The Crystal Series for Software Developers
Alistair Cockburn, Series Editor

Writing Effective Use Cases

Alistair Cockburn

FREE SAMPLE CHAPTER

The Writing Process

1. Name the system scope and boundaries.
   Track changes to this initial context diagram with the in/out list.
2. Brainstorm and list the primary actors.
   Find every human and non-human primary actor, over the life of the system.
3. Brainstorm and exhaustively list user goals for the system.
   The initial Actor-Goal List is now available.
4. Capture the outermost summary use cases to see who really cares.
   Check for an outermost use case for each primary actor.
5. Reconsider and revise the summary use cases. Add, subtract, or merge goals.
   Double-check for time-based triggers and other events at the system boundary.
6. Select one use case to expand.
   Consider writing a narrative to learn the material.
7. Capture stakeholders and interests, preconditions and guarantees.
   The system will ensure the preconditions and guarantee the interests.
8. Write the main success scenario (MSS).
   Use 3 to 9 steps to meet all interests and guarantees.
9. Brainstorm and exhaustively list the extension conditions.
   Include all that the system can detect and must handle.
10. Write the extension-handling steps.
    Each will end back in the MSS, at a separate success exit, or in failure.
11. Extract complex flows to sub use cases; merge trivial sub use cases.
    Extracting a sub use case is easy, but it adds cost to the project.
12. Readjust the set: add, subtract, merge, as needed.
    Check for readability, completeness, and meeting stakeholders' interests.

The Agile Software Development Series

Alistair Cockburn and Jim Highsmith, Series Editors

Visit **informit.com/agileseries** for a complete list of available publications.

Agile software development centers on four values, which are identified in the Agile Alliance's Manifesto*:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The development of Agile software requires innovation and responsiveness, based on generating and sharing knowledge within a development team and with the customer. Agile software developers draw on the strengths of customers, users, and developers to find just enough process to balance quality and agility.

The books in The Agile Software Development Series focus on sharing the experiences of such Agile developers. Individual books address individual techniques (such as Use Cases), group techniques (such as collaborative decision making), and proven solutions to different problems from a variety of organizational cultures. The result is a core of Agile best practices that will enrich your experiences and improve your work.

Writing Effective Use Cases

This page intentionally left blank

Writing Effective Use Cases

Alistair Cockburn

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit us on the Web at *www.awl.com/cseng/*

Library of Congress Cataloging-in-Publication Data

Cockburn, Alistair.
Writing effective use cases / Alistair Cockburn.
p. cm. -- (The Crystal Collection for software professionals)
Includes bibliographical references and index.
ISBN 0-201-70225-8 (alk. paper)
1. Application software—Development. 2. Use cases (Systems engineering)
I. Title. II. Series.

QA76.76A65 C63 2000
005.3--dc2 00-040179

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
Twenty-third printing, April 2011

Contents

Part 3 Reminders for the Busy 203

Chapter 20 Reminders for Each Use Case 205

Chapter 21 Reminders for the Use Case Set 215

Chapter 22 Reminders for Working on the Use Cases 221

Appendices

Appendix C *Glossary* 253

Appendix D *Readings* 257

Index 259

This page intentionally left blank

Preface

More and more people are writing use cases, for behavioral requirements, for software systems or to describe business processes. It all seems easy enough—just write about using the system. But, faced with writing, one suddenly confronts the question, "Exactly what am I supposed to write—how much, how little, what details?" That turns out to be a difficult question to answer. The problem is that writing use cases is fundamentally an exercise in writing prose essays, with all the difficulties in articulating *good* that comes with prose writing in general. It is hard enough to say what a good use case looks like, but we really want to know something harder: how to write them so they will come out being good.

These pages contain the guidelines I use in my use case writing and in coaching: how a person might think, what he or she might observe, to end up with a better use case and use case set.

I include examples of good and bad use cases, plausible ways of writing differently, and, best of all, the good news that a use case need not be the *best* to be *useful*. Even mediocre use cases are useful, more so than are many of the competing requirements files being written. So relax, write something readable, and you will have done your organization a service.

Audience

This book is predominantly aimed at industry professionals who read and study alone, and is therefore organized as a self-study guide. It contains introductory through advanced material: concepts, examples, reminders, and exercises (some with answers, some without).

Writing coaches should find suitable explanations and samples to show their teams. Course designers should be able to build course material around the book, issuing reading assignments as needed. (However, as I include answers to many exercises, they will have to construct their own exam material. :-))

Organization

The book is organized as a general introduction to use cases followed by a close description of the use case body parts, frequently asked questions, reminders for the busy, and end notes.

The **Introduction** contains an initial presentation of key notions, to get the discussion rolling: "What does a use case look like?," "When do I write one?," and "What variations are legal?" The brief answer is that they look different depending on when, where, with whom, and why you are writing them. That discussion begins in this early chapter, and continues throughout the book

Part 1, The Use Case Body Parts, contains chapters for each of the major concepts that need to mastered, and parts of the template that should be written. These include "The Use Case as a Contract for Behavior," "Scope," "Stakeholders and Actors," "Three Named Goal Levels," "Preconditions, Triggers, and Guarantees," "Scenarios and Steps," "Extensions," "Technology and Data Variations," "Linking Use Cases," and "Use Case Formats."

