Agile Development with ICONIX Process

People, Process, and Pragmatism

DOUG ROSENBERG, MATT STEPHENS, AND MARK COLLINS-COPE

Apress®

Lead Editor: Jim Sumser
Technical Reviewer: Dr. Charles Suscheck
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Assistant Publisher: Grace Wong
Project Manager: Tracy Brown Collins
Copy Manager: Nicole LeClerc
Production Manager: Kari Brooks-Copony
Production Editor: Beth Christmas
Compositor: Diana Van Winkle, Van Winkle Design Group
Proofreader: Elizabeth Berry
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Interior Designer: Diana Van Winkle, Van Winkle Design Group
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data

Rosenberg, Doug.
Agile development with ICONIX process : people, process, and pragmatism / Doug Rosenberg, Matt Stephens, Mark Collins-Cope.
p. cm.
Includes index.
ISBN 1-59059-464-9
1. Computer software--Development. I. Stephens, Matt. II. Collins-Cope, Mark. III. Title.

QA76.76.D47R666 2005
005.1--dc22
2005000163

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The latest information on this book can be found at www.softwarereality.com/design/agileiconix.jsp.

To Irv, and in loving memory of Vivian
—Doug Rosenberg

For Michelle and Alanah
—Matt Stephens

To Oliver, Nathan, and Lewis
—Mark Collins-Cope

Contents at a Glance

Contents

About the Authors

DOUG ROSENBERG is the founder and president of ICONIX Software Engineering, Inc. (`www.iconixsw.com`). Doug spent the first 15 years of his career writing code for a living before moving on to managing programmers, developing software design tools, and teaching object-oriented analysis and design.

Doug has been providing system development tools and training for nearly two decades, with particular emphasis on object-oriented methods. He developed a unified Booch/Rumbaugh/Jacobson design method in 1993 that preceded Rational's UML by several years. He has produced more than a dozen multimedia tutorials on object technology, including "COMPREHENSIVE COM" and "Enterprise Architect for Power Users," and is the co-author of *Use Case Driven Object Modeling with UML* (Addison-Wesley, 1999) and *Applying Use Case Driven Object Modeling with UML* (Addison-Wesley, 2001), both with Kendall Scott, and *Extreme Programming Refactored: The Case Against XP* (Apress, 2003), with Matt Stephens.

A few years ago, Doug started a second business, an online travel website (`www.VResorts.com`) that features his virtual reality photography and some innovative mapping software, which you can read about in this book.

MATT STEPHENS is a senior architect, programmer, and project leader based in Central London. He has led a number of agile projects through successive customer releases. He's also spoken at several software conferences, and he regularly writes on software development, having written for magazines including *Dr. Dobb's Journal*, *Software Development* magazine, *Application Development Advisor*, and *Application Development Trends*. His key interests include software agility, architecture, and interaction design. Check out his latest thoughts at `www.softwarereality.com`.

Matt co-authored *Extreme Programming Refactored: The Case Against XP* (Apress, 2003) with Doug Rosenberg. In fact, Matt and Doug are collaborating on yet another book, *Use Case Driven Object Modeling: Theory and Practice* (Addison-Wesley, 2005).

MARK COLLINS-COPE is technical director of Ratio Group Ltd., a UK-based company undertaking development, training, consultancy, and recruitment in the object and component technology arena (see `www.ratio.co.uk`). Collins-Cope has undertaken many roles in his 20 years in the software development industry, including analysis, design, architecture definition/technical lead; project manager; lecturer; and writer. His key interests include use-case analysis, software architecture, and component-based development and software process.

About the Technical Reviewer

DR. CHARLES SUSCHECK is an assistant professor of computer information systems at Colorado State University, Pueblo campus. He specializes in software development methodologies and project management, and has over 20 years of professional experience in information technology. Dr. Suscheck has held positions of process architect, director of research, principal consultant, and professional trainer at some of the most recognized companies in America. He has spoken at national and international conferences on topics related to project management.

Acknowledgments

Our thanks go to:

All the folks from ESRI, including Dave Lewis, Amir Soheili, and Jim McKinney for getting the mapplet built; Jack Dangermond for approving the project; and the ESRI marketing team (Debra Van Gorden, Paige Spee, and friends) for providing us with technical information on ArcGIS Server.

The fabulous Apress production team, including the "PM," Tracy Brown Collins; "the world's greatest copy editor," Nicole LeClerc; our production editor, Beth Christmas; our editor, Jim Sumser; and, of course, "Mr. Apress," Gary Cornell.

Our technical reviewer, Chuck Suscheck.

Our "unofficial" reviewer, David Putman of `www.exoftware.com`, for his extensive feedback on all the chapters (in particular the "agile core subset" and TDD chapters).

Geoff Sparks and Dermot O'Bryan from Sparx Systems.

