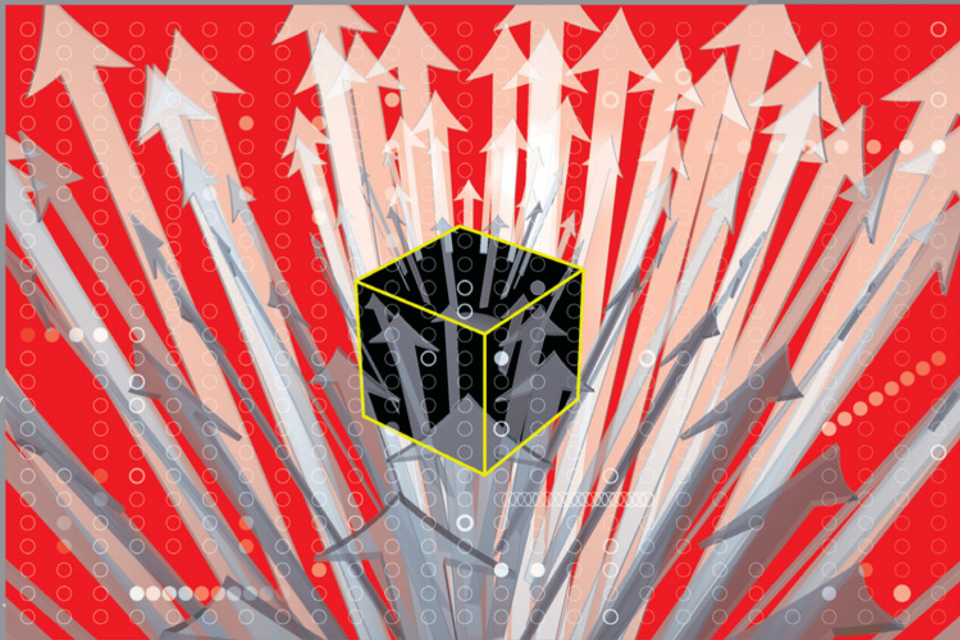
Microsoft

BEST PRACTICES

MORE ABOUT SOFTWARE REQUIREMENTS



Thorny Issues and Practical Advice

Karl E. Wiegers
Two-time winner of the *Software Development* Productivity Award

Praise for

More About Software Requirements: Thorny Issues and Practical Advice

"A must-have—Weigers goes well beyond aphorisms with practical insights for everyone involved in the requirements process. This book is an experience-based, insightful discussion of what the software requirements expert ought to know to get better at his or her job. It's the book you should read after whatever book you read to get an introduction to the topic of software requirements.

There's a whole lot of reality in this relatively small volume. There is a section on 'Managing Scope Creep,' and another on 'The Fuzzy Line Between Requirements and Design,' that are worth the price of the book just because of the light they shine on some particularly difficult issues. I like this book. Its author speaks with experience and authority, in a readable way, with lots of nice illuminating anecdotes."

—Robert L. Glass, Publisher/Editor, *The Software Practitioner*; Visiting Professor, Griffith University, Australia

"In an easy-to-read format, *More About Software Requirements* offers practical answers to the most vexing requirements problems faced by analysts and project managers. An essential companion to Wiegers's *Software Requirements*, it cuts to the chase with real-world wisdom for practitioners. Use this book to find the bottom-line advice you need to succeed with your requirements efforts."

—Ellen Gottesdiener, EBG Consulting; Author, *Requirements by Collaboration* and *Software Requirements Memory Jogger*

"Karl has done it again, he has captured the difficult requirement questions, provided sensible solutions, and shown us the related pitfalls.

If you do anything with software requirements you will find answers to your difficult questions clearly addressed and solution pros and cons plainly delineated."

—Ivy Hooks, Compliance Automation, Inc.

"Sage advice from a requirements master. And, fortunately, Wiegers is a great writer, too. Professionals who take requirements seriously—and we all should take requirements seriously—must read this book."

—Norm Kerth, Principal, Elite Systems

"If you have requirements problems (and who doesn't), this book will help. *More About Software Requirements* is filled with pragmatic down-to-earth advice. Wiegers helps the entire project team—project managers, developers, testers, and writers, not just the requirements analysts—understand what's happening with their requirements and how to fix those problems. Make this book required reading for your whole project team."

—Johanna Rothman, President of Rothman Consulting Group, Inc.; Coauthor, *Behind Closed Doors: Secrets of Great Management*; Author, *Hiring the Best Knowledge Workers, Techies & Nerds: The Secrets and Science of Hiring Technical People*

"Karl Wiegers has added to the treasure trove of advice in *Software Requirements, Second Edition*, by addressing some of the trickiest and most controversial issues in requirements engineering. I get asked questions in the areas covered in *More About Software Requirements* all the time.

The advice is sound; the writing is clear. This is a very useful, practical book."

—Erik Simmons, Requirements Engineering Program Lead, Corporate Quality Network, Intel Corporation

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2006 by Karl Wiegers

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number 2005936071

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWE 0 9 8 7 6 5 4 3

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, Excel, and Microsoft Press are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Ben Ryan
Project Editor: Devon Musgrave
Copy Editor: Michelle Goodman
Indexer: Brenda Miller

Body Part No. X11-66734

Contents at a Glance

Part I On Essential Requirements Concepts

1 Requirements Engineering Overview . . . 3
2 Cosmic Truths About Software Requirements . . . 11

Part II On the Management View of Requirements

3 The Business Value of Better Requirements . . . 21
4 How Long Do Requirements Take? . . . 29
5 Estimating Based on Requirements . . . 33

Part III On Customer Interactions

6 The Myth of the On-Site Customer . . . 51
7 An Inquiry, Not an Inquisition . . . 57
8 Two Eyes Aren't Enough . . . 69

Part IV On Use Cases

9 Use Cases and Scenarios and Stories, Oh My! . . . 77
10 Actors and Users . . . 85
11 When Use Cases Aren't Enough . . . 89

Part V On Writing Requirements

12 Bridging Documents . . . 103
13 How Much Detail Do You Need? . . . 105
14 To Duplicate or Not to Duplicate . . . 113
15 Elements of Requirements Style . . . 117
16 The Fuzzy Line Between Requirements and Design . . . 129

Part VI On the Requirements Process

17 Defining Project Scope . . . 137
18 The Line in the Sand . . . 147
19 The Six Blind Men and the Requirements . . . 153

Part VII On Managing Requirements

20 Handling Requirements for Multiple Releases 165
21 Business Requirements and Business Rules 171
22 Measuring Requirements 177
23 Exploiting Requirements Management Tools 183

Table of Contents

Preface xiii

Part I On Essential Requirements Concepts

1 Requirements Engineering Overview 3

"Requirement" Defined 3
Different Types of Requirements 4
Business Requirements 5
User Requirements 5
Functional Requirements 6
System Requirements 6
Business Rules 7
Quality Attributes 7
External Interfaces 7
Constraints 7
Requirements Engineering Activities 7
Looking Ahead 9

2 Cosmic Truths About Software Requirements 11

Requirements Realities 11
Requirements Stakeholders 14
Requirements Specifications 16

Part II On the Management View of Requirements

3 The Business Value of Better Requirements 21

Tell Me Where It Hurts 21
What Can Better Requirements Do for You? 23
The Investment 25
The Return 26
An Economic Argument 28

What do you think of this book?
We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: *www.microsoft.com/learning/booksurvey/*

4 How Long Do Requirements Take? 29
Industry Benchmarks 29
Your Own Experience 30
Incremental Approaches 31
Planning Elicitation 32

5 Estimating Based on Requirements 33
Some Estimation Fundamentals 33
Estimation Approaches 35
Goals Aren't Estimates 37
Estimating from Requirements 37
Measuring Software Size 38
Story Points 40
Use Case Points 41
Testable Requirements 45
The Reality of Estimation 47

Part III On Customer Interactions

6 The Myth of the On-Site Customer 51
User Classes and Product Champions 51
Surrogate Users 53
Now Hear This 54

7 An Inquiry, Not an Inquisition 57
But First, Some Questions to Avoid 57
Questions for Eliciting Business Requirements 59
User Requirements and Use Cases 60
Questions for Eliciting User Requirements 62
Open-Ended Questions 65
Why Ask Why? 66

8 Two Eyes Aren't Enough 69
Improving Your Requirements Reviews 69

Part IV On Use Cases

9 Use Cases and Scenarios and Stories, Oh My! . . . 77

Use Cases . . . 77
Scenarios . . . 79
User Stories . . . 82

10 Actors and Users. . . . 85

11 When Use Cases Aren't Enough . . . 89

The Power of Use Cases . . . 89
Project Type Limitations . . . 90
Event-Response Tables . . . 91
Use Cases Don't Replace Functional Requirements . . . 93
Use Cases Reveal Functional Requirements . . . 96

Part V On Writing Requirements

12 Bridging Documents . . . 103

13 How Much Detail Do You Need? . . . 105

Who Makes the Call? . . . 105
When More Detail Is Needed . . . 107
When Less Detail Is Appropriate . . . 108
Implied Requirements . . . 110
Sample Levels of Requirements Detail . . . 110

14 To Duplicate or Not to Duplicate . . . 113

Cross-Referencing . . . 114
Hyperlinks . . . 114
Traceability Links . . . 115
Recommendation . . . 116

15 Elements of Requirements Style . . . 117

I Shall Call This a Requirement . . . 117
System Perspective or User Perspective? . . . 119
Parent and Child Requirements . . . 120
What Was That Again? . . . 121
  Complex Logic . . . 121
  Negative Requirements . . . 122

