WRITING EFFECTIVE USE CASES

(* * PRE-PUB. DRAFT #3 * *)

Alistair Cockburn
Humans and Technology

Addison-Wesley

date: 2000.02.21

PREFACE

More and more people are writing use cases to describe business processes and the behavioral requirements for software systems. It all seems easy enough - just write about using the system.

Faced with writing, however, one suddenly asks, "Exactly what am I supposed to write - how much, how little, what details?" That is 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 writing and in coaching: how a person might think, what they 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 *best* to be *useful*. Even mediocre use cases are useful, more useful than many of the competing requirements files being written. So relax, write something readable, and you will have done your organization a service already.

Audience

The book is aimed at professionals who read and study alone. It is organized as a self-study guide. It contains introductory, intermediate and advanced concepts, examples, reminders, and exercises with answers.

Project and use case coaches should find suitable explanations and samples to show their teams.

Course designers and instructors 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 into four main parts: introduction to use cases, the use case body parts, frequently asked questions, reminders for the busy, and end notes.

The **Introduction to Use Cases** 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

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 & Actors” , “Three Named Goal Levels” , “Preconditions, Triggers, Guarantees” , “Scenarios and Steps” , “Extensions” , “Technology & Data Variations” , “Linking Use Cases” , and “Use Case Formats” .

Frequently Asked Questions addresses particular topics that come up repeatedly: “When are we done?” , “Scaling up to Many Use Cases” , “Two Special Use Cases” ("CRUD use cases" and "Parameterized use cases"), “Business Process Modeling” , “The Missing Requirements” , “Use Cases in the Overall Process” , “Use Cases Briefs and eXtremeProgramming” , and “Mistakes Fixed” .

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 reminders are organized as “Each Use Case” , “The Use Case Set” , and “Working on the Use Cases” .

The **End Notes** contains four topics: “Appendix A: Use Cases in UML” , “Appendix B: Answers to (some) Exercises” , “Appendix C: Glossary” , and “Appendix D: Reading” .

Heritage of the ideas in this book

Ivar Jacobson invented use cases in the late 1960s while working on telephony systems at Ericsson. Two decades later, he introduced them to the object-oriented programming community, where they were recognized as filling a significant gap in the development process. I took Jacobson’s course in the early 1990's. The ideas here are generally compatible with Jacobson’s descriptions, but I have slowly extended his model to accommodate recent insights regarding the writing. While neither he nor his team used the words *goal* and *goal failure*, it became clear to me over time that they had been using these notions in their teaching. In several comparisons, he and I have found there are no significant contradictions between his and my models.

I constructed the Actors & Goals conceptual model in 1994 while writing use case guides for the IBM Consulting Group. The Actors & Goals model explained a lot of the mystery of use cases, and gave guidance as to how to structure and write use cases. It circulated informally since 1995 from http://members.aol.com/acockburn, later at www.usecases.org, and it finally appeared in the Journal of Object-Oriented Programming in 1997, 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 & Goals model, led to the explanations in this book and the Stakeholders & Interests model, which is new in this book.

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 retained Jacobson's heritage of use cases. However, the UML standards group has a strong drawing-tools influence, with the effect that the textual nature of use cases was 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 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 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.

The place of use cases in the *Crystal* book collection

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

Crystal works from two basic principles:

- Software development is a cooperative game of group invention and communication. Software development improves as we improve people's personal skills and improve the team's collaboration effectiveness.
- Different projects have different needs. Systems have different characteristics, and are built by teams of differing sizes, containing people having differing values and priorities. It cannot be possible to describe the one, best way of producing software.

The foundation book for the Crystal collection is Software Development as a Cooperative Game. It works out the ideas of software development as a cooperative game, of methodology as a coordination culture, and of methodology families. It separates the different aspects of methodologies, techniques from activities, work products and standards. The essence of the discussion, as needed for use cases, is contained in "Your use case is not my use case" on page 20.

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 the needs of each individual project.

The samples used

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

Acknowledgements

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 for his encouragement and very specific feedback, as a practiced person with a sharp eye for the direct and practical needs of the team. Thanks to Firepond and Fireman's Fund Insurance Company for the live use case samples. Pete McBreen was the first to try out the Stakeholders & Interests model, and added his usual common sense, practiced eye, and suggestions for improvement. Thanks to the Silicon Valley Patterns Group for their careful reading on early drafts and their educated commentary on various papers and ideas. Mike Jones at Beans & Brews thought up the bolt icon for subsystem use cases.

Susan Lilly deserves special mention for the extremely exact reading she did, correcting everything imaginable: content, format, examples, ordering. The huge amount of work she gave me is reflected in much improved final copy.

Other specific reviewers who offered detailed comment and encouragement include: Paul Ramney, Andy Pols, Martin Fowler, Karl Waclawek, Alan Williams, Brian Henderson-Sellers, and Russell Gold.

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 Ft. Union Beans & Brews who once again provided lots of caffeine and a convivial atmosphere. Just to save us some future embarassment, my name is pronounced Cō-burn, with a long o.

Table of Contents

PART 1
The Use Case Body Parts

PART 2
Frequently Asked Questions

PART 3
Reminders for the Busy

PART 4
End Notes

1. Introduction to Use Cases

What do use cases look like?

Why would different project teams need different writing styles?

Where do they fit into the requirements gathering work?

How do we warm up for writing use cases?

It will be useful to have some thoughts on these questions in place before getting into the details of use cases themselves. Feel free to bounce between this introduction and Use Case Body Parts, picking up background information as you need.

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 it 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 conditions surrounding the requests. The use case collects together those different scenarios.

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 to communicate from one person to another, often to people with no special training. Simple text is, therefore, usually the best choice.

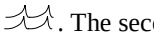
The use case, as a form of writing, can be put into service to stimulate discussion within a team about an upcoming system. They might later use that the use case form to document the actual requirements. Another team might later document the final design with the same use case form. They might do this 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 these different situations, even though the people will write with different amounts of rigor, at different levels of technical detail.

When the use cases document an organization's business processes, the system under discussion is the organization itself. The stakeholders are the company shareholders, customers, vendors, and

government regulatory agencies. The primary actors will include the company's customers and perhaps their suppliers.

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

I show several examples of use cases below. The parts of a use case are described in the next chapter ("Use Case Body Parts"). For now, just note that the *primary actor* is the one with the goal that the use case addresses. *Scope* identifies the system that we are discussing, the *preconditions* and *guarantees* say what must be true before and after the use case runs. The *main success scenario* is a case in which nothing goes wrong. The *extensions* section describes what can happen differently during that scenario. The numbers in the extensions refer to the step numbers in the main success scenario at which each different situation gets detected (for instance, steps *4a* and *4b* indicate two different conditions that could show up at step 4). When a use case references another use case, the second use case is written in *italics* or underlined.

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 the use case 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 *level* as "summary", and tag the use case with the "above sea level" kite symbol  . These symbols are all explained in more detail later.

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

Here are the first two use cases.

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, 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 that PAF can detect that something went wrong and can 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. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwabb, etc.) from user.
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site, and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.
Extensions:
2a. User wants a web site PAF does not support:
    2a1. System gets new suggestion from user, with option to cancel use case.
3a. Web failure of any sort during setup:
    3a1. System reports failure to user with advice, backs up to previous step.
    3a2. User either backs out of this use case, or tries again.
4a. Computer crashes or gets 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 user about the outcome.
    4b2. (see use case *Update questioned purchase*)
5a. Web site does not return the needed information from the purchase:
    5a1. PAF logs the lack of information, has the user *Update questioned purchase*.

USE CASE 2: GET PAID FOR CAR ACCIDENT

Primary Actor: The Claimant
Scope: The insurance company ("MyInsCo")
Level: Summary
Stakeholders and Interests:
the claimant - to get paid the most possible
MyInsCo - to pay the smallest appropriate amount
the dept. 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.
Main success scenario:
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 declines 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 declines claim, notifies claimant, records all this, terminates proceedings.
4b. Accident violates some minor policy guidelines:
    4b1. Insurance company begins negotiation with claimant as to degree of payment to be made.

Most of the use cases for 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 pretty 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". Sufficient is fine. It is all that is necessary. I show these real samples because you will rarely be able to generate perfect use cases yourself, despite whatever coaching I offer in the book. I can't even write perfect use cases most of the time.

Here is a use case written by a programmer for his user representative, his colleague and himself. It shows how the form can be modified without losing value. The writer adds additional business context to the story, illustrating how the computer application operates in the context of a working day. This is practical, as it saves having to write a separate document describing the business process or omitting the business context entirely. It confused no one, and was informative to the people involved. Thanks to Torfinn Aas, Central Bank of Norway.

USE CASE 3:  REGISTER ARRIVAL OF A BOX 

RA means "Receiving Agent".
RO means "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 TransportCompany TC
2. RA validates box id with TC registered ids.
3. RA maybe signs paper form for delivery person
4. RA registers arrival into system, which stores:
    RA id
    date, time
    box id
    TransportCompany
    <Person name?>
    # bags (?with bag ids)
    <estimated value?>
5. RA removes bags from box, puts onto cart, takes to RO.

Extensions:
2a. box id does not match transport company
4a. fire alarm goes off and interrupts registration
4b. computer goes down
 leave the money on the 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 use in different situations, to describe

* a business' work process,
* to focus discussion *about* upcoming software system requirements, but not be the requirements description,
* to be the functional requirements for a system, or
* to document the design of the system.
* They might be written in a small, close-knit group, or in a formal setting, 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*. They are further explained in "Use Case Body Parts," but you should become familiar with these notions right away.

- A close-knit group gathering requirements, or a larger group discussing upcoming requirements will write **casual** as opposed to the **fully dressed** use cases 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 below). All of the use cases shown above are fully dressed, using the full use case template and step numbering scheme. A example of casual form is shown below in Use Case 4:.
- Business process people will write **business** use cases to describe the operations of their business, while a hardware or software development team will write **external, system** use cases for their requirements. The design team may write **internal**, **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 - use cases that do not discuss the insides 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 yourself disagreeing on some matter of writing, just because you are writing use cases for different purposes. And you really 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, 5 were written for system requirements purposes, so they are fully dressed, black-box, system use cases, at the user-goal level. Use case 4 is the same, but casual instead of fully dressed. Use case 2 was written as the context-setting use case for business process documentation. It is fully dressed in form, it is black-box, and it is a summary-level business use case.

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 extra ceremony is worth the extra cost, that a) the use case template needs to be longer and more detailed, b) the writing team should write very much in the same style, to reduce ambiguity and cost of misunderstanding, c) the reviews should be tighter, to scrutinize the use cases closer for omissions and ambiguities. Having little tolerance for mistakes, they decide to reduce tolerances (variation between people) in the use cases writing also.
- 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 above ceremony a waste of time, energy and money. The team chooses a) a simpler template, b) to tolerate more variation in writing style, 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 the users. They can tolerate more errors in their written communication, and so more casual writing and more variation between people.

Neither is wrong. Those 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 5 years. Of course we've been saying, "One size doesn't fit all" for years, but just how to translate that into concrete advice has remained a mystery for methodologists.

The mistake is getting too caught up in precision and rigor, when it is not needed. That mistake will cost your project a lot in expended 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 it is incorrect to publish just one use case template. 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 their situation. The next two use cases show the same use case 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, check the price of the goods, completes the request for submission, and sends it to the Buyer. The Buyer checks the contents of storage, finding best vendor for goods. Authorizer: validate Approver's signature. Buyer: complete request for ordering, initiate PO with Vendor. Vendor: deliver goods to Receiving, get receipt for delivery (out of scope of system under design). Receiver: register delivery, send goods to Requestor. Requestor: mark request delivered.

At any time prior to receiving goods, Requestor can change or cancel the request. Canceling it removes it from any active processing. (delete from system?)Reducing the price leaves it intact in process. Raising the price sends it back to 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 non-electronic, 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 only be billed 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: *initiate a request*

2. Approver: check money in the budget, check price of goods, *complete request for submission*

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 the request.
Canceling it removes it from any active processing. (delete from system?)
Reducing price leaves it intact in process.
Raising price sends it back to Approver.
2a. Approver does not know vendor or price: leave blank and let Buyer fill in or call back.
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: gets resent to Approver.
4a. Authorizer declines Approver: send back to Requestor and remove from active processing. (what does this mean exactly?)
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 does *refuse 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
Freq 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 exactly 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 yet say very much, and any recommendation or process definition that simply says "write use cases" is incomplete. A use case valid on one project is not a valid use case 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. We don't need the full discussion in order to learn how to write use cases. We do need to separate the *writing technique* from *use case quality* and the *project standards*.

"Techniques" are the moment-to-moment thinking or actions people use while constructing the 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 both large and small projects.

"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, 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, along with how strongly to enforce them.

"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.

In most of this book, I deal with the most demanding problem, writing precise requirements. In the following eyewitness account, Steve Adolph describes using use cases to *discover* requirements rather 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 story-like 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 printing. 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 brand-new direct-to-plate system, gather requirements for something so totally new?

You could first find the use cases of the existing system, identify the actors and services of the existing system. But 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, "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 proof rather than a digital copy?” 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, after you 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, you should 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 still only a fraction.

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, Addison-Wesley, 1999). Their template is fantastically complete (intimidating in its completeness), so I cut it down to the following form, which I use as a guideline. This is still too large for most projects I encounter, and so we tend to cut it down further, as needed. However, it asks many interesting questions that otherwise would not get 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 file, I have run into many people who have never seen a requirements outline. I pass along this outline for your consideration. Its main purpose in this book is to illustrate the place of use cases in the overall requirements, to make the point that use cases will not hold all the requirements. They only describe the behavioral portion, the required function.

A Plausible Requirements File 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. The 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 to be used
4a. What technology requirements are there for this system?
4b. What systems will this system interface with, with what requirements?

Chapter 5. Other various requirements
5a. Development process
Q1. Who are the project participants?
Q2. What values will be reflected in the project (simple, soon, fast, or flexible)?
Q3. What feedback or project visibility do the users and sponsors wish?
Q4. What can we buy, what must we build, what is our competition to this system?
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, 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 only occupy chapter 3 of the requirements. They are not all of the requirements. They are *only* but *all of* the behavioral requirements. Business rules, glossary, performance targets, process requirements, and many other things simply do not fall in the category of behavior. They need their own chapters (see Figure 1.).