Andy Carmichael for the material on three-point estimation.

Dino Fancellu and Robin Sharp from `www.javelinsoft.com`.

Introduction
Rigor Without the Mortis

Many people (especially agilists) associate a high-ceremony software development process with a dead project (i.e., rigor mortis), and this association is not entirely incorrect. Our approach aims to put back the rigor while leaving out the mortis—that is, we can do rigorous analysis and design without killing the project with an excessively high-ceremony approach. The goal of this book is to describe that process in full detail.

Agility *in theory* is about moving ahead at speed, making lots of tiny course corrections as you go. The theory (and it's a good one) is that if you spend months or years producing dry specifications at the start of the project and then "set them in concrete," this doesn't necessarily (and in practice, doesn't) lead to a product that meets the customer's requirements, delivered on time and with an acceptably low defect count.

It's likely that the requirements will change over time, so we need to be prepared for that, and it's likely that a lot of the original requirements will turn out to be wrong or new requirements will be discovered after the requirements "concrete" has set. Agile methods answer this problem in a number of different ways, but the overriding principle is to break things down into smaller chunks and not to go setting anything in concrete (least of all your requirements specs).

That's great in theory, but it might leave you wondering how to go about doing it in practice. There are plenty of agile books out there that set out to tell you how to "be agile" at a management or process level, but this book is different for a number of reasons. First, we look at each level of agility and show how it can be applied to a real-life project, from issues with team adoption of agile practices down to a specific logical design process.

We also believe that *being agile doesn't mean you should have to abandon up-front analysis and design* (although we do agree with agilists that *too much* up-front analysis and design can be a bad thing). So, in this book we define a core subset of UML diagrams and techniques, and show how to apply them to your own project using a core subset of established agile practices. Our goal is to show how to make your own process *minimal yet sufficient*.[1]

Another reason this book is different is because we're essentially "outsiders" in the agile world. We've been in the industry for a long time and have taken part in a number of agile projects, but we're approaching agility from the standpoint of more traditional software engineering. We examine how agility can be applied in organizations that may approach "all-out, party-on extreme agility" with a heavy dose of caution and distrust. As a result, the process we describe in this book could well be more applicable to traditional or more disciplined organizations than other agile processes.

In our attempt to find the "sweet spot" between agility and discipline, the spot we've landed on is probably far removed from where some agilists (most notably the creators of Extreme Programming [XP]) would expect it to be. For example, we place a higher emphasis on UML modeling than XP does. However, if you're an XP practitioner, you may well find that the object modeling process this book demonstrates works very well in collaborative modeling workshops (in fact, we place a high emphasis on the value of collaborative modeling).

1. This isn't dissimilar to Agile Modeling's principle of making documentation "just barely good enough," though we distinguish it because there are some important differences in our guidance, as we describe in this book.

Plan Driven and Feedback Driven

Agile ICONIX Process is plan-driven and feedback-driven, small-increment, short-iteration development, using the following artifacts:

- Use case model (including personas)
- Domain model
- Release plan

Then for each release, we aim to produce the following:

- Robustness and sequence diagrams for each use case
- An updated class diagram
- Source code
- Unit and acceptance tests as appropriate

Using this process, we tie the persona modeling[2] into the iteration plan and drive the feature set, via the iteration plan, from the usage requirements. (This is similar in some ways to Feature-Driven Development [FDD], which we discuss in Chapter 4.)

Recurring Themes

Throughout this book, you'll encounter the themes described in the sections that follow.

People, Process, and Pragmatism

The ICONIX approach to software agility starts with a subtle but significant refactoring of the Agile Manifesto's value of "individuals and interactions over processes and tools."

We have redefined this as "people *and* processes *and* pragmatism." We believe that neither people nor processes are second-class citizens, and that both (in conjunction with a pragmatic approach) are equally important factors in a project's eventual success.

Real Agility vs. Illusory Agility

A software-engineering maxim that has fallen out of fashion lately is "There is no such thing as a shortcut." (Another one that seems to have been abandoned recently is "Get it right the first time.") In high-velocity projects that must be produced in "Internet time," there is always pressure to cut corners, by skimping on testing, designing, reviews, and so on. This is not the same as agility (at least not in theory, although in practice it often gets justified in the name of agility).

In fact, this approach almost invariably delays the project. We will explain how to distinguish between the elimination of practices that are weighing down the project ("excess fat") and practices that are actually there to save time. Cutting out these latter practices might appear to save some time early on, but in reality doing so stores up trouble for later.