Omissions 124
Boundaries 125
Avoiding Ambiguous Wording 126

16 The Fuzzy Line Between Requirements and Design 129
Solution Ideas and Design Constraints 130
Solution Clues 131

Part VI On the Requirements Process

17 Defining Project Scope 137
Vision and Scope 137
Context Diagram 138
Use Case Diagram 140
Feature Levels 141
Managing Scope Creep 143

18 The Line in the Sand 147
The Requirements Baseline 148
When to Baseline 149

19 The Six Blind Men and the Requirements 153
Limitations of Natural Language 154
Some Alternative Requirements Views 154
Why Create Multiple Views? 157
Selecting Appropriate Views 159
Reconciling Multiple Views 161

Part VII On Managing Requirements

20 Handling Requirements for Multiple Releases 165
Single Requirements Specification 166
Multiple Requirements Specifications 167
Requirements Management Tools 168

21 Business Requirements and Business Rules 171
Business Requirements 171
Business Rules 172
Business Rules and Software Requirements 173

22 **Measuring Requirements** 177

Product Size. 177

Requirements Quality 178

Requirements Status. 179

Requests for Changes. 180

Effort. 181

23 **Exploiting Requirements Management Tools** 183

Write Good Requirements First 183

Expect a Culture Change. 184

Choose a Database-Centric or Document-Centric Tool 184

Don’t Create Too Many Requirement Types or Attributes 185

Train the Tool Users 186

Assign Responsibilities. 186

Take Advantage of Tool Features 186

References 189

Index 195

What do you think of this book?
We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: *www.microsoft.com/learning/booksurvey/*

Preface

Requirements engineering continues to be a hot topic in the software industry. More and more development organizations realize that they cannot succeed unless they get the software requirements right. Too often, the people responsible for leading the requirements process are ill-equipped for this challenging role. They do the best they can, but it's an uphill climb without adequate training, coaching, resources, and experience.

As a consultant, trainer, and author, I receive many questions from practitioners about how to handle difficult requirements issues. Certain questions come up over and over again. Alas, there aren't simple answers for many of these. Many books on requirements engineering have been published during the last several years, including my own, *Software Requirements, Second Edition* (Wiegers 2003a). These books provide solid guidance on the challenges of requirements elicitation, analysis, specification, validation, and management. However, additional requirements topics are not covered well by the existing books. Also, some books contain guidance that I believe is ill-founded.

This book addresses some of these recurrent questions that puzzle and frustrate requirements analysts, such as the following:

- "How do I keep too much design from being embedded in the requirements?" (I heard this question again the day before I wrote these words.)
- "When should I baseline my requirements?"
- "How can I convince my managers that we need to do a better job on our project requirements?"
- "What are some good questions to ask in requirements interviews?"
- "Are use cases all I need for documenting the requirements?"
- "We can't get our customers to review the requirements specification. What should I do?"
- "What are some good metrics our organization should collect about our requirements?"
- "We're collecting requirements for multiple releases concurrently. How should I store those?"
- "How can I use requirements to estimate how long it will take to finish the project?"
- "How can I write better requirements?"

I've addressed some other topics in this book simply because little has been written about them. For instance, everyone talks about project scope, but the current books on requirements engineering say little about how to actually define scope. See Chapter 17, "Defining Project Scope," for some recommendations. Still other topics are included because I don't see

practitioners using some of the established techniques that can help them do a better job. As an example, nearly all requirements specifications I see consist entirely of written text—there's not a picture to be found. However, the skilled analyst should have a rich tool kit of techniques available for representing requirements information. Text is fine in many cases, but other sorts of requirements "views" sometimes are more valuable. Chapter 19, "The Six Blind Men and the Requirements," addresses this topic.

The suggestions I propose in this book augment the "good practices" approach I took in my earlier book. Many cross-references are provided to chapters in *Software Requirements, Second Edition*, marked with the icon shown to the left of this paragraph. As with all such advice, you need to think about how best to apply these suggestions to your specific situation. Organizations are different, projects are different, and cultures are different, so techniques that work in one situation might not be just right for another. To illustrate the application of these practices, I've included many examples of actual project experiences, marked with the "true stories" icon shown to the left here.

Anyone involved with defining or understanding the requirements for a new or enhanced software product will find this book useful. The primary audience consists of those team members who perform the role of the requirements analyst on a software development project, be this their full-time job or just something they do once in awhile. Part II of this book, "On the Management View of Requirements," is focused on aspects of requirements engineering that are of particular interest to project managers and senior managers. Customer representatives who work with the software team will also find certain chapters valuable, particularly those in Part III, "On Customer Interactions," and Part IV, "On Use Cases."

I should point out that all the practices I recommend assume that you're dealing with reasonable people. Sometimes an unreasonable customer will insist on a specific solution that isn't a good fit for the problem. Unreasonable funding sponsors might impose their own inappropriate preferences, overriding the thoughtful decisions made by actual user representatives. Senior managers or influential customers sometimes demand impossible delivery dates for an overly constrained project. If you face such a situation, try educating the difficult people to help them understand the risks posed by the approaches they are demanding and the value of using a better approach. People who appear unreasonable often are just uninformed. Sometimes, though, they truly are unreasonable. I can't help you much with that.

You may download the templates and other process assets described in this book from the Process Impact Web site, *http://www.processimpact.com*. Feel free to share your experiences with me at kwiegers@acm.org.

I hope you'll find this book a valuable supplement to your other resources for software requirements engineering. But don't just read the chapters and say, "That's interesting." Set yourself a personal goal of finding at least three new practices that you want to try the next few times you wear your analyst hat.

Acknowledgments

First, I thank the many people in my seminars who have asked some of these challenging questions over the past several years, as well as the readers who have sent me e-mails with their own thorny requirements issues. I'm grateful for the review input provided by Wayne Allen, Michael Beshears, Steven Davis, Chris Fahlbusch, Lynda Fleming, Betty Luedke, Jeannine McConnell, Terry Nooyen-Coyner, Debbie Shyne, David Standerford, Donna Swaim, and Robin Tucker. The many comments I received from reviewers Ellen Gottesdiener, Andre Gous, and Shannon Jackson were especially valuable. A special thanks goes to Erik Simmons, who provided incisive suggestions on every chapter and greatly helped me sharpen the message. Thanks also to the Microsoft Learning editorial and production teams, including acquisitions editor Ben Ryan, copy editor Michelle Goodman, proofreaders Becka McKay and Sandi Resnick, and artist Joel Panchot. I especially enjoyed the opportunity to work with project editor Devon Musgrave again.

And finally, a big thank-you once again to my ever-cheerful wife, Chris Zambito. Fooled you this time, hon!

Chapter 2

Cosmic Truths About Software Requirements

In this chapter:

Requirements Realities 11
Requirements Stakeholders 14
Requirements Specifications 16

As every consultant knows, the correct answer to nearly any question regarding software is, "It depends." This isn't just a consultant's cop-out–it's true. The best advice for how to proceed in a given situation depends on the nature of the project, its constraints, the culture of the organization and team, the business environment, and other factors. But having worked with many organizations, I've made some observations about software requirements that really do seem to be universally applicable. This chapter presents some of these "cosmic truths" and their implications for the practicing requirements analyst.

Requirements Realities

Cosmic Truth #1: If you don't get the requirements right, it doesn't matter how well you execute the rest of the project.

Requirements are the foundation for all the project work that follows. I don't mean the initial SRS you come up with early in the project, but rather the full set of requirements knowledge that is developed incrementally during the course of the project.

The purpose of a software development project is to build a product that provides value to a particular set of customers. Requirements development attempts to determine the mix of product capabilities and characteristics that will best deliver this customer value. This understanding evolves over time as customers provide feedback on the early work and refine their expectations and needs. If this set of expectations isn't adequately explored and crafted into a set of product features and attributes, the chance of satisfying customer needs is slim.

As mentioned in the previous chapter, requirements validation is one of the vital subcomponents of requirements development, along with elicitation, analysis, and specification. Validation involves demonstrating that the specified requirements will meet customer needs. One useful technique for validating requirements is to work with suitable customer representatives

to develop *user acceptance criteria*. These criteria define how customers determine whether they're willing to pay for the product or to begin using it to do their work. User acceptance criteria typically stipulate that the product allows the users to properly perform their most significant tasks, handles the common error conditions, and satisfies the users' quality expectations. User acceptance criteria aren't a substitute for thorough system testing. They do, however, provide a necessary perspective to determine whether the requirements are indeed right.

Cosmic Truth #2: Requirements development is a discovery and invention process, not just a collection process.

People often talk about "gathering requirements." This phrase suggests that the requirements are just lying around waiting to be picked like flowers or to be sucked out of the users' brains by the analyst. I prefer the term *requirements elicitation* to *requirements gathering*. Elicitation includes some discovery and some invention, as well as recording those bits of requirements information that customer representatives and subject matter experts offer to the analyst. Elicitation demands iteration. The participants in an elicitation discussion won't think of everything they'll need up front, and their thinking will change as the project continues. Requirements development is an exploratory activity.