Use cases as a project linking structure

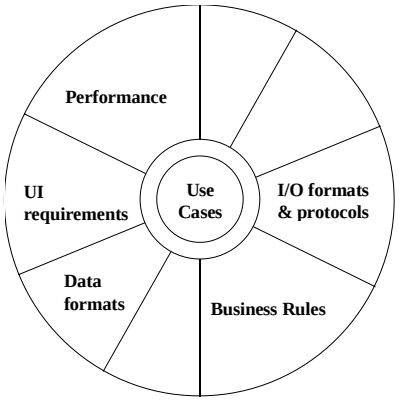


Figure 1. "Hub-and-spoke" model of requirements.

Use cases connect many other requirements details.

The use cases provide a scaffolding that connects information in different parts of requirements. they help crosslink user profile information, business rules and data format requirements.

Outside the requirement document, they help structure project planning information such as release dates, teams, priorities, and development status.

They help the design team track certain results, particularly the design of the user interface and system tests.

While not in the use cases, all these are connected to the use cases. The use cases act as the hub of a wheel (see Figure 1.), and the other information acts as spokes leading in different directions. It is for these reasons that people seem to consider use cases as the central element of the requirements, or even the central element of the project's development process.

Requirements File Exercises

Exercise 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.

Exercise 2 Design another plausible requirements file outline, suited to be put on an HTML-linked intranet. Pay attention to your subdirectory structure and date-stamping conventions (why will you need date-stamping conventions?).

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 ways they contribute value, and possibly not the greatest.

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

That list will be examined by user representatives, executives, expert developers, and project managers. They will estimate the cost and complexity of the system starting from that list. They will negotiate over which functions get built first, how the teams are to be set up. The list is a framework onto 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, they are 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. 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 should the system do here?"

Without the discrete use case steps and failure brainstorming activity, many error conditions stay undetected until some programmer discovers them while in the middle of typing a code fragment. That is very late to be discovering new functions and business rules. The business experts usually are gone, 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 the 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 next, then you can:

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

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

Precision is how much you care to say. When you say, "A 'Customer' will want to rent a video", you are not saying very many words, but you actually communicate 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, however, quite far off, or inaccurate. 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, adding 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 & Goals*. List what actors and which of their goals the system will support. Review this list, for accuracy and completeness. Prioritize and assign 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, write the trigger and 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 all. 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*. Finally, 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 a new goal that needs to be supported.

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

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, worth my describing, and worth 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 get comfortable with the material, sketch out a *vignette*, a few moments in the day of the life of one of the actors.

In this narrative, invent a fictional but specific actor, and capture, briefly, the mental state of that person, why they want what they want or what conditions drive them to act as they do. As with all of use case writing, we need not 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.

Here is an example of a usage narrative.

USAGE NARRATIVE: GETTING "FAST CASH"

> Mary, taking her two daughters to the day care 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 that 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, and she doesn't have to get out of her car to use it.

The narratives take little energy to write, little space, and lead the reader into the use case itself easily and gently.

People write usage narratives to help 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 the use case chapter, or just before the specific use cases they illustrate. One group described that they get a users, analyst and 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 generic actor name instead of the actual name used in the usage narrative.

Usage Narrative Exercises

Exercise 3 Write two user stories for the ATM you use. How and why do they differ from the one above? How significant are those differences for the designers about to design the system?

Exercise 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".

Exercise 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?

PART 1
THE USE CASE BODY PARTS

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 of training.

Learning to write good use cases 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 those three concepts straight is difficult. The difficulty shows up as soon as you start to write your first use case. These 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?

This part of the book covers these concepts at length, along with the other elements of the use case: action steps, scenarios, preconditions and guarantees, alternate flows, and technology and data variations.

While reading, hang onto these summary definitions:

* An actor is anyone or anything with behavior.
* A stakeholder is someone or something with a vested interest in the behavior of the system under discussion (SuD).
* The primary actor is the stakeholder who or which initiates an interaction with the SuD to achieve a goal.
* The use case is a contract for the behavior of the SuD.

2. The Use Case as a Contract for Behavior

The system under design is a mechanism to carry out a contract between various stakeholders. The use cases give the behavioral part of that contract. Every sentence in a use case is there because it describes an action that protects or furthers some interest of some stakeholder. A sentence might describe an interaction between two actors, or what the system must do internally to protect the stakeholders' interests.

Let's look first at a use case purely in the way it captures interactions between actors with goals. Once we have that, we can broaden the discussion to cover the use case as a contract between stakeholders with interests. I refer to the first part as the Actors & Goals conceptual model, and the second as the Stakeholders & Interests conceptual model.

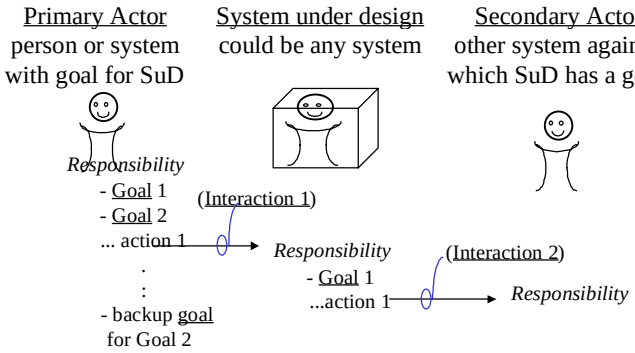
2.1 Interactions between Actors with Goals

Actors have goals

Imagine a clerk sitting by the phone, with the job responsibility to take service requests over the phone (the clerk is the primary actor in Figure 2.). When a call comes in, the clerk has a goal: to have the computer register and initiate the request.

The system also has a job responsibility, to register and initiate the service request in our example. (It actually has the responsibility to protect the interests of *all* the stakeholders; with the clerk (primary actor) being just one of those stakeholders. For now, however, let us just focus on the system's responsibility as providing a service to the primary actor.)

Figure 2. An actor with a goal calls upon the responsibilities of another.



To carry out its job responsibility, the system formulates subgoals. It can carry out some subgoals internally. It needs the help of another, *supporting*, actor to carry out others. This supporting actor may be a printing subsystem or it may be another organization, such as a partner company or government agency.

The supporting actor usually carries out its promise and delivers the subgoal to the SuD. The SuD interacts some more with external actors. It achieves its subgoals in some sequence, until it finally delivers its responsibility, its service promise.

Delivering a service promise is a topmost goal, achieved through subgoals. The subgoals can be broken down into sub-subgoals, *ad nauseam*. There is potentially no end to listing sub-sub-(...sub)-goals, if we want to break down the actions of the actors finely enough. As the English satirist and poet Jonathan Swift wrote (not about use cases):

> So, naturalists observe, a flea
> Hath smaller fleas that on him prey
> And these have smaller still to bite 'em
> And so proceed *ad infinitum.*
>
> - Jonathan Swift, from "On Poetry, A Rhapsody"

Probably the most difficult part of writing good use cases is controlling the fleas on the fleas, the sub-sub-goals in the writing. Read more on this in

* Section 5.“Three Named Goal Levels” on page 69,
* Reminder 6.“Get the goal level right” on page 202, and
* Guideline 6:“It contain a ’reasonable’ set of actions.” on page 98.

This Actors & Goals conceptual model is handy, since it applies equally to businesses as to computer systems. The actors can be individual people, organizations, or computers. We can describe mixed systems, consisting of people, companies and computers. We can describe a

software system driven by another computer system, calling upon a human supporting actor, or an organization calling upon a computer system or an individual. It is a useful and general model.

Goals can fail

What is the clerk with the customer on the phone supposed to do if the computer goes down in the middle of taking down the request? If the system cannot deliver its service promise, the clerk must invent a *backup* goal - in this case, probably using pencil and paper. The clerk still has a main job responsibility, and must have a plan in case the system fails to perform its part.

Similarly, the system might encounter a failure in one of its subgoals. Perhaps the primary actor sent in bad data, or perhaps there is an internal failure, or perhaps the supporting actor failed to deliver its promised service. How is it supposed to behave? *That* is a really interesting section of the SuD's behavioral requirements.

In some cases, the system can repair the failure and resume the normal sequence of behavior. In some cases it must simply give up on the goal. If you go to your ATM and try to withdraw more money than you have access to, your goal to withdraw cash will simply fail. It will also fail if the ATM has lost its connection with the network computer. If you merely mistype your personal code, the system will give you a second chance to type it in correctly.

This focus on goal failures and failure responses are two reasons use cases make good behavioral descriptions of systems, and excellent functional requirements in general. People who have done functional decomposition and data-flow decompositions mention this as the most significant improvement they see that use cases offer them.

Interactions are compound

The simplest interaction is simply sending a message. "Hi, Jean," I say, as we pass in the hall. That is a simple interaction. In procedural programming, the corresponding simple interaction is a function call, such as print(value). In object-oriented programming it is one object sending a message to another: objectA->print(value).

A *sequence of messages* is also an interaction, a compound interaction. Suppose I go to the soda machine and put in a dollar bill for an 80-cent drink, and get told I need exact change. My interaction with the machine is:

1. I insert a dollar bill
2. I press "Coke"
3. Machine says "Exact change required"
4. I curse, push Coin Return
5. Machine returns a dollar's worth of coins
6. I take the coins (and walk away, mumbling).

We can compact a sequence, as though it were a single step ("I tried to buy a Coke from the machine, but it needed exact change."), and put that compacted step into a larger sequence:

1. I went to the company bank and got some money.
2. I tried to buy a Coke from the machine, but it needed exact change.
3. So I walked down to the cafeteria and bought one there.

So interactions can be rolled up or broken down as needed, just as goals can be. Each step in a scenario captures a goal, and so each step can be unfolded into its own use case! It seems interactions have fleas with fleas, just as goals do.

The good news is that we can present the system's behavior at a very high level, with rolled-up goals and interactions. Unrolling them bit by bit, we can specify the system's behavior as precisely as we need. I often refer to the set of use cases as an *ever-unfolding story.* Our job is to write this ever-unfolding story in such a way that the reader can move around in it comfortably.

The astute reader will spot that I have used the word *sequence* rather loosely. In many cases, the interactions don't have to occur in any particular sequence. To buy that 80-cent Coke, I could put in 8 dimes, or three quarters and a nickel, or ... (you can fill in the list). It doesn't matter which coin goes in first.

Officially, *sequence* is not the right word. The correct phrase from mathematics is *partial ordering*. However, *sequence* is shorter, close to the point, and more easily understood by people writing use cases. If someone asks, "What about messages that can happen in parallel?", say, "Fine, write a little about that," and see what they come up with. My experience is that people write wonderfully clear descriptions with very little coaching. I therefore continue to say *sequence*. See gUse Case 22:"Register Loss" on page 83 for a sample with complex sequencing.

If you are interested in creating a formal language for use cases, it is easy to get into difficulty at this point. Most language designers either force the writer to list all possible orders or invent complex notations to permit the arbitrary ordering of events. But since we are writing use cases for another person to read, not a computer, we are more fortunate. We simply write, "Buyer puts in 80 cents, in nickels, dimes or quarters, in any order."

Sequences are good for describing interactions in the past, because the past is fully determined. To describe interactions in the future, we need *sets of possible sequences*, one for each possible condition of the world in the future. If I tell you about my asking for raise yesterday, I say:

> "I had a serious interaction with the boss today: I said, ' ...' She said, ' ...' I said, '...' etc."

But speaking into the future, I would have to say:

> "I am really nervous about this next interaction with the boss."
> "Why?"
> "I'm going to ask for a raise."
> "How?"
> "Well, first I'm going to say, ' ... ' Then if she says, ' ... ' then I'll respond with, ' ... ' But if she says,

' ... ,' then I'll try, ' ... ' " etc.

Similarly, if we tell another person how to buy a soda, we say:

First get your money ready.
If you have exact change, put it in and press the Coke button.
If you don't, put in your money and see whether it can give change. If it can ...

To describe an interaction in the future, we have to deal with different conditions, creating sets of sequences. For each sequence, or scenario, we say what the condition is, what the sequence will be, and what the outcome will be.

We can fold a set of sequences into a single statement. "First go and buy a Coke from the machine," or "Then you ask your boss for a raise." As with sequences, we can fold them into brief, high-level descriptions, or unfold them into detailed descriptions, to suit our needs.

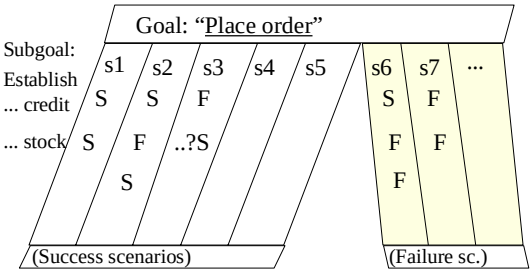
So far, we have seen that a use case contains the set of possible scenarios for achieving a goal. To be more complete, we need so add that

- All the interactions relate to the same goal of the same primary actor.
- The use case starts at the triggering event, and continues until the goal is delivered or abandoned, *and* the system's completes its responsibilities with respect to the interaction

A use case collects scenarios

The primary actor has a goal; the system should help the primary actor reach that goal. Some scenarios show the goal being achieved, some end with it being abandoned. Each scenario contains a sequence of steps showing how their actions and interactions unfold. A use case collects all those scenarios together, showing all the ways that the goal can be accomplished or fail.

Figure 3. Striped trousers: scenarios succeed or fail.

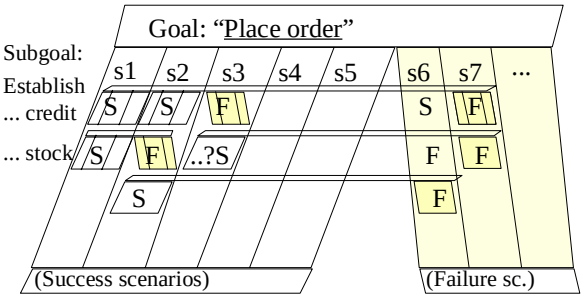


A useful metaphor for this is illustrated in the *striped trousers* image (Figure 3.). The belt on the trousers names the goal that holds all the scenarios together. There are two legs, one for the scenarios that end in success, one for the scenarios that end in failures. Each stripe corresponds to a scenario, any one being on the success leg or the failure leg. We'll call the first stripe on the success

leg the *main success scenario*. The other stripes are other scenarios that ultimately end in success, some through alternate success paths, and some after recovering from an intermediate failure. All of the stripes on the failure leg run through failures, possibly recovering and then failing in the end.

