# SOFTWARE REQUIREMENTS ESSENTIALS

## Core Practices for Successful Business Analysis

KARL WIEGERS | CANDASE HOKANSON

*Foreword by* **Joy Beatty**

FREE SAMPLE CHAPTER |

## Praise for *Software Requirements Essentials*

"As research for a book, I once read the ten best-selling requirements engineering books of the prior ten years. This one book succinctly presents more useful information than those ten books combined. I wish I'd had it as a reference then."

—***Mike Cohn**, author of* User Stories Applied *and co-founder of the Scrum Alliance*

"Diamonds come about when a huge amount of carbon atoms are compressed. The compression crystallizes to form diamonds. Karl and Candase have done something very similar: they have compressed their vast requirements knowledge into 20 gems they call 'core practices.'

"These 20 practices give you the essence of requirements discovery, and for extra convenience they are categorized to make your requirements journey more effective. These practices are potent stuff, and I recommend that they become part of everyone's requirements arsenal."

—***James Robertson**, author of* Mastering the Requirements Process *and* Business Analysis Agility

"What a valuable resource for new and experienced business analysts alike, who want an accessible, clearly written, and well-organized introduction to key business analyst practices. Karl and Candase do a great job of breaking down a complex role into a straightforward set of practices that can be integrated into your business analysis process to make it more effective."

—***Laura Brandenburg**, author of* How to Start a Business Analyst Career

"Candase and Karl have drawn upon their deep knowledge and experience of what it takes to elicit, identify, represent, communicate, and validate requirements for software products effectively. They have produced a useful, accessible, and clear book, which is full of practical advice, great examples, and answers to the hard questions that people building software products face in the real world. If you're involved in building software in any role, this book will give you guidance on ways to make sure the product meets customer needs and delivers real value."

—***Shane Hastie**, Global Delivery Lead at SoftEd and Lead Editor, Culture and Methods at InfoQ.com*

"*Software Requirements Essentials* will be a high-value addition to your business analysis library. I give the book high marks, as it does an excellent job of selecting and comprehensively covering the most essential business analysis practices teams should be considering. I thoroughly appreciated that the content was not overdone. Lessons were succinct while remaining extremely usable. Care was taken to ensure the guidance was applicable

whether you are using a waterfall, agile, or hybrid delivery approach. I believe anyone looking to improve their business analysis practices will find great practical advice they'll be able to apply immediately."

—***Laura Paton**, Principal Consultant, BA Academy, Inc.*

"Here is a book that all business analysts should have on their shelves, a readable reference that pulls together all the best practices we've been applying in business analysis for 50 years or so. While the book is aimed at the experienced BA, Karl and Candase thoughtfully provide an opening chapter reviewing the basic precepts and principles of business analysis. The book is written in Karl's inimitable easy-to-read style, so even beginning BAs can understand and apply the practices. Karl and Candase have made the book 'agile' with lots of practices applicable both to the traditional BA approach and to the BA who's defining user stories for the agile software developers.

"*Software Requirements Essentials* encapsulates all of the excellent advice and counsel Karl has given us over the years into this one touchstone of a book. I wish that I had written it."

—***Steve Blais**, author of* Business Analysis: Best Practices for Success *and co-author of* Business Analysis for Practitioners

"One of the many aspects of Karl Wiegers's latest book that we love is the universality of the requirements techniques he describes. Using real-life examples and easy-to-understand illustrations, Wiegers and Candase Hokanson describe practices that can be applied regardless of the project at hand or the methodology followed. They emphasize that there is no one right way to elicit and manage requirements; rather, they present many tried-and-true practices that lead to successful outcomes. Also helpful are the dozens of questions that business analysts can use to elicit various types of requirements.

"The authors emphasize concepts over methodology-specific terminology to ensure that the practices can be understood and applied as methodologies change. The recurrent themes they mention are spot-on and apply to any development effort. *Software Requirements Essentials* is a must-read for every business analyst who wants to avoid the pitfall of achieving 'project success but product failure.'"

—***Elizabeth Larson** and **Richard Larson**, past co-owners of Watermark Learning and authors of* CBAP Certification Study Guide

"So many product development projects face challenges because the stated requirements are ill-defined. This issue can be addressed by business analysts, or anyone conducting business analysis, if they possess the necessary toolkit of techniques and skills. *Software Requirements Essentials* offers an excellent introduction to the requirements engineering framework, and the techniques it encompasses, in an accessible and engaging way. The book offers invaluable guidance and insights via 20 best practices that are highly relevant, if not essential, for anyone working to define requirements. All business analysts need a mental map of the requirements definition service; this book provides it and more."

—***Dr. Debra Paul**, Managing Director, Assist Knowledge Development*

# Software Requirements Essentials

*This page intentionally left blank*

# Software Requirements Essentials

## Core Practices for Successful Business Analysis

**Karl Wiegers**
**Candase Hokanson**

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: dani3315/Shutterstock



Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2023931578



ISBN-13: 978-0-13-819028-6
ISBN-10: 0-13-819028-3

ScoutAutomatedPrintCode

## Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

*This page intentionally left blank*

*For Chris, naturally*
—K.W.

*For Peter and Edward*
—C.H.

*This page intentionally left blank*

# Contents

*This page intentionally left blank*

# Foreword

Long story short: If you are going to read only one requirements book, this is it. Karl and Candase have created the long-story-short version of how to develop good requirements on a software project.

Let's back up for the long story. If you made it this far, you already appreciate that good requirements are the foundation of any successful software or systems development project. Whether you're a business analyst, product owner, product manager, business stakeholder, or developer, it's well worth investing the time to elicit, analyze, document, and manage requirements to avoid paying for it later—quite literally. Good requirements lead to high-quality software.

*Software Requirements Essentials* is designed for the busy practitioner (and who isn't?) as a quick read about the most important requirements practices. It applies to projects using either traditional or agile approaches. The terminology and cadence of these practices may vary, but this book does a nice job of simplifying the differences and pointing out the similarities in those approaches. The practices described apply to virtually any kind of team building virtually any kind of product.

I know Karl and Candase very well personally and can attest to the strength of their collaboration. They each have areas of deep knowledge that complement one another, both extending and balancing each other's ideas. They also both live by what they say, having used the techniques themselves on many projects.

When it comes to comprehensive requirements books, I'm slightly biased in that I do love *Software Requirements, Third Edition*, which I coauthored with Karl. What many don't know is that I learned to be a business analyst from the first edition of *Software Requirements*. In fact, that's when I first met Karl. My job in the late 1990s was to define requirements practices for an agile-like iterative development approach at my software company. Boy, do I ever wish I had had this book back then!

*Software Requirements Essentials* distills the wealth of information found in *Software Requirements* and many other texts down to twenty of the most important requirements activities that apply on nearly all projects. Today's busy BA simply doesn't have the time to read a lengthy instructive guide front to back. But they should find the time to read this book.

This is the CliffsNotes version of many software requirements books, rolled into one. By nature of it being consciously focused and condensed, you should not expect massive details or full examples of every topic in *Software Requirements Essentials*.

For each of the many techniques presented, you'll get a little what, a little why, and a little how—enough to get you started and motivated. When you want more, follow the numerous links provided to reference materials.

As with any book by Karl, there is lots of practicality to it, with a dash of humor. Candase brings a long history of agile experience, keeping the text modern for today's common practices. Together, they've done a fine job of making this book highly relatable by pulling from their collective wealth of project experiences. The many real-life anecdotes make the recommended techniques real and justify their validity.