2. See Chapter 10.

Cookbook Development

A common view held in the agile world is that cookbook approaches to software development don't work. We agree with this to an extent, because analysis and programming are massive, highly complex fields, and the number of different types of software project is roughly equal to the number of software projects. However, we firmly believe that the core "logical" analysis and design process can (and in fact should) be a specific sequence of repeatable steps. These steps aren't set in stone (i.e., they can be tailored), but it helps to have them there. In a world of doubt and uncertainty, it's nice to have a clearly defined, definitive sequence of "how-to" steps to refer back to.

Way back in the pre-UML days when Doug first started teaching a unified Booch/Rumbaugh/Jacobson modeling approach (around 1992/93), one of his early training clients encouraged him to "write a cookbook, because my people like following cookbook approaches." While many have claimed that it's impossible to codify object-oriented analysis and design (OOAD) practices into a simple, repeatable set of steps (and it probably isn't possible in its entirety), ICONIX Process probably comes as close as anything out there to a cookbook approach to OOAD.

While there's still room for significant flexibility within the approach (e.g., adding in more state or activity diagrams), ICONIX Process lays down a simple, minimal set of steps that generally lead to pretty good results that have proven to be consistent and repeatable over the last 12 years.

So, Let's Get Started

There are three major parts in this book. In Part 1, we examine what agility is and isn't, explore the characteristics of a good software process, introduce ICONIX Process and its core UML subset and, in Chapter 4, introduce a core subset of agile practices.

In Part 2, we illustrate our core subset in action by exploring the design and code of an example project, a C#/.NET mapping application for the travel industry, which we call "the mapplet." The mapplet example represents "the authors practicing what they preach," as this is the real-life story of a commercial application for a website that's owned by one of them. So, if you've ever wondered what one of those #%$%& methodologists would really do if they needed to get something built, and if they'd follow the methodology they preach, you'll find this example interesting.

Part 3 presents some extensions to ICONIX Process related to persona-driven analysis and test-driven development. This part complements the material presented in the first two parts, and it also contains some last-minute innovations in the area of use case–driven testing that we hope you find as interesting as we do.

PART 1

ICONIX and Agility

The first part of this book lays the groundwork for the discussions and examples that follow. We begin in Chapter 1 by trying to define what *agility* is and what it isn't. In Chapter 2 we take a generic look at software processes in general, with some specific focus on finding the "sweet spot" between agility and discipline, and on the human-centric and communication issues that are strongly stressed in agile processes. In Chapter 3 we summarize ICONIX Process (as defined in Doug's first two books), and in Chapter 4 we attempt to define a minimalist core-subset of agile practices, which we illustrate by example in Part 2.

CHAPTER 1

What Is Agility? (And Why Does It Matter?)

"What is agility?" is a simple question with a complex answer. The answer is complex because there is no central authority to dictate the "agile standard" and define in legally binding terms what agility is or isn't.[1] While many (including us) would argue that this is a good thing, the result is that not everyone will ever agree on a single definition. To some, agility is all about the people: emphasizing people over tools. To others, agility is all about the ability to change requirements late in a project (in fact, this is what most people mean when they talk about "being agile"). To the industry thought-leaders, agility is a combination of these things. But in true agile spirit, the very definition of agility evolves and warps over time.

To be honest, we're glad that there isn't a central standards authority for software agility, handing down the "agile spec" from on high and stipulating that all agile processes must conform to its ideals to be allowed to call themselves AGILE™, because that would stifle the exciting air of innovation and exploration of new development techniques that have ignited the industry over the past few years.

Of course, with the excitement of new ideas and new frontiers, there's always the danger that we will disappear down a blind alley or two in our search for better ways of doing things. There's also a danger that previously learned lessons will be forgotten in the rush to discover new improvements. These lessons include the benefits of doing an up-front design and exploring requirements in sufficient detail before coding. We discussed the dangers of forgetting these important lessons in our previous book, *Extreme Programming Refactored: The Case Against XP* (Apress, 2003).

The pendulum has swung from one extreme to the other. It's gone from the inflexible, inefficient, old-guard, waterfall, high-ceremony methodologies to the highly fluid, trendy agile processes that we're seeing today.[2] As you might expect, the pendulum has already begun to swing back the other way, and it will hopefully come to rest somewhere between the two extremes.

We're hoping that this book will contribute toward finding the ideal middle ground. This book tells you how to create software using a small increment, short iteration, feedback-driven strategy while still modeling up-front and using that up-front thinking to avoid lots of rework (i.e., it has the benefits of agility but not the penalties of skipping up-front design).

ICONIX Process is a use case–driven analysis and design methodology. Its main focus is on how to get reliably from use cases to code in as few steps as possible. In this book, we describe ICONIX Process and show how it was applied to a real-life project. We also describe in detail how to apply ICONIX Process to a broader agile project environment. This combination of process and practices is shown in Figure 1-1. Informally, we refer to this combined process as *Agile ICONIX*.