We won't actually write every scenario separately from top to bottom. That is a poor writing strategy: tedious, redundant and hard to maintain. The stripes trousers image is useful to help keep in mind that every use case has two exits, that the primary actor's goal binds all the scenarios, and that every scenario is a simple description of the goal succeeding or failing.

Figure 4. The striped trousers showing subgoals.



In Figure 4., I add to the striped trousers image to show a sub use case fitting into the use case that names it. A customer who wants to *Place an Order*. One of the customer's subgoals is to *Establish Credit*. That subgoal is complex, and might succeed or fail: it is a use case we have rolled up into a single step. The step "Customer establishes credit" is the belt on another set of trousers. In the stripe or scenario containing that step, the subgoal either succeeds or not. In scenarios 1 and 2 on the drawing, the subgoal works. In scenarios 3 and 7, the subgoal fails. In scenario 3, however, the next subgoal for establishing credit succeeds, and the scenario ends with success. In scenario 7, the second attempt also fails, and the entire use case ends with failure to place an order.

The point of showing the little stripes on the sub use case in the figure is to illustrate that the outer use case doesn't care what the sub use case went through in getting to its end state. It either succeeded or not. The outer, or calling, use case simply builds on the success or failure of the step naming the sub use case.

The principles we see from the trousers image are that:

- Some scenarios end with success, some end with goal failure.
- A use case collects together all the scenarios, success and failure.
- Each scenario is a straight description for one set of circumstances with one outcome.
- Use cases contain scenarios (stripes on the trousers), and a scenario contains sub use cases as its steps.
- A step in a scenario does not care which stripe in the sub use case was used, only whether it

ended with success or failure.

We shall make use of these principles throughout our writing.

2.2 Contract between Stakeholders with Interests

The Actors & Goals portion of the model explains very nicely how to write sentences in the use case, but it does not cover the need to describe internal behavior in the system under discussion. It is for that reason that the Actors & Goal model needs to be extended with the idea of a use case as a contract between stakeholders with interests, which I'll refer to as the Stakeholders & Interactions conceptual model. The Stakeholders & Interests portion identifies what to include in the use case and what to exclude.

The system under design operates a contract between stakeholders, the use cases detailing the behavioral part of that contract. However, not all of the stakeholders are present while the system is running. The primary actor is usually present, but not always. The other stakeholders are not present. We might call them *off-stage actors*. The system acts to satisfy the interests of these off-stage actors. That includes gathering information, running validation checks, and updating logs.

The ATM must keep a log of all interactions, to protect the stakeholders in case of a dispute. It logs other information so they can find out how far a failed transaction got before it failed. The ATM and banking system verify that the account holder has adequate funds before giving out cash, to make sure the ATM only gives out money that customers really have in the bank.

The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfy the stakeholders' interests.

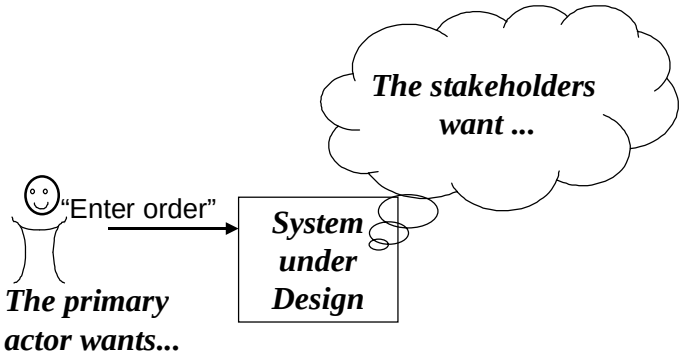


Figure 5. The SuD serves the primary actor, protecting off-stage stakeholders**.**

To carefully complete a use case, we list all the stakeholders, name their interests with respect to the operation of the use case, state what it means to each stakeholder that the use case completes successfully, and what guarantees they want from the system. Having those, we write the use case steps, ensuring that all the various interests get satisfied from the moment the use case is triggered until it completes. That is how we know when to start and when to stop writing, what to include and what to exclude from the use case.

Most people do not write use cases this carefully, and often they happily get away with it. Good writers do this exercise in their heads when writing casual use cases. They probably leave some out, but have other ways of catching those omissions during software development. That is fine on many projects. However, sometimes there is a large cost involved. See the story about forgetting some interests in the section 4.1“Stakeholders” on page 61.

To satisfy the interests of the stakeholders, we shall need to describe three sorts of actions:

* An interaction between two actors (to further a goal).
* A validation (to protect a stakeholder).
* An internal state change (on behalf of a stakeholder).

The Stakeholders & Interests model makes only a small change in the overall procedure in writing a use case: list the stakeholders and their interests and use that list as a double check to make sure that in none was omitted in the use case body. That small change makes a big change in the quality of the use case.

2.3 The Graphical Model

Note: *The Graphical Model is only intended for people to like to build abstract models. Feel free to skip this section if you are not one of them.*

A use case describes the behavioral contract between stakeholders with interests. We organize the behavior by the operational goals of a selected set of the stakeholders, those who will ask the system to *do* something for them. Those we call *primary actors.* The use case’s name is the primary actor’s goal. It contains all the behavior needed to describe that part of the contract.

The system has the responsibility to satisfy the agreed-upon interests of the agreed-upon stakeholders with its actions. An action is of one of three sorts:

* An interaction between two actors, in which information may be passed.
* A validation, to protect the interests of one of the stakeholders.
* An internal state change, also to protect or further an interest of a stakeholder.

A scenario consists of action steps. In a "success" scenario, all the (agreed-upon) interests of the stakeholders are satisfied for the service it has responsibility to honor. In a "failure" scenario, all those interests are protected according to the system's guarantees. The scenario ends when all of the interests of the stakeholders are satisfied or protected.

Three triggers that request a goal's delivery are the primary actor initiating an interaction with the system, the primary actor using an intermediary to initiate that interaction, or a time- or state-based initiation.

The model of use cases described in this chapter is shown in the figures below using UML (Unified Modeling Language). All of the relations are 1-to-many along the arrows, unless otherwise marked.

Here is a bit of truth-in-advertising. I don't know how to debug this model without several years of testing it on projects using a model-based tool. In other words, it probably contains some subtle errors. I include it for those who wish to experiment, perhaps to create such a model-based tool.

The primary actor is a stakeholder

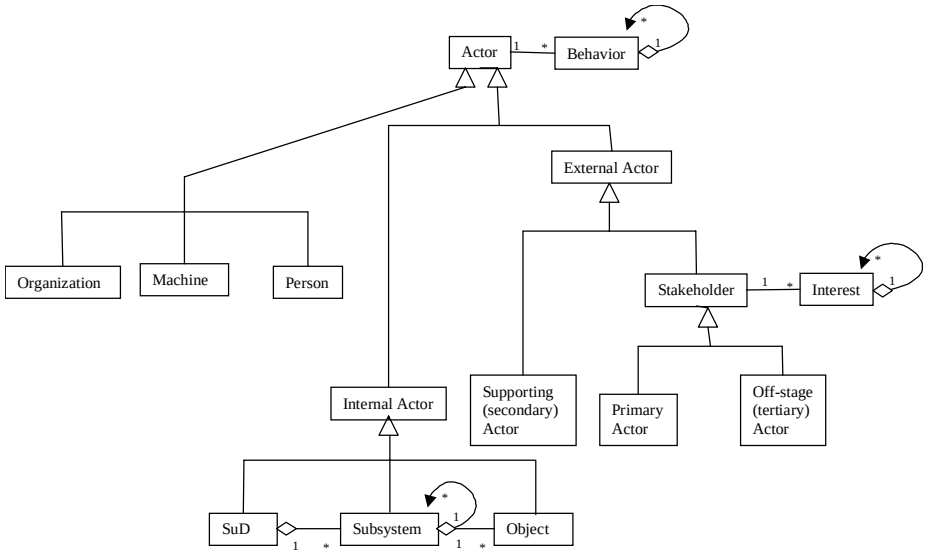


Figure 6. A stakeholder has interests**.** An actor has behaviors. A primary actor is also a stakeholder.

Behavior

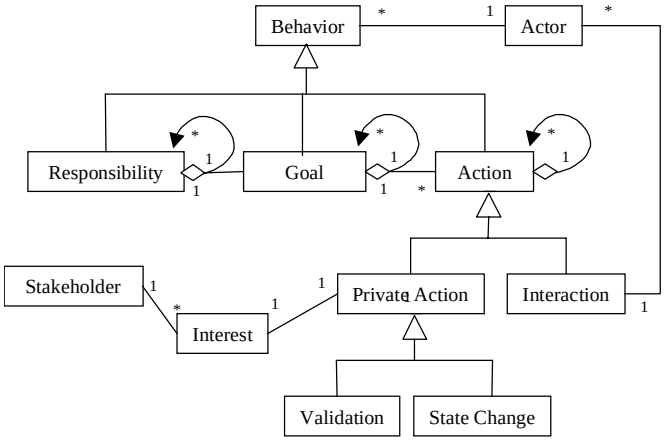


Figure 7. Goal-oriented behavior made of responsibilities, goals and actions.

The private actions we will write are those forwarding or protecting the interests of stakeholders. Interactions connect the actions of one actor with another.

Use case as contract

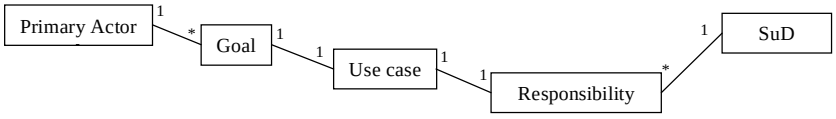


Figure 8. The use case as responsibility invocation. The use case is the primary actor's goal, calling upon the responsibility of the system under design.

Interactions

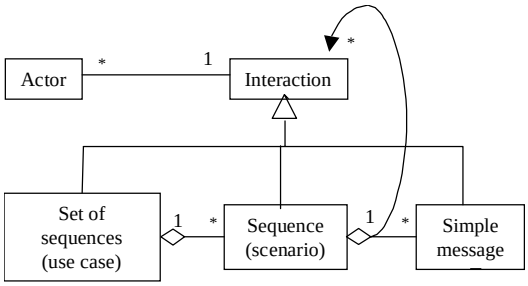


Figure 9. Interactions are composite**.** 'N' actors participate in an interaction. Interactions are composite, decomposing into use cases, scenarios, and simple messages. Once again, the word *sequence* is used as a convenience.

3. Scope

Scope is the word we use for the extent of what we consider to be designed by us, as opposed to already existing or someone else's design job.

Keeping track of the scope of the project, or even just the scope of a discussion can be difficult. Rob Thomsett introduced me to a wonderful little tool for tracking and managing scope discussions, the *In/Out List*. It is 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 containing any topic at all, the next two columns saying "in" or "out". Whenever it appears there might confusion as to whether a topic is within the scope of the discussion, you 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, they often have opposite views! Rob relates that sometimes it requires an appeal up to the project's steering committee to settle whether a particular topic really is inside the scope of work or not. In or out can make a difference of many work-months. Try this little technique out on your next projects, or perhaps your next meeting!

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

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 to one PO	In	
Partial deliveries, late deliveries, wrong deliveries	In	
All new system services, software	In	
Any non-software parts of the system		Out
Identification of any pre-existing software that can be used	In	
Requisitions	In	

Use the in/out list right at the beginning of the requirements or use case writing activity, to separate those things that are going to be within the scope of work, from those that are out of scope. Refer back to the chart 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 in/out list for topics relating to both the functional scope of the system under discussion, and the design scope of the system under discussion.

3.1 Functional scope

Functional scope refers to the services your system offers. It will eventually be captured by the use cases. As you start your project, however, it is quite likely that you don't precisely know the functional scope. 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 names all the user goals that the system supports, showing the functional content of the system. 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 actually will be supported by the system. Here is one project's actor-goal list for the purchase-request tracking system:

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 an 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 non-delivery	4
Authorizer	Validate Approver's signature	3
Receiver	Register delivery	1

To make this list, construct a table of three columns. Put into the left column the names of the primary actors, the actors having the goals. Put into the middle column each actors' goals with respect to the system. In the third column write the priority or an initial guess as to the release

number in which the system will support that goal. You will 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 those that will get triggered by time instead of a person; or the three items *Business Priority*, *Development Complexity*, *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.

Note: *For people drawing use case diagrams in the Unified Modeling Language*

The use case diagram's main value is as a visual actor-goal list. People like the way it shows the clusters of the use cases to the primary actors. In a good tool, it acts as a context diagram and a graphical table of contents, hot-linked to the use case text.

Although it serves those two purposes, the diagram does not prvide a worksheet format for studying the project estimation data. If you use the diagram, you will still need to collect the priority and estimation information and build a work format for it.

To have it serve its main purpose well, keep the diagram clear of clutter. Show only use cases at user goal level and higher.

The Use Case Briefs

I shall repeat several times the importance of managing your energy, of working at low levels of precision wherever possible. The actor-goal list is the lowest level of precision in describing the behavior of the system. 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 else a *use case brief*.

The *brief* of a use case is a 2-6 sentence description of the 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 from commercial, off-the-shelf components (COTS) use this description in preparing to select the components. Some projects, 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 put the use case brief either in a table, as an extension to actor-goal list, or directly into the use case body as its first draft. Here is a sample table of briefs, thanks to Paul Ford and Paul Bouzide of Navigation Technologies.

A SAMPLE OF USE CASE BRIEFS

Actor	Goal	Brief
Production Staff	Modify the administrative area lattice	Production staff add admin area metadata (administrative hierarchy, currency, language code, street types, etc.) to reference database and contact info for source data is cataloged. This is a special case of updating reference data.
Production Staff	Prepare digital cartographic source data	Production staff convert external digital data to a standard format, validate and correct it in preparation for merging with an operational database. The data is catalogued and stored in a digital source library.
Production & Field staff	Commit update transactions of a shared checkout to an operational database	Staff apply accumulated update transactions to an operational database. Non-conflicting transactions committed to operational database. Application context synchronized with operational database. Committed transactions cleared from application context. Leaves operational database consistent, with conflicting transactions available for manual/interactive resolution.

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 you are charged with designing or discussing. It is that boundary. If we are charged with designing 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 shall 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.

