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

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 on recycled and acid-free paper.

ISBN 0-201-70225-8

Text printed in the United States at Edwards Brothers in Ann Arbor, Michigan.

20th Printing February 2008

Chapter 1

Introduction

What do use cases look like?
Why do different project teams need different writing styles?
Where do use cases fit into the requirements gathering work?
How do we warm up for writing use cases?

It will be useful to have some answers to these questions before getting into the details of use cases themselves.

1.1 WHAT IS A USE CASE (MORE OR LESS)?

A use case captures a contract between the stakeholders of a system about its behavior. The use case describes the system's behavior under various conditions as the system responds to a request from one of the stakeholders, called the *primary actor*. The primary actor initiates an interaction with the system to accomplish some goal. The system responds, protecting the interests of all the stakeholders. Different sequences of behavior, or scenarios, can unfold, depending on the particular requests made and the conditions surrounding the requests. The use case gathers those different scenarios together.

Use cases are fundamentally a text form, although they can be written using flow charts, sequence charts, Petri nets, or programming languages. Under normal circumstances, they serve as a means of communication from one person to another, often among people with no special training. Simple text is, therefore, usually the best choice.

The use case, as a form of writing, can stimulate discussion within a team about an upcoming system. The team might or might not document the actual requirements with use cases. Another team might document their final design with use cases. All of the above might be done for a system as large as an entire company or as small as a piece of a software application program. What is interesting is that the same basic rules of writing apply to all of these situations, even though the teams will write with different amounts of rigor and at different levels of technical detail.

When use cases document an organization's business processes, the *system under discussion* (SuD) is the organization itself. The stakeholders are the company shareholders, customers, vendors, and government regulatory agencies. The primary actors include the company's customers and perhaps their suppliers.

When use cases record behavioral requirements for a piece of software, the SuD is the computer program. The stakeholders are the people who use the program, the company that owns it, government regulatory agencies, and other computer programs. The primary actor is the user sitting at the computer screen or another computer system.

A well-written use case is easy to read. It consists of sentences written in only one grammatical form—a simple action step—in which an actor achieves a result or passes information to another actor. Learning to read a use case should not take more than a few minutes.

Learning to write a good use case is harder. The writer has to master three concepts that apply to every sentence in the use case and to the use case as a whole. Odd though it may seem at first glance, keeping these three concepts straight is not easy. The difficulty shows up as soon as you start to write your first use case. The three concepts are

- *Scope:* What is really the system under discussion?
- *Primary actor:* Who has the goal?
- *Level:* How high- or low-level is that goal?

Several examples of use cases follow. The parts of a use case are described in the next chapter. For now, remember these summary definitions:

- *Actor:* anyone or anything with behavior.
- *Stakeholder:* someone or something with a vested interest in the behavior of the system under discussion (SuD).
- *Primary actor:* the stakeholder who or which initiates an interaction with the SuD to achieve a goal.
- *Use case:* a contract for the behavior of the SuD.
- *Scope:* identifies the system that we are discussing.
- *Preconditions and guarantees:* what must be true before and after the use case runs.

- *Main success scenario:* a case in which nothing goes wrong.
- *Extensions:* what can happen differently during that scenario.
- Numbers in the extensions refer to the step numbers in the main success scenario at which each different situation is detected (for instance, steps 4a and 4b indicate two different conditions that can show up at step 4).
- When a use case references another use case, the referenced use case is <u>underlined</u>.

The first use case describes a person about to buy some stocks over the web. To signify that we are dealing with a goal to be achieved in a single sitting, I mark the use case as being at the *user-goal level*, and tag it with the *sea-level* symbol, . The second use case describes a person trying to get paid for a car accident, a goal that takes longer than a single sitting. To show this, I mark the use case as being at the *summary level*, and tag it with the *above-sea-level* symbol, . These symbols are explained in Chapter 5, and summarized inside the front cover.

The first use case describes the person's interactions with a program ("PAF") running on a workstation connected to the Web. The black-box symbol, , indicates that the system being discussed is a computer system. The second use case describes a person's inte tion with a company, which I indicate with a building symbol, . The use of symbols is completely optional. Labeling the scope and level is not.

