting a Knowledge Management Course." Computerworld*, 21 August 2000.

Auer, Ken, and Roy Miller. *Extreme Programming Applied: Playing to Win*. Boston: Addison-Wesley, 2002.

Austin, Rob. *"Surviving Enterprise Systems: Adaptive Strategies for Managing Your Largest IT Investments."* Cutter Consortium Business-IT Strategies Advisory Service, *Executive Report* 4, no. 4 (April 2001).

Axelrod, Robert, and Michael Cohen. *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. New York: Free Press, 1999.

Bayer, Sam. *"Customer-Focused Development: The Art and Science of Conversing with Customers."* Cutter Consortium Agile Project Management Advisory Service, *Executive Summary* 2, no. 4 (April 2001).

Bayer, Sam, and Jim Highsmith. *"RADical Software Development." American Programmer* 7, no. 6 (June 1994): 35–42.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000.

Beck, Kent, and Martin Fowler. *Planning Extreme Programming*. Boston: Addison-Wesley, 2001.

Bennis, Warren. *"Will the Legacy Live On?" Harvard Business Review* (February 2002): 95–99.

Berinato, Scott. *"The Secret to Software Success." CIO*, 1 July 2001, 76–82.

Boehm, Barry. *"Software Engineering." IEEE Transactions on Computers* 25, no. 12 (December 1976): 1226–41.

———. *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall, 1981.

Bonabeau, Eric, and Christopher Meyer. *"Swarm Intelligence: A Whole New Way to Think About Business." Harvard Business Review* (May 2001): 106–14.

Brown, Shona L., and Kathleen M. Eisenhardt. *Competing on the Edge: Strategy as Structured Chaos*. Boston: Harvard Business School Press, 1998.

Brown, John Seely, and Paul Duguid. *The Social Life of Information*. Boston: Harvard Business School Press, 2000.

Buckingham, Marcus, and Curt Coffman. *First, Break All the Rules*. New York: Simon & Schuster, 1999.

———. *Now, Discover Your Strengths*. New York: Simon & Schuster, 2001.

Charette, Robert N. *Foundations of Lean Development: The Lean Development Manager's Guide*. Vol. 2, *The Foundations Series on Risk Management* (CD). Spotsylvania, Va.: ITABHI Corporation, 2002.

Christensen, Clayton. *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Boston: Harvard Business School Press, 1997.

Coad, Peter, Eric Lefebvre, and Jeff De Luca. *Java Modeling In Color With UML: Enterprise Components and Process*. Upper Saddle River, N.J.: Prentice Hall, 1999.

Cockburn, Alistair. *Surviving Object-Oriented Projects*. Reading, Mass.: Addison-Wesley, 1998.

———. *"Balancing Lightness with Sufficiency." Cutter IT Journal* 13, no. 11 (November 2000): 26–33.

———. *Writing Effective Use Cases*. Boston: Addison-Wesley, 2001.

———. *Agile Software Development*. Boston: Addison-Wesley, 2002.

———. *"Agile Software Development Joins the 'Would-Be' Crowd." Cutter IT Journal* 15, no. 1 (January 2002a): 6–12.

———. *Crystal Clear*. Manuscript in preparation.

Constantine, Larry. *"Methodological Agility." Software Development* 9, no. 6 (June 2001).

Cusumano, Michael A., and Richard Selby. *Microsoft Secrets*. New York: Free Press, 1995.

Davenport, Thomas H., and Laurence Prusak. *Working Knowledge: How Organizations Manage What They Know*. Boston: Harvard Business School Press, 1998.

De Geus, Arie. *The Living Company*. Boston: Harvard Business School Press, 1997.

DeGrace, Peter, and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions: A Catalogue of Modern Engineering Paradigms*. Upper Saddle River, N.J.: Prentice Hall, 1998.

DeMarco, Tom. *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. New York: Broadway Books, 2001.

———. Preface to *Planning Extreme Programming*, by Kent Beck and Martin Fowler. Boston: Addison-Wesley, 2001a.

DeMarco, Tom, and Tim Lister. *Peopleware: Productive Projects and Teams*. New York: Dorset House, 1987.

Dixon, Nancy. *Common Knowledge: How Companies Thrive by Sharing What They Know*. Boston: Harvard Business School Press, 2000.

Dove, Rick. *Response Ability: The Language, Structure, and Culture of the Agile Enterprise*. New York: John Wiley & Sons, 2001.

Drucker, Peter F. *"Management's New Paradigms." Forbes*, 5 October 1998.

DSDM Consortium. *Dynamic Systems Development Method*, Version 3. Ashford, Eng.: DSDM Consortium, 1997.

Eisenhardt, Kathleen M., and Donald N. Sull. *"Strategy as Simple Rules." Harvard Business Review* (January 2001): 106–116.

Elbow, Peter. *Writing without Teachers*, 2d ed. New York: Oxford University Press, 1998.

Fowler, Martin. *Analysis Patterns: Reusable Object Models*. Reading, Mass: Addison-Wesley, 1996.

———. *Refactoring: Improving the Design of Existing Code*. Reading, Mass.: Addison-Wesley, 1999.