You don't have to read *Software Requirements Essentials*. But if you deal with requirements in any capacity on a software project, I'd consider it … a requirement!

*—Joy Beatty, COO, ArgonDigital*

# Acknowledgments

In preparing this book, we had valuable discussions with Jim Brosseau, Mike Cohn, Jennifer Colburn, David Mantica, Ramsay Millar, and Meilir Page-Jones. We thank them sincerely for their time and expert input. James Robertson eloquently reminded us of how important it is to understand the problem rather than assuming a proposed solution is correct. We appreciate Holly Lee Sefton sharing her expertise on data elicitation and governance. Eugenia Schmidt kindly provided an insightful quotation on requirements analysis, and Tim Lister allowed us to share his succinct definition of project success.

We greatly appreciate the helpful manuscript review input provided by Jeremy Beard, Tanya Charbury, Jennifer Colburn, James Compton, Mihai Gherghelescu, Lisa Hill, Fabrício Laguna, Reneé Lasswell, Linda Lewis, Geraldine Mongold, Meilir Page-Jones, Laura Paton, Maud Schlich, Eugenia Schmidt, James Shields, and Tom Tomasovic. Review comments from Joy Beatty, Runna Hammad, and Holly Lee Sefton were especially valuable.

Many thanks to Noor Ghafoor, Joyce Grapes, and Joe Hawes at ArgonDigital, who helped with prototype wireframes, example models, and glossary entries. Early editorial reviews by Erin Miller were particularly helpful.

Special thanks go to Jim Brosseau of Clarrus for his generous permission to include a version of his quality attribute prioritization spreadsheet tool in the supplementary materials for the book.

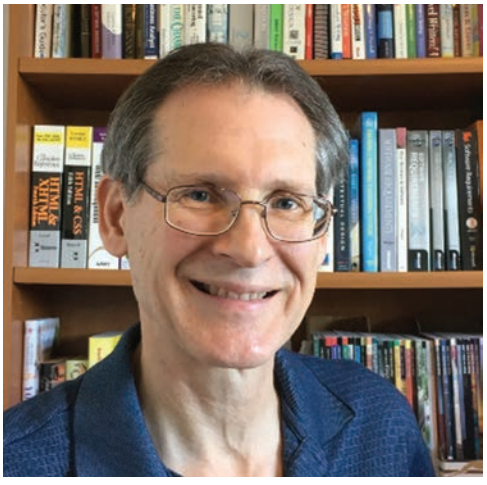
We're grateful to Haze Humbert, Menka Mehta, and the production team at Pearson Education for their fine editorial and production work on the manuscript. We also thank ArgonDigital and particularly Joy Beatty for their steadfast support for, and many contributions to, this project.

Working with a coauthor brings numerous benefits. It's tremendously helpful to have someone to bounce ideas off, to clarify your thinking, to improve your presentation, and to contribute new content, fresh perspectives, and unique project experiences. Two authors generate a synergy that lets them tell a richer story than either could on their own. Karl thanks Candase for contributing all those benefits, sharing her extensive experience on agile projects, and adding many illuminating true stories to this book.

As always, Karl is indebted to his wife, Chris, for patiently tolerating yet another book project. She's heard way more about software development and book writing over the years than she ever expected or cared to.

Candase is extremely grateful to her family for supporting her in her first book-writing experience. Special thanks go to her project teammates at ArgonDigital and at her major consulting client for their encouragement even during long and arduous product launches. Two people at ArgonDigital stand out for particular thanks: Joy Beatty for her encouragement and guidance in becoming an author and Megan Stowe for always inspiring Candase to continue learning. Finally, Candase would like to thank Karl for giving her the opportunity to coauthor with him, for being a great mentor through the publishing process, and for making the work fun and enjoyable.

# About the Authors



Since 1997, Karl Wiegers has been Principal Consultant with Process Impact, a software development consulting and training company in Happy Valley, Oregon. He has delivered more than 650 presentations to thousands of students and conference attendees worldwide. Previously, he spent eighteen years at Kodak, where he held positions as a photographic research scientist, software developer, software manager, and software process and quality improvement leader. Karl received a PhD in organic chemistry from the University of Illinois.

Karl is the author of thirteen previous books, including *Software Requirements*, *More About Software Requirements*, *Software Development Pearls*, *The Thoughtless Design of Everyday Things*, *Practical Project Initiation*, *Peer Reviews in Software*, and a forensic mystery novel titled *The Reconstruction*. He has written many articles on software development, management, design, consulting, chemistry, military history, and self-help. Several of Karl's books have won awards, most recently the Society for Technical Communication's Award of Excellence for *Software Requirements, Third Edition* (coauthored with Joy Beatty). Karl has served on the Editorial Board for *IEEE Software* magazine and as a contributing editor for *Software Development* magazine.

When he's not at the keyboard, Karl enjoys wine tasting, volunteering at the public library, delivering Meals on Wheels, wine tasting, playing guitar, writing and recording songs, wine tasting, reading military history, traveling, and wine tasting. You can reach him through www.processimpact.com and www.karlwiegers.com.

Candase Hokanson is a Business Architect and PMI-Agile Certified Practitioner at ArgonDigital, a software development, professional services, and training company based in Austin, Texas. With over ten years of experience in product ownership and business analysis, Candase works with clients to identify and implement the requirements that generate the best return on investment for their projects, regardless of the development life cycle. She has also trained or coached several hundred fellow product owners and business analysts. Her current passions are understanding how to optimize agile in large enterprises and agile requirements for very technical or back-end systems. Candase graduated from Rice University with a BS and an MS in civil engineering and a BA in religious studies.

Candase is an active member of the product management and business analysis communities, previously serving as a co-chair for the Keep Austin Agile conference in 2019 and president of the Austin IIBA. She has authored multiple articles on using visual models in agile, requirements in agile, and agile in the large enterprise.

Outside of work, Candase enjoys spending time with her family, all things Disney related, reading about British history, traveling, and wine tasting. You can reach her through www.argondigital.com and candase.hokanson@argondigital.com.

# Chapter 1

# Essentials of Software Requirements

Many years ago, I (Karl) would sometimes dive into writing a new program based on nothing more than an initial idea. I'd spend time coding, executing, fixing, and making a mess in my source code editor as I fumbled around, trying to get results. Eventually, I realized that the root of the problem was rushing to code without having an end point in mind—coding's fun! Those frustrating experiences taught me the importance of thinking through some requirements—objectives, usage tasks, data elements, and more—before doing anything else. After I adjusted my process to understand my requirements first, I never again felt like a software project was out of control.

All projects have requirements. Some teams begin with crisply defined business objectives, other teams receive a rich description of the desired solution's capabilities and characteristics, and still others start with only a fuzzy new product concept. Regardless of the starting point, all participants eventually must reach a shared understanding of what the team is supposed to deliver.

Some project participants aren't very interested in requirements. Certain managers may claim they're too busy to engage in requirements discussions. But then their expectations surface after the product has progressed to the point where major changes mean expensive rework. Some technical people might regard the time spent exploring and documenting requirements as a distraction from the real work of crafting code. However, a good set of requirements lets you answer some important—and universal—questions.

- Why are we working on this?
- Who are we trying to satisfy?

- What are we trying to build?
- What functionality do we implement first? Next? Maybe never?
- How can we tell if our solution[1] is good enough?
- How do we know when we're done?