Part 2, Frequently Discussed Topics, addresses particular topics that come up repeatedly: "When Are We Done?," "Scaling Up to Many Use Cases," "CRUD and Parameterized Use Cases," "Business Process Modeling," "The Missing Requirements," "Use Cases in the Overall Process," "Use Case Briefs and eXtreme Programming," and "Mistakes Fixed."

Part 3, Reminders for the Busy, contains a set of reminders for those who have finished reading the book, or already know this material and want to refer back to key ideas. The chapters are organized as "Reminders for Each Use Case," "Reminders for the Use Case Set," and "Reminders for Working on the Use Cases."

There are four appendices: Appendix A discusses "Use Cases in UML" and Appendix B contains "Answers to (Some) Exercises." The book concludes with Appendix C, Glossary; and a list of materials used while writing, Appendix D, Readings.

Heritage of the Ideas

In the late 1960s, Ivar Jacobson invented what later became known as use cases while working on telephony systems at Ericsson. In the late 1980s, he introduced them to the object-oriented programming community, where they were recognized as filling a significant gap in the requirements process. I took Jacobson's course in the early 1990s. While neither he nor his team used my phrases *goal* and *goal failure*, it eventually became clear to me that they had been using these notions. In several comparisons, he and I have found no significant contradictions between his and my models. I have slowly extended his model to accommodate recent insights.

I constructed the Actors and Goals conceptual model in 1994 while writing use case guides for the IBM Consulting Group. It explained away much of the mystery of use cases and provided guidance as to how to structure and write them. The Actors and Goals model has circulated informally since 1995 at *http://members.aol.com/acockburn* and later at *www.usecases.org*, and finally appeared in the *Journal of Object-Oriented Programming* in 1997, in an article I authored entitled "Structuring Use Cases with Goals."

From 1994 to 1999, the ideas stayed stable, even though there were a few loose ends in the theory. Finally, while teaching and coaching, I saw why people were having such a hard time with such a simple idea (never mind that I made many of the same mistakes in my first tries!). These insights, plus a few objections to the Actors and Goals model, led to the explanations in this book and to the Stakeholders and Interests model, which is a new idea presented here.

The Unified Modeling Language (UML) has had little impact on these ideas—and vice versa. Gunnar Overgaard, a former colleague of Jacobson's, wrote most of the UML use case material and kept Jacobson's heritage. However, the UML standards group has a strong drawing-tools influence, with the effect that the textual nature of use cases has been lost in the standard. Gunnar Overgaard and Ivar Jacobson discussed my ideas and assured me that most of what I have to say about a use case fits *within* one of the UML ellipses, and hence neither affects nor is affected by what the UML standard has to say. That means that you can use the ideas in this book quite compatibly with the UML 1.3 use case standard. On the other hand, if you only read the UML standard, which does not discuss the content or writing of a use case, you will not understand what a use case is or how to use it, and you will be led in the dangerous direction of thinking that use cases are a graphical, as opposed to a textual, construction. Since the goal of this book is to show you how to write effective use cases and the standard has little to say in that regard, I have isolated my remarks about UML to Appendix A.

Samples Used

The writing samples in this book were taken from live projects as much as possible, and they may seem slightly imperfect in some instances. I intend to show that they were sufficient to the needs of the project teams that wrote them, and that those imperfections are within the variations and economics permissible in use case writing.

The Addison-Wesley editing crew convinced me to tidy them up more than I originally intended, to emphasize correct appearance over the actual and adequate appearance. I hope you will find it useful to see these examples and recognize the writing that happens on projects. You may apply some of my rules to these samples and find ways to improve them. That sort of thing happens all the time. Since improving one's writing is a never-ending task, I accept the challenge and any criticism.

Use Cases in The Crystal Collection

This is just one in a collection of books, The Crystal Collection for Software Professionals, that highlights lightweight, human-powered software development techniques. Some books discuss a single technique, some discuss a single role on a project, and some discuss team collaboration issues.

Crystal works from two basic principles:

- Software development is a cooperative game of invention and communication. It improves as we develop people's personal skills and increase the team's collaboration effectiveness.
- Different projects have different needs. Systems have different characteristics and are built by teams of differing sizes, with members having differing values and priorities. It is impossible to name one, best way of producing software.

The foundation book for the Crystal Collection, *Software Development as a Cooperative Game*, elaborates the ideas of software development as a cooperative game, of methodology as a coordination of culture, and of methodology families. That book separates the different aspects of methodologies, techniques and activities, work products and standards. The essence of the discussion, as needed for use cases, appears in this book in Section 1.2, Your Use Case Is Not My Use Case on page 7.

Writing Effective Use Cases is a technique guide, describing the nuts-and-bolts of use case writing. Although you can use the techniques on almost any project, the templates and writing standards must be selected according to each project's needs.

Acknowledgments

Thanks to lots of people. Thanks to the people who reviewed this book in draft form and asked for clarification on topics that were causing their clients, colleagues, and students confusion. Special thanks to Russell Walters, a practiced person with a sharp eye for the direct and practical needs of the team, for his encouragement and very specific feedback. Thanks to FirePond and Fireman's Fund Insurance Company for the live use case samples. Pete McBreen, the first to try out the Stakeholders and Interests model, added his usual common sense, practiced eye, and suggestions for improvement. Thanks to the Silicon Valley Patterns Group for their careful reading of early drafts and their educated commentary on various papers and ideas. Mike Jones at the Fort Union Beans & Brew thought up the bolt icon for subsystem use cases.