It is incredibly important that the writer and reader are in agreement about the design scope for a use case - and correct. The cost of being wrong can be a factor of two or more in cost or price, with disastrous results for the outcome of a contract. The readers of a use case must quickly see what you intend as 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.

A small, 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! You think we just use a little laser printer to print our invoices? We have a huge printing system, with chain printer, batch I/O and everything. We produce invoices by the boxfull!"

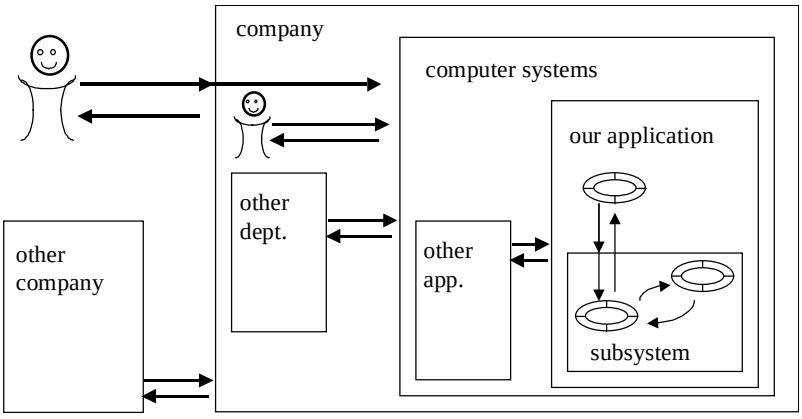
I was shocked, "You mean the printer is not in the scope of the system?"

"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 read the tape and printed what it could. It prepared 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, it was for us to use. It was out of our design scope. (It was, as described in the next section, a *supporting actor.*) Had we not detected this mistake, we would have written the use case to include it in scope, and turned in a bid to build more system than was needed.

Figure 10. Design scope can be any size.



Typically, the writer considers it obvious what the design scope of the system is. It is so obvious that they don't mention it. However, once there are multiple writers and multiple readers, then the design scope of a use case is not at all obvious. One writer is thinking of the entire corporation as the design scope (see Figure 10.), 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. The readers, having no clue as to what is meant, get lost or misunderstand the document.

What can we do to clear up these misunderstandings?

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 design scopes. To be concrete, let us suppose that MyTelCo is designing NewApp system, which includes a Searcher subsystem. The design scope names are:

- "**Enterprise**" scope (put the real name here, e.g. **MyTelCo**)  signifies that 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, e.g., *MyInsCo*, rather than just writing "the company". If discussing a department, use the department name. Business use cases are written at enterprise scope.

- "**System**" scope (put the system name in here, e.g., **NewApp**)  means just the piece of hardware/software you are charged with building. Outside the system are all the pieces of hardware, software and humanity that it is to interface with.

- "**Subsystem**" scope (put the subsystem name in here, e.g. **Searcher**)  means 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 the reader before they start reading. There are no tools at this time to manage the icons, but I find that drawing them reduces the confusion about a use case's scope. In this book I label each use case with its appropriate icon to make it easier for you to note the design scope of each example.

Recall, in the following, 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 design 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 inside the organization.
- A *system* use case has a computer system as its design 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 (as in nuts and bolts):  . See the use case set *Documenting a Design Framework* for an example of a component use case.

Examples of design scope

I offer three samples to illustrate descriptions of systems at different scopes.

Enterprise to system scope

This is the most common situation.

> 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 computer. The server will be connected to a mainframe computer running the old system, BSSO. BSSO is just a computer terminal attached to the mainframe. We are not allowed to make any changes to BSSO. We can only use its existing interfaces.

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

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

The immediate question to ask 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, multi-day 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 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 system  scope, with the Scope field labeled Acura. It will freely mention clerks and departments and other computer systems, but not mention 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, 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 the five- to twenty-minute handling of the request in detail in the user-goal use case having Acura as design 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 design scope Acura Workstation or Acura Server, as they are not of interest to us. Actually, they are not of interest to *us, now.* Later, someone in the design team may choose to document Acura's subsystem design using use cases. At that time, they would write two use cases, one with Scope: Acura Workstation, the other with Scope: Acura Server. My experience is that, typically, these use cases are never written, since there are other adequate techniques for documenting subsystem architecture.

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 over more function. The two systems must co-exist and synchronize with each other. Use cases have to be written for both systems, system Acura being entirely new, and system BSSO being modified to synchronize with Acura.

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. They look redundant.

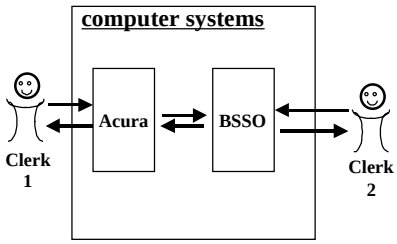
To document this situation, I first write a summary-level use case whose scope is both computer systems together. This gives me a chance to document their interactions over time. In that use case I reference the specific use cases that comprise each system's requirements. This first use case will be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

The situation is complicated enough that I also include inline diagrams of the design scope of each use case.

Figure 11. System scope diagram for Acura - BSSO. In this case, I put the system scope diagram directly within Use Case 8:.

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

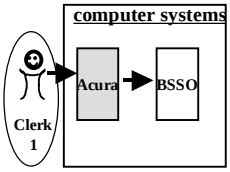


Main success scenario:
1. Clerk *adds new service* into Acura.
2. Acura *notes new service request* in BSSO.
3. Some time later, Clerk *updates service request* in BSSO.
4. BSSO *notes the updated request* in Acura.

The four use cases called out above 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 inline 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 to all that, I gave the use cases slightly different verb names, using *Note* to indicate the synchronization activity.

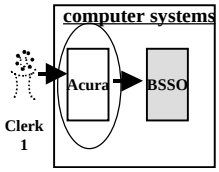
USE CASE 9: *ADD NEW SERVICE (INTO ACURA)*

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



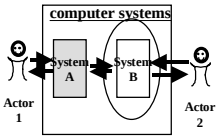
USE CASE 10: *NOTE NEW SERVICE REQUEST (IN BSSO)*

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



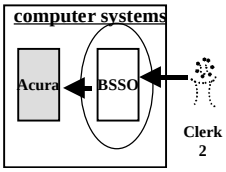
USE CASE 11: *UPDATE SERVICE REQUEST (IN BSSO)*

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-style use case diagrams, you will draw , possibly instead of writing the summary level use case. 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 inline scope diagrams.

Personally, I do not find that eliminates the confusion very much. I would consider drawing the non-standard use case diagram in Figure 12. to show the connection between the two systems. This

diagram is clearer, but harder to maintain over time.You should draw whichever you and your readers find communicates best for you.

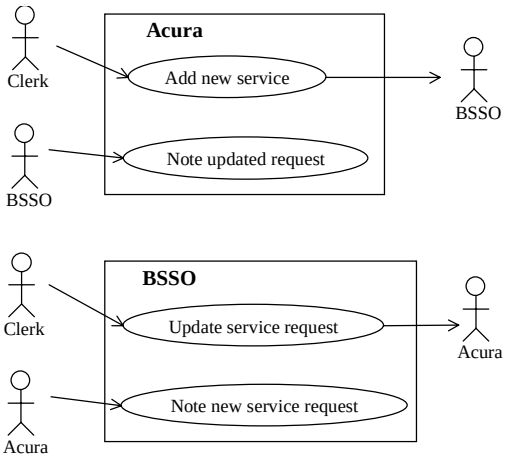


Figure 12. Use case diagrams for Acura - BSSO. This is the UML style of denoting the interactions between the two systems. The upper section shows that BSSO is a supporting actor to one use case of Acura, and a primary actor to another use. In the lower section, it shows the roles reversed.

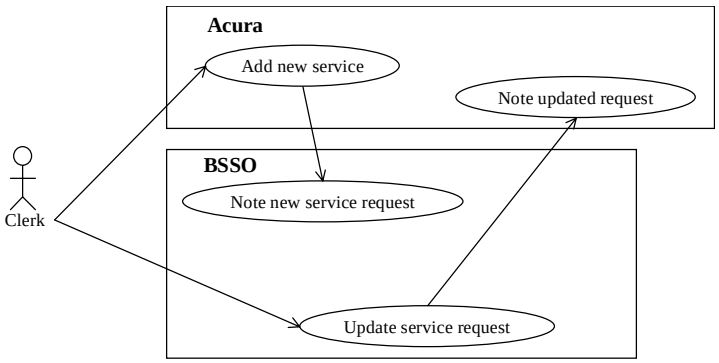


Figure 13. A combined use case diagram for Acura-BSSO. This drawing shows the relationships of the four use cases most clearly, but is non-standard, since it shows one system's use case triggering another system's use case.

Nuts and bolts use cases

At the far end of the scale, let us look 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.

They 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 use cases.

You will find these use cases incomprehensible unless you are in that business. However, I expect some readers to be technical programmers looking for ways to document their designs. 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 when they are used. 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 non-zero:
    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 non-zero 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:
   - · For Exclusive access
   - · For Shared access

2c. The Lock holding time-out can be specified by:
   - · The Service Client
   - · A Resource Locking Policy
   - · 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 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 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 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
2aWaiting Timer expires:
2a1.Signal Service Client that requested access could not be granted.

2a2.Fail!

Variations:

The Lock waiting time-out can be specified by

- The Service Client
- A Resource Locking Policy
- A global default value

Design Scope Exercises

Exercise 6 * Name at least 5 different system design scopes 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..."*

Exercise 7 * Draw a picture of the multiple scopes in action for an ATM, including hardware and software.

Exercise 8 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.

Exercise 9 Draw a picture of the multiple scopes in action for the Personal Advisors/Finance system.

Exercise 10 Draw a picture of the multiple scopes in action for a web application in which a user's workstation is connected through the web to your company's web server, attached to a legacy mainframe system.

Exercise 11 Describe the difference between "*enterprise-scope white-box business use cases"* and "*enterprise-scope black-box business use cases"*.

3.3 The Outermost Use Cases

In "Enterprise to system scope" on page 50, 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. Keep widening the scope until you reach the point widening it farther would bring the primary actor inside it. That is the *outermost scope.* Sometimes the outermost scope is the enterprise, sometime the department, and sometimes it is just the computer. Often, the computer department is the primary actor on the computer security use cases, the

marketing department is the primary actor on the advertising use cases, and the customer the primary actor on the main system function use cases.

Typically, there are only 2-5 outermost use cases for the entire system, so it is not the case that every use case gets written twice. There are so few of them because each outermost use case 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. It takes very little time, and provides excellent context for the set of use cases. The outermost use cases show how the system ultimately benefits the most external users of the system, and 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, described a little earlier.

> MyTelCo decides to let web-based customers access Acura directly. This will 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 shall 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, and so there is one outermost use case with the customer as primary actor and MyTelCo as scope. This use case will be a summary level use case, 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 51.

The clerk is inside MyTelCo. The outermost scope for *Add Feature (By Staff)* is All Computer Systems. This use case will collect together 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 the 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 the computer system 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 there are more than these four

outermost use cases to write 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, 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 design.

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 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, the four work products that bind the system's scope are the

* vision statement,
* design scope drawing,
* in/out list, and
* 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.

4. Stakeholders & Actors

A stakeholder is someone who gets to participate in the contract. An actor is anything having behavior. As one student said, "It must be able to execute an 'IF' statement." An actor might be a person, a company or organization, a computer program or a computer system, hardware or software or both.

Look for actors in:

- the *stakeholders* of the system.
- the *primary* actor of a use case;
- the system under design, itself (*SuD*);
- the *supporting* actors of a use case;
- *internal actors*, components within the SuD;

4.1 Stakeholders

A stakeholder is someone with a vested interest in the behavior of the use case, even if they never interact directly with the system. Every primary actor is, of course, a stakeholder. But there are stakeholders who never interact directly with the system., even though they have a right to care how the system behaves. Examples are the owner of the system, the company's board of directors, and regulatory bodies such as the Internal Revenue Service and the Department of Insurance.

Since these other stakeholders never appear directly in the action steps of the use case, students have nicknamed them *offstage* actors, *tertiary* actors, and *silent* actors.

Paying attention to these silent actors improves the quality of a use case significantly. Their interests show up in the checks and validations the system performs, the logs it creates, and the actions it performs. The business rules get documented because the system must enforce them on behalf of the stakeholders. The use cases need to show how system protects these stakeholders' interests. Here is a story that illustrate the cost of forgetting some of the stakeholders' interests.2

A short, true story

> In the first year of operation following selling several copies of its new system, a company received some change requests for their system. That all seemed natural

enough, until they took a use case course and were asked to brainstorm the stakeholders and interests in their recently delivered system.

Much to their surprise, they found they were naming their recent change request items in the stakeholders and interests! Evidently, while developing the system, they had completely overlooked some of the interests of some of the stakeholders. It didn't take the stakeholders long to notice that the system wasn't serving them properly, and hence the change requests started coming in.

The leader has since become adamant about naming stakeholders and interests early on, to avoid a repeat of this expensive mistake.

My colleagues and I find that we identify significant and otherwise unmentioned requirements early by asking about the stakeholders and their interests. It does not take much time to do this, and it saves a great deal of effort later on.

4.2 The primary actor of a use case

The primary actor of a use case is the stakeholder that calls upon the system to deliver one of its services. The primary actor has a goal with respect to the system, one that can be satisfied by its operation. The primary actor is often, but not always, the actor who triggers the use case.

Usually, the use case starts because the primary actor sends a message, pushes a button, enters a keystroke, or in some other way initiates the story. There are two common situations in which the initiator of the use case is not the primary actor. The first is when a company clerk or phone operator initiates the use case on behalf of the person who really cares, the second is when the use case is triggered by time.

A company clerk or phone operator is often a technological convenience for the real person who cares, what I call the *ultimate primary actor.* With technology shifting, it becomes more likely that the ultimate primary actor will initiate the use case directly, using the web or an automated phone systems. An example of this is the customer who currently phones in with a request. In a web redesign of the system, the customer may enter their request directly (as with Amazon.com).

Similarly, the Marketing or Auditing Division might insist on the presence of use cases which are to be operated by a clerk. It is not really the clerks' goal to have the use case run, they are a technological convenience for the Marketing managers. Under slightly different circumstances, the Marketing managers would run the use cases themselves.