The analyst is not simply a scribe who records what customers say. The analyst is an investigator who asks questions that stimulate the customers' thinking, seeking to uncover hidden information and generate new ideas. (See Chapter 7, "An Inquiry, Not an Inquisition.") It's fine for an analyst to propose requirements that might meet customer needs, provided that customers agree that those requirements add value before they go into the product (Robertson 2002). An analyst might ask a customer, "Would it be helpful if the system could do <whatever idea he has>?" The customer might reply, "No, that wouldn't do much for us." Or the customer might reply, "You could do that? Wow, that would be great! We didn't even think to ask for that feature, but if you could build it in, it would save our users a lot of time." This creativity is part of the value that the analyst adds to the requirements conversation. Just be careful that analysts and developers don't attempt to define a product from the bottom up through suggested product features, rather than basing the requirements on an understanding of stakeholder goals and a broad definition of success.

Cosmic Truth #3: Change happens.

It's inevitable that requirements will change. Business needs evolve, new users or markets are identified, business rules and government regulations are revised, and operating environments change over time. In addition, the business need becomes clearer as the key stakeholders become better educated about what their true needs are.

The objective of a change control process is not to inhibit change. Rather, the objective is to *manage* change to ensure that the project incorporates the right changes for the right reasons. You need to anticipate and accommodate changes to produce the minimum disruption and

cost to the project and its stakeholders. However, excessive churning of the requirements after they've been agreed upon suggests that elicitation was incomplete or ineffective—or that agreement was premature. (See Chapter 18, "The Line in the Sand.")

To help make change happen, establish a change control process. You can download a sample from my Web site, *http://www.processimpact.com/goodies.shtml*. When I helped to implement a change control process in an Internet development group at Eastman Kodak Company, the team members properly viewed it as a structure, not as a barrier (Wiegers 1999). The group found this process invaluable for dealing with its mammoth backlog of change requests.

Every project team also needs to determine who will be evaluating requested changes and making decisions to approve or reject them. This group is typically called the change (or configuration) control board, or CCB. A CCB should write a charter that defines its composition, scope of authority, operating procedures, and decision-making process. A template for such a charter is available from *http://www.processimpact.com/goodies.shtml*.

Nearly every software project becomes larger than originally anticipated, so expect your requirements to grow over time. According to consultant Capers Jones (2000), requirements growth typically averages 1 to 3 percent per month during design and coding. This can have a significant impact on a long-term project. To accommodate some expected growth, build contingency buffers—also known as management reserve—into your project schedules (Wiegers 2002b). These buffers will keep your commitments from being thrown into disarray with the first change that comes along.

I once spoke with a manager on a five-year project regarding requirements growth. I pointed out that, at an average growth rate of 2 percent per month, his project was likely to be more than double the originally estimated size by the end of the planned 60-month schedule. The manager agreed that this was a possibility. When I asked if his plans anticipated this growth potential, he gave the answer I expected: No. I'm highly skeptical that this project will be completed without enormous cost and schedule overruns.

When you know that requirements are uncertain and likely to change, use an incremental or iterative development life cycle. Don't attempt to get all the requirements "right" up front and freeze them. Instead, specify and *baseline* the first set of requirements based on what is known at the time. A baseline is a statement about the state of the requirements at a specific point in time, such as "We believe that these requirements will meet customer needs and are a suitable foundation for proceeding with design and construction." Then implement that fraction of the product, get some customer feedback, and move on to the next slice of functionality. This is the intent behind agile development methodologies, the spiral model, iterative prototyping, evolutionary delivery, and other incremental approaches to software development.

Finally, recognize that change always has a price. Even the act of reviewing a proposed change and rejecting it consumes time. Software people need to educate their project stakeholders so they understand that, sure, we can make that change you just requested, and here's what it's going to cost. Then the stakeholders can make appropriate business decisions about which desired changes should be incorporated and at what time.

Requirements Stakeholders

Cosmic Truth #4: The interests of all the project stakeholders intersect in the requirements process.

Consultant Tim Lister once defined project success as "meeting the set of all requirements and constraints held as expectations by key stakeholders." A *stakeholder* is an individual or group who is actively involved in the project, who is affected by the project, or who can influence its outcome. Figure 2-1 identifies some typical software project stakeholder groups. Certain stakeholders are internal to the project team, such as the project manager, developers, testers, and requirements analysts. Others are external, including customers who select, specify, or fund products; users who employ the systems; compliance certifiers; auditors; and marketing, manufacturing, sales, and support groups. The requirements analyst has a central communication role, being responsible for interacting with all these stakeholders. Further, the analyst is responsible for seeing that the system being defined will be fit for use by all stakeholders, perhaps working with a system architect to achieve this goal.

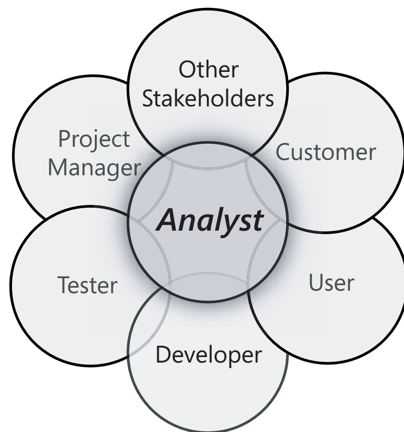


Figure 2-1 Some typical software project stakeholders.

At the beginning of your project, identify your key stakeholder groups and determine which individuals will represent the interests of each group. You can count on stakeholders having conflicting interests that must be reconciled. They can't all have veto power over each other. You need to identify early on the decision makers who will resolve these conflicts, and these decision makers must determine what their decision-making process will be. As my colleague Christian Fahlbusch, a seasoned project manager, points out, "I have found that there is usually one primary decision maker on a project, oftentimes the key sponsor within the organization. I don't rest until I have identified that person, and then I make sure he is always aware of the project's progress."

Cosmic Truth #5: Customer involvement is the most critical contributor to software quality.

Various studies have confirmed that inadequate customer involvement is a leading cause of the failure of software projects. Customers often claim they can't spend time working on requirements. However, customers who aren't happy because the delivered product missed the mark always find plenty of time to point out the problems. The development team is going to get the customer input it needs eventually. It's a lot cheaper—and a lot less painful—to get that input early on, rather than after the project is ostensibly done.

Customer involvement requires more than a workshop or two early in the project. Ongoing engagement by suitably empowered and enthusiastic customer representatives is a critical success factor for software development. Following are some good practices for engaging customers in requirements development:

- **Identify user classes.** Customers are a subset of stakeholders, and users are a subset of customers. You can further subdivide your user community into multiple *user classes* that have largely distinct needs (Gause and Lawrence 1999). Unrepresented user classes are likely to be disappointed with the project outcome.
- **Select product champions.** You need to determine who will be the literal voice of the customer for each user class. I call these people *product champions*. Ideally, product champions are actual users who represent their user-class peers. See Chapter 6, "The Myth of the On-Site Customer," for more about product champions.

- **Build prototypes.** Prototypes provide opportunities for user representatives to interact with a simulation or portion of the ultimate system. (See Chapter 13 of *Software Requirements, Second Edition*.) Prototypes are far more tangible than written requirements specifications. However, prototypes aren't a substitute for documenting the detailed requirements.
- **Agree on customer rights and responsibilities.** People who must work together rarely discuss the nature of their collaboration. The analyst should negotiate with the customer representatives early in the project to agree on the responsibilities each party has with respect to the requirements process. An agreed-upon collaboration strategy is a strong contributor to the participants' mutual success. See Chapter 2 of *Software Requirements, Second Edition* for some suggestions of customer rights and responsibilities in the requirements process.

Cosmic Truth #6: The customer is not always right, but the customer always has a point.

It's popular in some circles to do whatever any customer demands, claiming "The customer is always right." Of course, the customer is *not* always right! Sometimes customers are in a bad mood, uninformed, or unreasonable. If you receive conflicting input from multiple customers, which one of those customers is "always right"?

The customer may not always be right, but the analyst needs to understand and respect whatever point each customer is trying to make through his request for certain product features or attributes. The analyst needs to be alert for situations in which the customer could be in the wrong. Rather than simply promising anything a customer requests, strive to understand the rationale behind the customer's thinking and negotiate an acceptable outcome. Following are some examples of situations in which a customer might not be right:

- Presenting solutions in the guise of requirements.
- Failing to prioritize requirements or expecting the loudest voice to get top priority.
- Not communicating business rules and other constraints, or trying to get around them.
- Expecting a new software system to drive business-process changes.
- Not supplying appropriate representative users to participate in requirements elicitation.
- Failing to make decisions when analysts or developers need issues resolved.
- Not accepting the need for tradeoffs in both functional and nonfunctional requirements.
- Demanding impossible commitments.
- Not accepting the cost of change.

Requirements Specifications

Cosmic Truth #7: The first question an analyst should ask about a proposed new requirement is, "Is this requirement in scope?"

Anyone who's been in the software business for long has worked on a project that has suffered from scope creep. It is normal and often beneficial for requirements to grow over the course of a project. Scope creep, though, refers to the uncontrolled and continuous increase in requirements that makes it impossible to deliver a product on schedule.

To control scope creep, you need to have the project stakeholders agree on a scope definition, a boundary between the desired capabilities that lie within the scope for a given product release and those that do not. (See Chapter 17, "Defining Project Scope," for some scope-definition techniques.) Then, whenever some stakeholder proposes a new functional requirement, feature, or use case, the analyst can ask, "Is this in scope?" To help answer this question, some project teams write their scope definition on a large piece of cardstock, laminate it, and bring it to their requirements elicitation discussions.

If a specific requirement is deemed out of scope one week, in scope the next, then out of scope again later, the project's scope boundary is not clearly defined. And that's an open invitation to scope creep.