1. Of course, there is the Agile Alliance (see `www.agilealliance.org`), although this isn't quite the same sort of thing. We discuss the Agile Alliance later in this chapter and in Chapter 4.
2. If we think of a big, high-ceremony process as elephantine, and an extremely minimal/agile one as mouselike, it's amusing to note that some folks try to equate the mouse to the elephant because both are gray mammals with tails. In reality, however, there are significant differences between a mouse and an elephant, and neither of those animals does the work of a horse very well.

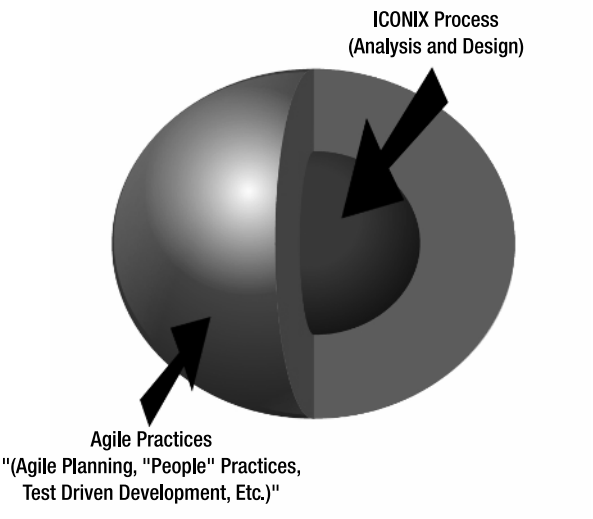

Figure 1-1. *Agile ICONIX in a nutshell*

This book is all about how to be agile and accommodate the real-world situations of changing requirements, new technology baselines, and so on, without skipping analysis and design, using a small increment, short iteration, model-driven *and feedback-driven* approach. This book also teaches how to drive the development from UML models, and then shows how to adjust both the models and the code using an interleaved approach in a way that the model and code become more tightly linked over time.

Although much of ICONIX Process can be considered agile, some parts of it stand in stark contrast to recent agile thinking. In particular, agile methods often tell us not to bother keeping design documentation and source code synchronized. The theory is that once code has been written, we don't need the diagrams anymore.[3] ICONIX Process, on the other hand, suggests exactly the opposite: the more tightly synchronized the design documentation is with the code, the faster, more maintainable, and more accurate (i.e., closer to the customer's requirements) your project will be over successive releases.

Luckily, ICONIX Process also aims to make this process easier (again in contrast to other agile processes). To achieve this tight synchronization between diagrams and code, we need to cut down on the number of diagrams that we have to draw (and therefore maintain). It's also important to know which diagrams are going to be important to ongoing iterations and which can safely be discarded. "Minimal but sufficient" analysis and design is right at the core of ICONIX Process.

Of course, there's an ever-increasing number of development processes claiming to be agile. Project leaders may decide that the project they're embarking on will be agile, or their existing project may not be working out too well, so they decide to introduce some agile practices. But what exactly does "agile" mean? What's the yardstick by which agility is measured? Where's the magic point of latitude when a project flips from being nonagile to being agile?

In this chapter, we examine the somewhat nebulous term "agility" and attempt to disambiguate it. In doing so, we aim to answer the questions in the previous paragraph.

3. This mind-set comes from thinking of diagrams as documentation rather than as tools that enable the design process and efficient communication about the design across the team. In our estimation, this is 100% backward. Modeling should be about doing design and communicating the design to the rest of the team, *not* about documentation.

What Software Agility Isn't

To better explain what software agility is, it's worth beginning with a description of what it isn't. Software agility isn't about

- Big up-front requirements gathering and design
- Hacking
- High ceremony (i.e., lots of paperwork and bureaucratic hoops to jump through)
- Low levels of communication
- "Carrying" mediocre developers
- Politics and secrets

Let's take a quick look at each of these topics in turn.

Big Up-front Requirements Gathering and Design

Up-front requirements gathering and design are, of course, vital parts of any software process, and they have their place in an agile project. What agile practitioners are generally opposed to is the "pure waterfall" concept of eliciting all requirements and modeling the architecture in detail for the entire project before starting on individual sections. Instead, you gather together everything you think you'll need just for the next iteration, and proceed with that. This approach is similar to the iterative and incremental process that has been developed in the Rational Unified Process (RUP).

Hacking

Adopting an agile process isn't (or shouldn't be) a license to hack. Agile processes are typically high-discipline, and it often takes hard work and patience for developers to get the full benefit.

High Ceremony

Sometimes the organization requires a high level of paperwork (e.g., change requests to be signed off on by three levels of management). If your organization doesn't, then why bury yourself out of choice?

Low Levels of Communication

As with any process, agile processes benefit from teams that tell each other what's going on, and that both talk and listen to the customer. Agile processes are very much opposed to "low-communication" environments—for example, environments that separate programmers into isolated cubicles.