These days I write, "sales rep for the customer" or "clerk for Marketing Department" to capture that the user of the system is acting for someone else. This lets us know that the user interface and security clearances need to be designed for a clerk, but that the customer or Marketing Department are the ones who really care about the outcome.

Time is the other example of a non-operator trigger. There is no clerk triggering the use cases that to run every midnight, or at the end of the month. It is easy, in this case, to see that the primary actor is whichever stakeholder cares that the use case runs at that time.

It is easy to get into long arguments on the topic of users versus ultimate primary actors. I suggest you don't spend too much time on it, or else you argue in a pub. When the team starts investigating the user interface design, they will - or should - put a good deal of effort into studying the real users' characteristics. When they review the requirements', they will find it useful to know the ultimate primary actor for each use case, who it is that really cares.

As one student astutely asked, "How much damage is there if we get the primary actor wrong at this point?" The answer is, "Not much." We need to understand...

WHY PRIMARY ACTORS ARE UNIMPORTANT (AND IMPORTANT)

Primary actors are important at the beginning of requirements gathering and just before system delivery. Between those two points, they become remarkably unimportant.

At the beginning of use case production

Listing primary actors helps us get our minds around the entire system for one brief moment (it will escape us soon enough). We brainstorm to name all the actors in order to brainstorm and name all the goals. It is really the goals that interest us. If we brainstorm the goals directly, we will miss too many of them. Brainstorming the primary actors sets up a work structure. We can then traverse that structure to get a better goal list.

Creating a slightly larger number of primary actors does not hurt, since we will, at the worst, generate the same goal twice. When we go over the actors and goals to prioritize the work, we will find and remove the duplicates.

Even with this careful double brainstorming, it is quite unlikely that we will name all of the goals our system needs to support. New ones have a tendency to show up while writing the failure handling steps of a use case. However, that is something we can't affect at this early stage, and we do our best to capture all the goals by listing first all the primary actors.

A rich list of primary actors confers three other advantages.

* It focuses our minds on the people who will use the system. In the requirements document, we write down who we expect the primary actors to be, their job descriptions, their typical background and skills. We do this so that the user interface and system designers can match the system to that expertise.
* It sets up the structure for the actor-goal list, which will be used to prioritize and partition the development work.
* It will be used to partition a large set of use cases into packages that can be given to different design teams.

During use case writing, and during design

Once we start developing the use cases in detail, the primary actors become almost unimportant. That may surprise you. What happens is that, over time, the use case writers discover that a use case can be used by multiple sorts of actors. For example, anyone higher than *Clerk* might answer the phone and talk to a customer. The use case writers often start to name the primary actor in an increasingly generic way, using role names such as *Loss Taker*, *Order Taker*, *Invoice Producer*, and so on. This produces use cases that say, "The invoice producer produces the invoice...", and "The order taker takes the order..." (not terribly enlightening, was that?).

You can handle this fragmentation of roles in several ways, each with a small advantage and disadvantage. No strategy is clearly superior, so you just have to choose one.

Alternative 1: Break apart the primary actors into the roles that they play. Maintain a table that lists all the different sorts of people and the systems that play primary actor to any use case, and all the roles they all can play. Use the role names in the Primary Actor field. Use the actor-role list to get from the use cases to the people and systems in the world.

This strategy allows the writers to ignore the intricacies surrounding job titles and simply get on with writing the use cases. Someone, perhaps the user interface designer or the software packager, will use the actor-to-role table to match up the use cases with the eventual users. The trouble with alternative 1 is that there is a separate list to maintain and to read.

Alternative 2: Write, somewhere in the front of the use case section, "The Manager can do any use case the Clerk can, plus more. The Regional Manager can do any use case the Manager does, plus more. Therefore, wherever we write that the primary actor is (e.g.) Clerk, it is to be understood that any person with more seniority, the Manager and Regional Manager in this case, can also do the use case."

This is easier to maintain than the actor-to-role table, as it is unlikely to change. The disadvantage is that people will spend more time reminding each other that when *Clerk* is written as the primary actor, really the *Manager* also can run the use case.

People achieve adequate results with both methods. For what it's worth, I chose to use the second method. I like having one less work product to write, review and maintain.

The point is, the *Primary Actor* field of the use case template becomes devalued over time. This is normal, and you shouldn't worry about it.

After design, preparing to deploy the system

Just before delivering the system, the primary actors become important again. We need the list of all the *people*, and which use cases they will run. We need these to:

* Package the system into units that get loaded onto the various users' machines;
* Set security levels for each use case (web users, internal, supervisor, etc.);
* Create training for the various user groups.

In short, act terribly concerned over the completeness and correctness of the list of primary actors at the beginning. The damage of omitting an actor is large: we might

entirely overlook a section of requirements. The damage of naming too many actors is small: some extra work in the early stages until the unnecessary actors are eliminated.

After that, the actors become relatively unimportant. Fussing over the exact and correct name for the primary actor adds little to the value of the behavior description.

Just before deployment, the primary actors become important again, in preparing system packaging and training.

Note: *About actors as opposed to roles*

The word *actor* implies an *individual* in action. Sometimes, in a use case, we mean an individual, but we also mean the general category of individuals who can play that role.

Suppose that Kim is a customer of MyTelCo, Chris is a clerk, and Pat is a sales manager. Any one of them can *Place an Order.* Using the language of *actors*, we say that Kim, Chris and Pat can be primary actors for the use case *Place an Order.* We also say that *Customer*, *Clerk*, and *Sales Manager* are the allowed primary actors for *Place an Order.* We might write that "a sales manager can perform any use case a clerk can." These are all fine ways of speaking.

Using the language of *roles*, we say that Kim, Chris and Pat are individual actors. Any of them can play the role of *Customer*, but only Chris and Pat can play the role of *Clerk*, and only Pat can play the role of *Sales Manager.* Then we say the role that drives the *Place an Order* use case is *Order Taker*, and that any of *Customer, Clerk*, and *Sales Manager* can play the role of *Order Taker.* This way of speaking is more precise than the previous, and so some people prefer it. It is, however, non-standard in the use case world.

The point of this note is that you should use whichever terms your team prefers. In “Why primary actors are unimportant (and important)” , I show why the matter should not cause you too much stress, and how to deal with some of the situations that arise. In the meantime, actor is the word the industry has accepted, it works quite adequately, and that is what I use in this book.

Note: *Unified Modeling Language diagrams and actor/role specialization*

This note is for people drawing use case diagrams in UML.

UML provides a special, hollow-headed arrow to indicate that one actor *specializes* another (see, Figure 34.“Correctly closing a big deal” on page 231). This arrow resolves part, but not all of the actor-role controversy.

The good part about this arrow is that it allows you to express succinctly that a Manager can do anything a Clerk can do: simply draw the arrow with the arrowhead at Clerk and the tail at Manager.

The bad part is that many people think the resulting drawing appears backwards. Most people don't think of a Manager as *a special kind of* a Clerk, or a Clerk as a *special kind of* Customer, which is what the drawing seems to assert (it really asserts that the one *can do anything the other can do*). They think of a Manager as *more than a* Clerk. It is not not a big thing, but you will have to deal with this reaction.

The specialization arrow does not help at all with the main part of the actor-role question. A *Sales Clerk* and an *Auditing Clerk* have overlapping use case sets. You cannot use the specialization arrow to indicate their overlap, since neither can do all that the other can do. This puts you back into the middle of the actor-role controversy.

Characterizing the primary actors

Just having a list of actors is not of much help to the designers. They should know what sorts of skills the users will have, so they can design the system behavior and the user interface to match. Teams that create an *Actor Profile Map* say they keep a better view of how their software will suit the needs of the end users. This is because they get to think about the skill base of their end users, and have those characterizations in front of them during development.

The simplest actor profile map has just two columns, as shown in the following example. Some people also capture other names or *aliases* the actors are known by. Variations of this list are discussed in Software for Use by Constantine and Lockwood.

A SAMPLE ACTOR PROFILE MAP:

Name	Profile: Background & Skills
Customer	Person off the street, able to use a touch-screen display, but not expected to operate a GUI with subtlety or ease. May have difficulty reading, be shortsighted, colorblind, etc.
Returned Goods Clerk	Person working with this software continuously. Touch-types, sophisticated user. May want to customize UI.
Manager	Occasional user. Used to GUIs but will not be familiar with any particular software function. Impatient.

4.3 Supporting actors

A supporting actor of a use case is an external actor that provides a service to the system under design. It might be a high-speed printer, it might be a web service, or it might be a group of humans who have to do some research and get back to us (for example, the coroner's office, which provides the insurance company with confirmation of a person's death). We used to call this a *secondary*

actor, but people found the term confusing. More people are now using the more natural term, supporting actor.

Identify supporting actors in order to identify the external interfaces the system will use, and the protocols that cross those interfaces. This feeds the other requirements, such as the data formats and external interfaces (see Figure 1.“"Hub-and-spoke" model of requirements” on page 28).

An actor can be primary actor in one use case and supporting actor in another.

4.4 The system under discussion, itself

The system under discussion itself is an actor, a special one. We usually refer to it by its name, e.g, *Acura*. Otherwise we say the system, system under discussion, system under design, or SuD. It gets named or described in the Design Scope field of the use case, and is referred to either by its name or just as the system inside the use case.

The SuD is not a primary or supporting actor for any use case, although it is an actor. There is not more to say about the SuD as an actor, since it was discussed at length in *Design Scope*.

4.5 Internal actors and white-box use cases

Most of the time, we treat the system under discussion as a *black box*, which we can not peek inside. Internal actors are carefully not mentioned. This makes good sense when we use the use cases to name requirements for a system that has not yet been designed.

There are occasions when we want to use the use-case form to document how the parts of the system cooperate to deliver the correct behavior. We would do this when documenting business processes, as in Use Case 5:“Buy Something (Fully dressed version)” on page 22, and Use Case 19:“Handle Claim (business)” on page 78. We might do this when showing the larger design for a multi-computer system, as in Use Case 8:“Enter and Update Requests (Joint System) .” on page 52.

In these circumstances, components of the system show up as actors.

When we look inside the system, and name the components and their behavior, we treat the system as a *white box*. Everything about writing use cases still works with a white box use case, it is just that we discuss the behaviors of the internal actors as well as the external actors. There are more than just two actors in a white-box use case, since the components of the system are being shown as well as the external actors.

It is extremely rare and usually a mistake, to write white-box use cases as behavioral requirements for a computer system to be designed.

Actor-Stakeholder Exercises

Exercise 12 Identify a use case for a vending machine in which the owner is the primary actor.

Exercise 13 * We are hired to create the requirements document for a new ATM. Name whether each item in the following list is a stakeholder, primary actor, supporting actor, the system under design, or not an actor at all (or multiple of the above).

The ATM
The customer
The ATM card
The bank
The front panel
The bank owner
The serviceman
The printer
The main bank computer system
The bank teller
The bank robber

Exercise 14 * The ATM is a component in a larger system. In fact, it is part of several larger systems. Repeat the previous exercise for one such containing system.

Exercise 15 "The PAF System" Personal Advisors, Inc. (a hypothetical company) is coming out with a new product, which will allow people to review their financial investment strategies, such as retirement, education funds, land, stock, etc. The idea is that the product, "Personal Advisor/ Finance" (PAF, for short) comes on a CD. The user installs it, and then runs various kinds of financial scenarios to learn how to optimize his or her financial future. The product can interrogate various tax packages, such as Kiplinger Tax Cut, for tax laws. Personal Advisors is setting up an agreement with Kiplinger to allow direct exchange. They are also setting up an agreement with various mutual fund and web stock services, such as Vanguard and E*Trade, to directly buy and sell funds and stocks over the web. Personal Advisors thinks it would also be a great idea to have a version of PAF that is web-active, on a pay-per-use basis.

Name and identify PAF's actors, primary actors, supporting actors, system under design, and containing system (a system having PAF as a component).

5. Three Named Goal Levels

We have seen that both the goals and the interactions in a scenario can be unfolded into finer and finer-grained goals and interactions. This is normal, and we handle it well in everyday life. The following two paragraphs illustrate how our goals contain sub- and sub-sub-goals.

> "I want this sales contract. To do that I have to take this manager out to lunch. To do that I have to get some cash. To do that I have to withdraw money from this ATM. To do that I have to get it to accept my identity. To do that I have to get it to read my ATM card. To do that I have to find the card slot."
>
> "I want to find the tab key so I can get the cursor into the address field, so I can put in my address, so I can get my personal information into this quote software, so I can get a quote, so I can buy a car insurance policy, so I can get my car licensed, so I can drive."

However normal this is in everyday life, it causes confusion when writing a use case. A writer is faced with the question, "What level of goal shall I describe?" at every sentence.

Giving names to goal levels helps. The following sections describe the goal level names and icons I have found useful, and finding the goal level you need at the moment. Figure 14.illustrates the names and visual metaphors I use.

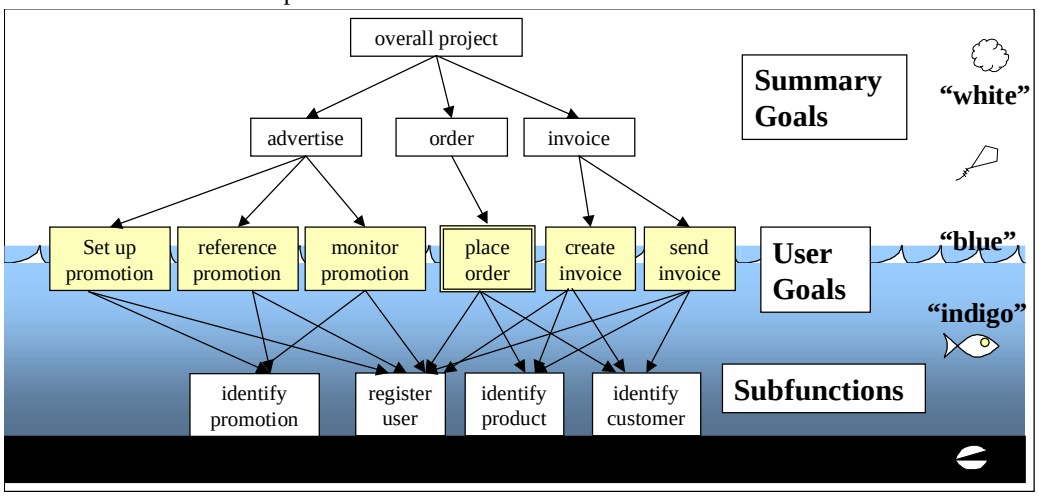


Figure 14. The levels of use cases. The use case set reveals a hierarchy of goals, the *ever-unfolding story.*

5.1 User-goals (blue, sea-level)

The *user goal* is the goal of greatest interest. It is the goal the primary actor has in trying to get work done, or the user has in using the system at all. It corresponds to "elementary business process" in the business process engineering literature.