This book describes the twenty most important practices that help software teams create a set of requirements to serve as the foundation for the subsequent development work. These practices broadly apply regardless of the type of product the team is creating or their development approach. Some software teams work not on discrete development projects but on existing products that demand ongoing modifications and new functionality. The people who are responsible for requirements work on product teams like those will find the practices in this book equally applicable to their work.

The requirements terminology differs between traditional (plan-driven or predictive) and agile (change-driven or adaptive) projects. Regardless of the terminology used, developers still need the same information to build the right solution correctly (Wiegers and Beatty, n.d.a). Some teams will perform certain practices iteratively, delivering value in small chunks. Others may do much of the requirements work early in the project because the problem is well understood. A startup that's trying to assess its product's market fit will focus on exploring ideas and approaches rather than trying to assemble a detailed specification. Whichever way you plan your development cycles, performing these twenty practices well can make the difference between delivering a solution that satisfies your stakeholders and creating one that does not.

## Requirements Defined

Now that we've used the word *requirement* several times, we should define what we mean. A software team must deal with many types of requirements-related knowledge, and people will be confused if they lack a common understanding of them. Although it's not fully inclusive, one useful definition of *requirement* comes from Ian Sommerville and Pete Sawyer (1997):

> Requirements are … a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

1. A *project* is an initiative that's launched to create a solution for one or more business problems or to exploit a business opportunity. A *solution* involves creating or modifying one or more products, which could include software systems, manual operations, and business processes. This book uses the terms *product*, *system*, and *application* interchangeably to refer to whatever your team is building.

This definition points out that requirements encompass multiple types of information. However, one aspect lacking from that definition is the concept of a requirement as a statement of a stakeholder need, which is the real starting point for all discussions about requirements.

Several classification schemas and models are in common use to describe various kinds of requirements information (Robertson and Robertson 2013, Wiegers and Beatty 2013, IIBA 2015). They generally agree but differ in some terminology details. In this book, we'll use the model shown in Figure 1.1.

This model shows various categories of requirements information (ovals) as well as containers in which to store that information (rectangles). For simplicity, this book will refer to those containers as documents. They could just as well be spreadsheets, databases, requirements management tools, issue tracking tools, wikis, or a wall covered with sticky notes—whatever works for your team. The container itself is less important than the information it holds and how you choose to record, organize, and communicate that information.

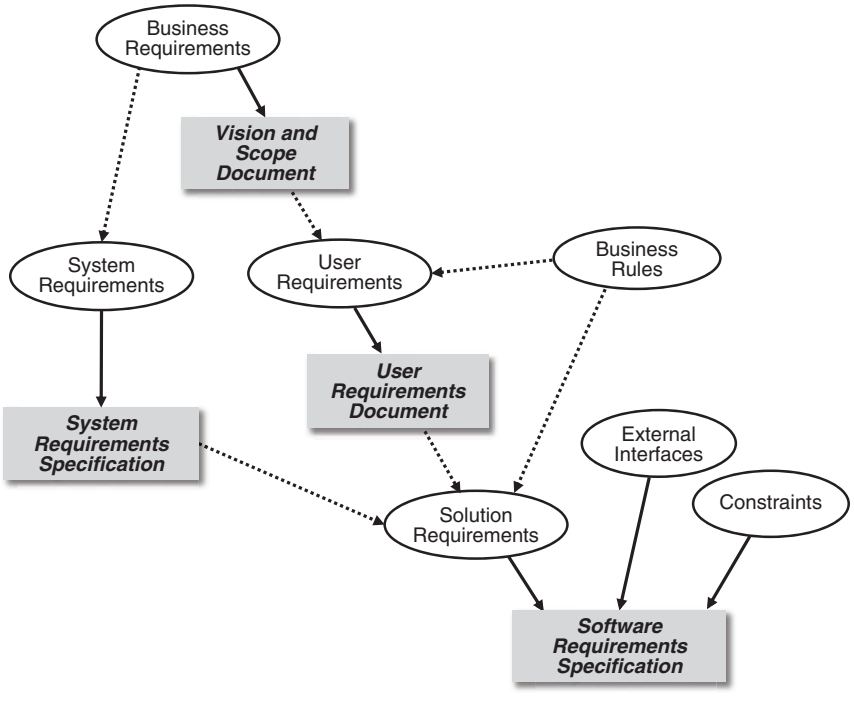



**Figure 1.1** *Connections between several types of requirements information and containers that store them. Solid lines mean "are stored in." Dotted lines mean "are the origin of" or "influence."*

Models like that in Figure 1.1 illustrate that there are many types of requirements information. This book uses the definitions in Table 1.1, which are broadly accepted in the requirements engineering and business analysis domains. Note that solution requirements encompass functional, nonfunctional, and data requirements (IIBA 2015). You'll see examples of these various items in later chapters. This book uses the collective term *requirements* to refer to all of these types of information, whether your local terminology focuses on features, use cases, user stories, or anything else.

**Table 1.1** *Definitions of several types of requirements information*

| Type of information | Definition |
|---|---|
| Business requirement | Information that describes why the organization is undertaking the project, establishes business objectives, defines a product vision, and includes other direction-setting information. (See Practice #2, "Define business objectives.") |
| Business rule | A directive that defines or restricts actions within an organization's operations. A policy, regulation, law, or standard that leads to derived solution requirements that enforce or comply with it. (See Practice #16, "Identify and document business rules.") |
| Constraint | A restriction imposed on the requirements, design, or implementation activities. |
| Data requirement | A definition of a data object or element that the system must manipulate, its composition and attributes, relationships among data objects, and their input and output formats. (See Practice #8, "Assess data concepts and relationships.") |
| External interface requirement | A description of a connection between the solution being built and other elements of the world around it, including users, other software systems, hardware devices, and networks. |
| Functional requirement | A description of some behavior that the product will exhibit under specified circumstances. |
| Nonfunctional requirement | Most commonly refers to what is also known as a *quality attribute* requirement. Quality attributes describe various quality, service, or performance characteristics of the solution. (See Practice #9, "Elicit and evaluate quality attributes.") |
| Solution requirement | A description of a capability or characteristic that the product being created must possess to satisfy certain user requirements and help achieve the project's business objectives. Solution requirements include functional, nonfunctional, and data requirements, as well as manual operations. |
| System requirement | A description of a top-level capability or characteristic of a complex system that has multiple subsystems, often including both hardware and software elements. System requirements serve as the origin of derived software solution requirements. |

**Table 1.1** (continued)

| Type of information | Definition |
|---|---|
| User requirement | A description of a task or goal that a user wishes to accomplish with the solution. The International Institute of Business Analysis generalizes this category to "stakeholder requirements," but in actuality, all requirements originate from some stakeholder (IIBA 2015). Here, we're specifically referring to things the *user* needs to do and user-specific expectations the solution must satisfy. (See Practice #6, "Understand what users need to do with the solution.") |

The fact that the diagonal arrows in Figure 1.1 that lead from Business Requirements down to the Software Requirements Specification are all aligned is no accident. Developers do not directly implement business requirements or user requirements. They implement functional requirements, including those derived from other categories of requirements information. The goal is to implement the right set of functionality that lets users perform their tasks and satisfies their quality expectations, thereby (hopefully) achieving the project's business requirements, within all imposed constraints. That "right set" of functional requirements comes from a foundation of well-understood business and user requirements.

Not every requirement will fit tidily into one or another of the categories in Table 1.1. Debating exactly what to call a specific statement is not important. What's important is that the team recognizes the need, analyzes it, records it in an appropriate form and location, and builds whatever is necessary to satisfy it.