Here are Use Cases 1 and 2.

Use Case 1 Buy Stocks over the Web

Primary Actor: Purchaser
Scope: Personal Advisors / Finance package (PAF)
Level: User goal
Stakeholders and Interests:
Purchaser—wants to buy stocks and get them added to the PAF portfolio automatically.
Stock agency—wants full purchase information.
Precondition: User already has PAF open.
Minimal Guarantee: Sufficient logging information will exist so that PAF can detect that something went wrong and ask the user to provide details.
Success Guarantee: Remote web site has acknowledged the purchase; the logs and the user's portfolio are updated.
Main Success Scenario:
1. Purchaser selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwab, etc.) from user.
3. PAF opens web connection to the site, retaining control.
4. Purchaser browses and buys stock from the web site.
5. PAF intercepts responses from the web site and updates the purchaser's portfolio.
6. PAF shows the user the new portfolio standing.

Extensions:
2a. Purchaser wants a web site PAF does not support:
  2a1. System gets new suggestion from purchaser, with option to cancel use case.
3a. Web failure of any sort during setup:
  3a1. System reports failure to purchaser with advice, backs up to previous step.
  3a2. Purchaser either backs out of this use case or tries again.
4a. Computer crashes or is switched off during purchase transaction:
  4a1. (What do we do here?)
4b. Web site does not acknowledge purchase, but puts it on delay:
  4b1. PAF logs the delay, sets a timer to ask the purchaser about the outcome.
5a. Web site does not return the needed information from the purchase:
  5a1. PAF logs the lack of information, has the purchaser <u>update questioned purchase</u>.

Use Case 2 Get Paid for Car Accident

Primary Actor: Claimant
Scope: Insurance company ("MyInsCo")
Level: Summary
Stakeholders and Interests:
Claimant—to get paid the most possible.
MyInsCo—to pay the smallest appropriate amount.
Department of Insurance—to see that all guidelines are followed.
Precondition: None.
Minimal Guarantees: MyInsCo logs the claim and all activities.
Success Guarantees: Claimant and MyInsCo agree on amount to be paid; claimant gets paid that.
Trigger: Claimant submits a claim.
Main Success Scenario:
1. Claimant submits claim with substantiating data.
2. Insurance company verifies claimant owns a valid policy.
3. Insurance company assigns agent to examine case.
4. Insurance company verifies all details are within policy guidelines.
5. Insurance company pays claimant and closes file.

Extensions:
1a. Submitted data is incomplete:
  1a1. Insurance company requests missing information.
  1a2. Claimant supplies missing information.
2a. Claimant does not own a valid policy:
  2a1. Insurance company denies claim, notifies claimant, records all this, terminates proceedings.
3a. No agents are available at this time.
  3a1. (What does the insurance company do here?)
4a. Accident violates basic policy guidelines:
  4a1. Insurance company denies claim, notifies claimant, records all this, terminates proceedings.
4b. Accident violates some minor policy guidelines:
  4b1. Insurance company begins negotiation with claimant as to amount of payment to be made.

Most of the use cases in this book come from live projects, and I have been careful not to touch them up (except to add the scope and level tags if they weren't there). I want you to see samples of what works in practice, not just what is attractive in the classroom. People rarely have time to make the use cases formal, complete, and pretty. They usually only have time to make them "sufficient," which is all that is necessary. I show these real samples because you rarely will be able to generate perfect use cases yourself, despite my coaching. Even I can't write perfect use cases most of the time.

Use Case 3 was written by Torfinn Aas of Central Bank of Norway for his colleague, his user representative, and himself. It shows how the form can be modified without losing value. The writer added additional business context to the story, illustrating how the computer application operates in the course of a working day. This was practical, as it saved having to write a separate document describing the business process. It confused no one and was informative to the people involved.

Use Case 3 Register Arrival of a Box

RA—"Receiving Agent"
RO—"Registration Operator"
Primary Actor: RA
Scope: Nightime Receiving Registry Software
Level: User goal
Main Success Scenario:
1. RA receives and opens box (box ID, bags with bag IDs) from Transport Company (TC)
2. RA validates box ID with TC registered IDs.
3. RA maybe signs paper form for delivery person.
4. RA registers box's arrival into system, which stores:
   RA ID
   Date, time
   Box ID
   Transport Company
   <Person name?>
   # bags (With bag IDs?)
   <Estimated value?>
5. RA removes bags from box, puts on cart, takes to RO.