A user goal addresses the question, "Can the primary actor go away happy after having done this?" For a clerk, it would be, "Does your job performance depend on how many of these you do today?" or the *coffee break test*: "After I get done with this, I can take a coffee break." In most circumstances, it passes the test:

* One person, one sitting (2-20 minutes).

Neither "Complete an on-line auction purchase" nor "Log on" generally count as a user goals. On-line auctions take several days, so fail the single-sitting test. Logging on 42 times in a row does not (usually) satisfy the person's job responsibilities or purpose in using the system.

"Register a new customer" and "Buy a book" are likely to be user goals. Registering 42 new customers has some significance to a sales agent. A book purchase can be completed in a single sitting.

So far it should all look easy. Faced with a slew of phrases on a whiteboard, or a use case that isn't looking right for some reason, it is easy to become uncertain. I find most people can find their bearings when expressing goal levels either in *colors* or *altitudes*.

The color gradient runs white to blue, to indigo, to black. The user goal is blue. Longer-range, higher-level goals, such as "Complete an online auction" and "Get paid for car accident" are white. Shorter-range, lower-level goals are indigo. Black is used to indicate that a goal is so low level that it would be a mistake to write a use case for it. "Hit tab key" would be one such.

The idea with the sea level metaphor is this: The sky goes upwards for a long distance above sea level, and the water goes down for a long distance below sea level, but there is only one level where sky and sea meet: sea level. The same holds for goals. There are many goal levels above the user goal, and many below. The goals that are really important to write are the user goals. Therefore, sea level corresponds to user goals. A cloud or a kite indicates higher than sea level, a fish or a clam indicates lower than sea level.

The system is justified by its support of sea-level goals. He is one such:

> "You are a clerk sitting at your station. The phone rings, you pick it up. The person on the other end says, '...'. You turn to your computer. What is on your mind at that moment is that you need to accomplish X. You work with the computer and the customer for a while, and finally accomplish X. You turn away from computer, say "Good-bye", and hang up the phone."

X is the user goal, blue, or sea-level. In accomplishing X, you accomplished a number of lower-level (indigo) goals. The person on the phone probably had a higher-level goal in mind, and accomplishing X was only one step in that. That person's higher level goals are white.

The sea-level / blue / user goals are incredibly important. It is worth a large amount of effort to understand and internalize just what constitutes one. You justify the existence of the system by the user goals it supports of various primary actors. The shortest summary of a system's function is the list of user goals it supports. That list is the basis for prioritization, delivery, team division, estimation and development.

Use cases will be written at levels above and below sea level. It is handy to think of the enormous number of lower level goals and use cases as being "underwater" -- it implies, particularly, that we don't really want to write them or read them.

A small, true story

> I was once sent over a hundred pages of use cases, all indigo ("underwater" was an appropriate phrase to describe them). That requirements document was so long and boring that it did not serve either its writers or readers. The sender later sent me the six sea-level use cases that had replaced them, and said everyone found them easier to understand and work with.

Two levels of blue

Usually, a blue use case has one white use case above it and several indigo use cases below it. However, a blue use case occasionally refers to another blue use case. I have only seen this occur in one sort of situation, but that situation shows up repeatedly.

Suppose I walk past the video rental store, doing some errands. I think, "I might as well register now, while I'm here." So I walk in and ask to *Set up a Membership*. That is my user goal, the blue use case. The next week, I go in with my membership card and *Rent a Video*. I execute the two user goals on different days.

However, you rent differently. You walk into the video store and want to *Rent a Video*. The clerk asks, "Are you a member?" You say, no, and so the clerk has you *Set up a Membership* within the process of renting your first video. *Set up a Membership* is a step inside *Rent a Video*, even though both are blue goals.

This "register a person in passing" is the only situation in which I recall seeing a blue use case inside a blue use case. When asked about this, I say that both are at sea level, but *Rent a Video* sits at the top of the wave in Figure 14., and *Set up a Membership* sits in the trough of the wave :-).

5.2 Summary level (white, cloud / kite)

Summary[1]*-level* goals involve multiple user goals. They serve three purposes in the describing the system:

- They show the context in which the user goals operate,
- They show life-cycle sequencing of related goals,
- They provide a table of contents for the lower-level use cases, both lower white use cases and blue use cases.

Summary use cases are white on the color gradient. White use cases have steps that are white, blue, or, occasionally, even indigo ("Log in" is an indigo goal likely to be found in a white use case). I have not found it useful to distinguish between various levels of white, but occasionally a speaker will say something like, "That use case is *really* white, 'way-up-in-the-clouds' white." In terms of the sea-level metaphor, we would say that most summary use cases are "like a kite, just above sea level", and others are "way up in the clouds".

Summary use cases typically execute over hours, days, weeks, months, or years. Here is the main scenario from a long-running use case, whose purpose is to tie together blue use cases scattered over years. You should be able to recognize the graphics as highlighting that the use case deals with the company as a black box, and that the goal level is very white (up in the clouds). The phrases in italics are lower-level use cases being referred to.

USE CASE 18: OPERATE AN INSURANCE POLICY

Primary Actor: The customer
Scope: The insurance company ("MyInsCo")
Level: summary ("white")
Steps:
1. Customer *gets a quote for a policy.*
2. Customer *buys a policy.*
3. Customer *makes a claim against the policy.*
4. Customer *closes the policy.*

Other examples of white use case are

* Use Case 19:“Handle Claim (business)” on page 78,
* Use Case 20:“Evaluate Work Comp Claim” on page 79, and
* Use Case 21:“Handle a Claim (systems)” on page 80.

1.(In previous writing, I used both "strategic" and "summary". I recently decided "summary" causes the least confusion, and chose that word for the book.)

The outermost use cases revisited

Earlier, I recommended that you write a few outermost use cases for whatever system you are designing. Here is the more precise description of finding those use cases.

5 Start with a user goal.

6 Ask, "what (preferably outside the organization) primary actor AA does this goal really serve?" Actor AA is the u*ltimate primary actor* of the use cases that we are about to collect.

7 Find the outermost design scope S such that AA is still outside S. Name the scope S. I typically find three such outermost design scopes:

* the company,
* the software systems combined,
* the specific software system being designed.

8 Find all the user goals having ultimate primary actor AA and design scope S.

9 Work out the summary goal GG that actor AA has against system S.

10 Finally, write the summary use case for goal GG of actor AA against system S. This use case ties together a number of your sea-level use cases.

All told, there are usually only about four or five of these topmost use cases (GG) even in the largest systems I have been associated with. They summarize the interests of three or four ultimate primary actors (AA):

* the customer as outermost primary actor to the company,
* the marketing department as outermost primary actor to the software systems combined,
* the security department to the software system itself.

These outermost use case are very useful in holding the work together, and I highly recommend writing them, for the reasons given earlier. They will not, however, provide your team with the functional requirements for the system to be built. Those reside in the user goal (blue) use cases.

5.3 Subfunctions (indigo/black, underwater/clam)

Subfunction-level goals are those required to carry out user goals. Include them only as you have to. They are needed on occasion for readability, or because many other goals use them. Examples of subfunction use cases are "Find a product", "Find a Customer", "Save as a file." See, in particular, the unusual indigo use case Use Case 23:“Find a Whatever (problem statement)” on page 86.

Subfunction use cases are underwater, indigo. Some are *really* underwater, so far underwater that they sit in the mud on the bottom. Those we color black, to mean "this is so low level, please don't even expand it into a use case" ("It doesn't even swim... it's a clam!"). It is handy to have a special name for these ultra-low-level use cases, so that when someone writes one, you can indicate it shouldn't be written, that its contents ought to rolled into another use case.

Blue use cases have indigo steps, and indigo use cases have deeper indigo steps (Figure 15.). That figure also shows that to find a higher goal level for your goal phrase, you answer the question "Why is the actor doing this?". This "how/why" technique is discussed more in 5.5"Finding the right goal level" on page 75.5.5

Note that even the farthest underwater, lowest subfunction use case has a primary actor that is outside the system. I wouldn't bother to mention this, except that people occasionally talk about subfunctions as though they were somehow internal design discussions or without a primary actor. A subfunction use case follows all the rules for use cases. It is probable that a subfunction has the same primary actor as the higher-level use case that refers to it.

Summarizing goal levels

For now, three points about goal levels are important.

- Put a lot of energy into detecting the sea-level use cases. These are the important ones.
- Write a few outermost use cases, to provide context for the others.
- Don't make a big fuss over whether your favorite phrase among the system requirements sentences "makes it" as a use cases title.

Making it as a use case title does not mean "most important requirement", and not becoming a use case title does not mean unimportant. I see people upset because their favorite requirement is *merely* a step in a use case, and did not get promoted to being a use case that is tracked on its own.

Don't fuss about this. One of the points of the goal model of use cases is that it is a relatively small change to move a complex chunk of text into its own use case, or to fold a trivial use case back into a higher-level use case. Every sentence is written as a goal, and every goal can be unfolded into its own use case. We cannot tell by looking at the writing which sentences have been unfolded into separate use cases, and which have not (except by following the links). This is good, since it preserves the integrity of the writing across minor changes of writing. The only goals that are guaranteed to have their own use cases are the blue ones.

5.4 Using graphical icons to highlight goal levels

In *Using graphic icons to highlight the design scope*, I showed some icons that are usefully put to the left of the use case title. Because goal levels are at least as confusing, I put a goal-level icon at the top right of the use case title. This is in addition to filling the fields in the template. My experience is that readers (and writers) can use all the help they can get in knowing the level.

In keeping with the altitude nomenclature, I separate five altitudes. You will only use the middle three, in most situations.

- Very summary (very white) use cases get a cloud,  . Use this on that rarest of occasions, when you see that the steps in the use case are themselves white goals.
- Summary (white) use cases get a kite,  . This is for most summary use cases, whose steps are blue goals.
- User-goal (blue, sea-level) use cases get waves,  .
- Subfunction (indigo) use cases get a fish,  . Use this for most indigo use cases.
- Some subfunctions (black) should never be written. Use a clam,  , to mark a use case that needs to be merged with its calling use case.

With these icons, you can mark the design scope and goal level even on UML use case diagrams, as soon as the tool vendors support them. If your template already contains Design Scope and Goal Level fields, you may choose to use them as redundant markers. If your template does not contain those fields, then add them.

5.5 Finding the right goal level

Finding the right goal level is the single hardest thing about use cases. Focus on these:

* Find the user's goal.
* Use 3-10 steps per use case.

Find the user’s goal

In all of the levels of goal, only one level stands out from the others:

> You are describing a system, whether a business or a computer. You care about someone using the system. That person wants something from your system NOW. After getting it, they can go on and do something else. What is it they want from your system, now?

That level has many names. In business process modeling, they call it an *elementary business process*. In French one would say the system's *raison d'être*. In use cases, it the *user's goal*.

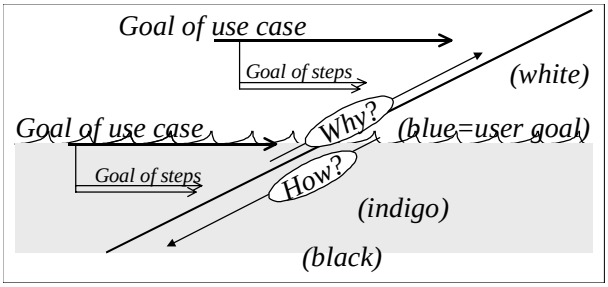
The very first question is, "Is this what the primary actor really wants from the system, now?" For most people's first drafts of use cases, the answer is "no." Most beginners draft underwater use cases, thinking they are at sea level. To find the higher level goal, ask:

* "What does the primary actor really want?", or
* "Why is this actor doing this?"

The answer to the question might be the actor's real goal. But ask the question again, until you find the real user goal. The interesting thing is that even though the tests for a user goal are subjective, people soon come to consensus on the matter. Experienced people nominate surprisingly similar answers for user goals. It seems to be a stable concept.

Merge steps, keep asking "why"

Figure 15. Ask "why" to shift levels.



The second point of focus is the length of the use case. Most well-written use cases have 3-8 steps. I have never seen one longer than 11 steps that didn't get better when it was shortened. I have no idea why this is. I doubt there is anything magical about those numbers. If I were to guess, I would say that people do not tolerate or think in terms of processes that take more than 10 intermediate steps. I keep waiting for a legitimate counter-example, just to prove that the numbers have no deep significance.

Whatever the reason, use the observation to improve your writing. If you have more than 10 steps, you probably included user interface details, or wrote action steps at too low a level.

* Remove the user interface details. Show the actor's intent, not movement.

* Raise the goal level by asking the "why" question to find the next higher goal level.
* Merge steps.
* Compare your use cases with the writing samples in the next section and in Chapter 19.“Mistakes Fixed” on page 185.

Goal-Level Exercises

Exercise 16 * "*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.*" Name a white, a blue and an indigo goal Jenny.

Exercise 17 * List at least ten goals that the ATM’s various primary actors will have with respect to the ATM, and label their goal levels.

Exercise 18 List the summary and user goals of all the primary actors for the PAF software package (PAF system described in Exercise 15 on page 68). Identify the highest-level, outermost, actor-scope-goal combinations.

5.6 A longer writing sample: "Handle a Claim" at several levels

I would like to thank the people at Fireman’s Fund Insurance Corporation in Novato, California for allowing me to include the following use cases as writing samples. They were written by claims handling professionals directly from the field, working with business analysts from the IT department and the technical development staff. The field staff had insights about the use of the system that the IT staff could not have guessed, and the IT staff help them make the writing precise enough. Between them, they combined field, corporate and technical viewpoints.

The writing team included Kerry Bear, Eileen Curran, Brent Hupp, Paula Ivey, Susan Passini, Pamela Pratt, Steve Sampson, Jill Schicktanz, Nancy Jewell, Trisha Magdaleno, Marc Greenberg, and Nicole Lazar, Dawn Coppolo, and Eric Evans. I found that the team demonstrated that usage experts with no software background can work with IT staff in writing requirements.