## Good Practices for Requirements Engineering

The domain of requirements engineering is broadly divided into *requirements development* and *requirements management*. Requirements development encompasses the activities a team performs to identify, understand, and communicate requirements knowledge. Requirements management deals with taking care of requirements once you have them in hand. Requirements management activities include handling the inevitable changes, tracking versions of requirements and their status over time, and tracing individual requirements to related requirements, design components, code, tests, and other elements.

Requirements development is further partitioned into four subdomains:

| | |
|---|---|
| Elicitation | Activities to collect, discover, and invent requirements. Sometimes called gathering requirements, but elicitation is much more than a collection process. |
| Analysis | Activities to assess requirements for their details, value, interconnections, feasibility, and other properties to reach a |

| | |
|---|---|
| | sufficiently precise understanding to implement the requirements at low risk. |
| Specification | Activities to represent requirements knowledge in appropriate and persistent forms so that they can be communicated to others. |
| Validation | Activities to assess the extent to which requirements will satisfy a stakeholder need. |

These four sets of activities are not simply performed in a linear, one-pass sequence. As Figure 1.2 illustrates, they are interwoven and repeated until a particular set of requirements is understood well enough that the development team can build and verify that part of the solution with confidence. Requirements development is an incremental and iterative process by necessity, frustrating though that can be for the participants. Exploring requirements is an investment that reduces uncertainty and improves efficiency. The process might feel slow, but requirements thinking saves time in the end.

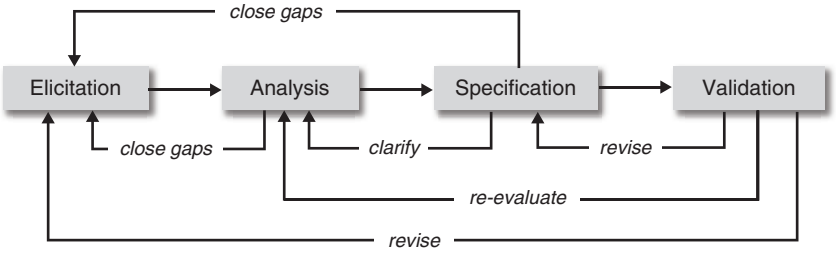



**Figure 1.2** *Requirements elicitation, analysis, specification, and validation are performed incrementally, iteratively, and often concurrently.*

Each of the requirements engineering subdomains encompasses numerous discrete practices. That's what this book is about. It describes twenty core practices that are particularly strong contributors to success on nearly all projects. Whether you lead requirement efforts, take part in them, or depend on them to perform your own work, you'll be more effective if you apply these core practices. Several of the practices refer to templates, spreadsheet tools, checklists, and other work aids, which you may download from the website associated with this book at www.informit.com.

We've grouped the practices by requirements engineering subdomain, four for requirements development and one for requirements management. Chapter 3 addresses requirements elicitation, Chapter 4 describes analysis practices, Chapter 5 deals with requirements specification, and Chapter 6 discusses key validation practices. The most important requirements management practices appear in Chapter 7.

Each practice description presents numerous practical techniques, identifies related practices, and suggests several Next Steps to help you put the practice into action right away. The practice descriptions are relatively short, so we've provided many references to other sources where you can get more detailed information.

Some practices in the elicitation chapter also describe related analysis and specification activities for topics like quality attributes and data. This grouping underscores the intrinsic entanglement of these requirements subdomains. It's not a clean separation.

You might have noticed that we skipped past Chapter 2. That chapter discusses five additional requirements-related activities that every project should perform to lay a solid foundation for a successful outcome. You're well served to conduct those activities early on to align all the stakeholders toward common goals, rather than going back to address them later when the team runs into problems.

This set of practices does not constitute a one-size-fits-all requirements process. When developing software, whoever leads the requirements work should work with other leaders to decide which requirements approaches will be most effective. Factors to consider include the project's nature and size, the team's experience with similar products, the access the team will have to stakeholders, particular areas of requirements risk, constraints, and organizational cultures (IIBA 2015). Select those practices that you believe will add the most value to the work, and adapt the practice descriptions from this book and other sources to best meet your specific needs.

The Appendix lists all twenty practices we address. These are by no means the only available requirements techniques. Numerous comprehensive (meaning long) books describe dozens of practices for requirements engineering and business analysis. These are some of the most useful resources:

- *Software Requirements, 3rd Edition* by Karl Wiegers and Joy Beatty (Microsoft Press, 2013)
- *Mastering the Requirements Process: Getting Requirements Right, 3rd Edition* by Suzanne Robertson and James Robertson (Addison-Wesley, 2013)
- *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise* by Dean Leffingwell (Addison-Wesley, 2011)
- *Business Analysis: Best Practices for Success* by Steven P. Blais (John Wiley & Sons, Inc., 2012)
- *Business Analysis, 4th Edition* by Debra Paul and James Cadle (BCS, The Chartered Institute for IT, 2020)
- *A Guide to the Business Analysis Body of Knowledge (BABOK Guide), 3rd Edition* (International Institute of Business Analysis, 2015)

- *Business Analysis for Practitioners: A Practice Guide* (Project Management Institute, Inc., 2015)
- *The PMI Guide to Business Analysis* (Project Management Institute, Inc., 2017)

We encourage you to refer to books like those for more information on the topics we discuss here, as well as to learn about other practices you might find helpful. A professional in the requirements field must accumulate a rich tool kit of practices and techniques, along with the experience to know which tool is the best one to use in each situation.

Some books or development frameworks recommend that you discard certain established practices and replace them with others. That's poor advice. You should *add* new practices to your tool kit, discarding older ones only when you can replace them with something that's demonstrably better in all situations. If something works for you, why throw it away?

## Who Does All This Stuff?

Historically, someone responsible for developing and managing requirements on a software project was called a *requirements analyst*, *systems analyst*, *business systems analyst*, or simply *analyst*. Large projects, particularly those building systems with both hardware and software components, might have *requirements engineers* who perform this function. Organizations that create commercial software products use *product managers* to bridge the gap between marketing and the development team. Agile development teams often include a *product owner* who defines and manages the requirements and other work items—collectively called product backlog items—that will lead to the solution.

In recent years, the term *business analyst* has largely replaced those historical job titles. This book uses *business analyst*, or BA, to refer to whomever on a development team has responsibility for requirements. In many organizations, a BA's role extends beyond dealing with requirements, but we will focus on their requirements activities.

Note that *business analyst* refers to a role, not necessarily a job title. Even if the team lacks an official BA, someone still must elicit, analyze, specify, validate, and manage its requirements. This work could be divided among multiple individuals, possibly including a project manager, quality assurance leader, and developers. When a team member who has another title is performing this kind of work, they are acting as a BA.

Because the requirements domain is both critical and complex, it's unrealistic to expect any random team member to perform the BA role without some education about how to do it well. A capable BA brings a particular set of knowledge, experience, personality characteristics, and skills to the process, including those listed in Table 1.2 (Wiegers and Beatty 2013). If you're working in this role, assess your capabilities in each category and then work to improve those that aren't as strong as others.