Susan Lilly deserves special mention for the exact reading she did, correcting everything imaginable: sequencing, content, formatting, and even use case samples. The huge amount of work she contributed is reflected in the much improved final copy.

Other reviewers who contributed detailed comments and encouragement include Paul Ramney, Andy Pols, Martin Fowler, Karl Waclawek, Alan Williams, Brian Henderson-Sellers, Larry Constantine, and Russell Gold. The editors at Addison-Wesley did a good job of cleaning up my usual ungainly sentences and frequent typos.

Thanks to the people in my classes for helping me debug the ideas in the book.

Thanks again to my family, Deanna, Cameron, Sean, and Kieran, and to the people at the Fort Union Beans & Brew who once again provided lots of caffeine and a convivial atmosphere.

More on use cases is at the web sites I maintain: *members.aol.com/acockburn* and *www.usecases.org*. Just to save us some future embarassment, my name is pronounced Cō-burn, with a long o.

This page intentionally left blank

This page intentionally left blank

Chapter 3

Scope

Scope is the word we use for the extent of what we design as opposed to someone else's design job or an already existing design.

Keeping track of the scope of a project, or even just the scope of a discussion, can be difficult. The consultant Rob Thomsett introduced me to a wonderful little tool for tracking and managing scope discussions—the *in/out list*. Absurdly simple and remarkably effective, it can be used to control scope discussions for ordinary meetings as well as project requirements.

Simply construct a table with three columns. The left column contains any topic; the next two columns are labeled "In" and "Out." Whenever there might confusion as to whether a topic is within the scope of the discussion, add it to the table and ask people whether it is in or out. The amazing result, as Rob described and I have seen, is that while is it completely clear to each person in the room whether the topic is in or out, the views are often opposing. Rob relates that sometimes it requires an appeal to the project's steering committee to settle whether a particular topic really is within the scope of work or not. In or out can make a difference of many work-months. Try this technique on your next project or perhaps your next meeting.

Table 3.1 is a sample in/out list we produced for our purchase request tracking system.

Use the in/out list right at the beginning of the requirements or use case writing activity, to separate the things that are within the scope of work from those that are out of scope. Refer to it whenever the discussion seems to be going off track or some requirement is creeping into the discussion that might not belong. Update the chart as you go.

Use the in/out list for topics relating to both the functional scope and the design scope of the system under discussion.

Table 3.1. *A Sample In/Out List*

Topic	In	Out
Invoicing in any form		Out
Producing reports about requests (e.g., by vendor, by part, by person)	In	
Merging requests into one PO	In	
Partial deliveries, late deliveries, wrong deliveries	In	
All new system services, software	In	
Any nonsoftware parts of the system		Out
Identification of any preexisting software that can be used	In	
Requisitions	In	

3.1 FUNCTIONAL SCOPE

Functional scope refers to the services your system offers and that will eventually be captured by the use cases. As you start your project, however, it is quite likely that you won't know it precisely. You are deciding the functional scope at the same time you are identifying the use cases—the two tasks are intertwined. The in/out list helps with this, since it allows you to draw a boundary between what is in and what is out of scope. The other two tools are the *actor-goal list* and the *use case briefs*.

The Actor-Goal List

The actor-goal list names all the user goals that the system supports, showing the system's functional content. Unlike the in/out list, which shows items that are both in and out of scope, the actor-goal list includes only the services that will actually be supported by the system. Table 3.2 is one project's actor-goal list for the purchase request tracking system.

To make this list, construct a table of three columns. Put the names of the primary actors—the actors having the goals—in the left column; put each actor's goals with respect to the system in the middle column; and put the priority, or an initial guess as to the release in which the system will support that goal, in the third column. Update this list continually over the course of the project so that it always reflects the status of the system's functional boundary.

Some people add additional columns—*trigger*, to identify the use cases that will get triggered by time instead of by a person, and *business priority, development complexity,*

Table 3.2. *A Sample Actor-Goal List*

Actor	Task-level Goal	Priority
Any	Check on requests	1
Authorizor	Change authorizations	2
Buyer	Change vendor contacts	3
Requestor	Initiate a request	1
	Change a request	1
	Cancel a request	4
	Mark request delivered	4
	Refuse delivered goods	4
Approver	Complete request for submission	2
Buyer	Complete request for ordering	1
	Initiate PO with vendor	1
	Alert of nondelivery	4
Authorizer	Validate Approver's signature	3
Receiver	Register delivery	1

and *development priority*, so they can separate the business needs from the development costs to derive the development priority.

The actor-goal list is the initial negotiating point between the user representative, the financial sponsor, and the development group. It focuses the layout and content of the project.

The Use Case Briefs

I will keep repeating the importance of managing your energy and working at low levels of precision wherever possible. The actor-goal list is the lowest level of precision in describing system behavior, and it is very useful for working with the total picture of the system. The next level of precision will either be the main success scenario or a *use case brief*.