I include five use cases over the next pages, to illustrate the things we have discussed so far, particularly the use of design scopes and goal levels. These use cases also illustrate good writing style for steps and extensions. I provide a commentary before each use case, indicating some points of interest or contention.

The set starts with a cloud-level white-box business use case, which shows business processes involved in handling a claim. Watch how the goals go into lower levels, and the system scope shrinks from "company operations" to "all computer systems" to just the system under design. The

underlined phrases are references to other use cases. The template was modified a little to make the main success scenario closer to the top, faster to read.

Commentary on Use Case 19:: The system is the operations of the company. The computer system is not even mentioned. The use case will be used by the business to anchor its business procedures, and to search for a way to use the computer to facilitate its operations. At the moment, this use case is only in its first stage of sketching. As usual, the main success scenario looks trivial. It should, because it shows how things work in the best success situation! The interesting bits will show up in the failure conditions, and in how the company uses this information to nominate improvements to its IT support of operations. Note the stakeholders.

USE CASE 19: HANDLE CLAIM (BUSINESS)

Scope: Insurance company operations
Level: Business summary
Release: Future **Status**: Draft **Revision**: Current
Context of use: Claims Adjuster handles claim.
Preconditions: A loss has occurred
Trigger: A claim is reported to Insurance company

Main Success Scenario:
1. A reporting party who is aware of the event registers a loss to Insurance company.
2. Clerk receives and assigns the claim to a claims adjuster.
3. The assigned Claims Adjuster
   conducts an investigation
   evaluates damages
   sets reserves
   negotiates the claim
   resolves the claim and *closes it.*

Extensions:
   to be written
Success Guarantee: Claim is resolved and closed
Minimal Guarantee:
Stakeholders & interests:
   Insurance company Divisions who sell Insurance company policies
   Insurance company Customers who have purchased policies
   Department of Insurance who sets market conduct
   Claimants who have loss as a result of act of an insured
   Insurance company Claims Division
   Future Customers

Commentary on Use Case 20:: Here is another a business use case. The system under discussion is still the operations of the company, but the goal is at a lower level than the preceding one. In this, the work of an adjuster that may take days, weeks or months is shown. It is a kite-level summary use case - it contains many single-sitting activities.

The writer does not mention the computer directly, but only names the goals of the adjuster. The team must make a leap of imagination to invent what in this process the computer can help with. This use case is the raw material for their act of invention.

Step 7 shows a step that was added because of the interests of the Department of Insurance.

USE CASE 20: EVALUATE WORK COMP CLAIM

Scope: Insurance company Operations
Level: -White (summary, above single user goal level)
Context of use: Claims Adjuster completes thorough evaluation of the facts of a loss.
Primary Actor: Claims Adjuster
Preconditions:
Trigger:
Main Success Scenario:
Please Note: Investigation has ideally been completed prior to evaluation,
although the depth of the investigation can vary from claim to claim.

1. Adjuster reviews and evaluates the medical reports, lien documents, benefits paid to date and other supporting documents.
2. Adjuster rates the permanent disability by using a jurisdictional formula to determine % of disability.
3. Adjuster sums the permanent disability owed, taking credit for advances and payment of liens to arrive at the claims full value.
4. Adjuster determines the final settlement range.
5. Adjuster checks reserves to make sure they are in line with settlement range.
6. Adjuster seeks authorization for settlement and reserve increase if above their authority levels.
7. Adjuster documents the file.
8. Adjuster sends any correspondence and or documentation to parties involved as necessary.
9. Adjuster continues to document file regarding all settlement activity.

Extensions: ...

Frequency of occurrence: Every claim is evaluated, this can happen several times a day.
Success Guarantee: Claim is evaluated and settlement range determined.
Minimal Guarantee: Additional investigation or medical evaluations are completed until claim is ready to be re-evaluated for settlement.
Stakeholders & interest:

Claimant, wants maximum settlement
Adjuster, wants lowest legitimate settlement
Insurance company, same
Attorney (defense and plaintiff) Insureds,
Division of Insurance, and state governing offices (each state has a separate governing department that oversees the administration of benefits and fairness of settlements), wants fairness and adherence to procedures.

Open Issues: Jurisdictional issues will have to be addressed when writing the business rules

---

Commentary on Use Case 21:: To many people on the project, this system use case seemed so vague as to be useless. However, it paid for its writing time in several ways.

It glues together a number of user-goal use cases, showing how they fit within the business guidelines. It describes closing, purging and archiving a claim, which was a mystery to a number of people on the project. Although the last three steps do not generate work for the programmers, they are part of the story of handling a claim, useful contextual information for every reader.

It put into the official files certain business rules which were unknown to some of the team. The team had expended 3 work hours the day before trying to guess those rules. Once this use case was written, the rest of the team was saved many more work hours of discussion on the topic.

This use case serves as an introduction and table of contents to new readers, people ranging from the company executives to new hires. Executives can see that the key processes are included. Newcomers can learn how the company worked, and drill down into the user-goal use cases. Extension *a1 was interesting, since it called out a failure handling use case that couldn't be written by the claims adjustors, but had to be given to the technical group to write.

USE CASE 21: HANDLE A CLAIM (SYSTEMS)

Scope: "System" means all computer systems combined
Level: Summary (white)
Release: 1st **Status:** Ready for review **Revision:** Current
Context of use: Customer wants to get paid for an incident
Primary Actor: Customer
Preconditions: none
Trigger: Customer reports a claim
Main Success Scenario:
1. Customer reports a claim (paper, phone or fax) to Clerk
2. Clerk finds the policy, captures loss in System, and assigns an Adjuster.

3. Adjuster investigates the claim and updates claim with additional information.
4. Adjuster enters progress notes over time.
5. Adjuster corrects entries and sets monies aside over time.
6. Adjuster receives documentation including bills throughout the life of the claim and enters bills.
7. Adjuster evaluates damages for claim and documents the negotiation process in System.
8. Adjuster settles and closes claim in System.
9. System purges claim 6 months after close.
10. System archives claim after time period.

Extensions:
*a. At any time, System goes down:
    *a1. System group repairs system.
1a. Submitted data is incomplete:
    1a1. Insurance company requests missing information
    1a2. Claimant supplies missing information
     1a2a: Claimant does not supply information within time period:
     1a2a1. Adjuster closes claim in System.
2a. Claimant does not own a valid policy:
    2a1. Insurance company declines claim, notifies claimant, updates claim, closes claim.
3a. No agents are available at this time
    3a1. (What do we do here?)
8a. Claimant notifies adjuster of new claim activity:
    8a1. Clerk reopens claim. Reverts to step 3.

Technology and Data Variations List:

Frequency of occurrence: to be documented
Success Guarantee: Claim is closed, settled and archived.
Minimal Guarantee: Claim closed but may be reopened later.
Stakeholders & interests:
The company - make smallest accurate settlement.
Customer - get largest settlement.
Depart of Insurance - ensure correct procedures

Business Rules:
Data descriptions: Will be defined in other use cases
UI Links:
Open Issues: What are the time periods for archiving claims?

Commentary on Use Case 22:: This is one of the most complex use cases I have seen. It shows why it is handy that use cases are written in natural language prose.

The first source of complexity is the sequencing. A clerk on the phone talking to a distraught customer must be able to enter information in any order at all, while still attempting to follow a standard question sequence. Simultaneously, the computer is to use information as it is entered to do whatever processing can be done, such as pulling the customer's records, and assigning a claim number and an adjuster. The writers wrote at least four completely versions of this use case, trying to be clear, show the normal work path, and show the computer working asynchronously. Perhaps on the 7th or 8th revision, they would have found something better, but they felt they had passed the point of diminishing returns and stopped with this version.

This use case makes several invocations to the use case *Find a whatever*, each time mentioning a different thing to find, and different search criteria. The team came up with an ingenious solution to avoid rewriting the standard steps for searching for something: match lists, sorting criteria, resorting, researching, no items found, etc. I ask you to do the same in Exercise 19, below.

Handling of extension, '*a. Power failure' generated surprising new requirements questions. It introduced the notion of intermediate saves. Having an intermediate save suddenly implied the clerk could search for one later, which was a surprise to the people writing the use case. That introduced questions of storing and searching for temporarily saved losses, more surprises for the team. It all ended with the failure condition '6b', which dealt with time-out on a temporarily saved loss, and confronted the writers with the very detailed question, "What are the business rules having to do with an allegedly temporarily entered loss, which cannot be committed because it is missing key information, but shouldn't be deleted because it passed the minimum entry criteria?" The team toyed with the unacceptable alternatives, from not doing intermediate saves to deleting the loss, before settling on this solution.

Extension '1c' shows failures within failures. The writers could have turned this into its own use case. They decided that would introduce too much complexity into the use case set: a new use case would have to be tracked, reviewed and maintained. So they made the extension of the extension. Many people take use case extensions this far for that reason. They also comment that this is about as far as they feel comfortable before making the extension into its own use case.

Extension '2-5a' shows how malleable the medium is. The condition could arise in any of the steps 2-5. How should they write them - once for each time it could occur? That seemed such a waste of energy. The solution was just to write '2-5a' and '2-5b'. It was clear to the readers what this meant.

USE CASE 22: REGISTER LOSS

Scope: "System" means the claims-capturing computer system

Level: Blue (User Goal)
Release: 2 **Status:** Reviewed **Revision:** Current
Context of use: Capture loss fully
Primary Actor: Clerk
Preconditions: Clerk already logged in.
Trigger: Clerk has started entering loss already
Success Guarantee: loss information is captured and stored
Minimal Guarantee: Nothing happens.
Stakeholder & interest: as before
Main Success Scenario:
To speed up the clerk's work, the System should do its work asynchronously, as soon as the required data is captured. The Clerk can enter data in any order to match the needs of the moment. The following sequence is foreseen as the most likely.

1. Clerk enters insured's policy number or else name and date of incident. System populates available policy information and indicates claim is matched to policy.
2. Clerk enters basic loss information, System confirms there are no existing, possibly competing claims and assigns a claim number.
3. Clerk continues entering loss information specific to claim line.
4. Clerk has System pull other coverage information from other computer systems.
5. Clerk selects and assigns an adjuster.
6. Clerk confirms they are finished, System saves and triggers acknowledgement be sent to agent.

Extensions:
*a. Power failure during loss capture:
    *a1. System autosaves intermittently (Possibly at certain transaction commit points, open issue.)
*b. Claim is not for our company to handle:
    *b1. Clerk indicates to System that claim is entered "only for recording purposes" and either continues or ends loss.
1a. Found policy information does not match the insured's information:
    1a1. Clerk enters correct policy number or insured name and asks System to populate with new policy index information.
1b. Using search details, System could not find a policy:
    1b1. Clerk returns to loss and enters available data.
1c. Clerk changed policy number, date of loss or claim line after initial policy match:
    1c1. System validates changes, populates loss with correct policy information, validates and indicates claim is matched to policy
    1c1a. System cannot validate policy match:

1c1a1. System warns Clerk.
1c1a2. Clerk Finds the policy using the search details for "policy"
1c2. System warns Clerk to re-evaluate coverage.
1d. Clerk wants to restart a loss which has been interrupted, saved or needs completion:
1d1. Clerk Finds a loss using search details for "loss".
1d2. System opens it for editing.
2-5a. Clerk changes claim line previously entered and no line specific data has been entered:
2-5a1. System presents appropriate line specific sections of loss based on Clerk entering a different claim line.
2 - 5b. Clerk changes claim line previously entered and there is data in some of the line specific fields:
2-5b1. System warns that data exists and asks Clerk to either cancel changes or proceed with new claim line.
2-5b1a. Clerk cancels change: System continues with the loss.
2-5b1b. Clerk insists on new claim line: System blanks out data which is line specific (it keeps all basic claim level data).
2c. System detects possible duplicate claim:
2c1. System displays a list of possible duplicate claims from within loss database.
2c2. Clerk selects and views a claim from the list. This step may be repeated multiple times.
2c2a. Clerk finds that the claim is a duplicate:
2c2a1. Clerk opens duplicate claim from list of claims for editing if not yet marked completed (base on Clerks security profile). Clerk may delete any data in previously saved.
2c2b. Clerk finds that the claim is not a duplicate: Clerk returns to loss and completes it.
2d. Preliminary loss information is changed after initial duplicate claim check is done:
2d1. System performs duplicate claim check again.
2e. Clerk can save the loss any time before completion of steps 2 through 6. (some reasons to save may be just a comfort level or that the Clerk must interrupt entry for some reason (e.g.. claim must be handled by & immediately transferred to higher level adjuster)).
2e1. Clerk has System save the loss for completion at a later time.
4-5a. Either, claim line or loss description (see business rules) are changed after coverage was reviewed by Clerk:
4-5a1. System warns Clerk to re-evaluate coverage.
6a. Clerk confirms they are finished without completing minimum information:
6a1. System warns Clerk it cannot accept the loss without date of loss, insured name or policy number and handling adjuster:
6a1a. Clerk decides to continue entering loss or decides to save without marking complete.
6a1b. Clerk insists on existing without entering minimum information: System discards any intermediate saves and exits.
6a2. System warns Clerk it cannot assign claim number without required fields (claim line, date of loss, policy number or insured name): System directs Clerk to fields that require entry.
6b. Time-out: Clerk has saved the loss temporarily, intending to return, System decides it is

time to commit and log the loss, but handling adjuster has still not been entered:
6b1. System assigns default adjuster (see business rule)

Frequency of occurrence: ??
Business Rules:
*. When does saved loss goes to main system (timelines)?
1. Minimum fields needed for saving an loss (and be able to find it again) are: ...
2. Claim number, once assigned by system, cannot be changed.
2. Business rules for manual entry of claim number - needed?
4. Loss description consists of two fields, one being free form, the other from a pull down menu.
4. System should know how to find coverage depending on policy prefix
6. Required fields in order to confirm a loss is finished are: ....
6b. Rules for default adjuster are: ...

Data descriptions used: Search details for policy, Policy index information, Preliminary loss information, claim-line-specific loss information, additional information, Search details for loss, duplicate claim check criteria, list of possible duplicate claims, a claim from the list, Finds a policy, Get Coverage
UI Links:
Owner: Susan and Nancy
Critical Reviewers: Alistair, Eric, ...
Open Issues:
How often does it autosave.
Agent acknowledgement cards, where and how do they print etc.?