"Carrying" Mediocre Developers

Agile processes typically require high-caliber staff if they are to work at all. There is no place for slouchers and "bug jockeys" on such projects!

Politics and Secrets

Agility places a big emphasis on increased transparency—that is, allowing the customer to see exactly what's going on in the project at any time. If a project is behind schedule, the agile planning methods generally make it difficult to hide this from the customer. Although this might seem alarming at first, it has the very important benefit that you don't find yourself storing up nasty surprises for later—in particular, on the day before the delivery deadline, having to tell the customer that it's going to be at least another 6 months before any working software can be delivered.

Increased transparency also extends to individuals and teamwork. Because agile teams tend to work more closely together, it's difficult for one team member to operate deviously (e.g., hanging on the coattails of more talented programmers and taking the credit for their work).

The Goals of Agility

When deciding whether to adopt agile practices for your project, it's important to take a step back from the hype and ask what's in it for you (or, more accurately, for your project). So, what does your project stand to gain from adopting agile practices? What are the benefits, the goals, the problems that agility sets out to solve? And do these problems exist on your project?

The Agile Manifesto website (see `www.agilemanifesto.org`) describes the principles and values that agile teams need to follow to describe themselves as "agile." But oddly, it doesn't describe the actual goals of software agility—that is, why you would want to make your project agile in the first place (although some goals are described by the group's 12 principles, listed at `www.agilemanifesto.org/principles.html`).

We've interpreted the goals of agility as the ability to

- Respond to changing requirements in a robust and timely manner.
- Improve the design and architecture of a project without massively impacting its schedule.
- Give customers exactly what they want from a project for the dollars they have to invest.
- Do all this without burning out your staff to get the job done.

These goals are sometimes overlooked in the rush to address the values. The agile values are important but, we feel, are really there to fulfill these goals.

The overriding agile goal is responsiveness to changing requirements. This was the main problem out of which agile processes grew. Offset against responsiveness is the need for the project to be robust. Usually, *robustness* means contingency—that is, having safety nets in place to mitigate risk and provide a backup plan if (when) things do go wrong. Robustness implies more effort; more practices to put in place that, while reducing risk, can slow down the project's overall pace (see Figure 1-2). (Sometimes, though, robustness can also mean simply using smarter practices to reduce risk).

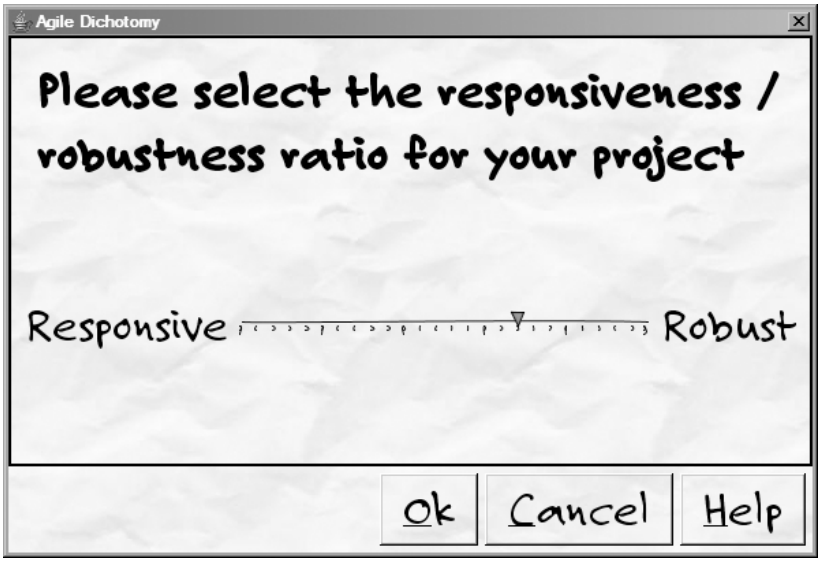

Figure 1-2. *The agile software dichotomy: responsiveness versus robustness*

Responsiveness in this context means the ability to react quickly (without regard for keeping all your safety nets in place, which is where robustness comes in). If you react to a change quickly but immediately introduce lots of defects as a result, that's very responsive but not particularly robust! Safety nets include things like unit tests, customer acceptance tests, design documents, defect tracking software, requirements traceability matrices, and so on.

So, ironically perhaps, to be really responsive (and to not introduce lots of defects every time the code changes), you need to push the slider further toward robustness. But then, the more safety nets you have in place, the more difficult it becomes to react quickly (i.e., the project becomes less agile).