The use case brief is a two-to-six sentence description of use case behavior, mentioning only the most significant activity and failures. It reminds people of what is going on in the use case. It is useful for estimating work complexity. Teams constructing

Table 3.3. Sample Use Case Briefs

Actor	Goal	Brief
Production Staff	Modify the administrative area lattice	Production staff adds administrative area metadata (administrative hierarchy, currency, language code, street types, etc.) to the reference database. Contact information for source data is cataloged. This is a special case of updating reference data.
Production Staff	Prepare digital cartographic source data	Production staffs convert external digital data to a standard format and validate and correct it in preparation for merging with an operational database. The data is cataloged and stored in a digital source library.
Production and Field Staff	Commit update transactions of a shared check-out to an operational database	Staff applies accumulated update transactions to an operational database. Nonconflicting transactions are committed to the operational database. The application context is synchronized with the operational database. Committed transactions are cleared from the application context, leaving the operational database consistent, with conflicting transactions available for manual/interactive resolution.

from commercial, off-the-shelf components (COTS) use this description in selecting the components. Some project teams, such as those having extremely good internal communications and continual discussion with their users, never write more than these use case briefs for their requirements; they keep the rest of the requirements in the continual discussions, prototypes, and frequently delivered increments.

You can prepare the use case brief as a table, as an extension to the actor-goal list, or directly as part of the use case body in its first draft. Table 3.3 is a sample of briefs, thanks to Paul Ford, Steve Young, and Paul Bouzide of Navigation Technologies.

3.2 DESIGN SCOPE

Design scope is the extent of the system—I would say "spatial extent" if software took up space. It is the set of systems, hardware and software, that we are charged with designing or discussing; it is that boundary. If we are to design an ATM, we are to produce hardware and software that sits in a box—the box and everything in it is ours to design. The computer network that the box will talk to is not ours to design—it is out of the design scope.

From now on, when I write *scope* alone, I mean *design scope*. This is because the functional scope is adequately defined by the actor-goal list and the use cases, while the design scope is a topic of concern in every use case.

As the following story illustrates, it is very important that the writer and reader are in agreement about the design scope for a use case—and correct. The price of being wrong can be a factor of two or more in cost, with disastrous results for the outcome of a contract. The readers of a use case must quickly see what you intend to be inside the system boundary. That will not be obvious just from the name of the use case or the primary actor. Systems of different sizes show up even within the same use case set.

Typically, writers consider the scope of the system to be so obvious that they don't mention it. However, once there are multiple writers and multiple readers, the design scope of a use case is not obvious at all. One writer is thinking of the entire corporation as the scope (see Figure 3.1), one is thinking of all of the company's software systems, one is thinking of the new, client–server system, and one is thinking of only the client or only the server. Readers, having no clue as to what is meant, get lost or misunderstand the document.

What can we do to clear up the misunderstanding?

The only answer I have found is to *label each and every use case with its design scope*, using specific names for the most significant scopes. To be concrete, let us suppose

υ A Short, True Story

To help with constructing a fixed-time, fixed-cost bid of a large system, we were walking through some sample designs. I picked up the printer and spoke its function. The IS expert laughed. "You personal computer people crack me up, "he said," You think we just use a little laser printer to print our invoices? We have a huge printing system, with a chain printer, batch I/O, and everything. We produce invoices by the boxful!"

I was shocked. "You mean the printer is not in the scope of the system?"

"Of course not! We'll use the printing system we already have."

Indeed, we found that there was a complicated interface to the printing system. Our system was to prepare a magnetic tape with things to be printed. Overnight, the printing system would read the tape and print what it could. It would prepare a reply tape describing the results of the printing job, with error records for anything it couldn't print. The following day, our system would read back the results and note what had not been printed correctly. The design job for interfacing to that tape was significant, and completely different from what we had been expecting.

The printing system was not for us to design, but was for us to use. It was out of our design scope. (It was, as described in Section 3.3, a *supporting actor*.) Had we not detected this mistake, we would have written the use case to include it in our scope and turned in a bid to build more system than was needed.

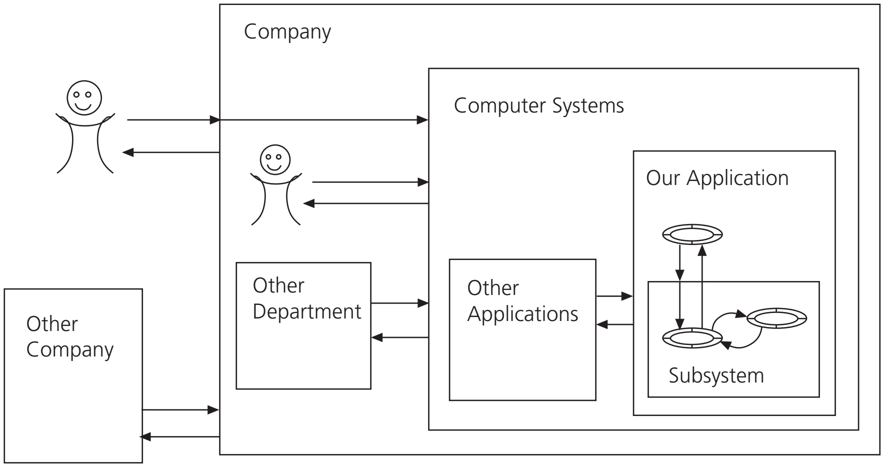


Figure 3.1 *Design scope can be any size*

that MyTelCo is designing a NewApp system, which includes a Searcher subsystem. The design scope names are these:

- *Enterprise* (i.e., *MyTelCo*). You are discussing the behavior of the entire organization or enterprise in delivering the goal of the primary actor. Label the *Scope* field of the use case with the name of the organization—*MyTelCo*—rather than just "the company." If discussing a department, use the department name. Business use cases are written at the enterprise scope.
- *System* (i.e., *NewApp*). This is the piece of hardware or software you are charged with building. Outside the system are all the pieces of hardware, software, and humanity that the system is to interface with.
- *Subsystem* (i.e., *Searcher*). You have opened up the main system and are about to talk about how a piece of it works.