Extensions:
2a. Box ID does not match transport company ID.
4a. Fire alarm goes off and interrupts registration.
4b. Computer goes down.
    Leave money on desk and wait for computer to come back up.

Variations:
4'. With and without Person ID.
4''. With and without estimated value.
5'. RA leaves bags in box.

1.2 YOUR USE CASE IS NOT MY USE CASE

Use cases are a form of writing that can be put to work in different situations, including the following:

- To describe a business's work process.
- To focus discussion *about* upcoming software system requirements, but not to be the requirements description.
- To be the functional requirements for a system.
- To document the design of the system.
- To be written in a small, close-knit group, or in a large or distributed group.

Each situation calls for a slightly different writing style. Here are the major subforms of use cases, driven by their *purpose*.

- A close-knit group gathering requirements, or a larger group discussing upcoming requirements, will write *casual* as opposed to *fully dressed* use cases, which are written by larger, geographically distributed, or formally inclined teams. The casual form "short-circuits" the use case template, making the use cases faster to write (see more on this later). Use Cases 1 through 3 are fully dressed, using the full use case template and step-numbering scheme. An example of casual form is shown in Use Case 4.
- Business process people write *business* use cases to describe the operations of their business, while a hardware or software development team writes *system* use cases for their requirements. The design team may write other *system* use cases to document their design or to break down the requirements for small subsystems.
- Depending on the level of view needed at the time, the writer will choose to describe a multi-sitting, or *summary*, goal; a single-sitting, or *user* goal; or a part of a user goal, or *subfunction*. Communicating which of these is being described is so important that my students have come up with two different gradients to describe them: by height relative to sea level (above sea level, at sea level, underwater), and by color (white, blue, indigo).
- Anyone writing requirements for a new system to be designed, whether business process or computer system, will write *black-box* use cases—those that do not discuss the innards of the system. Business process designers will write *white-box* use cases, showing how the company or organization runs its internal processes. The technical development team might do the same to document the operational context for the system they are about to design, and they might write white-box use cases to document the workings of the system they just designed.

It is wonderful that the use case writing form can be used in such varied situations, but it is confusing. Several of you sitting together are likely to find yourselves disagreeing on some matter of writing, just because your use cases are for different purposes. And you are likely to encounter several combinations of those characteristics over time.

Finding a general way to talk about use cases, while allowing all those variations, will plague us throughout the book. The best I can do is outline the issue now and let the examples speak for themselves.

You may want to test yourself on the use cases in this chapter. Use Cases 1, 3, and 5 were written for system requirements purposes, so they are the fully dressed, black-box, system type, at the user-goal level. Use Case 4 is the same, but casual, not fully dressed. Use Case 2, written as the context-setting use case for business process documentation, is fully dressed, black-box, and at the summary level.

The largest difference between use case formats is how "dressed up" they are. Consider these quite different situations:

- A team is working on software for a large, mission-critical project. They decide that the extra ceremony is worth the extra cost, so (a) the use case template needs to be longer and more detailed, (b) the writing team should write in the same style to reduce ambiguity and misunderstanding, and (c) the reviews should be tighter to more closely scrutinize the use cases for omissions and ambiguities. Having little tolerance for mistakes, they decide to reduce tolerances (variation between people) in the use case writing as well.
- A team of three to five people is building a system whose worst damage is the loss of comfort, easily remedied with a phone call. They consider all the ceremony a waste of time, energy, and money. They therefore choose (a) a simpler template, (b) to tolerate more variation in writing style, and (c) fewer and more forgiving reviews. The errors and omissions in the writing are to be caught by other project mechanisms, probably conversations among teammates and with users. They can tolerate more errors in their written communication and so more casual writing and more variation between people.

Neither is wrong. Such choices must be made on a project-by-project basis. This is the most important lesson that I, as a methodologist, have learned in the last five years. Of course we have been saying, "One size doesn't fit all" for years, but just how to translate that into concrete advice has remained a mystery.