Cosmic Truth #8: Even the best requirements document cannot—and should not—replace human dialogue.

Even the best requirements specification won't contain every bit of information the developers and testers need to do their jobs. There will always be tacit knowledge that the stakeholders assume (rightly or wrongly) that other participants already know, along with the explicit knowledge that must be documented in the SRS. Analysts and developers will always need to talk with knowledgeable users and subject matter experts to refine details, clarify ambiguities, and fill in the blanks. This is the rationale behind having some key customers, such as product champions, work intimately with the analysts and developers throughout the project. The person performing the role of requirements analyst (even if this is one of the developers) should coordinate these discussions to make sure that all the participants reach the same understanding so that the pieces all fit together properly. A written specification is still valuable and necessary, though. A documented record of what stakeholders agreed to at a point in time is more reliable than human memory.

You need more detail in the requirements specifications if you aren't going to have opportunities for frequent conversations with user representatives and other decision makers. (See Chapter 13, "How Much Detail Do You Need?") A good example of this is when you're outsourcing the implementation of a requirements specification that your team created. Expect to spend considerable time on review cycles to clarify and agree on what the requirements mean. Also expect delays in getting questions answered and decisions made, which can slow down the entire project. This very issue was a major contributing factor in a lawsuit I know of between a software package vendor and a customer (Wiegers 2003b). The vendor allowed no time in the schedule for review following some requirements elicitation workshops, planning instead to begin construction immediately. Months later, many key requirements issues had not yet been resolved and the actual project status didn't remotely resemble the project plan.

Cosmic Truth #9: The requirements might be vague, but the product will be specific.

Specifying requirements precisely is hard! You're inventing something new, and no one is exactly sure what the product should be and do. People sometimes are comfortable with vague requirements. Customers might like them because it means they can redefine those requirements later on to mean whatever they want them to mean at any given moment. Developers sometimes favor vague requirements because they allow the developers to build whatever they want to build. This is all great fun, but it doesn't lead to high-quality software.

Ultimately, you are building only one product, and someone needs to decide just what that product will be. If customers and analysts don't make the decisions, the developers will be forced to. This is a sign that the key stakeholders are abdicating their responsibility to make requirements-level decisions, leaving those decisions to people who know far less about the problem.

Don't use uncertainty as an excuse for lack of precision. Acknowledge the uncertainty and find ways to address it, such as through prototyping. A valuable adjunct to simply specifying each requirement is to define *fit criteria* that a user or tester could employ to judge whether the requirement was implemented correctly and as intended (Robertson and Robertson 1999). Attempting to write such fit criteria will quickly reveal whether a requirement is stated precisely enough to be verifiable.

Cosmic Truth #10: You're never going to have perfect requirements.

Requirements are never finished or complete. There is no way to know for certain that you haven't overlooked some requirement, and there will always be some requirements that the analyst won't feel it is necessary to record. Rather than declaring the requirements "done" at some point, define a baseline. (See Chapter 18.) Once you've established a baseline, follow your change control process to modify the requirements, recognizing the implications of making changes. It's folly to think you can freeze the requirements and allow no changes after some initial elicitation activities.

Striving for perfection can lead to analysis paralysis. Analysis paralysis, in turn, can have a backlash effect. Stakeholders who have been burned once by a project that got mired in requirements issues are reluctant to invest in requirements development on their next project.

You don't succeed in business by writing a perfect SRS. From a pragmatic perspective, requirements development strives for requirements that are *good enough* to allow the team to proceed with design, construction, and testing at an acceptable level of risk. The risk is the threat of having to do expensive and unnecessary rework. Have team members who will need to base their own work on the requirements review them to judge whether they provide a suitably solid foundation for that subsequent work. Keep this practical goal of "good enough" in mind as you pursue your quest for quality requirements.

Chapter 11

When Use Cases Aren't Enough

In this chapter:

The Power of Use Cases 89
Project Type Limitations 90
Event-Response Tables 91
Use Cases Don't Replace Functional Requirements 93
Use Cases Reveal Functional Requirements 96

Use cases are recognized as a powerful technique for exploring user requirements. The great benefit they provide is to bring a user-centric and usage-centered perspective to requirements elicitation discussions. The analyst employs use cases to understand what the user is trying to accomplish and how he envisions his interactions with the product leading to the intended user value. Putting the user at the center is much better than focusing on product features, menus, screens, and functions that characterize traditional requirements discussions. And the structure that use cases provide is far superior to the nearly worthless technique of asking users, "What do you want?" or "What are your requirements?"

As with most new software development techniques, use cases have acquired a bit of a mystique, some misconceptions, overblown hype, and polarized camps of enthusiasts who will all try to teach you the One True Use Case Way. In this chapter, I share my perspectives on when use cases work well, when they don't, and what to do when use cases aren't a sufficient solution to the requirements problem.

The Power of Use Cases

I'm a strong believer in the use case approach. Use cases are an excellent way to structure the dialogue with users about the goals they need to accomplish with the help of the system. Users can relate to and review use cases because the analyst writes them from the user's point of view, describing aspects of the user's business. In my experience, once they get past the discomfort of trying a new technique, users readily accept the use case method as a way to explore requirements.

I'm often asked how to write requirements specifications so that users can read and understand them and also so that they contain all the detail that developers need. In many cases, one requirements representation won't meet both of these objectives. Users can comprehend use cases, but they might balk at reviewing a more detailed functional requirements specification. Use cases give developers an overall understanding of the system's behavior

that fragments of individual functionality cannot. However, developers usually need considerably more information than use cases provide so that they know just what to build. In many circumstances, the combination of employing use cases to represent user requirements and a software requirements specification to contain functional and nonfunctional requirements meets both sets of needs.

Project Type Limitations

My experience has shown that use cases are an effective technique for many, but not all, types of projects. Use cases focus on the user's interactions with the system to achieve a valuable outcome. Therefore, use cases work great for interactive end-user applications, including Web sites. They're also useful for kiosks and other types of devices with which users interact.

However, use cases are less valuable for projects involving data warehouses, batch processes, hardware products with embedded control software, and computationally intensive applications. In these sorts of systems, the deep complexity doesn't lie in the user-system interactions. It might be worthwhile to identify use cases for such a product, but use case analysis will fall short as a technique for defining all the system's behavior.

I once worked on a computational program that modeled the behavior of a multi-stage photographic system. This software used a Monte Carlo statistical simulation to perform many complex calculations and it presented the results graphically to the user. The user-system dialog needed to set up each simulation run was quite simple. (I know this because I built the user interface.) The complexity resided behind the scenes, in the computational algorithms used and the reporting of results. Use cases aren't very helpful for eliciting the requirements for these aspects of a system.

Use cases have limitations for systems that involve complex business rules to make decisions or perform calculations. Consider an airline flight reservation system, one of the classic examples used to illustrate use cases. Use cases are a fine technique for exploring the interactions between the traveler and the reservation system to describe the intended journey and the parameters associated with it. But when it comes to calculating the fare for a specific flight itinerary, a use case discussion won't help. Such calculations are driven by the interaction of highly complex business rules, not by how the user imagines conversing with the system.

Nor are use cases the best technique for understanding certain real-time systems that involve both hardware and software components. Think about a complex highway intersection. It includes sensors in the road to detect cars, traffic signals, buttons pedestrians can press to cross the street, pedestrian walk signals, and so forth. Use cases don't make much sense for specifying a system of this nature. Here are some possible use cases for a highway intersection:

- A driver wants to go through the intersection.
- A driver wants to turn left when coming from a particular direction.
- A pedestrian wants to cross one of the streets.

These approximate use cases, but they aren't very illuminating. Exploring the interactions between these actors (drivers and pedestrians) and the intersection-control software system just scratches the surface of what's happening with the intersection. The use cases don't provide nearly enough information for the analyst to define the functionality for the intersection-control software.

Use cases aren't particularly helpful for specifying the requirements for batch processes or time-triggered functions, either. My local public library's information system automatically sends me an e-mail to remind me when an item I've borrowed is due back soon. This e-mail is generated by a scheduled process that checks the status of borrowed items during the night (the one I received today was sent at 1:06 AM) and sends out notifications. Some analysts regard "time" to be an actor so that they can structure this system behavior in the form of a use case. I don't find that helpful, though. If you know the system needs to perform a time-triggered function, just write the functional requirements for that function, instead of packaging it into a contrived use case.

Event-Response Tables

A more effective technique for identifying requirements for certain types of systems is to consider the external events the system must detect. Depending on the state of the system at the time a given event is detected, the system produces a particular response. Event-response tables are a convenient way to collect this information (Wiegers 2003a). Events could be signals received from sensors, time-based triggers (such as scheduled program executions), or user actions that cause the system to respond in some way. Event-response tables are related to use cases. In fact, the trigger that initiates a use case is sometimes termed a *business event*.

The highway intersection system described earlier has to deal with various events, including these:

- A sensor detects a car approaching in one of the through lanes.
- A sensor detects a car approaching in a left-turn lane.
- A pedestrian presses a button to request to cross a street.
- One of many timers counts down to zero.

Exactly what happens in response to an external event depends on the state of the system at the time it detects the event. The system might initiate a timer to prepare to change a light from green to amber and then to red. The system might activate a Walk sign for a pedestrian (if the sign currently reads Don't Walk), or change it to a flashing Don't Walk (if the sign currently says Walk), or change it to a solid Don't Walk (if it's currently flashing). The analyst needs to write the functional requirements to specify ways to detect the events and the decision logic involved in combining events with states to produce system behaviors. Table 11-1 presents a fragment of what an event-response table might look like for such a system. Each expected system behavior consists of a combination of event, system state, and response.