———. *"Put Your Process on a Diet." Software Development* 8, no. 12 (December 2000): 32–36.

Fowler, Martin, and Jim Highsmith. *"Agile Methodologists Agree on Something." Software Development* 9, no. 8 (August 2001): 28–32.

Fowler, Martin, and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Refactoring: Improving the Design of Existing Code*. Reading, Mass.: Addison-Wesley, 1997.

Freedman, David H. *"A Few Good Principles: What the Marines Can Teach Silicon Valley." Forbes*, 29 May 2000.

Gilb, Tom. *Principles of Software Engineering Management*. Wokingham, Eng.: Addison-Wesley, 1988.

Gleick, James. *Chaos: Making a New Science*. New York: Penguin, 1989.

Goldman, Steven, Roger Nagel, and Kenneth Preiss. *Agile Competitors and Virtual Organizations: Strategies for Enriching the Customer*. New York: Van Nostrand Reinhold, 1995.

Goldratt, Eliyahu M., and Jeff Cox. *The Goal: Excellence in Manufacturing*. Croton-on-Hudson, N.Y.: North River Press, 1984.

Goranson, H. T. *The Agile Virtual Enterprise: Cases, Metrics, Tools*, Westport, Conn.: Quorum Books, 1999.

Hamm, Steve, et al. *"E-Biz: Down but Hardly Out." Business Week*, 26 March 2001.

Hammer, Michael. *Beyond Reengineering: How the Process-Centered Organization Is Changing our Work and Lives*. New York: HarperBusiness, 1996.

Haeckel, Stephan H. *Adaptive Enterprise: Creating and Leading Sense-and-Respond Organizations*. Boston: Harvard Business School Press, 1999.

Higgins, David A. *Data Structured Software Maintenance: The Warnier/Orr Approach*. New York: Dorset House, 1986.

Highsmith, Jim. *"Synchronizing Data with Reality." Datamation*, November 1981.

———. *"Software Ascents." American Programmer* 5, no. 6 (June 1992): 20–26.

———. *"Messy, Exciting, and Anxiety-Ridden: Adaptive Software Development."* American *Programmer* 10, no. 4 (April 1997): 23–29.

———. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House, 2000.

———. *"Retiring Lifecycle Dinosaurs,"* Software Testing & Quality Engineering 2, No. 4 (July/August 2000a).

———. *"Harnessing Innovation and Speed."* Cutter Consortium Agile Project Management Advisory Service (formerly e-Project Management Advisory Service), *Executive Summary* 1, no. 1 (November 2000b).

———. *"The CHAOS Report—Reality Challenged."* Cutter Consortium Agile Project Management Advisory Service, *E-Mail Advisor* (20 September 2001).

Hock, Dee. *Birth of the Chaordic Age*. San Francisco: Berrett-Koehler Publishers, 1999.

———. *"Institutions in the Age of Mindcrafting."* Paper presented at the 1994 Bionomics Annual Conference, San Francisco, Calif., 1994.

Hohmann, Luke. *GUIs with Glue*. Manuscript in preparation.

Holland, John H. *Emergence: From Chaos to Order*. Reading, Mass.: Addison-Wesley, 1998.

Humphrey, Watts S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

———. *Introduction to the Personal Software Process*. Reading, Mass.: Addison-Wesley, 1997.

———. *Introduction to the Team Software Process*. Boston: Addison-Wesley, 2000.

Iansiti, Marco. *Technology Integration: Making Critical Choices in a Dynamic World*. Boston: Harvard Business School Press, 1998.

Jacobson Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley, 1999.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Boston: Addison-Wesley, 2001.

Jones, Capers. *Software Assesssments, Benchmarks, and Best Practices*. Boston: Addison-Wesley, 2000.

Kaner, Sam, with Lenny Lind, Catherine Toldi, Sarah Fisk, and Duane Berger. *A Facilitator's Guide to Participatory Decision-Making*. Philadelphia: New Society Publishers, 1996.

Kanter, Rosabeth Moss. *e-Volve!: Succeeding in the Digital Culture of Tomorrow*. Boston: Harvard Business School Press, 2001.

Katzenbach, Jon R., and Douglas K Smith. *The Wisdom of Teams: Creating the High-Performance Organization*. Boston: Harvard Business School Press, 1993.

Kerth, Norman L. *Project Retrospectives*. New York: Dorset House, 2001.

Kosko, Bart. *Heaven in a Chip: Fuzzy Visions of Society and Science in the Digital Age*. New York: Three Rivers Press, 2000.

Kruchten, Phillipe. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2000.

Larman, Craig. *Applying UML and Patterns*, 2d ed. Upper Saddle River, N.J.: Prentice Hall, 2002.

Lewin, Roger, and Birute Regine. *The Soul at Work: Listen, Respond, Let Go: Embracing Complexity Science for Business Success*. New York: Simon & Schuster, 2000.

Mathiassen, Lars, Jan Pries-Heje, and Ojelanki Ngwenyama. *Improving Software Organizations*. Boston: Addison-Wesley, 2002.

McBreen, Pete. *Software Craftsmanship: The New Imperative*. Boston: Addison-Wesley, 2001.