The mistake is getting too caught up in precision and rigor when they are not needed, which will cost your project a lot in time and energy. As Jim Sawyer wrote in an email discussion, ". . . as long as the templates don't feel so formal that you get lost in a recursive descent that worm-holes its way into design space. If that starts to occur, I say strip the little buggers naked and start telling stories and scrawling on napkins."

I have come to the conclusion that just one use case template isn't enough. There must be at least two: a casual one for low-ceremony projects and a fully dressed one for higher-ceremony projects. Any one project will adapt one of the two forms for its situation. The next two use cases are the same but written in the two styles.

Use Case 4 Buy Something (Casual Version)

The Requestor initiates a request and sends it to her or his Approver. The Approver checks that there is money in the budget, checks the price of the goods, completes the request for submission, and sends it to the Buyer. The Buyer checks the contents of storage, finding the best vendor for goods. The Authorizer validates Approver's signature. The Buyer completes request for ordering, initiates PO with Vendor. The Vendor delivers goods to Receiving, gets receipt for delivery (out of scope of system under design). The Receiver registers delivery, sends goods to Requestor. The Requestor marks request delivered.

At any time prior to receiving goods, the Requestor can change or cancel the request. Canceling it removes it from any active processing (deletes it from system?). Reducing the price leaves it intact in processing. Raising the price sends it back to the Approver.

Use Case 5 Buy Something (Fully Dressed Version)

Primary Actor: Requestor

Goal in Context: Requestor buys something through the system, gets it. Does not include paying for it

Scope: Business—the overall purchasing mechanism, electronic and nonelectronic, as seen by the people in the company

Level: Summary

Stakeholders and Interests:

Requestor: Wants what he/she ordered, easy way to do that.

Company: Wants to control spending but allow needed purchases.

Vendor: Wants to get paid for any goods delivered.

Precondition: none

Minimal Guarantees: Every order sent out has been approved by a valid authorizer. Order was tracked so that company can be billed only for valid goods received.

Success Guarantees: Requestor has goods, correct budget ready to be debited.

Trigger: Requestor decides to buy something.

Main Success Scenario:

1. *Requestor:* <u>initiate a request</u>.
2. *Approver:* check money in budget, check price of goods, <u>complete request for submission</u>.

3. *Buyer:* check contents of storage, find best vendor for goods.
4. *Authorizer:* validate Approver's signature.
5. *Buyer:* complete request for ordering, initiate PO with Vendor.
6. *Vendor:* deliver goods to Receiving, get receipt for delivery (out of scope of system under design).
7. *Receiver:* register delivery; send goods to Requestor.
8. *Requestor:* mark request delivered.

Extensions:

1a. Requestor does not know vendor or price: Leave those parts blank and continue.

1b. At any time prior to receiving goods, Requestor can change or cancel request:
   Canceling it removes it from active processing (Delete from system?).
   Reducing price leaves it intact in processing.
   Raising price sends it back to Approver.

2a. Approver does not know vendor or price: Leave blank and let Buyer fill in or callback.

2b. Approver is not Requestor's manager: Still OK as long as Approver signs.

2c. Approver declines: Send back to Requestor for change or deletion.

3a. Buyer finds goods in storage: Send those up, reduce request by that amount, and carry on.

3b. Buyer fills in Vendor and price, which were missing: Request gets resent to Approver.

4a. Authorizer declines Approver: Send back to Requestor and remove from active processing. (What does this mean?)

5a. Request involves multiple Vendors: Buyer generates multiple POs.

5b. Buyer merges multiple requests: Same process, but mark PO with the requests being merged.

6a. Vendor does not deliver on time: System does alert of non-delivery.

7a. Partial delivery: Receiver marks partial delivery on PO and continues.

7b. Partial delivery of multiple-request PO: Receiver assigns quantities to requests and continues.

8a. Goods are incorrect or improper quality: Requestor refuses delivered goods. (What does this mean?)

8b. Requestor has quit the company: Buyer checks with Requestor's manager: either reassign Requestor or return goods and cancel request.

Technology and Data Variations List: None.