State-transition diagrams and statechart diagrams are other ways to represent this information at a higher level of abstraction. Use cases just aren't enormously helpful in this situation.

Table 11-1 Partial Event-Response Table for a Highway Intersection

Event	System State	Response
Road sensor detects vehicle entering left-turn lane.	Left-turn signal is red. Cross-traffic signal is green.	Start green-to-amber countdown timer for cross-traffic signal.
Green-to-amber countdown timer reaches zero.	Cross-traffic signal is green.	1. Turn cross-traffic signal amber. 2. Start amber-to-red countdown timer.
Amber-to-red countdown timer reaches zero.	Cross-traffic signal is amber.	1. Turn cross-traffic signal red. 2. Wait 1 second. 3. Turn left-turn signal green. 4. Start left-turn-signal countdown timer.
Pedestrian presses a specific walk-request button.	Pedestrian sign is solid Don't Walk. Walk-request countdown timer is not activated.	Start walk-request countdown timer.
Pedestrian presses walk-request button.	Pedestrian sign is solid Don't Walk. Walk-request countdown timer is activated.	Do nothing.
Walk-request countdown timer reaches zero plus the amber display time.	Pedestrian sign is solid Don't Walk.	Change all green traffic signals to amber.
Walk-request countdown timer reaches zero.	Pedestrian sign is solid Don't Walk.	1. Change all amber traffic signals to red. 2. Wait 1 second. 3. Set pedestrian sign to Walk. 4. Start don't-walk countdown timer.

Here's another type of project for which use cases aren't sufficient. I enjoy watching auto races, probably because I raced stock cars myself (alas, with little success) as a teenager. Several shopping malls throughout the United States have NASCAR race-car simulators. These consist of small car bodies mounted on motion-control bases. The customer is a driver in a computer-controlled simulated race. Each driver's view of the racetrack is projected on a screen in front of his car. The driver is racing against whatever other customers happen to be competing in that race as well as against several simulated drivers that the computer controls. The car body tilts and sways on its motion-control base during the race in response to the driver's actions. A synthesized voice provides information to the driver over a speaker, warning when other cars are nearby and reporting the driver's position and how many laps are left in the race. It's a blast!

Defining the requirements for this complex system of interacting hardware and software components demands more than use cases. There aren't that many use cases for the driver. He can

press the accelerator and the brake pedal, turn the steering wheel, and shift gears, but these aren't truly use cases—they're events. A great deal of the product's complexity lies not in the user interactions but under the hood (literally, in this case). An event-response approach will go much farther toward understanding the requirements for this kind of system. So although use cases are valuable for systems in which much of the complexity lies in the interactions between the user and the computer, they are not effective for some other types of products.

Use Cases Don't Replace Functional Requirements

One book about use cases states, "To sum up, all functional requirements can be captured as use cases, and many of the nonfunctional requirements can be associated with use cases" (Bittner and Spence 2003). I agree with the second part of this sentence but not with the first part. It is certainly true that use cases are a powerful technique for discovering the functional requirements for a system being developed. However, this statement suggests that use cases are the only tool needed for representing a software system's functionality.

This notion that all functional requirements can fit into a set of use cases and that use cases contain all the functional requirements for a system appears in many of the books and methodologies that deal with use cases. The thinking seems to be that the use cases *are* the functional requirements. If the analyst writes good use cases, the developers are supposed to be able to build the system from that information, along with nonfunctional requirements information that's included in a supplementary specification.[1] Nonfunctional requirements, such as performance, usability, security, and availability goals, typically relate to a specific use case or even to a particular flow within a use case.

Unfortunately, despite the thousands of students I've taught in requirements seminars over the years, I have yet to meet a single person who has found this pure use case approach to work! Perhaps some people have successfully done it, but I haven't met any of them. On the contrary, dozens of requirements analysts have told me, "We gave our use cases to the developers and they got the general idea, but the use cases just didn't contain enough information. The developers had to keep asking questions to get the additional requirements that weren't in the use cases." I suppose you could argue that they must not have been very good use cases. But when dozens of people report the same unsatisfactory experience when trying to apply a particular methodology, I question the methodology's practicality.

---

1. There's an interesting conflict in the current use case literature. Bittner and Spence (2003) provide the following definition for *supplementary requirements*: "Functional or nonfunctional requirements that are traceable to a particular use case are said to *supplement* the use case description" (their italics). However, the Unified Software Development Process, which is heavily use case driven, offers a definition of *supplementary requirement* that is directly contradictory: "A generic requirement that cannot be connected to a particular use case..." (Jacobson, Booch, and Rumbaugh 1999). It's no wonder practitioners get confused. It's generally agreed that a supplemental specification is needed to contain at the very least those nonfunctional requirements that the use cases do not describe.

There are three problems with adhering to this philosophy of use case purity. First, your use cases must contain all the functional detail that the analysts need to convey to the developers. That requires writing highly detailed use cases. The sample use cases in some books do include some complex examples. But elaborate use cases become hard to read, review, and understand.

The second problem with this approach is that it forces you to invent use cases to hold all the functional requirements because a use case is the only container you have available to describe system functionality. However, some system functionality does not fit appropriately into a use case. I have seen many new use case practitioners struggle to create inappropriate use cases to hold all the bits of functionality, to no useful end.

Logging in to an ATM or a Web site is an example that illustrates this problem. Bittner and Spence (2003) provide this good definition of use case:

> *Describes how an actor uses a system to achieve a goal and what the system does for the actor to achieve that goal. It tells the story of how the system and its actors collaborate to deliver something of value for at least one of the actors.*

By this definition, logging in to a system is not a legitimate use case because it provides no value to the person who is logging in. No one logs in to a system and feels as though he accomplished something as a result. Logging in is a means to an end, a necessary step to being able to perform use cases that do provide value. Nevertheless, the functionality to permit login and everything associated with it (such as business rules or integrity requirements regarding passwords) must be defined somewhere. If you're using only use cases to capture functional requirements, you wind up inventing artificial use cases—those that don't provide user value—just to have a place to store certain chunks of functionality. This artificiality does not add value to the requirements development process.

A third shortcoming of the use case–only philosophy is that use cases are organized in a way that makes good sense to users but not to developers. As Figure 11-1 illustrates, a use case consists of multiple small packages of information. A typical use case template contains sections for preconditions, postconditions, the normal flow of events, zero or more alternative flows (labeled with *Alt.* in Figure 11-1), zero or more exceptions (labeled with *Ex.*), possibly some business rules, and perhaps some additional special requirements.

These small packages are easy to understand and review, but they make it hard for the developer to see how the pieces fit together. As a developer, I find it more informative to see all the related requirements grouped together in a logical sequence. Suppose I read a functional requirement that implements a step in the normal flow. I want to see the requirements dealing with branch points into alternative flows and conditions that could trigger exceptions immediately following that one functional requirement. I'd like to see the requirements that handle each business rule in context, juxtaposed with the relevant system functionality.

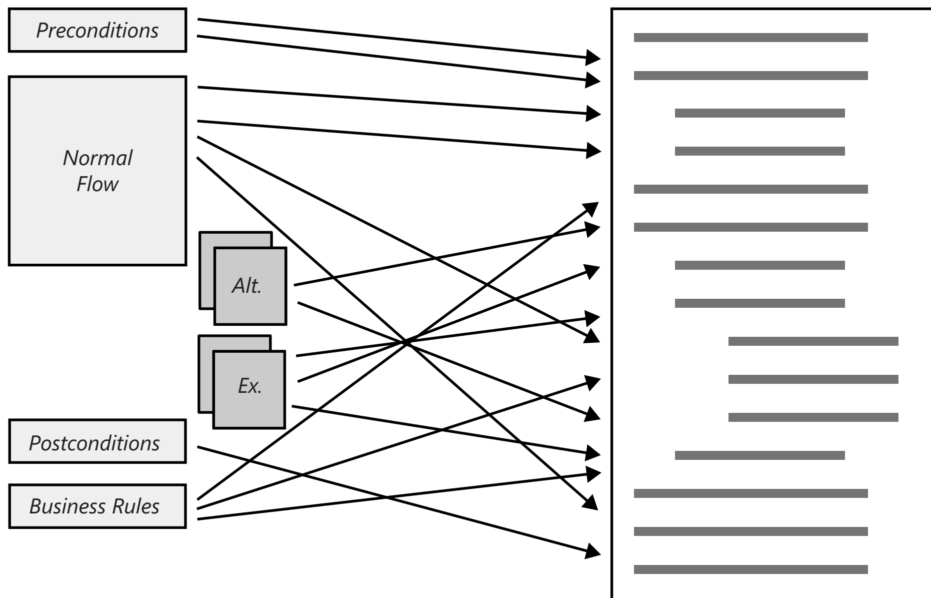


Figure 11-1 Use case organization (left) differs from SRS organization (right).

As Figure 11-1 illustrates, the functional requirements that come from the various chunks of a use case can be sprinkled throughout a hierarchically organized SRS. Traceability analysis becomes important so that you can make sure every functional requirement associated with the use case traces back to a specific part of the use case. You also want to ensure that every piece of information in the use case leads to the necessary functionality in the SRS. In short, the way a use case is organized is different from the way many developers prefer to work.