**Table 1.2** *Some valuable business analyst skills and characteristics*

| | |
|---|---|
| Listening | Writing |
| Interviewing and questioning | Modeling |
| Facilitation | Flexibility across the abstraction scale |
| Nonverbal communication | Organizing information and activities |
| Analytical thinking | Handling interpersonal interactions |
| Systems thinking | Leadership |
| Quick thinking | Creativity |
| Observation | Curiosity |

In recent years, several organizations have recognized the great value that business analysts and requirements engineers can contribute. These organizations have developed bodies of knowledge and professional certifications that people working in these fields can pursue. Such professional organizations include

- The International Institute of Business Analysis (IIBA), iiba.org
- The International Requirements Engineering Board (IREB), ireb.org
- The Project Management Institute (PMI), pmi.org

The bodies of knowledge these organizations have accumulated are rich sources of information about the many requirements processes, techniques, and tools that contribute to success.

## Some Recurrent Themes

Some common themes run through this book. Keep the following themes in mind as you select practices to use on your projects and tailor them to suit each situation.

- **Requirements development demands an incremental and iterative approach.** It's highly unlikely that anyone will think of all the requirements before development begins and that they will remain unchanged. People get more information, have fresh ideas, remember things they had overlooked, change their minds, and must adapt to changing business and technical realities.
- No matter how you choose to represent requirements knowledge, **the goal of all specification activities is clear and effective communication**. The artifacts the BA produces have multiple audiences. Those audiences may wish to see information presented in different forms and at various levels of detail. Consider those diverse audiences as you create requirements deliverables.
- **Requirements engineering is a collaborative process.** Requirements affect all stakeholders. Many people can supply input to the requirements, many people do work based on them, and many people use the resultant solution. Customer engagement is a powerful contributor to a successful outcome. The BA must work with people who can accurately present the needs of diverse stakeholder communities. Most requirements decisions involve multiple participants with different, and sometimes conflicting, interests and priorities.
- **Change happens.** A solution-development effort is chasing a moving target. Business needs, technologies, markets, regulations, and users change. A BA must keep up with evolving needs and make sure that changes are clearly understood, recorded, and communicated to those they affect.
- A powerful way to increase development productivity is to minimize the amount of rework the team must perform. Therefore, try to **push quality activities to the front** of the development cycle—that is, earlier rather than later. Better requirements pay off with less rework later in development or following delivery.
- **Use risk thinking** to decide which requirements practices to employ, when to perform them, when to stop, and how much detail is necessary. For instance, the risks of miscommunication and wasted effort are greater when development is outsourced or teams are remote than when participants work in proximity. Therefore, requirements for such projects must be written more precisely and in more detail than when developers can quickly get answers from the people around them.

## The Life and Times of Requirements

Neither requirements development nor requirements management activities end when the initial project team delivers the solution. They continue throughout the product's operational or market life, as it evolves through an ongoing series of enhancement and maintenance cycles. As change requests arrive, someone must elicit the corresponding requirements details and evaluate their impact on the current solution. They must then document the new or changed requirements, validate them, track their implementation status, trace them to other system elements, and so forth.

The BA should look for existing requirements-related items from other projects they could reuse. At times, they might create deliverables that have reuse potential elsewhere in the organization. Glossaries, business rules, process descriptions, stakeholder catalogs, data models, security requirements, and the like can apply to multiple situations. Once an organization invests in creating these artifacts, it should organize them to enable reuse and look for opportunities to leverage that investment further (Wiegers and Beatty 2013).

## Getting Started

This book contains a lot of information and recommends many practices and techniques. Some of these you no doubt already perform; others might be new to you. We have two pieces of advice about getting started with the practices we suggest.

1. Don't feel bad if you don't already perform all these activities on your projects.
2. Don't try to do everything at once.

As you read, identify those practices that you think would add the most value to your project. Look for opportunities to try them and situations in which they might yield better results. Recognize the reality that the learning curve will slow you down a bit as you try to figure out how to make new methods work for you and your colleagues. Follow the references we've provided to learn more about those practices that look interesting to you. Over time, new ways of working will become part of your BA tool kit—and you will get better results.

Whether you call it business analysis or requirements engineering, it's a challenging, yet vital, function. The core practices described in this book give you solid tools to tackle this critical activity with confidence.

*This page intentionally left blank*

# Chapter 2

# Laying the Foundation

In the classical pure (and hypothetical) waterfall software development model, the team accumulates a complete set of requirements for the product, designs a solution, builds the entire solution, tests it all, and delivers it. We all know that approach doesn't work well in most cases.

Projects will vary in how much requirements work can, and should, be done up front. Sometimes it's possible to specify a good portion of the requirements for an information system before getting too far into implementation. Complex products with multiple hardware and software components demand careful requirements engineering because the cost of making late changes is high. For applications that change rapidly or lend themselves to incrementally releasing ever more capable software versions, developing requirements just-in-time in small chunks is an effective approach. Innovative apps may involve a lot of concept exploration, prototyping, feasibility studies, and market assessment.

No single approach to the development life cycle or requirements work optimally fits every situation. However, there are several interconnected activities related to requirements that every team should perform at the beginning. This chapter describes five essential practices that collectively provide a solid foundation for both technical and business success:

Practice #1. Understand the problem before converging on a solution.

Practice #2. Define business objectives.

Practice #3. Define the solution's boundaries.

Practice #4. Identify and characterize stakeholders.

Practice #5. Identify empowered decision makers.

**Practice #1** Understand the problem before converging on a solution.

Imagine that you worked for more than a year on a project that had executive support and high visibility. In your business analyst role, you performed the requirements elicitation, analysis, and specification. The development team built what the stakeholders asked for and deployed the product on schedule. But just three months later, the product is considered a failure and decommissioned. Why? Because it didn't solve the right problem.

Far too often, teams build and release requirements, features, and even entire products that go unused because those teams didn't fully understand the business situation and the problems they were trying to solve. Understanding the problems or opportunities that your solution will address aligns all participants on the core issues and provides confidence that the solution will indeed achieve the desired outcomes.

## Business Problems

A business problem is any issue that prevents the business from achieving its goals or exploiting an opportunity (Beatty and Chen 2012). A business problem can be small, such as a user complaint that some task takes too long, which can perhaps be solved by streamlining some functionality. Or it can be as large as organization-level business challenges—spending too much money, not making enough money, or losing money—that demand major projects or entirely new products.

Organizations launch initiatives to solve one or more business problems. Each activity gets funded because management expects its business value to outweigh its costs. However, those problems or opportunities often are neither explicitly stated nor documented. Rather than presenting a clear problem statement, the executive sponsor or lead customer might simply tell the team what to build. This can cause the scenario described above: project success but product failure. If you don't understand the problem adequately, or if you begin with a specific solution in mind, there's a good chance that the team will solve only part of the problem—or perhaps none of it.

It's a good idea to avoid presuming that either a presented problem or a presented solution is necessarily correct. That initial presentation might come from a business case, project charter, senior manager, or product visionary. But can you trust it as setting the right direction for all the work that will follow?

When you're presented with a stated problem, perform a *root cause analysis* until you're confident that the real issue and its contributing factors are well understood (Tableau 2022). Then you can derive possible solutions that you know will address those very issues. If you're presented with a solution, explore this question: "If <*solution*> is the answer, what was the question?" In other words, ask "Why do

you think that's the right solution?" You might discover that the underlying issue demands a different approach: possibly simpler, possibly more complex, possibly more specific, possibly more general. You won't know until you perform the analysis.

## Eliciting the Real Problems

A stakeholder might request a solution such as "Combine several systems into one," with the expectation that such a strategy would address multiple, unspecified objectives. However, system consolidation could be overkill if a simpler answer is appropriate. If the problem is that you're spending too much money on maintenance and support for four existing systems, combining them could be the right approach. However, suppose that the most pressing concern instead is that your users are unhappy. A root cause analysis using the *5 Whys* technique with the pertinent stakeholders could sort all this out (Tableau 2022).