Priority: Various

Releases: Several

Response Time: Various

Frequency of Use: 3/day

Channel to Primary Actor: Internet browser, mail system, or equivalent

Secondary Actors: Vendor

Channels to Secondary Actors: Fax, phone, car
Open Issues:
When is a canceled request deleted from the system?
What authorization is needed to cancel a request?
Who can alter a request's contents?
What change history must be maintained on requests?
What happens when Requestor refuses delivered goods?
How does a requisition work differently from an order?
How does ordering reference and make use of the internal storage?

I hope it is clear that simply saying, "We write use cases on this project" does not say very much, and that any recommendation or process definition that simply says, "Write use cases" is incomplete. A use case valid on one project is not valid on another project. More must be said about whether fully dressed or casual use cases are being used, which template parts and formats are mandatory, and how much tolerance across writers is permitted.

The full discussion of tolerance and variation across projects is described in *Software Development as a Cooperative Game* (Cockburn, 2001). We don't need the full discussion to learn how to write use cases. We do need to separate the *writing technique* from *use case quality* and *project standards*.

"Techniques" are the moment-to-moment thinking or actions people use while constructing use cases. This book is largely concerned with technique: how to think, how to phrase sentences, in what sequence to work. The fortunate thing about techniques is that they are largely independent of the size of the project. A person skilled in a technique can apply it on projects both large and small.

"Quality" says how to tell whether the use cases that have been written are acceptable for their purpose. In this book, I describe the best way of writing I have seen for each use case part, across use cases, and for different purposes. In the end, though, the way you evaluate the quality of your use cases depends on the purpose, tolerance, and amount of ceremony you choose.

"Standards" say what the people on the project agree to when writing their use cases. In this book, I discuss alternative reasonable standards, showing different templates and different sentence and heading styles. I come out with a few specific recommendations, but ultimately it is for the organization or project to set or adapt the standards and to decide how strongly to enforce them.

In most of this book, I deal with the most demanding problem—writing precise requirements. In the following eyewitness account, Steve Adolph, a consultant, describes using use cases to *discover* requirements rather than to document them.

◆ Steve Adolph: "Discovering" Requirements in New Territory

Use cases are typically offered as a way to capture and model known functional requirements. People find the storylike format easier to comprehend than long shopping lists of traditional requirements. They actually understand what the system is supposed to do.

But what if no one knows what the system is supposed to do? The automation of a process usually changes the process. The printing industry was recently hit with one of the biggest changes since the invention of offset printing—the development of direct-to-plate/direct-to-press technology. Formerly, setting up a printing press was a labor-intensive, multi-step process. Direct-to-plate and direct-to-press made industrial scale printing as simple as submitting a word processor document for printing.

How would you, as the analyst responsible for workflow management for that direct-to-plate system, gather requirements for something so totally new? You could first find the use cases of the existing system and identify the system's actors and services. However, that only gives you the existing system. No one has done the new work yet, so all the domain experts are learning the system along with you. You are designing a new process and new software at the same time. Lucky you. How do you find the tracks on this fresh snow? Take the existing model and ask the question, "What changes?" The answer could well be "Everything."

When you write use cases to *document* requirements, someone has already created a vision of the system. You are simply expressing that vision so everyone clearly understands it. In *discovering* the requirements, however, you are creating the vision.

Use the use cases as a brainstorming tool. Ask the question, "Given the new technology, which steps in the use case no longer add value to the use case goal?" Create a new story for how the actors reach their goals. The goals are still the same, but some of the supporting actors are gone or have changed.

Use a *dive-and-surface* approach. Create a broad, high-level model of how you think the new system may work. Keep things simple, since this is new territory. Discover what the main success scenario might look like. Walk it through with the former domain experts.

Then dive down into the details of one use case. Consider the alternatives. Take advantage of the fact that people find it easy to comprehend stories, to flush out missing requirements. Read a step in a use case and ask the question, "Well, what happens if the client wants a hard-copy rather than a digital-copy proof?" This is easier than trying to assemble a full mental model of how the system works.

Finally, come back to the surface. What has changed now that you have submerged yourself in the details? Adjust the model, then repeat the dive with another use case.