Using Graphical Icons to Highlight the Design Scope

Consider attaching a graphic to the left of the use case title to signal the design scope to readers before they start reading. There are no tools at this time to manage the icons, but I find that drawing them reduces confusion. In this book I label each use case with its appropriate icon to make it easier for you to note its scope.

As you read the following list, remember that a *black-box* use case does not discuss the internal structure of the system under discussion while a *white-box* use case does.

- A *business* use case has the enterprise as its scope. Its graphic is a building. Color it grey ( ) if you treat the whole enterprise as a black box. Color it white ( ) if you talk about the departments and staff within the organization.
- A *system* use case has a computer system as its scope. Its graphic is a box. Color it grey ( ) if you treat it as a black box, white ( ) if you reveal how its componentry works.
- A *component* use case is about a subsystem or component of the system under design. Its graphic is a bolt . See Use Cases 13 through 17 for an example.

Design Scope Examples

I offer three examples to illustrate systems at different scopes.

(1) Enterprise-to-System Scope

Suppose that we work for telephone company, *MyTelCo*, which is designing a new system, *Acura*, to take orders for services and upgrades. Acura consists of a workstation connected to a server. The server will be connected to a mainframe running the old system, *BSSO*. BSSO is just a terminal attached to the mainframe. We are not allowed to make any changes to it; we can only use its existing interfaces.

The primary actors for Acura include the customer, the clerk, various managers, and BSSO (we are clear that BSSO is not within our scope).

Let's find a few of the goals the system should support. The most obvious is "Add a new service." We decide that the primary actor for that is the company clerk, acting on behalf of the customer. We sit down to write a few use cases.

The immediate question is "What is the system under discussion?" It turns out that there are two that interest us:

- *MyTelCo*. We are interested in the question, "What does MyTelCo's service look like to the customer, showing the new service implementation in its complete form, from initial request to implementation and delivery?" This question is of double interest. The company managers will want to see how the new system appears to the outside world, and the implementation team will want to see the context in which the new system will sit.

  This use case will be written at the enterprise scope ( ), with the Scope field labeled MyTelCo and the use case written without mention of company-internal players (no clerks, no departments, no computers). This sort of use case is often referred to as a *business use case*, since it is about the business.
- *Acura*. We are interested in the question, "How does Acura's service appear, at its interface to the clerk or customer on one side and to the BSSO system on the

other side?" This is the use case the designers care most about, since it states exactly what they are to build. The use case will be written at the system scope ( ), with the Scope field labeled "Acura." It will freely mention clerks and departments and other computer systems, but not the workstation and the server subsystems.

We produce two use cases. To avoid having to repeat the same information twice, we write the enterprise use case at a higher level (the kite symbol), showing MyTelCo responding to the request, delivering it, and perhaps even charging for it and getting paid. The purpose of the enterprise use case is to show the context around the new system. Then we describe in detail the 5- to 20-minute handling of the request in the user-goal use case having Acura as its scope.

Use Case 6 Add New Service (Enterprise)

Primary Actor: Customer
Scope: MyTelCo
Level: Summary

1. Customer calls MyTelCo, requests new service . . .
2. MyTelCo delivers . . . etc. . . .

Use Case 7 Add New Service (Acura)

Primary Actor: Clerk for external customer
Scope: Acura
Level: User goal

1. Customer calls in, clerk discusses request with customer.
2. Clerk finds customer in Acura.
3. Acura presents customer's current service package . . . etc. . . .

No use case will be written with a scope of Acura workstation or Acura server, as these are not of interest to us. Later, someone in the design team may choose to document Acura's subsystem design using use cases. At that time, they will write two use cases, one with a scope of Acura workstation, the other with a scope of Acura server. My experience is that these use cases are never written, since there are other adequate techniques for documenting subsystem architecture.

(2) Many Computers to One Application

The following is a less common situation, but one that is very difficult. Let us build onto the MyTelCo situation.

> Acura will slowly replace BSSO. New service requests will be put into Acura and then modified using BSSO. Over time, Acura will take on more function. The two systems must co-exist and synchronize with each other. Thus, use cases have to be written for both systems: Acura being entirely new and BSSO being modified to synchronize with it.

The difficulty in this situation is that there are four use cases, two for Acura and two for BSSO. There is one use case for each system having the clerk as primary actor and one having the other computer system as the primary actor. There is no way to avoid these four use cases, but people looking at them get confused because they look redundant.

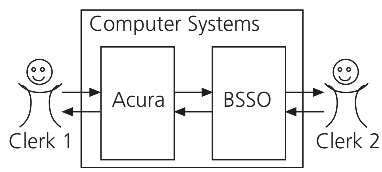
To document this situation, I first write a summary-level use case whose scope is both computer systems. This gives me a chance to document their interactions over time. In that use case, I reference the specific use cases that comprise each system's requirements. This first use case will be of the white-box type (note the white-box symbol).