It gets even more confusing if you're employing use cases to describe the bulk of the functionality but have placed additional functional requirements that don't relate to specific use cases into a supplemental specification (Leffingwell and Widrig 2003). This approach forces the developer to get some information from the use case documentation and then to scour the supplemental specification for other relevant inputs. Before your analysts impose a particular requirements-packaging strategy on the developers, have these two groups work together to determine the most effective ways to communicate requirements information. (See Chapter 12, "Bridging Documents.")

My preference is for the analyst to create an SRS as the ultimate deliverable for the developers and testers. This SRS should contain all the known functional and nonfunctional requirements, regardless of whether they came from use cases or other sources. Functional requirements that originated in use cases should be traced back to those use cases so that readers and analysts know where they came from.

Use Cases Reveal Functional Requirements

Rather than expecting use cases to contain 100 percent of the system's functionality, I prefer to employ use cases to help the analyst discover the functional requirements. That is, the use cases become a tool rather than being an end unto themselves. Users can review the use cases to validate whether a system that implemented the use cases would meet their needs. The analyst can study each use case and derive the functional requirements the developer must implement to realize the use case in software. I like to store those functional requirements in a traditional SRS, although you could add them to the use case description if you prefer.

I'm often asked, "Which comes first: use cases or functional requirements?" The answer is use cases. Use cases represent requirements at a higher level of abstraction than do the detailed functional requirements. I like to focus initially on understanding the user's goals so that we can see how they might use the product to achieve those goals. From that information, the analyst can derive the necessary functionality that must be implemented so that the users can perform those use cases and achieve their goals.

Functional requirements—or hints about them—lurk in various parts of the use case. The remainder of this chapter describes a thought process an analyst can go through to identify the less obvious functional requirements from the elements of a use case description.

Preconditions

Preconditions state conditions that must be true before the actor can perform the use case. The system must test whether each precondition is true. However, use case descriptions rarely state what the system should do if a precondition is *not* satisfied. The analyst needs to determine how best to handle these situations.

Suppose a precondition for one use case states, "The patron's account must be set up for payroll deduction." How does the system behave if the patron attempts to perform this use case but his account is not yet set up for payroll deduction? Should the system notify the patron that he can't proceed? Or should the system perhaps give the patron the opportunity to register for payroll deduction and then proceed with the use case? Someone has to answer these questions and the SRS is the place to provide the answers.

Postconditions

Postconditions describe outcomes that are true at the successful conclusion of the use case. The steps in the normal flow naturally lead to certain postconditions that indicate the user's goal has been achieved. Other conditions, however, might not be visible to the user and therefore might not become a part of a user-centric use case description.

Consider an automated teller machine. After a cash withdrawal, the ATM software needs to update its record of the amount of cash remaining in the machine by subtracting the amount withdrawn. Perhaps if the cash remaining drops below a predetermined threshold the system

is supposed to notify someone in the bank to reload the machine with additional money. I doubt any user will ever convey this information during a discussion of user requirements, yet the developer needs to know about this functionality.

How can you best communicate this knowledge to the developers and testers? There are two options. One is to leave the use case at the higher level of abstraction that represents the user's view and have the requirements analyst derive the additional requirements through analysis. The analyst can place those requirements in an SRS that is organized to best meet the developer's needs. The second alternative is for the analyst to include those additional details directly in the use case description. That behind-the-scenes information is not part of the user's view of the system as a black box. Instead, you can think of that information as being white-box details about the internal workings of the use case that the analyst must convey to the developer.

Normal and Alternative Flows

The functionality needed to carry out the dialogue between the actor and the system is usually straightforward. Simply reiterating these steps in the form of functional requirements doesn't add knowledge, although it might help organize the information more usefully for the developer. The analyst needs to look carefully at the normal flow and alternative flows to see if there's any additional functionality that isn't explicitly stated in the use case description. For example, under what conditions should the system offer the user the option to branch down an alternative flow path? Also, does the system need to do anything to reset itself so that it's ready for the next transaction after the normal flow or an alternative flow is fully executed?

Exceptions

The analyst needs to determine how the system could detect each exception and what it should do in each case. A recovery option might exist, such as asking the user to correct an erroneous data entry. If recovery isn't possible, the system might need to restore itself to the state that existed prior to beginning the use case and log the error. The analyst needs to identify the functionality associated with such recovery and restore activities and communicate that information to the developer.

Business Rules

Many use cases are influenced by business rules. The use case description should indicate which business rules pertain. It's up to the analyst to determine exactly what functionality the developer must implement to comply with each rule or to enforce it. These derived functional requirements should be recorded somewhere (I recommend documenting them in the SRS), rather than just expecting every developer to figure out the right way to comply with the pertinent business rules.

Special Requirements and Assumptions

The use case might assume that, for instance, the product's connection to the Internet is working. But what if it's not? The developer must implement some functionality to test for this error condition and handle it in an appropriate way.

In my experience, the process of having the analyst examine a use case in this fashion to derive pertinent functional requirements adds considerable value to the requirements development process. There's always more functionality hinted at in the use case than is obvious from simply reading it. Someone needs to discern this latent functionality. I would prefer to have an analyst do it rather than a developer. If your developers are sufficiently skilled at requirements analysis they could carry out this task, but they might not view it as part of their responsibilities.

I recently spoke to a highly experienced developer who said it was much more helpful to receive requirements information organized in a structured way from the analyst than to have to figure out those details on his own. This developer preferred to rely on the analyst's greater experience with understanding the problem domain and deriving the pertinent functional requirements. Not only did this result in better requirements, but it also allowed the developer to focus his talents and energy where he added the most value—in designing and coding the software.

My philosophy of employing use cases as a tool to help me discover functional requirements means that I don't feel a need to force every bit of functionality into a use case. It also gives me the option of writing use cases at whatever level of detail is appropriate. I might write some use cases in considerable detail to elaborate all their alternative flows, exceptions, and special requirements. I could leave other use cases at a high level, containing just enough information for me to deduce the pertinent functional requirements on my own. That is, I view use cases as a means to an end. The functional requirements are that "end," regardless of where you choose to store them or whether you even write them down at all.

Deriving functional requirements from the use case always takes place on the journey from ideas to executable software. The question is simply a matter of who you want to have doing that derivation and when. (See Chapter 13, "How Much Detail Do You Need?") If you deliver only use cases without all the functional detail to developers, each developer must derive the additional functionality on his own. A developer with little requirements analysis expertise might overlook some of these requirements. It's also unlikely that all developers will record the functional requirements they identify. This makes it hard for testers to know exactly what they need to test. It also increases the chance that someone will inadvertently fail to implement certain necessary functionality. If you're outsourcing construction of the software, you can't expect the vendor's developers to accurately determine the unstated functionality from a use case description.

You will almost always have additional functional requirements that do not fit nicely into a particular use case. Earlier in this chapter, I mentioned the example of logging in to a system.

Clearly, that functionality must be implemented, but I don't consider it to be a use case. You might also have functional requirements that span multiple use cases. Consider the behavior the system should exhibit if a required connection to the Internet goes down. The Internet connection could fail while the user is executing any use case. Therefore, this error condition doesn't constitute an exception flow associated with a specific use case. It needs to be detected and handled in multiple operations. The analyst can place all the functional requirements that are not associated with or derived from a particular use case into the logically appropriate section of the SRS.

As an alternative to creating separate use case and SRS documents, you could select use cases as the organizing structure for the bulk of the functional requirements in the SRS. IEEE (Institute of Electrical and Electronics Engineers) Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications," provides guidance on how to create an SRS (IEEE 1998). According to this standard, you might organize the functional requirements by system mode, user class, objects, feature, stimulus (or external event), response, or functional hierarchy. You could also combine or nest these organizing schemes. You might have functional requirements grouped by stimulus within user class, for example. There is no universally optimal way to organize your functional requirements. Remember that the paramount objective is clear communication to all stakeholders who need to understand the requirements.

I have found use cases to be a highly valuable technique for exploring requirements on many types of projects. But I also value the structured software requirements specification as an effective place to adequately document the functional requirements. Keep in mind that these documents are simply containers for different types of requirements information. You can just as readily store use cases, functional requirements, and other types of requirements information in the database of a requirements management tool. Just don't expect use cases to replace all your other strategies for discovering and documenting software requirements.