Root cause analysis involves working backward from a stated problem or a proposed solution to identify the underlying problems and the factors that contribute to them. Assessing those factors then leads to the appropriate solution choice. With the 5 Whys technique, you ask questions like "Why is that a problem?" or "Why are we not already achieving that goal today?" repeatedly until you unveil the compelling issue that drove launching the initiative in the first place. The conversation between a business analyst and a key stakeholder might go something like this:

*Analyst:* "You requested that we combine your four current systems into one. Why do we need to combine them?"

*Stakeholder:* "Because our customers complain that they must keep signing in between webpage clicks. It's annoying. This is because they're accessing different backend systems that all have separate user accounts."

*Analyst:* "Why is it an issue if your customers are complaining?"

*Stakeholder:* "According to our market research, 25 percent of our customers have left us for the competition because of their frustrations with usability on our site."

*Analyst:* "If that's the case, why not just implement single sign-on to improve usability?"

*Stakeholder:* "That would help, but we'd still have to maintain and support all four systems."

*Analyst:* "If we combined them, wouldn't you still need the same number of support people for the new system?"

*Stakeholder:* "We don't believe so. The four current systems use different programming languages. We need at least one engineer fluent in each language to support each system, although there's not enough work to keep them busy. By combining the systems into one using a single language, we could free up the additional engineers to work on other products."

*Analyst:* "Ah, so it looks like you're trying to solve multiple problems. You want higher customer retention, and you also want to reduce support costs and free up staff by using fewer technologies."

By asking "why" several times in this conversation, the analyst now understands that the stakeholder expects their proposed solution to address two significant concerns. The request to combine several systems into one might indeed be the best long-term strategy. However, an interim solution using single sign-on could appease the disgruntled customers quickly, while the consolidation initiative works on the larger concern of support and maintenance.

A root cause analysis diagram, also called a fishbone or Ishikawa diagram, is a way to show the analysis results. Suppose the BA drills down into the first problem the stakeholder brought up: losing frustrated customers. The BA could apply the 5 Whys technique to determine exactly why the customers are frustrated and then draw a diagram like the one in Figure 2.1. The problem goes at the head of the "fish." Place the highest-level causes in the boxes on diagonal lines coming off the fish's backbone. Add contributing causes on the short horizontal lines from each diagonal. Continue the exploration until you reach the ultimate, actionable root causes. Then you can devise one or more solutions to address them.

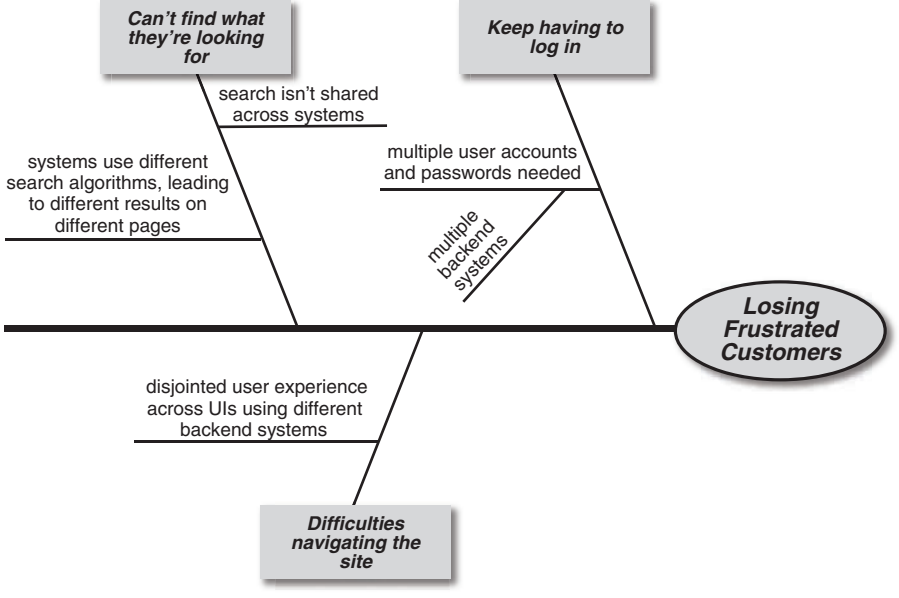



**Figure 2.1** *A root cause analysis (fishbone or Ishikawa) diagram example shows the factors that contribute to the stated problem.*

Once you've identified the primary and contributing issues, consider all their implications before committing to a solution. The requested or most apparent solution could be the wrong strategy. On one of Candase's projects, the problem was that the version of the commercial off-the-shelf (COTS) package the company used was going end-of-life soon and the vendor would no longer support it. After that, any production issue could have cost the company its entire business because it wouldn't have any vendor assistance. Nor could it currently make its own enhancements to the vendor product. The obvious solution was to upgrade to the latest version of the vendor's product. However, the company would have had to pay the vendor high service fees to resolve problems and add enhancements. Consequently, the company considered both acquiring a new COTS package from a different vendor and building an in-house replacement as better solutions to both the initial end-of-life concern and the additional issues.

Problem analysis can reveal other, unobvious challenges. You might confront conflicting problems from different stakeholders or be trying to solve multiple, disparate problems with a single fix. As you explore the issues, look for situations where you might need several solutions, rather than seeking a single silver bullet.

## Keeping the Business Problem in Focus

When the key stakeholders have agreed upon a clear understanding of the core business concerns, consider writing a problem statement (Kyne 2022). A template like this can be helpful (Compton 2022):

| | |
|---|---|
| ***Situation*** | Describe the background, context, and environment. |
| ***Problem*** | Describe the business problems or opportunities as you now understand them. |
| ***Implication*** | Describe the likely results if the problem isn't solved. |
| ***Benefit*** | State the business value of solving the problem. |
| ***Vision*** | Describe what the desired future state would look like. |

A concise problem statement serves as the reference point for the rest of the work. It feeds directly into crafting the specific business objectives that management or your customers expect the solution to achieve (see Practice #2, "Define business objectives"). The problem statement also helps the team make decisions throughout the project. When prioritizing requirements, favor those items that are the most critical or timely contributors to solving the highest-value problem (see Practice #13, "Prioritize the requirements"). In the combine-several-systems-into-one example above, implementing single sign-on to relieve customer frustration

would be a quicker fix than combining multiple systems and would address the immediate concern of losing customers.

Whenever someone requests a new system capability, ask how it relates to the business problems (see Practice #20, "Manage changes to requirements effectively"). If you can't tie each new requirement to any of the defined business problems, either there are more problems yet to explore or you don't need the new requirement.

Stakeholders often will propose a specific deliverable as a requirement: "Build me product X or feature Y." The stakeholder's solution may indeed be the correct one—but not necessarily. Take the time to thoroughly understand the real business problem to ensure that the team focuses on achieving the proper outcomes. If your analysis reveals that the real problem doesn't quite match what you found in a business case or other originating document, revise that document to match the newly understood reality. That insight could profoundly change the project's direction.

## Related Practices

Practice #2. Define business objectives.

Practice #3. Define the solution's boundaries.

Practice #13. Prioritize the requirements.

Practice #20. Manage changes to requirements effectively.