The situation is complicated enough that I also include diagrams of each use case's scope.

Use Case 8 Enter and Update Requests (Joint System)

Primary Actor: Clerk for external customer

Scope: Computer systems, including Acura and BSSO (see diagram)



Level: Summary

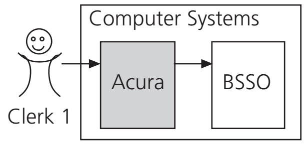
Main Success Scenario:

1. Clerk adds new service into Acura.
2. Acura notes new service request in BSSO.
3. Some time later, Clerk updates service request in BSSO.
4. BSSO notes the updated request in Acura.

The four sub use cases are all user-goal use cases and get marked with the sea-level symbol. Although they are all system use cases, they are for different systems—hence the diagrams. In each diagram, I circle the primary actor and shade the SuD. The use cases are black-box this time, since they are requirements for new work. In addition, I give them slightly different verb names, using the verb "note" to indicate one system synchronizing with the other.

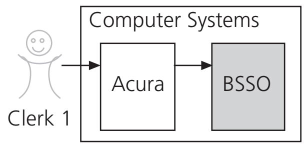
Use Case 9 Add New Service (into Acura)

Primary Actor: Clerk for external customer
Scope: Acura
Level: User goal
. . . use case body follows. . .



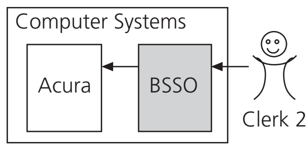
Use Case 10 Note New Service Request (in BSSO)

Primary Actor: Acura
Scope: BSSO
Level: User goal
. . . use case body follows. . .



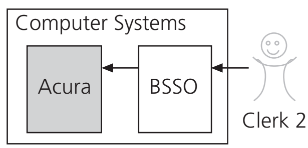
Use Case 11 Update Service Request (in BSSO)

Primary Actor: Clerk for external customer
Scope: BSSO
Level: User goal
. . . use case body follows. . .



Use Case 12 Note Updated Request (in Acura)

Primary Actor: BSSO
Scope: Acura
Level: User Goal
. . . use case body follows. . .



If you are using UML use case diagrams, you might draw the summary-level use case instead of writing it. That still does not reduce the confusion within the four user-goal use cases, so you should still carefully mark their primary actor, scope, and level, and possibly still draw the scope diagrams within the use cases.

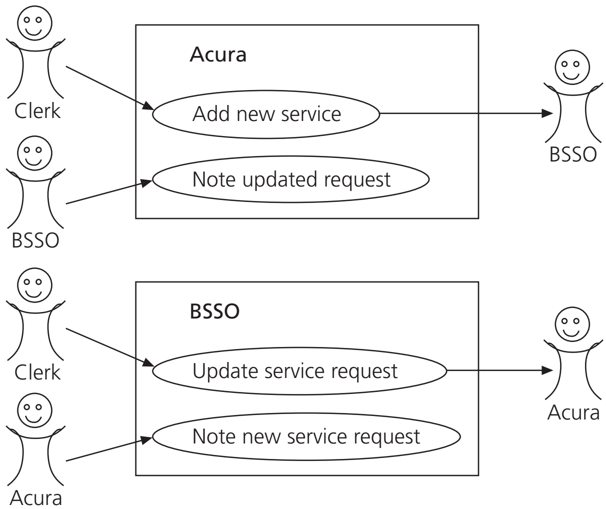


Figure 3.2 Use case diagrams for Acura–BSSO*. This is the UML style of denoting the interactions between the two systems. The upper section shows that BSSO is a supporting actor to one use case of Acura and a primary actor to another use case. In the lower diagram, the roles are reversed.*

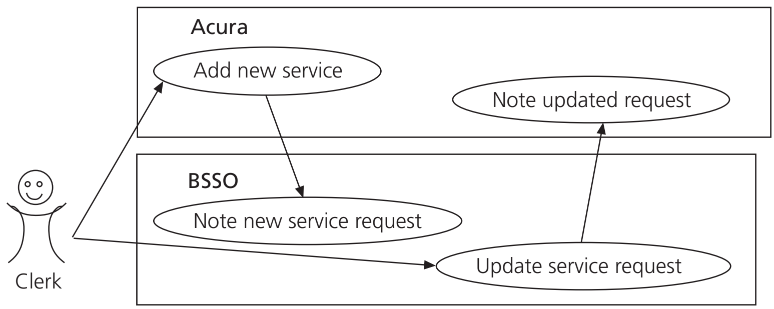


Figure 3.3 A combined use case diagram for Acura-BSSO*. This drawing shows the relationships of the four use cases most clearly, but is nonstandard, since it shows one system's use case triggering another system's use case.*

Personally, I do not find that this eliminates much confusion. I would consider drawing the nonstandard use case diagram in Figure 3.3 to show the connection between the two systems. This diagram is clearer but harder to maintain over time. Draw whichever you and your readers find communicates best for you.

(3) Nuts and Bolts Use Cases

At the far end of the scale, let's look at the way one group documented their design framework with use cases. They started with an 18-page, diagram-loaded description of the rules for their framework. They decided it was too hard to read and experimented with use cases as the descriptive technique.