Agility, then, is the ability to adapt to change in a timely and economic manner, or the ability to change direction while moving at speed. So a leopard is agile; an elephant isn't. But an elephant can carry a lot more than a leopard. But a horse is a lot more agile than an elephant and can carry a lot more than a leopard. ICONIX Process, without the agile extensions we present in this book, can be thought of as a "workhorse" process. By adding in a core subset of agile practices, we're aiming to create a "thoroughbred racehorse" process.

In other words, agile techniques work best for small projects but don't always scale very well. The techniques we describe in this book scale better than those in some agile processes because they place a greater emphasis on up-front design and documentation.

SCRIBBLED ON THE BACK OF A NAPKIN

The screenshot in Figure 1-2 looks like it was drawn on a crumpled-up piece of paper, although it is actually from a "real" program. It uses a readily available Swing look and feel called Napkin Look & Feel (see `http://napkinlaf.sourceforge.net`; this page also includes a link to a Java WebStart demo).

The idea behind this look and feel is that sometimes, when a developer shows a prototype GUI to a manager or customer, the customer assumes that what he's seeing is working software. (Appearances *are* everything, after all. . .)

On the other hand, if the working prototype was presented looking like a user interface mockup that had been scrawled on the back of a napkin, then the customer would be more likely to see it for what it actually is: a slightly working but mostly nonfunctional prototype that was cobbled together quickly so as to give the customer a rough idea of what the finished product will look like.

Seems like a great idea to us!

Why Is Agility Important?

Agile development has virtually exploded onto the software development world, bringing with it a shift in the way that software is developed. Agility makes a lot of sense because it addresses some of the common reasons why projects have failed over the years.

Back in 1995, the Standish Group's CHAOS Report[4] (a commonly cited report focusing on reasons for project successes and failures) showed that the primary reason for project failure was "lack of user input." The same study also found that the top element that dramatically increases the chance of success is "user involvement." So it isn't surprising that user involvement (and, in particular, user feedback, as early in the process as possible) is a hugely important driver behind agility.

Another commonly recognized contributor toward project failure is the "big monolithic project" syndrome, where a project doesn't get to see the light of day for a year or two. Finally, when this monstrosity rolls out of the programmers' workshop, the customer grows purple-faced and demands to know what this has to do with his original specification. It's quite likely that what the customer thought he was specifying was completely different from the way in which the programmers interpreted the spec, or the requirements might simply have changed beyond recognition in those 2 years, and somebody forgot to tell the programmers. It sounds absurd, but it happens—a lot.

Agile development addresses these issues: it cuts down the amount of time before the customer sees working software, and it encourages increased communication among everyone involved in the project (the programmers, the analysts, the customer, the users, the managers, the tea lady, and so on). If business customers are concerned

4. See `www.standishgroup.com/press/article.php?id=2`.

about trusting their millions of dollars to your IT consultancy, you need a pretty good story to convince them that you will deliver what they need, when they need it.

In an ailing software industry in which it is still unacceptably common for projects to fail, the most visible (and probably the most often cited) reason for failure is unstable requirements. In other words, requirements that were thought to be "set in stone" are liable to change midproject, causing code to be hacked and the most insidious and time-consuming bugs to creep into the product. Often the requirements change so much, and the code becomes so hacked about as a result, that many of the bugs are not discovered until the software is released. Projects that follow heavyweight (or *high-ceremony*) methodologies may also suffer as a result, due to the amount of documentation that must be kept up to date (doubling the amount of work to be done), sent back and forth, reviewed, and signed off on to cover each and every change. Such projects face the danger of collapsing beneath their own weight.

Another problem faced by many different project teams is that the design will warp and evolve over the course of a single project. In other words, the design that you began with will likely not be the same as the finished design; rather, it will change as the team's understanding of the problem improves. The extent of this problem varies from project to project, depending on a number of factors such as how well understood the problem is, whether a similar project has been written before by anyone on the team, the skill level of the people involved, the quality of the design process and supporting environment and, sometimes, simply whether or not there is a prevailing wind to send the project on its way.

Given this situation, it was inevitable that the industry would begin looking for ways to accommodate changing requirements and evolving designs more easily. A number of agile processes have grown out of this need, each taking a slightly different approach. We describe some of these agile processes in the next section.

What Makes a Project Agile?

A variety of practices, principles, values, and other factors go into making a project agile. In this section, we describe the more important ones (we also sum up these practices as a "top 10" list at the end of this chapter).

Tuning the Process As You Go Along

A doomed project is one that fails to adapt its development process as it goes along. If a particular practice obviously isn't working, it should be replaced with a practice that does work for your team.

Keeping It Low Ceremony

Keeping a process *low ceremony* is the art of producing just enough documentation to proceed, and having just enough process hoops to jump through to provide some structure and coordination, without slowing down the team. For example, if a team is unable to start programming because another 100 pages of design documentation need to be written (using the extensive company template with 12 pages of "front matter," of course), then that would classify as a high-ceremony project.