## Next Steps

1. If you haven't already done so, talk with project leadership and key stakeholders about why they're undertaking your initiative to make sure you understand the problem it is intended to solve.
2. Create a root cause analysis diagram for your core business problem, using a technique like 5 Whys to discover both major and contributing causes.
3. Write a problem statement using the template described in this section.
4. Based on the problem or problems identified, assess whether your current solution concept will address them adequately. If not, either change the solution or point out the risk that the current solution may not be sufficient.

# Chapter 3

# Requirements Elicitation

The first step in dealing with requirements is to get some. People often speak of "gathering requirements," as though it were a simple collection process: The requirements are sitting around in people's heads, and the business analyst merely asks for them and writes them down. It's never that simple. In reality, stakeholders begin with random fragments of information: dissatisfaction with their current systems, bits of functionality they want, tasks to perform, important pieces of data, and ideas of what screen displays might look like.

*Requirements elicitation* is a better term for this foundational activity. To elicit something means to draw it forth or bring it out, particularly something that's hidden or latent. The *Merriam-Webster Thesaurus* (2022) says, "*elicit* usually implies some effort or skill in drawing forth a response." That skill is a significant asset that a business analyst brings to software development. Requirements elicitation does involve collection, but it also involves exploration, discovery, and invention. The BA guides this imaginative journey, working with diverse stakeholders to understand the problem and then define a satisfactory solution. The BA looks for potential requirements from many sources, including these:

- User representatives and many other stakeholders
- Documentation about business processes, current systems, and competing products
- Laws, regulations, and business policies
- Existing systems, which may or may not be documented
- User problem reports, help desk records, and support staff

An experienced BA exploits multiple techniques for elicitation, choosing the appropriate tool for a particular situation. Factors to consider when selecting elicitation methods include the types of information needed; who has that information, where those people are located, and their availability; the effort that the method requires; the budget and time available; the development team's life cycle model and methodologies; and the cultures of the developing and customer organizations (IIBA 2015).

This book does not go into elicitation techniques in detail, as those are thoroughly described in other resources (e.g., Davis 2005, Robertson and Robertson 2013, Wiegers and Beatty 2013, IIBA 2015). Table 3.1 lists several commonly used elicitation activities and the typical participants.

**Table 3.1** *Some common requirements elicitation techniques*

| Participants | Activities |
|---|---|
| Business analyst | • Document analysis<br>• Existing product and process analysis<br>• System interface analysis<br>• User interface analysis<br>• Data mining and analysis |
| Business analyst and stakeholders | • Interviews<br>• Facilitated group workshops<br>• Scenario analysis<br>• Observing users at work<br>• Process modeling<br>• Focus groups<br>• Brainstorming<br>• Mind mapping<br>• Prototyping<br>• Collaboration tools such as wikis and discussion forums<br>• Questionnaires and surveys |

This chapter describes four core practices that are particularly valuable for eliciting both functional and nonfunctional requirements:

Practice #6. Understand what users need to do with the solution.

Practice #7. Identify events and responses.

Practice #8. Assess data concepts and relationships.

Practice #9. Elicit and evaluate quality attributes.

**Practice #6** Understand what users need to do with the solution.

If you were holding a requirements elicitation discussion with some users about a new information system, which of these questions do you think would yield the greatest insights?

- What do you want?
- What are your requirements?
- What do you want the system to do?
- What features would you like to see in the system?
- What do you need to do with the solution?

We favor the final question. While the first four questions can provide a good starting point to ask *why* a stakeholder wants those things, they all inquire about the solution, not the user's problems, needs, or goals. Focusing on features can lead the team to implement incomplete functionality that doesn't let users do all the things they must do. The feature-centered mindset also can lead to building functionality that seems like a good idea but goes unused because it doesn't directly relate to user tasks. Regardless of your development approach, if you don't understand what the users need to do with the features they request, you might release a product that you must rework later.

Karl once saw the limitations of elicitation questions that focus on the solution. A company held a daylong workshop with about sixty participants to brainstorm ideas for a large new commercial product. They stapled together the output from their six subgroups and called it a requirements specification. But it wasn't. It was a mishmash of functionality fragments, feature descriptions, user tasks, data objects, and performance expectations, along with extraneous information, all stirred together with no structure or organization. Simply asking people to imagine what they wanted to see in the new product didn't produce actionable requirements knowledge. Much more requirements development work was needed following the workshop.

## Focusing on Usage

The question "What do you need to do with the solution?" is a more effective opening for discussing requirements. By understanding what the users need to do, the BA can deduce just what functionality is needed. A usage-centric approach makes it

more likely that the solution will satisfy user needs, incorporating the necessary capabilities without wasting development effort on unneeded functions (Wiegers 2022).

Stories, scenarios, and use cases are variations on a common theme: asking users to describe an interaction they might have with a software system or a business to achieve some goal (Alexander and Maiden 2004). These descriptions of user goals and the interactions that lead to achieving them constitute the user requirements. User requirements appear in the middle section of the requirements information model in Figure 1.1, as reproduced in Figure 3.1. The user requirements should align with the business objectives from the vision and scope document and contribute to solving an identified business problem.

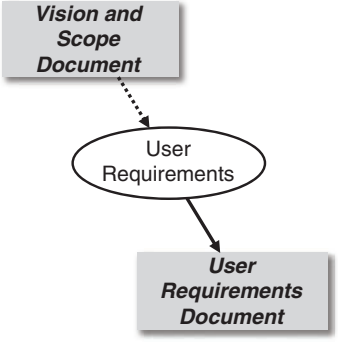



**Figure 3.1** *User requirements lie between business requirements and solution requirements.*

## Eliciting User Requirements

A person doesn't launch an application to use a particular feature; they launch it to do something. It's difficult for users to articulate their "requirements," but they can easily describe how they might perform a business activity. During an elicitation discussion, the BA might ask a user representative, "Please describe a session you might have with the product we're talking about. What would you be trying to accomplish? How do you imagine your dialogue with the system would go?" A description of a single interactive session like this is called a *scenario*.

A scenario identifies a sequence of steps that define a task to achieve a specific intent (Alexander and Maiden 2004). When you ask a user to describe a scenario, they'll usually begin with the most typical or frequent activity they perform. This is sometimes called the normal flow, main flow, main success scenario, or happy path. From that initial scenario, the BA and user can then explore *alternative* scenarios (or flows), variations that also lead to a successful outcome. They can also

discuss *exceptions*, possible conditions that could prevent a scenario from concluding successfully.

An effective way to organize these related scenarios is in the form of *use cases* (Cockburn 2001, Kulak and Guiney 2004). A use case structures all this information according to a template, which is described in the next section. The use case technique helps the team acquire and organize the mass of requirements information that any sizable system involves. If an elicitation participant says "I want to <*do something*>" or "I need to be able to <*do something*>," the <*do something*> likely is a use case.

The various user classes will have different use cases, different things they need to accomplish with the solution. That's why it's a good idea to conduct group elicitation activities with members of each user class separately. As an example, Table 3.2 lists a few use cases for each of the user classes named earlier for the hypothetical Speak-Out.biz publication platform in Practice #4, "Identify and characterize stakeholders."

**Table 3.2** *Some use cases for several Speak-Out.biz user classes*

| User class | Use cases |
|---|---|
| Author | Draft an Article |
| | Edit an Article |
| | Publish an Article |
| | Submit an Article to a Publication |
| | View Article Statistics |
| Reader | Read an Article |
| | Comment on an Article |
| | Subscribe to an Author |
| Publication Editor | Create a New Publication |
| | Accept or Reject a Submitted Article |
| | Reply to an Author |
| Administrator | Respond to a Reader Complaint |
| | Suspend an Author's Account |

Each use case name is a concise statement that clearly indicates the user's goal, the outcome of value that the user wishes to achieve. Notice that all the use cases in Table 3.2 begin with a definitive action verb. This is a standard use case naming convention.