The group spent one week on the task. First they drafted 40 use cases to make sure they had captured all the requests their framework would handle. Using extensions and the data variations list, they revised those down to just six.

Most readers will find these use cases incomprehensible because they are not in that business. However, I expect some readers to be technical programmers looking for ways to document their designs, so I include these use cases to show how this group documented an internal architecture and how they made use of the variations list. I find them fairly easy to read, given the complexity of their problem. Notice that sub use cases are underlined. Thanks to Dale Margel in Calgary for the writing.

General Description:

The overall architecture must be able to handle concurrent tasks. To do this, it must support Process Threads and Resource Locking. These services are handled by the Concurrency Service Framework (CSF). CSF is used by client objects to protect critical sections of code from unsafe access by multiple processes.

Use Case 13 Serialize Access to a Resource

Primary Actor: Service Client object

Scope: Concurrency Service Framework (CSF)

Level: User goal

Main Success Scenario:

1. Service Client asks a Resource Lock to give it specified access.
2. The Resource Lock returns control to the Service Client so that it may use the Resource.
3. Service Client uses the Resource.
4. Service Client informs the Resource Lock that it is finished with the Resource.
5. Resource Lock cleans up after the Service Client.

Extensions:

2a. Resource Lock finds that Service Client already has access to the resource:

 2a1. Resource Lock applies a lock conversion policy (Use Case 14) to the request.

2b. Resource Lock finds that the resource is already in use:

 2b1. The Resource Lock applies a compatibility policy (Use Case 15) to grant access to the Service Client.

2c. Resource Locking Holding time limit is nonzero:

 2c1. Resource Lock starts the holding timer.

3a. Holding Timer expires before the Client informs the Resource Lock that it is finished:

  3a1. Resource Lock sends an Exception to the Client's process.

  3a2. Fail!

4a. Resource Lock finds nonzero lock count on Service Client:

  4a1. Resource Lock decrements the reference count of the request.

  4a2. Success!

5a. Resource Lock finds that the resource is currently not in use:

  5a1. Resource Lock applies an access selection policy (Use Case 16) to grant access to any suspended service clients.

5b. Holding Timer is still running:

  5b1. Resource Lock cancels Holding Timer.

Technology and Data Variations List:

1. The specified requested access can be:

  υ For exclusive access

  υ For shared access

2c. The lock holding time-out can be specified by:

  υ The Service Client

  υ A Resource Locking policy

  υ A global default value

Use Case 14 Apply a Lock Conversion Policy

Primary Actor: Client object

Scope: Concurrency Service Framework (CSF)

Level: Subfunction

Main Success Scenario:

1. Resource Lock verifies that request is for exclusive access.
2. Resource Lock verifies that Service Client already has shared access.
3. Resource Lock verifies that there is no Service Client waiting to upgrade access.
4 Resource Lock verifies that there are no other Service Clients sharing the resource.
5. Resource Lock grants Service Client exclusive access to the resource.
6. Resource Lock increments Service Client lock count.

Extensions:

1a. Resource Lock finds that the request is for shared access:

  1a1. Resource Lock increments lock count on Service Client.

  1a2. Success!

2a. Resource Lock finds that the Service Client already has exclusive access:

  2a1. Resource Lock increments lock count on Service Client.

  2a2. Success!

3a. Resource Lock finds that there is another Service Client waiting to upgrade access:
 3a1. Signal Service Client that requested access could not be granted.
 3a2. Fail!
4a. Resource Lock finds that there are other Service Clients using the resource:
 4a1. Resource Lock makes Service Client wait for resource access (Use Case 17).

Use Case 15 Apply an Access Compatibility Policy

Primary Actor: Service Client object
Scope: Concurrency Service Framework (CSF)
Level: Subfunction
Main Success Scenario:
1. Resource Lock verifies that request is for shared access.
2. Resource Lock verifies that all current usage of resource is for shared access.

Extensions:
2a. Resource Lock finds that the request is for exclusive access:
 2a1. Resource Lock makes Service Client wait for resource access (Use Case 17) (the process is resumed later by the Lock serving strategy).
2b. Resource Lock finds that the resource is being exclusively used:
 2b1. Resource Lock makes Service Client wait for resource access (Use Case 17)

Variations:
1. The compatibility criterion may be changed.

Use Case 16 Apply an Access Selection Policy

Primary Actor: Client object
Scope: Concurrency Service Framework (CSF)
Level: Subfunction
Main Success Scenario:
Goal in Context: Resource Lock must determine which (if any) waiting requests should be served.
Note: This strategy is a point of variability.
1. Resource Lock selects oldest waiting request.
2. Resource Lock grants access to selected request(s) by making its process runnable.

Extensions:
1a. Resource Lock finds no waiting requests:
 1a1. Success!
1b. Resource Lock finds a request waiting to be upgraded from a shared to an exclusive access:
 1b1. Resource Lock selects the upgrading request.

1c. Resource Lock selects a request that is for shared access:
    1c1. Resource repeats [Step 1] until the next one is for exclusive access.

Variations:

1. The selection ordering criterion may be changed.

Use Case 17 Make Service Client Wait for Resource Access

Primary Actor: Client object
Scope: Concurrency Service Framework (CSF)
Level: Subfunction
Main Success Scenario:
Used By: CC 2,4 Resource Locking:

1. Resource Lock suspends Service Client process.
2. Service Client waits until resumed.
3. Service Client process is resumed.