Index

Numbers

3D function points, estimating, 39

A

A/B construct, avoiding ambiguous language and, 127
abstraction scale, 79, 129, 159
acceptance criteria, user, 11
activity diagrams, 159, 161
actors, 85–88
  interaction with use cases, 41–42, 140
  naming, 88
  primary, 77, 87
  secondary, 87
adjusted use case points, 44
adverbs, 63
  avoiding ambiguous language, 128
  cautions, 128
  use cases, 77
agile software development methodologies, 31, 51, 82
airline flight reservation kiosk project, 61
algorithms, 176
alternative flows (of use cases)
  additional functionality, 97
  vs. exceptions, 80
ambiguity, 121–128
  avoiding, 126–128
  confusing terminology, 4
  natural language, 154
analogy, 36
analysis, 8
analysis models, 138-139, 140, 155, 157, 158, 159–161
analysis paralysis, 18, 30, 161
assumptions, 67, 98
attributes (requirements types), 132, 148, 166, 168, 184–186
attributes (system properties), 4

B

baseline of project requirements, 9, 13, 18, 147–151
  checklist for when to establish, 150–151
  defined, 147
basic flow (of use cases), 80
batch processes, use case limitations and, 91
behavioral requirements, 6
better requirements, 21–28
blind men fable, 153
bookmarks, 114
boolean logic, ambiguity and, 121
bottom-up estimation, 35, 37
boundary, system, 138-–140
boundary values, ambiguity and, 125
Box, George, 4
bridging documents, 103–104
budgets, requirements baseline and, 149
buffers, contingency, 13, 35, 38, 145, 149
business case document, 137
business events, 91
business objectives, 171–172
business requirements, 5
  vs. business rules, 171–172
  elicitation questions, 59
business rules, 7, 97, 171–176
  defined, 172
  elicitation questions, 67
  requirements management tools, 115
  reuse opportunities, 175
  vs. software requirements, 173–172, 176
  use case limitations, 90
business rules catalogs, 172
business value of good requirements, 23–25

C

cafeteria ordering system project, 138–142
can, used in writing requirements, 118
case studies. *See* project case studies
case-management software system project (FBI), 22
CCB (change control board), 13
champions. *See* product champions
change control board (CCB), 13
change control process, 12, 18, 147, 149
changes, 179–181
  measuring change activity, 180
  origin of, 181
  status of change requests, 181
CHAOS Reports, 21
check-in and check-out procedures (SRS), 166
check-in and check-out version control, 166
checklists
  baseline of project requirements, 150–151
  requirements documentation reviews, 70
Chemical Tracking System project, 86
child requirements, 120, 177
class diagrams, 160
classes, 185
clients. *See* customers
code, estimating lines of, 38
commitments, 9
company policies, 172
complex boolean logic, ambiguity and, 121

complexity of use cases, 42
computations, business rules and, 176
conditions, ambiguity and, 121
cone of uncertainty, 34
configuration control board (CCB), 13
constraints, 4, 7, 130–133
Construction phase (RUP), 31
consultants, 26
consumers of bridging documents, 103
context diagrams, 138–140, 159
  vs. use case diagrams, 140
context-free questions, 65
contingency buffers, 13, 35, 38, 145, 149
cost models, 36
counts, 38, 39
creeping featurism. *See* scope creep
cross-referencing information, 114
customer is always right adage, 15
customers
  cosmic truths and, 14–16
  expectations, 11
  importance of involvement, 15
  on-site, 51–55
  project problems and failures, 22
  rights and responsibilities, 15

D

data dictionaries, 160
data field definitions, 160
data flow diagrams, 159
databases, storing requirements information in, 115, 148, 184
decision tables and decision trees, 155, 160
defects, cost and causes of, 27
design, 129–133
detail, level of when assessing requirements, 105–111
developers
  constraints, 7, 130
  ear of the developer (EOD), 55
  reactions to use cases, 89, 94
  as reviewers, 71
  story points, 40
  use cases, 93–98
  vague requirements, 17, 106
dialog, importance of, 17
dialog maps, 160
documentation, 17, 26
  cross-referencing, 114
  inspecting, 69–73, 178
  requirements baseline, 147
  requirements duplication, 113–116
  requirements reuse, 88
  reviewing, 69–73, 178
document-centric requirements management tools, 184
documents, limitations of, 113–116, 167
downloads
  CCB template, 13
  change control process sample, 13
  current requirements practice self-assessment, 25
  project documents templates, 25
  requirements documentation reviews checklist, 70
  SRS, 167
  use cases template, 78
  vision and scope documents template, 137, 172
duplicating requirements information, 113

E

ear of the developer (EOD), 55
EFactors (environmental factors), 43–44
effectiveness metric, 179
efficiency metric, 179
effort, 181
  estimation equation, 37
  story points, 40
  use case points, 44
e.g., avoiding ambiguous language and, 126
Elaboration phase (RUP), 31
elicitation, 8, 12, 57–68
  asking why, 66–68, 131
  planning, 32
entity-relationship diagrams, 160
environmental factors (EFactors), 43–44
EOD (ear of the developer), 55
e-projects, project problems and failures and, 22
estimates, 33–47
  facilitating, 23
  vs. goals, 37
  methods, 35
  realities, 47
  requirements level of detail, 108
estimating. *See* estimates
estimation equation, 37
event-response tables, 91–93, 160
events
  business rules, 97
  external, 91
  responses and, 91
exceptions, exception paths, and exception flows (of use cases), 80, 97
exclusions, 138
expectations, 11
experience, using to estimate future requirements, 30
expert opinion, 36
extensions (of use cases). *See* alternative flows (of use cases)
external entities, 138
external interfaces, 7
Extreme Programming methodology, 51, 82

F

FAA (Federal Aviation Administration) certification, requirements traceability and, 108
fable of six blind men, 153
favored user classes, 85
FBI, case-management software system project and, 22
FDA (Food and Drug Administration) certification, requirements traceability and, 108
feature creep. *See* scope creep
feature levels, 141–142
feature roadmaps, 142
features, 6, 140–141
  defined, 141
  vs. product vision and project scope, 142
featuritis. *See* scope creep
Federal Aviation Administration (FAA) certification, requirements traceability and, 108
Federal Bureau of Investigation, case-management software system project and, 22
financial business objectives, 171–172
fit criteria, 18
flow of events (of use cases), 79, 80–83
flowcharts, 159, 161
flows, 138
  context diagram, 138
  use cases, 42–43
Food and Drug Administration (FDA) certification, requirements traceability and, 108
format descriptions, 159
function points, estimating, 38
functional requirements, 3, 6
  aligning with business objectives, 172
  associated with features, 6, 141
  attributes of, 132, 148, 166, 168, 184–186
  example illustrating (home alarm system project), 105, 110–111
  measuring, 177
  object status, 160
  requirements traceability, 108
  triggered by business rules, 173–174
  vs. use cases, 41, 93–99
  user task descriptions, 161
  writing, 119–128

G

glossary of software engineering terminology, 147
goal-question-metric (GQM), 177
goals vs. estimates, 37
government regulations, 7, 172, 173, 176
GQM (goal-question-metric), 177
graphical analysis models, 138–139, 140, 155, 157, 158, 159–161

H

happy path (of use cases), 80
hierarchical requirements, 120
home alarm system project, 105, 110–111
human dialog, importance of, 17
hyperlinks, 114, 158

I

i.e., avoiding ambiguous language and, 126
IEEE (Institute of Electrical and Electronics Engineers), SRS and, 99
IEEE Standard Glossary of Software Engineering Terminology, 147
implied requirements, 67, 110
Inception phase (RUP), 31
incompleteness in requirements, 124–125
incremental approaches to requirements development, 31
industry benchmarks, 29
industry standards, 7, 172, 173
inspection of requirements documents. *See* documentation, inspecting
inspections, 69–73
Institute of Electrical and Electronics Engineers (IEEE), SRS and, 99
interface specifications, 159
interfaces, external, 7
Internet age, project problems and failures and, 22
inverse requirements, 122
iteration in requirements development, 9
iterative software development, 31, 148

K-L

Karner, Gustav, 41
laws, 172
levels, feature, 141–142
levels of requirements, 5
limitations, product, 138
limitations of requirements documents, 115, 169

M

main course (of use cases), 80
main success scenario (of use cases), 80
management tools. *See* requirements management tools
marketing requirements document, 137

mathematical expressions, 157
may, used in writing requirements, 118
measurement. *See* metrics
metaquestions, 66
metrics, 177–182
might, used in writing requirements, 118
models, analysis, 138–139, 140, 155, 157, 158, 159–161
multimedia technology, multiple requirements views and, 158
must, used in writing requirements, 118

N

naming
  actors, 88
  baselines, 147
  use cases, 61, 77, 79
NASA projects, cost and schedule overruns and, 24
natural language, 8, 159
  alternatives to, 73, 154–157
  limitations, 154
  user stories, 82
near-synonyms, avoiding ambiguous language and, 126
negative requirements, 122
nonfinancial business objectives, 171–172
nonfunctional requirements, 93, 95
  business rules as source of, 174
  requirements sizing, 178
  use case points and, 434
normal flow (of use cases), 80, 81, 97

O

omissions, 73, 157, 178
on-site customer, 51–55
open-ended questions, 65
outsourcing, 17
  requirements level of detail, 107
  use cases, 98

P

package solutions, 106, 109–110
parent requirements, 120
PBR (perspective-based reading), for requirements reviews, 72
peer deskchecks, 71
peer reviews, 69–73
perfection, 18
personas, 54, 85
perspective-based reading (PBR), for requirements reviews, 72
photographs, multiple requirements views and, 158
Planguage, 119, 161
postconditions, 96
preconditions, 96
predictions, 33
primary actor, 77, 87
primary scenario (of use cases), 80
prioritizing requirements, 23
problems attributable to poor requirements, 22–23
procedure descriptions, 159
procedures, check-in and check-out (SRS), 166
processes, developing, 25
product champions, 15, 51
  identifying user classes, 88
  implementing requirements reviews, 71
product design, 23
product features. *See* features
product releases, 147, 165–169
product requirements, 6
product size, 177
product vision, 137, 142
productivity
  estimation equation, 37
  measuring, 44, 181
  project velocity, 40
project case studies
  airline flight reservation kiosk, 61
  cafeteria ordering system, 138, 140, 141
  Chemical Tracking System, 86
  FBI case-management software system, 22
  home alarm system, 105, 110–111
project charter, 137
project scope, 137–145
  defined, 137
  exclusions, 138
  increasing, 144
  limitations, 138
  requirements baseline, 147, 148
  scope creep, 143–145
project velocity, 40
projects
  causes of problems and failures, 21–23
  estimation equation, 37
  life cycle, 31
  requirements development, time required for, 29–32
  selecting, 23
pronouns, avoiding ambiguous language and, 126
prototypes, 156
  building, 15
  user interfaces, 160