Agile projects often rely on *user stories* as a technique for discussing system capabilities. According to agile expert Mike Cohn (2004), "A user story describes functionality that will be valuable to either a user or purchaser of the system or software." A user story is intentionally brief, a starting point for further exploration of its details so that developers can learn enough to implement the story. User stories conform to a simple pattern, such as this one:

> As a <*type of user*>, I want to <*perform some task*> so that I can <*achieve some goal*>.

Stories that focus on what users need to do with the solution, rather than on bits of system functionality, can serve the goal of usage-centric requirements exploration. Here's a user story we might hear from a Speak-Out.biz author:

> As an author, I want to view the page-view statistics for my published articles so that I can see which topics my readers enjoy the most.

This story addresses a piece of the functionality for the final use case shown for the Author user class in Table 3.2, View Article Statistics. The user story format offers the advantages of naming the user class and describing the intent. That information would appear in a use case specification, but it's helpful to see it right up front like this.

There are ongoing debates about whether use cases are appropriate—or even allowed—for agile development. This isn't the place to rehash those debates, but the short answer is: They are (Leffingwell 2011). Both use cases and user stories have their advantages and limitations (Bergman 2010). Both can be used to explore what users need to accomplish with the solution.

One of the BA's challenges is to examine a particular scenario that describes a single usage session and consider how to generalize it to encompass a group of logically related scenarios. That is, the BA moves up the abstraction scale from a specific scenario to a more general use case. Similarly, the BA on an agile project might see that a set of related user stories can be abstracted into a larger *epic* that needs to be implemented over several iterations.

At other times, elicitation participants might begin with a complex usage description that the BA realizes should be split into multiple use cases. Those individual use cases often can be implemented, and executed, independently, although several could perhaps be chained together during execution to carry out a larger task. On an agile project, a user story that's too large to implement in a single iteration is split into several smaller stories. Moving between levels of abstraction like this is a natural part of exploring user requirements.

Use cases facilitate top-down thinking, describing multiple scenarios and fleshing out the details of the user–system interactions. Use cases provide a context for organizing related pieces of information. Epics perform an analogous top-down function on agile projects. User stories describe smaller user goals or pieces of system functionality without much context or detail. Stories generally are smaller than

use cases, describing slices of functionality that can be implemented in a single development iteration. Related user stories can be grouped together and abstracted into an appropriate use case or an epic. Any approach can be effective—use cases or user stories, top-down or bottom-up—provided the focus stays on usage.

## Anatomy of a Use Case

Unlike the simple user story format, a use case specification follows a rich template like the one in Figure 3.2 (Wiegers and Beatty 2013). You may download this template from the website that accompanies this book. A collection of use case descriptions could serve as the contents of the user requirements document ("container") that appears in Figure 3.1. Nothing says that you must complete this full template for each of your use cases. Write in whatever level of detail will clearly communicate the use case information to those who must validate, implement, or write tests based on it.

| Use Case Element | Description |
|---|---|
| ID and Name | Give each use case a unique identifier and a descriptive name. |
| Primary Actor | Identify the actor (user role) who initiates the use case and derives the principal benefit from it. |
| Secondary Actors | Identify other users or systems that participate in performing the use case. |
| Description | Provide a brief description of the use case in just a few sentences. |
| Trigger | Identify the event or action that initiates the use case's execution. |
| Preconditions | State any prerequisites that must be met before the use case can begin. |
| Postconditions | State conditions that are true after the use case is successfully completed. |
| Normal Flow | The core of a use case specification describes how the user visualizes interacting with the system to accomplish the goal. List the steps in the dialog that takes place between the primary actor, the system, and any other systems or actors that participate in the normal flow scenario. |
| Alternative Flows | Describe any alternative ways the use case might be performed and still satisfy the postconditions. Alternative flows often involve branching away from the normal flow at some step and then perhaps rejoining it. |
| Exceptions | Identify conditions for each flow that could terminate the scenario before it completes successfully. Describe how the system should respond or help the user resolve the problem. |
| Priority | State the relative priority of this use case compared to others. |
| Business Rules | Point to any business rules that influence how this use case is implemented or executed. |
| Assumptions | State any known assumptions that people are making with respect to this use case. |

**Figure 3.2** *A rich use case specification template.*

## Applying Usage-centric Requirements Information

User requirements serve as a starting point for several subsequent activities. Both use cases and user stories need to be further elaborated into a set of functional requirements, which is what developers implement. This step takes place whether a BA does it analytically and documents the resultant details or whether each developer does it in their head on the fly (not the recommended approach).

Use cases and user stories both facilitate starting testing early in the development cycle. If a BA derives functional requirements from a use case and a tester derives tests, you now have two representations of requirements knowledge that you can compare. That comparison can reveal requirements errors, ambiguities, and omissions. See Practice #18, "Review and test the requirements," for more on this topic. Documenting how the system should handle exceptions lets developers build more robust software and helps testers do a more thorough job.

User stories and use cases also lie at the core of requirements prioritization. The deciding stakeholder typically prioritizes user stories or use cases in a sequence that maximizes customer value. The team then fits them into iterations or increments based on the team's available capacity, considering any relevant technical and functional dependencies. While user stories are each prioritized on their own, the individual flows within a use case could have different priorities. You might opt to implement the normal flow and its exceptions in one development increment, and then implement alternative flows and their corresponding exceptions in upcoming increments.

Usage-centric requirements exploration won't reveal behind-the-scenes capabilities, such as a timeout to turn off some device or log out a user after a period of inactivity. Nonetheless, focusing elicitation on understanding what users must do with the system helps the team implement all the necessary—and no unnecessary—functionality. Usage-centric thinking also leads nicely into designing an optimal user experience (Constantine and Lockwood 1999).

## Related Practices

Practice #2. Define business objectives.

Practice #4. Identify and characterize stakeholders.

Practice #7. Identify events and responses.

Practice #13. Prioritize the requirements.

Practice #16. Identify and document business rules.

Practice #18. Review and test the requirements.

# Appendix

# Summary of Practices

## Laying the Foundation

Practice #1. Understand the problem before converging on a solution.

Practice #2. Define business objectives.

Practice #3. Define the solution's boundaries.

Practice #4. Identify and characterize stakeholders.

Practice #5. Identify empowered decision makers.

## Requirements Elicitation

Practice #6. Understand what users need to do with the solution.

Practice #7. Identify events and responses.

Practice #8. Assess data concepts and relationships.

Practice #9. Elicit and evaluate quality attributes.

## Requirements Analysis

Practice #10. Analyze requirements and requirement sets.

Practice #11. Create requirements models.

Practice #12. Create and evaluate prototypes.

Practice #13. Prioritize the requirements.

## Requirements Specification

Practice #14. Write requirements in consistent ways.

Practice #15. Organize requirements in a structured fashion.

Practice #16. Identify and document business rules.

Practice #17. Create a glossary.

## Requirements Validation

Practice #18. Review and test the requirements.

## Requirements Management

Practice #19. Establish and manage requirements baselines.

Practice #20. Manage changes to requirements effectively.

# Index

## Numbers



## A

## B

## C



## D

## E

## L



## M



## N



## O



## P

## Q



## R

## S

## T



## U

## V



## W