My experience has been that using use cases to *discover* requirements leads to higher-quality functional requirements. They are better organized and more complete.

1.3 REQUIREMENTS AND USE CASES

If you are writing use cases as requirements, keep two things in mind:

- *They really are requirements*. You shouldn't have to convert them into some other form of behavioral requirements. Properly written, they accurately detail what the system must do.
- *They are not all of the requirements*. They don't detail external interfaces, data formats, business rules, and complex formulae. They constitute only a fraction (perhaps a third) of all the requirements you need to collect—a very important fraction but a fraction nonetheless.

Every organization collects requirements to suit its needs. There are even standards available for requirements descriptions. In any of them, use cases occupy only one part of the total requirements documented.

The following requirements outline is one that I find useful. I adapted it from the template that Suzanne Robertson and the Atlantic Systems Guild published on their Web site and in the book *Managing Requirements* (Robertson and Robertson, 1999). Their template is intimidating in its completeness, so I've cut it down to the form shown in the outline that follows, which I use as a guideline. This is still too large for most projects I encounter, and so I tend to cut it down further as needed. Whatever its size, it asks many interesting questions that otherwise would not be asked, such as "What is the human backup to system failure?" and "What political considerations drive any of the requirements?"

While it is not the role of this book to standardize your requirements deliverable, I have run into many people who have never seen a requirements outline. I pass along this one for your consideration. Its main purpose is to illustrate the place of use cases in the overall requirements and to make the point that use cases will not hold all the requirements, but only describe the behavioral portion, the required function.

A Plausible Requirements Outline

Chapter 1. Purpose and Scope
- 1a. What is the overall scope and goal?
- 1b. Stakeholders (Who cares?)
- 1c. What is in scope, what is out of scope?

Chapter 2. Terms Used / Glossary

Chapter 3. The Use Cases
- 2a. The primary actors and their general goals
- 2b. The business use cases (operations concepts)
- 2c. The system use cases

Chapter 4. The Technology Used
- 4a. What technology requirements are there for this system?
- 4b. What systems will this system interface with, with what requirements?

Chapter 5. Other Requirements
- 5a. Development process
  - Q1. Who are the project participants?
  - Q2. What values will be reflected (simple, soon, fast, or flexible)?
  - Q3. What feedback or project visibility do the users and sponsors want?
  - Q4. What can we buy, what must we build, what is our competition?
  - Q5. What other process requirements are there (testing, installation, etc.)?
  - Q6. What dependencies does the project operate under?
- 5b. Business rules
- 5c. Performance
- 5d. Operations, security, documentation
- 5e. Use and usability
- 5f. Maintenance and portability
- 5g. Unresolved or deferred

Chapter 6. Human Backup, Legal, Political, Organizational Issues
- Q1. What is the human backup to system operation?
- Q2. What legal and what political requirements are there?
- Q3. What are the human consequences of completing this system?
- Q4. What are the training requirements?
- Q5. What assumptions, dependencies are there on the human environment?

The thing to note is that use cases occupy only Chapter 3 of the requirements. They are not all of the requirements—they are *only* the behavioral requirements but they are *all of* the behavioral requirements. Business rules, glossary, performance targets, process requirements, and many other things do not fall into the category of behavior. They need their own chapters (see Figure 1.1).

Use Cases as Project-Linking Structure

Use cases connect many other requirements details. They provide a scaffolding that connects information in different parts of the requirements and they help crosslink user profile information, business rules, and data format requirements.

Outside the requirements document, use cases help structure project planning information such as release dates, teams, priorities, and development status. Also, they help the design team track certain results, particularly the design of the user interface and system tests.