Q-R

quality attributes, 5, 7, 110, 185
  business rules as source of, 174–175
questions
  context-free, 65
  for eliciting business requirements, 59–60
  for eliciting user requirements, 62–65
  open-ended, 65–66
  to avoid asking, 57–58

rationale
 behind constraints, 130
 behind requirements, 67
 requirement attribute, 168, 185, 186
Rational Unified Process (RUP), 31
realities of software requirements, 11–13
real-time systems, 90
releases, 147, 165–169
report layouts, 159
requests for changes, 180
requirements, 3–10. *See also* functional requirements; nonfunctional requirements; software requirements specification (SRS)
 baseline for. *See* baseline of project requirements
 as basis for estimates, 37–47
 vs. business rules, 173–172, 176
 causes of project problems and failures, 21–23
 changes in, 12, 21, 22
 cosmic truths, 11–18
 duplicating in documentation, 113–116
 eliciting. *See* elicitation
 grouping into structured lists, 156
 implied, 67, 110
 importance of getting them right, 11
 incompleteness in, 124–125
 inverse, 122
 level of detail, 105–111
 measuring, 177–182
 missing, 73, 157, 178
 negative, 122
 prioritizing, 23
 quality assessments, 178
 reuse, 88
 ROI, 21–28
 special, 98
 specifying, 17
 status tracking, 179
 time required for, 29–32
 types of, 4–7
 views of, 153–161
 writing. *See* writing requirements
requirements baseline. *See* baseline of project requirements
requirements creep. *See* scope creep
requirements development, 7
 incremental approaches, 31
 time required, 29–32
requirements documents. *See* documents, limitations of; requirements management tools, software requirements specification; vision and scope document
requirements engineering, 3, 25
 activities of, 7–10
 ROI, 21
 types of requirements, 4–7
requirements information, 5–4
requirements levels, 5
requirements management, 7
requirements management tools, 26, 115, 183–187
 multiple software releases, 168–169
 requirements baseline document, 148, 168
requirements specification, 3–7
requirements traceability, 108, 156
requirements types, 185
requirements volatility, calculating, 180
resources for further reading
 context-free questions, 65
 elicitation, 57
 IEEE Standard Glossary of Software Engineering Terminology, 147
 management tools, 26
 open-ended questions, 65
 peer reviews, 69
 requirements engineering process, 3
 software estimation, 33
 testable requirements, 45
 word-usage errors, 127
 writing requirements, 117
return on investment. *See* ROI (return on investment)
reusing
 business rules, 175
 descriptions, 88
 requirements documentation, 88
reviewing requirements. *See* developers, as reviewers; documentation, reviewing; requirements specification
reviews, 69–73
rework, 27, 28
roadmaps, feature, 142
ROI (return on investment), 21–28
 expectations, 26
 measuring, 25
RUP (Rational Unified Process), 31

S

scenarios, 77, 79–82, 161
 defined, 80, 82
 executing, 81
 use cases and, 79–82
scheduling, requirements baseline and, 149
scope, project. *See* project scope
scope creep, 16, 137, 143–145
screen designs, 156
screen layouts, 160
secondary actors, 87
sequence diagrams, 161
shall, used in writing requirements, 117
should, used in writing requirements, 118
similar-sounding words, avoiding ambiguous language and, 127

six blind men fable, 153
size, 177–182
  correlating with project effort, 181
  estimation equation, 37
  measuring, 38–39
software requirements specification (SRS), 5
  ambiguity in, 121–128, 154
  baseline version of, 148–149
  bridging documents, 103–104
  check-in and check-out procedures, 166
  checklist for reviewing, 70
  IEEE recommendations for creating, 99
  limitations of, 115, 169
  master, 166
  multiple product releases and, 165–168
  perfection, 18
  product releases, 165–169
  requirements baseline document, 148
  requirements reviews, 72
  template, 167
  use cases, 95–97
software size, estimating. *See* size
solutions, focus on, 129–133
sound clips, multiple requirements views and, 158
special requirements, 98
specifications, 8, 16–18
SRS. *See* software requirements specification (SRS)
staffing, requirements baseline and, 149
stakeholders
  cosmic truths, 14–16
  defined, 14
  requirements baseline, 147
  use cases, 77
  users, 85–88
Standish Group's CHAOS Reports, 21
statechart (state-transition) diagrams, 91, 158, 159, 160
status tracking of requirements, 179
story points
  estimating, 39, 40
  requirements sizing, 177
storyboards, 160
structured lists, 156
surrogate users, 53
swimlane diagrams, 159
synonyms, avoiding ambiguous language and, 126
system boundary, 138–140
system requirements, 6

T

tables, storing similar requirements in, 156
technical complexity factors (TFactors), 43–44
technology limitations, 4
templates
  CCB charter, 13
  downloading, 25
  SRS, 167
  use cases, 78
  vision and scope documents, 137, 172
terminators, 138
terminology, as source of confusion, 4
test cases, 155, 161
testable requirements, 39, 45–47
  child requirements, 177
  requirements level of detail, 107
testing
  performing effectively, 24
  requirements level of detail, 107
TFactors (technical complexity factors), 43–44
time constraints, 37
tools
  requirements management, 26, 115, 183–187
  version control, 166
top-down estimation, 36, 37
traceability, of requirements. *See* requirements traceability
traceability links (traces), storing requirements information in, 115, 186
tracking status of requirements, 179
training, 26, 186
Transition phase (RUP), 31

U

UAW (unadjusted actor weights), 42
UCP. *See* use case points
UML (Unified Modeling Language)
  graphical analysis models, 155
  use case diagrams, 87, 140
unadjusted actor weights (UAW), 42
unadjusted use case points (UUCP), 42
unadjusted use case weights (UUCW), 42
uncertainty, cone of, 34
Unified Modeling Language (UML)
  graphical analysis models, 155
  use case diagrams, 87, 140
usage-centered perspective, 83, 89, 104
use case diagrams, 87, 140, 159
use case points, 41–45
  calculating (counting), 41
  estimating, 39
  limitations, 45
  requirements sizing, 177
use case specifications, 161

use cases, 39, 60–62, 77–82, 89–99
  actors, 85–88
  checklist, 70
  defined, 77, 94
  functional requirements, 93–99
  limitations, 90, 92
  merging, 79
  multiple requirements views, 157
  naming, 61, 77, 79
  rating complexity, 42
  requirements management tools, 115
  requirements reviews, 72
  scenarios, 79–82
  template for documenting, 78
  vs. user stories, 41
user acceptance criteria, 11
user classes, 51, 85
  identifying, 15, 88
  requirements baseline definition, 151
user interface control descriptions, 160
user representatives. *See also* user classes
  product champions, 51
  surrogates for, 53–54
user requirements, 5, 60–65
  aligning with business objectives, 172
  elicitation questions, 62–65
  use cases, 79
user stories, 40, 77, 82–83
  defined, 82
  splitting into multiple, 82
  vs. use cases, 41
  user task descriptions, 161
users, 85–88
  reactions to use cases, 89, 94
  requirements reviews, 71
  surrogate, 53
UUCP (unadjusted use case points), 42
UUCW (unadjusted use case weights), 42

V

vagueness in requirements, 17
validation, 9, 11
VCF case-management software system project (FBI), 22
verbs, 61, 63, 77
version control tools, 166
video clips, multiple requirements views and, 158
views, requirements, 153–154
vision and scope documents, 137, 172
vision statement, 137
visual representations of requirements, 155
voice of the customer (VOC), 55, 58
  requirements reviews, 71
  use cases and user stories, 77–83

W

waterfall software development life cycle, 8
Web sites
  Process Impact, 13
  requirements management tools, 115
  use case benefits, 90
  word-usage errors, 127
why questions, 66–68, 131
Wideband Delphi estimation technique, 36
words, avoiding ambiguity with similar-sounding, 127
writing requirements, 9, 117–128
  ambiguity, 121–128
  business rules influencing, 174
  requirements management tools, 183

About the Author

Karl E. Wiegers is Principal Consultant with Process Impact, a software process consulting and education company in Portland, Oregon. His interests include requirements engineering, peer reviews, process improvement, project management, risk management, and software metrics. Previously, he spent 18 years at Eastman Kodak Company, where he held positions as a photographic research scientist, software developer, software manager, and software process and quality improvement leader. Karl received a B.S. degree in chemistry from Boise State College, and M.S. and Ph.D. degrees in organic chemistry from the University of Illinois. He is a member of the IEEE, IEEE Computer Society, and ACM.

Karl's most recent book is *More About Software Requirements: Thorny Issues and Practical Advice* (Microsoft Press, 2006). He also wrote *Software Requirements, Second Edition* (Microsoft Press, 2003), *Peer Reviews in Software: A Practical Guide* (Addison-Wesley, 2002), and *Creating a Software Engineering Culture* (Dorset House, 1996), as well as 160 articles on software development, chemistry, and military history. Karl is a two-time winner of the Productivity Award from *Software Development* magazine. Karl has served on the Editorial Board for *IEEE Software* magazine and as a contributing editor for *Software Development* magazine. He is a frequent speaker at software conferences and professional society meetings. In his spare time, Karl enjoys playing guitar, drinking wine, watching movies, and studying military history. You can reach him at *http://www.processimpact.com*.