ICONIX Process aims to keep the project low ceremony, by identifying a minimum set of UML diagrams that can be produced to get reliably from use cases to code with as few hoops to jump through as possible.

Enhancing Agility Through Good Design

If sufficient time is spent on getting the architecture and design right near the start of the project, then it becomes a lot easier to extend and modify the design later on (when changes are traditionally more expensive).

As we describe later in this chapter, Robert C. Martin identifies the need for *continuous care*—that is, constant attention to the design and the state of the source code. If the design begins to fray at the edges (e.g., because some functionality is being added that the design wasn't originally intended to cope with), then time needs to be spent readjusting the design and tidying up the code through refactoring. The alternative—just shoehorning the new functionality in (also known as hacking)—means that the problem never goes away, and over time it just gets worse.

However, a vital extra component can also help to keep the design extensible: time spent on up-front design. Contrary to popular belief, designing for change doesn't mean adding layers of indirection to your code. Unless there's a really good reason for it, this sort of design ultimately just produces more code, making it more difficult to modify later.

Instead, designing for change (also known as *defensive programming*) means following sound, object-oriented (OO) design principles, the following in particular:

- *Keep the code in a highly modular state*. Make sure each class has just one responsibility, and that it handles that responsibility well. Also, make sure each method performs just one task (e.g., by delegating to other methods).
- *Keep your classes highly cohesive and loosely coupled*. Make sure everything in each class fits well and is there to fulfill the reason for the class's existence. Try to reduce the number of dependencies between classes.
- *Don't overcomment your code*. Any code that you have control over should be made self-explanatory (see the next point), reducing the need for comments. The exception is with calls to library code that you have no control over; often, some deftly placed comments can help a lot here.
- *Use method names that describe the method's purpose, not how the method achieves that purpose*. For example, instead of calling a method `addToList(String str)`, call it `addToCatalog(String productID)`.
- *Use class names that are closer to the problem domain than the implementation details*. We'll explore this in more detail when we talk about domain models in Chapter 3.
- *Don't overcommit*. That is, don't use a concrete type where a more abstract type would suffice. For example, if you've declared a return type as a `java.util.ArrayList`, you might find it's better to declare the return type as a `Collection` interface instead. And speaking of interfaces . . .
- *Use interfaces as return types and parameter types wherever possible*.

Improving Communication and Teamwork

One of the most important factors separating agile development methods from traditional development methods is the focus on people. For example, Agile Modeling promotes communication and teamwork through the following practices:

- Active stakeholder participation
- Modeling with others
- Displaying models publicly
- Collective ownership

You build systems for your stakeholders. It is their requirements that you need to fulfill, they are the source of critical information, and they need to make important decisions such as the setting of priorities. For iterative and incremental development to succeed, your stakeholders must be available to provide information, to work with you to explore their requirements, and to make decisions in a timely manner. In other words, your stakeholders must be actively involved with your project.

Software development is a lot like swimming: it is very dangerous to do it alone. Whenever you work on something by yourself, you risk going in an inappropriate direction and not realizing you're doing so. Furthermore, you risk making a common mistake due to lack of experience with or knowledge of a technique or technology. This is why it's so important for design work to be collaborative: by modeling with one or more people, the quality of your work is likely to be much greater, you're likely to get the job done faster, and the information gained during the modeling effort is communicated to everyone involved.

BUT WHAT ABOUT RESPECTING YOUR CO-WORKERS?

Agile processes do place a greater emphasis on the people factor, but it's debatable whether this is intrinsically what makes a project agile. Respecting your co-workers is equally important on nonagile projects, for example. If people feel valued, they'll be more productive, and more willing to participate and contribute actively to the project.

Reducing Exposure to the Forces of Change

The premise behind this idea is simple: making changes halfway through an iteration is expensive and can cause bugs as the new requirements get shoehorned into an unsuspecting design. So to reduce the likelihood of requirements being introduced halfway through an iteration, keep the iterations short. This is like a boxer creating a smaller target by turning side-on to his opponent.

We can also reduce the likelihood of requirements changing at the most inopportune moment by getting from use cases to code in as short a time frame as possible. We do this by spending a little extra time focusing on the requirements and the design modeling (when a change in the requirements is welcomed), so that we cut a straight line to the source code, developing short iterations in short spaces of time.

Measuring Progress with Working Software

If working software has been delivered to the customer, then we know we've made at least some progress (even delivering a small amount of working functionality to the customer is a lot further than some projects get). Working software can also be used to measure (in precise terms) how near to completion the project is (see Chapter 9 for more on agile planning).

But how do we know that the software is working? "Working" doesn't just mean "doesn't crash"; it means that the software matches up with the customer's requirements—that is, it does what it's meant to do. Testing the software against the requirements (often referred to as *customer acceptance testing* or *functional testing*) is generally how we verify that the software works.