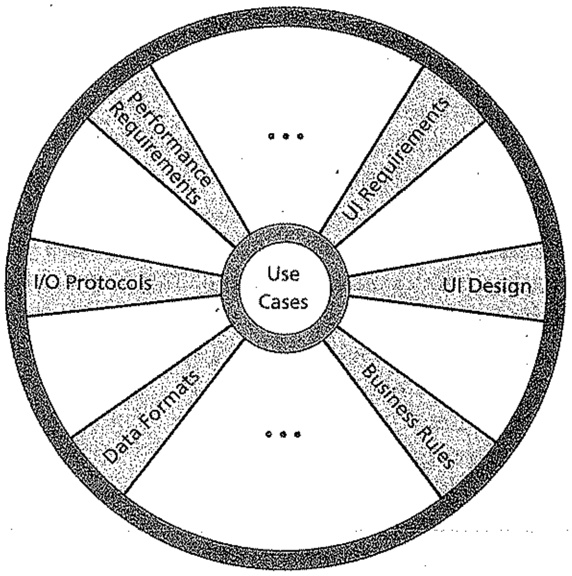

Figure 1.1 *The "Hub-and-Spoke" model of requirements*

While not in the use cases, all these requirements are connected to them. The use cases act as the hub of a wheel, as in Figure 1.1, and the other information acts as spokes leading in different directions. It is for this reason that people seem to consider use cases to be the central element of the requirements or even the central element of the project's development process.

1.4 WHEN USE CASES ADD VALUE

Use cases are popular largely because they tell coherent stories about how the system will behave in use. The users of the system get to see just what this new system will be. They get to react early to fine-tune or reject the stories ("You mean we'll have to do *what*?"). That is, however, only one of the ways use cases contribute value and possibly not the most significant.

The first moment at which use cases create value is when they are named as user goals that the system will support, and are collected into a list. This list announces what the system will do, revealing the scope of the system, its purpose in life. It becomes a communication device between the different stakeholders on the project.

The list of goals will be examined by user representatives, executives, expert developers, and project managers, who will estimate the cost and complexity of the system starting from it. They will negotiate which functions get built first and how the teams are to be set up. The list is a framework to which to attach complexity, cost, timing, and status measures. It collects diverse information over the life of the project.

The second particularly valuable moment is when the use case writers brainstorm all the things that could go wrong in the successful scenario, list them, and begin documenting how the system should respond. At that moment, the team is likely to uncover something surprising, something that they or their requirements givers had not thought about.

When I get bored writing a use case, I hold out until I get to the failure conditions. There I regularly discover a new stakeholder, system, goal, or business rule while documenting the failure handling. As we work out how to deal with one of these conditions, I often see the business experts huddled together or making phone calls to resolve what the system should be doing here.

Without discrete use case steps and failure brainstorming, many error conditions go undetected until some programmer discovers them while typing a code fragment. That is very late to be discovering new functions and business rules. The business experts usually are gone and time is pressing, and so the programmers type whatever they think up at the moment instead of researching the desired behavior.

People who write one-paragraph use cases save a lot of time by writing so little and already reap one of the benefits of use cases. People who perservere through failure handling save a lot of time by finding subtle requirements early.

1.5 MANAGE YOUR ENERGY

Save your energy. Or at least manage it. If you start writing all the details at the first sitting, you won't move from topic to topic in a timely way. If you write down just an outline to start with and then write just the essence of each use case, you can

- Give your stakeholders a chance to offer correction and insight about priorities early.
- Permit the work to be split across multiple groups, increasing parallelism and productivity.

People often say, "Give me the 50,000-foot view," "Give me just a *sketch*," or "We'll add *details* later." They mean "Work at low precision for the moment; we can add more later."

Precision is how much you care to say. When you state, "A 'Customer' will want to rent a video," you are not using very many words, but you are actually communicating a great deal to your readers. When you show a list of all the goals that your proposed system will support, you have given your stakeholders an enormous amount of information from a small set of words.

Precision is not the same as accuracy. If someone tells you, "π is 4.141592," they are using a lot of precision. They are wrong, though, and by a large amount. If they say, "π is about 3," they are not using much precision (there aren't very many digits), but they are accurate for as much as they said. The same ideas hold for use cases.

You will eventually add details to each use case, increasing precision. If you happen to be wrong (*inaccurate*) with your original, low-precision statement of goals, then the energy put into the high-precision description is wasted. Better to get the goal list correct before expending the dozens of work-months of energy required for a fully elaborated set of use cases.