McConnell, Steve. *Rapid Development: Taming Wild Software Schedules*. Redmond, WA: Microsoft Press, 1996.

Moore, Geoffrey A. *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*. New York: HarperBusiness, 1991.

———. *Inside the Tornado: Marketing Strategies from Silicon Valley's Cutting Edge*. New York: HarperBusiness, 1995.

———. *Living on the Fault Line: Managing for Shareholder Value in the Age of the Internet*. New York: HarperBusiness, 2000.

Musashi, Miyamoto. *The Book of Five Rings*. Translated by Thomas Gleary. Boston: Shambhala, 1993.

Palmer, Stephen, and John M. Felsing. *A Practical Guide to Feature Driven Development*. Upper Saddle River, N.J.: Prentice Hall, 2002.

Pascale, Richard T., Mark Millemann, and Linda Gioja. *Surfing the Edge of Chaos: The Laws of Nature and the New Laws of Business*. New York: Crown Business, 2000.

Petroski, Henry. *Evolution of Useful Things*, reprint ed. New York: Vintage Books, 1994.

Petzinger, Thomas, Jr. *The New Pioneers: The Men and Women Who Are Transforming the Workplace and Marketplace*. New York: Simon & Schuster, 1999.

Postrel, Virginia. *The Future and Its Enemies: The Growing Conflict over Creativity, Enterprise, and Progress*. New York: Touchstone, 1998.

Reinertsen, Donald G. *Managing the Design Factory: A Product Developer's Toolkit*. New York: Free Press, 1997.

Royce, Walker. *Software Project Management*. Reading, Mass.: Addison-Wesley, 1998.

Rummler, Geary A., and Alan P. Brache. *Improving Performance: How to Manage the White Space on the Organization Chart*, 2d. ed. San Francisco: Jossey-Bass Publishers, 1995.

Russell, Lou. *"The Middle Is Manic."* Cutter Consortium Business-IT Strategies Advisory Service, *E-Mail Advisor* (13 June 2001).

Schmidt, Douglas C., Hans Rohnert, and Michael Stal. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons, 2000.

Schneider, William. *The Reengineering Alternative: A Plan for Making Your Current Culture Work*. Burr Ridge, Ill.: Irwin Professional Publishing, 1994.

Schrage, Michael. *No More Teams: Mastering the Dynamics of Creative Collaboration*. New York: Currency Doubleday, 1989.

Schwaber, Ken. *"Controlled Chaos: Living on the Edge." American Programmer* 9, no. 5 (April 1996): 10–16.

———. *"The Agile Alliance Revolution."* Cutter Consortium e-Project Management Advisory Service, *Executive Update* 2, no. 8 (May 2001).

Schwaber, Ken, and Mike Beedle. *Agile Software Development with Scrum*. Upper Saddle River, N.J.: Prentice Hall, 2002.

Senge, Peter. *The Fifth Discipline: The Art & Practice of The Learning Organization*. New York: Currency Doubleday, 1990.

Sobek, Durward K., II, Allen C. Ward, and Jeffrey K. Liker. *"Toyota's Principles of Set-Based Concurrent Engineering." Sloan Management Review* (Winter 1999).

Swainson, Bill, ed. *Encarta Book of Quotations*. New York: St. Martin's Press, 2000.

Stapleton, Jennifer. *DSDM, Dynamic Systems Development Method: The Method in Practice*. Harlow, Eng.: Addison-Wesley, 1997.

Takeuchi, Hirotaka, and Ikujiro Nonaka. *"The New New Product Development Game." Harvard Business Review* (January–February 1986): 137–46.

Tapscott, Don, Alex Lowy, and David Ticoll (eds.). *Blueprint to the Digital Economy: Creating Wealth in the Era of E-Business*. New York: McGraw-Hill, 1998.

Vijayan, Jaikumar, and Gary H. Anthes. *"Lessons from India Inc." Computerworld*, 2 April 2001.

Waldrop, M. Mitchell. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York: Simon & Schuster, 1992.

Weinberg, Gerald M. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

———. *The Psychology of Computer Programming*, silver anniversary ed. New York: Dorset House, 1998.

Wenger, Etienne, *Communities of Practice: Learning, Meaning, and Identity*. Cambridge, Eng.: Cambridge University Press, 198.

Wheelwright, Steven C., and Kim B. Clark. *Revolutionizing Product Development: Quantum Leaps in Speed, Efficiency, and Quality*. New York: Free Press, 1992.

Wiegers, Karl E. *Software Requirements*. Redmond, Wash.: Microsoft Press, 1999.

Williams, Laurie, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. *"Strengthening the Case for Pair Programming." IEEE Software* 17, no. 4 (July/August 2000): 19–25.

Williams, Laurie, and Robert Kessler. *Pair Programming Illustrated*. Manuscript in preparation.

Womack, James P., Daniel T. Jones, and Daniel Roos. *The Machine That Changed the World: The Story of Lean Production*. New York: HarperPerennial, 1990.

Womack, James P., and Daniel T. Jones. *Lean Thinking: Banish Waste and Create Wealth in Your Corporation*. New York: Simon & Schuster, 1996.