During analysis and design, you should constantly ask yourself, "How am I going to test this?" If you can't test something, then you should seriously consider whether you should be building it. Considering testability while modeling is important because it keeps the team focused on realistic designs. Furthermore, it gives the team a head start on writing the test cases, as some thought has already been put into them. Model reviews, described in Chapter 3, are also a good option for validating both the requirements and the design.

Agile Project Management

As with improving communication and teamwork, the people factor plays a large part in agile project management. In many ways, this is no different from the "traditional" project management role: it's about managing groups of people and keeping them at their optimum efficiency without burning them out.

In practical terms, this might simply involve seating people near appropriate colleagues so that they're more inclined to communicate well, moving individuals across teams so that the correct skill sets are applied to individual problems, and so on.

Agile Planning

The very term "agile planning" might strike some people as oxymoronic. How can you create a plan for something that by its very nature keeps changing? In fact, the more volatile a project's requirements, the more agile planning is needed.

Agile planning operates at two levels:

- *Adaptive planning*: This involves planning ahead, tracking changes, and adapting the plan as the project progresses. *Predictive planning*, on the other hand, involves making predictions such as "On June 14, we'll start doing module A, and it will take 4 days for Eric to write it." Adaptive planning stands in stark contrast to predictive planning, because adaptive planning involves synchronizing the plan with reality as we go along.[5]
- *Planning* for *agility*: This entails preparing for the unknown so that when changes take place, their impact is kept to a minimum. Agile planning is a way of anticipating and controlling change.

Typically, agile planning consists of various practices, including performing short, fixed-length iterations (usually of 1 to 2 weeks); regularly reviewing the project plan; time boxing (i.e., specifying a fixed amount of time

5. See Martin Fowler's paper "The New Methodology" at `www.thoughtworks.com/us/library/newMethodology.pdf`.

in which to deliver a variable amount of functionality); and tracking velocity (i.e., the rate at which new working software is being delivered).

Managing Change

While some agile methodologies tell us to "embrace change," we prefer to see change as something that should be actively managed—controlled, even. This doesn't mean *discouraging* change (that wouldn't be very agile, after all), but instead keeping a pragmatic focus on the costs involved in making a change to the requirements at a late stage in the project.

If the customer is made aware that change isn't free, and that the cost to fix a requirements defect increases exponentially the longer it's left, she might prefer to take the hit and leave a requirement as is, at least until the next release. For the customer to make an objective decision, she needs accurate planning data, including cost estimates. So a big part of managing change is tracking *project velocity* (see Chapter 9).

Delivering the System That the Customer Wants at the End of the Project, Not What He Thought He Wanted at the Start

This goal is probably the most intrinsically *agile* of the agile factors listed here. The customer might, in all good faith, specify a set of requirements at the start of the project, believing them to be correct. During the project, however, things change. The business changes, causing the goalposts to shift; our understanding (and the customer's understanding) of the problem domain grows; our understanding of the required solution increases; and the design evolves, meaning a better solution is sometimes found.

Agility is all about recognizing that these things happen in just about every software project and, rather than trying to suppress them (e.g., by "freezing" requirements with another 8 months to go before the deadline), we use change to our advantage to drive the project toward the working system that is the most optimal for the user's needs.

We can use a variety of practices to achieve this goal. In fact, pretty much all of the agile practices contribute to this goal in some way.

Challenges of Being Agile

Often, introducing agility into your project can involve a trade-off. For example, certain levels of documentation and modeling may be scrapped so that the team can "travel light." Striking the right balance can be difficult.

Here, in our view, are the top 10 challenges of being agile.

10. Sticking to the Path

Many agile processes, though low ceremony, are high discipline and require consistent effort from the whole team to keep the project on track. You may well find that you start to do certain practices less (e.g., if you're using pair programming, team members might start to pair up less). Agile processes put various safety nets in place to lessen the impact when you fall. (Of course, some processes have more safety nets than others, as we discuss in *Extreme Programming Refactored: The Case Against XP*.)

If you're finding it increasingly difficult to maintain certain practices, it's time to review the process. Perhaps a particular practice is too difficult because it simply doesn't fit your project. There may be an equivalent practice (or group of practices)—something easier—that you could do instead and that your team is more likely to stick to.

9. Keeping Releases Short

Short releases (i.e., releasing working code to the customer on a rapid turnaround) can be difficult to keep on doing, for a number of reasons. Examples include interface changes, database changes, user retraining, and help file rewriting.

Similarly, it's tempting to stop doing short releases simply because they're so difficult. Funnily enough, the answer in many cases is to release more often (a similar solution to difficulties associated with continuous integration; see the section titled "Continuous Integration" later in this chapter for more information).