I divide the energy of writing use cases into four stages of precision, according to the amount of energy required and the value of pausing after each stage:

1. *Actors and goals*. List what actors and which of their goals the system will support. Review this list for accuracy and completeness. Prioritize and assign goals to teams and releases. You now have the functional requirements to the first level of precision.
2. *Use case brief or main success scenario*. For the use cases you have selected to pursue, sketch the main success scenario. Review these in draft form to make sure that the system really is delivering the interests of the stakeholders you care about. This is the second level of precision on the functional requirements. It is fairly easy material to draft, unlike the next two levels.
3. *Failure conditions*. Complete the main success scenario and brainstorm all the failures that could occur. Draft this list completely before working out how the system must handle them. Filling in the failure handling takes much more energy than listing the failures. People who start writing the failure handling immediately often run out of energy before listing all the failure conditions.
4. *Failure handling*. Write how the system is supposed to respond to each failure. This is often tricky, tiring, and surprising work. It is surprising because quite often a question about an obscure business rule will surface during this writing, or the failure handling will suddenly reveal a new actor or new goal that needs to be supported.

Most projects are short on time and energy. Managing the precision level to which you work should therefore be a project priority. I strongly recommend working in the order given here.

1.6 WARM UP WITH A USAGE NARRATIVE

A usage *narrative* is a situated example of the use case in operation—a single, highly specific example of an actor using the system. It is not a use case, and in most projects it does not survive into the official requirements document. However, it is a very useful device that is worth my describing and your writing.

On starting a new project, you or the business experts may have little experience with use case writing or may not have thought through the system's detailed operation. To become comfortable with the material, sketch out a *vignette*, that is, a few moments in the day of the life of one of the actors.

In this narrative, invent a fictional but specific actor and briefly capture the mental state of that person—why he wants what he wants or what conditions drive him to act as he does. As with all use case writing, you needn't write much. It is astonishing how much information can be conveyed with just a few words. Capture how the world works, in this particular case, from the start of the situation to the end.

Brevity is important so the reader can get the story at a glance. Details and motives, or emotional content, are important so that every reader, from the requirements validator to the software designer, test writer, and training materials writer, can see how the system should be optimized to add value to the user.

A usage narrative takes little energy to write and little space, and leads the reader into the use case itself easily and gently. Here is an example.

Usage Narrative: Getting "Fast Cash"

Mary, taking her two daughters to the day care center on the way to work, drives up to the ATM, runs her card across the card reader, enters her PIN code, selects FAST CASH, and enters $35 as the amount. The ATM issues a $20 and three $5 bills, plus a receipt showing her account balance after the $35 is debited. The ATM resets its screens after each transaction with FAST CASH, so Mary can drive away and not worry that the next driver will have access to her account. Mary likes FAST CASH because it avoids the many questions that slow down the interaction. She comes to this particular ATM because it issues $5 bills, which she uses to pay the day care provider, and she doesn't have to get out of her car to use it.

People write usage narratives to help them envision the system in use. They also use it to warm up before writing a use case, to work through the details. Occasionally, a team publishes the narratives at the beginning of all the use cases or just before the specific use cases they illustrate. One group wrote that they get a user, an analyst, and a requirements writer together and animate the narrative to help scope the system and create a shared vision of it in use.

The narrative is not the requirements; rather, it sets the stage for more detailed and generalized descriptions of the requirements. The narrative anchors the use case. The use case itself is a dried-out form of the narrative—a formula—with a generic actor name instead of the actual name used in the usage narrative.

1.7 EXERCISES

Requirements File

1.1. Which sections of the requirements file outline are sensitive to use cases, and which are not? Discuss this with another person and think about why you come up with different responses.

1.2. Design another plausible requirements outline that can be put on an HTML-linked intranet. Pay attention to your subdirectory structure and date-stamping conventions (why will you need date-stamping conventions?).

Usage Narrative

1.3. Write two usage narratives for the ATM you use. How and why do they differ from the narrative example? How significant are those differences for the designers about to design the system?

1.4. Write a usage narrative for a person going into a brand new video rental store, interested in renting the original version of *The Parent Trap*.

1.5. Write a usage narrative for your current project. Get another person to write a usage narrative for the same situation. Compare notes and discuss. Why are they different, what do you care to do about those differences—is that tolerance in action or is the difference significant?