Extensions:

1a. Resource Lock finds that a waiting time-out has been specified:
    1a1. Resource Lock starts timer.
2a. Waiting Timer expires:
    2a1. Signal Service Client that requested access could not be granted.
    2a2. Fail!

Technology and Data Variations List:

1a1. The Lock waiting time-out can be specified by:
- The Service Client
- A Resource Locking policy
- A global default value

3.3 THE OUTERMOST USE CASES

In the Enterprise-to-System Scope subsection on page 41, I recommend writing two use cases, one for the system under design and one at an outer scope. Now we can get more specific about that: For each use case, find the outermost design scope at which it still applies and write a summary-level use case at that scope.

The use case is written to a design scope. Usually, you can find a wider design scope that still has the primary actor outside it. If you keep widening the scope, you reach the point at which widening it farther would bring the primary actor inside. That is the *outermost scope*. Sometimes the outermost scope is the enterprise, sometime the department, and sometimes just the computer. Often, the computer department is the primary actor on computer security use cases, the marketing department is the primary actor on advertising use cases, and the customer is the primary actor on the main system function use cases.

Typically, there are only two to five outermost use cases for the entire system, so not every use case gets written twice. There are so few of them because each one merges the primary actors having similar goals on the same design scope, and pulls together all the lower-level use cases for those actors.

I highly recommend writing the outermost use cases because it takes very little time and provides excellent context for the use case set. The outermost use cases show how the system ultimately benefits the most external users of the system; they also provide a table of contents for browsing through the system's behavior.

Let's visit the outermost use cases for MyTelCo and its Acura system.

> MyTelCo decides to let web-based customers access Acura directly to reduce the load on the clerks. Acura will also report on the clerks' sales performance. Someone will have to set security access levels for customers and clerks. We have four use cases: *Add Service (by Customer)*, *Add Service (by Clerk)*, *Report Sales Performance*, and *Manage Security Access*.

We know we will have to write all four use cases with Acura as the scope of the SuD. We need to find the outermost scope for each of them.

The customer is clearly outside MyTelCo, so there is one outermost use case with the customer as primary actor and MyTelCo as scope. This use case will be at the summary level, showing MyTelCo as a black box, responding to the customer's request, delivering the service, and so on. In fact, the use case is outlined in Use Case 6, *Add New Service (Enterprise),* on page 42.

The clerk is inside MyTelCo. The outermost scope for *Add Feature (by Staff)* is All Computer Systems. This use case will gather all the interactions the clerks have with the computer systems. I would expect all the clerks' user-goal use cases to be in this outermost use case, along with a few subfunction use cases, such as *Log In* and *Log Out*.

Report Sales Performance has the Marketing Department as the ultimate primary actor. The outermost use case is at scope Service Department and shows the Marketing Department interacting with All Computer Systems and the Service Department for setting up performance bonuses, reporting sales performance, and so on.

Manage Security Access has the Security or IT Department as its ultimate primary actor and either the IT Department or All Computer Systems as the outermost design scope. The use case references all the ways the Security Department uses All Computer Systems to set and track security issues.

Notice that these four outermost use cases cover security, marketing, service, and customers, using Acura in all the ways that it operates. It is unlikely that more than these four need to be written for the Acura system, even if there are a hundred lower-level use cases to write.

3.4 USING THE SCOPE-DEFINING WORK PRODUCTS

You are defining the functional scope for your upcoming system, brainstorming, and moving between several work products on the whiteboard. On one part of the whiteboard, you have the in/out list to keep track of your scoping decisions ("No, Bob, we decided that a new printing system is out of scope—or do we need to revisit that entry in the in/out list?"). You have the actors and their goals in a list. You have a drawing of the design scope, showing the people, organizations, and systems that will interact with the system under discussion.

You find that you are evolving them all as you move between them, working out what you want your new system to do. You think you know what the design scope is, but a change in the in/out list moves the boundary. Now you have a new primary actor, and the goal list changes.

Sooner or later, you will probably find that you need a fourth item: a *vision statement* for the new system. The vision statement holds together the overall discussion. It helps you decide whether something should be in scope or out of scope in the first place.

When you are done, you have the four work products that bind the system's scope:

- Vision statement
- Design scope drawing
- In/out list
- Actor-goal list

What I want you to take from this short discussion is that the four work products are intertwined and that you are likely to change them all while establishing the scope of the work to be done.

3.5 EXERCISES

Design Scope

3.1. Name at least five system design scopes that the following user story fragment could be about: "*. . . Jenny is standing in front of her bank's ATM. It is dark. She has entered her PIN and is looking for the Enter button . . .*"

3.2. Draw a picture of the multiple scopes for an ATM, including hardware and software.

3.3. What system are you, personally, writing requirements for? What is its extent? What is inside it? What is outside it that it must communicate with? What is the system that encloses it, and what is outside that containing system that *it* must communicate with? Give the enclosing system a name.

3.4. Draw a picture of the multiple scopes for the Personal Advisors/Finance (PAF) system. (See Excercise 4.4.)

3.5. Draw a picture of the multiple scopes for a web application in which a user's workstation is connected through the web to your company's web server, which is attached to a legacy mainframe system.

3.6. Describe the difference between *enterprise-scope white-box business use cases* and *enterprise-scope black-box business use cases*.

Index