USE CASE 23: FIND A WHATEVER (PROBLEM STATEMENT)

The project team decided it would be silly to write almost identical use cases Find a Customer, Find a Policy, etc. So they created a generic mechanism that every writing team used.

They decided that writing any sentence of the form Find a..., such as Find a Customer or Find a Product, would mean that the use case Find a Whatever would be called, with an implicit substitution of whatever was really needed for the word *whatever*. Each different use needed its own searching, sorting and display criteria, so the writer would put the data and search restrictions on a different - hyperlinked - sheet. A sample sentence would therefore read as

> Find a customer using Customer search details.

With this neat convention, the logistical details of *Find a Whatever* could be written just once, and used in many similar but different contexts. The developers were happier, too, since they knew that all the searches would use the same mechanism, and they could develop a common one.

I want you to try your hand at doing this, so I'll defer showing this team's solution. Work the exercise before looking at this team's solution, which is discussed in Section 14.2"Parameterized use cases" on page 150.

Exercise 19 Find a Whatever. Write the use case for *Find a Whatever*, whose trigger is that the user decides to locate whatever they are looking for. The use case should allow the user to enter searching and sorting information. It should deal with all the situations that might arise, and in the success case, end with a whatever being identified by the computer, for whatever use the calling use case specifies next.

6. Preconditions, Triggers, Guarantees

6.1 Preconditions

The *Precondition* of the use case announces what the system will ensure is true before letting the use case start. Since it is enforced by the system and known to be true, it will not be checked again during the use case. A common example is, "the user has already logged on and been validated".

Generally, having a precondition indicates that some other use case has already run, to set up the condition. Let's say that use case 92, *Place an Order*, relies on a precondition ("logged on"). I would immediately look to see which use case set it up (perhaps use case 91 is called *Log On*). I usually create a higher-level use case that mentions both use cases 91 and 92, so the reader can see the way the two fit together. In this example, it might be summary use case 90, *Use the application.* We get, in this example, the following structure (I abbreviate the template to show just the relevant parts). Note the goal levels of the three use cases.

Use case 90: *Use the application*
Level: Summary (white)
Precondition: none
1. Clerk *logs on.*
2. Clerk *places an order.*

Use case 91: *Log On*
Level: Subfunction (indigo)
Precondition: none
...

Use case 92: *Place an Order*
Level: User goal (blue)
Precondition: Clerk is logged on
...

Not everybody follows my habit of writing the higher level use case to glue together the lower level ones. That means you are likely to pick up only use cases 91 and 92, in that example. You have to be able to deduce from the writing that the precondition is correct and will be enforced.

Let's continue that example to show a use case which sets a condition in one step, and relies on that condition in another step. Once again, the sub use case expects the condition to be true and will not check for mistakes.

Use case 92: *Place an Order*
Level: User goal (blue)
Precondition: Clerk is logged on
1. Clerk identifies customer, system pulls up customer record.
2. Clerk enters order information.
3. System *calculates charges.*
...

Use case 93: *Calculate Charges*
Level: Subfunction (indigo)
Precondition: Customer is established and known to system, order contents are known.
1. System calculates base charge for order.
2. System calculates discount for customer.
...

This example illustrates how one use case relies upon information captured in a calling use case. The writer of *Calculate Charges* declared the information that is already available and can go ahead and refer to the customer information.

Note on the example used: The alert reader will be suspicious about the use case *Calculate Charges*. I declared it as color indigo, but so far in the writing there is little to justify it even having its own use case. If I, as the writer, don't uncover complicated interactions with the user, or interesting failure cases, I would reclassify it from indigo to black (fish to clam, using the icons). That is the signal to merge the text back into the use case *Place an Order* and eliminate the use case *Calculate Charges* altogether.

Write a precondition is written as simple assertions about the state of the world at the moment the use case opens. Suitable examples are:

"Precondition: The user is logged on."
"Precondition: The customer has been validated."
"Precondition: The system already has located the customer's policy information."

A common mistake is to write into the precondition something that is often true, but not necessarily true.

Suppose we are writing *Request benefits summary*, with primary actor Claimant. We might think that a Claimant would surely have submitted at least one claim or bill before asking for the benefits summary. However, that is not always the case, the system cannot ensure it, and, in fact, it actually isn't essential. Claimants should be able to ask for their benefit summary at any time. So it is wrong to write "Precondition: The claimant has submitted a bill."

6.2 Minimal Guarantees

The Minimal Guarantees are the least promises the system makes to the stakeholders, particularly for when the primary actor's goal cannot be delivered. They hold for when the goal is delivered, of course, but they are of real interest when the main goal is abandoned. Most of the time, two or more stakeholders have to be addressed in the minimal guarantees, examples being the user, the company providing the system, and possibly a government regulatory body.

Do not bother listing in the Minimal Guarantees section all the ways the use case can fail. There are dozens of ways to fail, and little in common between them. All of the failure conditions and handling show up in the Extensions section, and it is both tiring and error prone to try to keep the two lists synchronized. The purpose of this section of the template is to announce what the system promises.

The most common Minimal Guarantee is, "The system logged how far it got." Don't think that logging transaction failures is either obvious or trivial. System logs are often forgotten in the requirements description, and sometimes rediscovered by the programmers. The log is crucial to the system owners as well as the user. The system uses the log to continue a transaction once normal operating conditions resume. The stakeholders use it to settle disputes . The use case writer should discover the need for log either by investigating the stakeholders' interests, or when brainstorming failure conditions.

The Minimal Guarantee is written as a number of simple assertions that will be true at the end of any running of the use case. It shows the interests of each stakeholder being satisfied.

> Minimal Guarantee: Order will be initiated only if payment received.
> Minimal Guarantee: If the minimum information was not captured, the partial claim has been discarded and no log made of the call. If the minimum information was captured (see business rules), then the partial claim has been saved and logged.

The pass/fail test for the Minimal Guarantee is that the stakeholders would agree that their interests had been protected under failure conditions for this goal.

Minimal Guarantee Exercises

Exercise 20 * Write the Minimal Guarantee for withdrawing money from the ATM.

Exercise 21 Write the Minimal Guarantee for the PAF system's main use case (PAF system described in Exercise 15 on page 68).

Exercise 22 Write the Minimal Guarantee for a sea-level use case for your current system. Show it to a colleague and have them analyze it with respect to the interests of the stakeholders.

6.3 Success Guarantee

The Success Guarantee states what interests of the stakeholders are satisfied after a successful conclusion of the use case, either at the end of the main success scenario or the end of a successful alternative path. The Success Guarantee is generally written to add onto the Minimal Guarantee: all of the Minimal Guarantee is delivered, *and* some extra conditions are true. Those additional conditions include at least the goal stated in the use case title.

Like the Minimal Guarantee, the Success Guarantee is written as simple assertions that apply at the end of a successful running of the use case. It should show the interests of each stakeholder being satisfied. Suitable examples are:

> Success Guarantee: The claimant will be paid the agreed-upon amount, the claim closed, the settlement logged.
> Success Guarantee: The file will be saved.
> Success Guarantee: The system will initiate an order for the customer, will have received payment information, and logged the request for the order."

The pass/fail test for the Success Guarantee section is that the stakeholders would agree, from reading the Success Guarantee, that their interests had been satisfied.

The best way to uncover the success guarantee is to ask, "What would make this stakeholder *unhappy* at the end of a successful run?" That question is usually easy to answer. Then write the negative of that. To see an example of this, do Exercise 23 and then read the discussion in the Answers to Exercises.

Success Guarantee Exercises

Exercise 23 * Write the Success Guarantee for withdrawing money from the ATM.

Exercise 24 Write the Success Guarantee for the PAF system's main use case (PAF system described in Exercise 15 on page 68).

Exercise 25 Write the Success Guarantee for a sea-level use case for your current system. Show it to a colleague and have them analyze it with respect to the interests of the stakeholders.

6.4 Triggers

The Trigger states what event gets the use case started. Sometimes the trigger precedes the first step of the use case. Sometimes it is the first step of the use case. To date, I have not seen a convincing argument that one form can be applied in all cases. Nor have I noticed any confusion from choosing one way or the other. You will have to develop your personal or project style.

Consider a person using a banking machine, the ATM. The ATM system wakes up only when the person inserts their card. It is not meaningful to say that the trigger is when someone decides to use the ATM. The trigger, "Customer inserts card,", is also the first step in the use case.

Use Case: Use the ATM
Trigger: Customer inserts card
1. Customer inserts card with bank id, bank account and encrypted PIN.
2. System validates ...

Consider, however, a clerk sitting all day at a workstation showing a number of graphical icons to run different application programs. The trigger is that a customer calls with a particular request. This can written using either form, but I'll write it the second way to illustrate.

Use Case: Log a complaint
Trigger: Customer calls in a complaint.
1. Clerk calls up the application.
2. The system brings up the clerk's recent complaints list.
...

7. Scenarios and Steps

A set of use cases is an ever-unfolding story of primary actors pursuing goals. Each individual use case has a criss-crossing storyline that shows the system delivering the goal or abandoning it. The criss-crossing storyline is presented as a main scenario and a set of scenario fragments as extension to that. Each scenario or fragment starts from a triggering condition that indicates when it runs, and goes until it shows completion or abandonment of the goal it is about. Goals come in all different sizes, as we have seen, so we use the same writing form to describe pursuit of any size of goal, at any level of scenario.

7.1 The Main Success Scenario, Scenarios

Main success scenario as the simple case

We often explain things to other people by starting with an easy to understand description and then adding,

> "Well, actually, there is a little complication. When such-and-such occurs, what really happens is ...".

People do very well with this style of explanation, and it is the way we write the criss-crossing storyline that is the use case. We first write a top-to-bottom description of one easy to understand and fairly typical scenario in which the primary actor's goal is delivered and all stakeholders' interests are satisfied. This is the *main success scenario*.

All other ways to succeed, and handling of all failures, are described in the extensions to the main success scenario.

Common surrounding structure

The main success scenario and scenario extensions sit within the same structure. That consists of:

- A condition under which the scenario runs. For the main success scenario, this is the use case precondition plus the use case trigger. For an extension scenario, this is the extension condition (perhaps with the step number or place in the scenario where that condition applies).

- A a goal to achieve. For the main success scenario, this is exactly the use case name, satisfying, of course, the stakeholders' interests. For an extension scenario, the goal is either to complete the use case goal or to rejoin the main success scenario after handling the condition.
- A set of action steps. These form the body of the scenario, and follow the same rules in every scenario or scenario fragment.
- An end condition. The goal is achieved at the end of the main success scenario. A scenario fragment may end with the goal either being achieved or abandoned.
- A possible set of extensions, written as scenario fragments. Extensions to the main success scenario are placed in the *Extensions* section of the use case template. Extensions to extensions are placed directly inline, inside or just after the extension body.

Here are two extracts from Use Case 1 to show the similarity. I have stripped these down to show just the similarities of their surrounding structures.

A main success scenario:

Use Case: Buy stocks over the web
Precondition: User already has PAF open.
Trigger: User selects "buy stocks"
1. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwabb, etc.) from user.
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site, and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.

An extension to the main success scenario:

3a. Web failure of any sort during setup:
3a1. System reports failure to user with advice, backs up to previous step.
3a2. User either backs out of this use case, or tries again.

In this chapter, we look in detail at the body of the scenario, which consists of action steps.

The scenario body

Every scenario or fragment is written as a sequence of goal-achieving actions by the various actors. I say *sequence* for convenience, but we are allowed to add notes to describe that steps can go in parallel, be taken in different orders, repeat, even that some are optional.

As per "Contract between Stakeholders with Interests" on page 40, any one step will describe

* an interaction between two actors ("Customer enters address"),
* a validation step to protect an interest of a stakeholder, ("System validate PIN code"),

* an internal change to satisfy a stakeholder ("System deducts amount from balance").

Here is an example of a typical sort of Main Success Scenario, borrowed from Use Case 22:“Register Loss” on page 83. You will notice that steps 1, 3, 5-8 are interactions, step 4 is a validation, and steps 2 and 9 are internal changes.

1. Clerk enters insured's policy number or else name and date of incident.
2. System populates available policy information and indicates claim is matched to policy.
3. Clerk enters basic loss information.
4. System confirms there are no competing claims and assigns a claim number.
5. Clerk continues entering loss information specific to claim line.
6. Clerk has System pull other coverage information from other computer systems.
7. Clerk selects and assigns an adjuster.
8. Clerk confirms they are finished.
9. System saves and triggers acknowledgement be sent to agent.

Each action step is written to show a simple, active action. I liken it to describing a soccer match: *Person 1 kicks ball to person 2. Person 2 dribbles ball. Person 2 kicks ball to person 3.*

Once you master your own writing of the three kinds of action steps, you are pretty well set for your writing style. The same style is used for action steps in every part of any use case, whether main success scenario or extension, whether business or system use case, high-level and low-level.

7.2 Action Steps

The action steps that make up a well-written use case are written in one grammatical form, a simple action in which one actor accomplishes a task or passes information to another actor.

"User enters name and address."
"At any time, user can request the money back."
"The system verifies that the name and account are current."
"The system updates the customer's balance to reflect the charge."

Most of the time, the timing can be omitted, since steps generally follow one after the other.

There are many minor variations in ways to write the action steps, as you have already seen in the use case samples. However you choose to write, preserve the following characteristics in each step.

Guidelines for an action step

Guideline 1: It uses simple grammar

The sentence structure should be absurdly simple:

Subject ... verb... direct object... prepositional phrase.

Example:

> The system ... deducts ... the amount ... from the account balance.

That's all there is to it. I mention this matter because many people accidently leave off the first noun. If you leave off the first noun, it is no longer clear who is controlling the action (who has the ball). If your sentence is badly formed, the story gets hard to follow.

Guideline 2: It shows clearly, "Who has the ball"

A useful visual image is that of friends battering a soccer ball around. Sometimes person 1 kicks it to person 2, and then person 2 dribbles it a while, then kicks it to person 3. Occasionally it gets muddy, and one of the players wipes the mud off.

A scenario has the same structure. At each step one actor "has the ball". That actor is going to be the subject of the sentence, the first actor named, probably as the first or second word in the sentence. The "ball" is the message and data that gets passed from actor to actor.

The actor with the ball will do one of three things: kick to him/her/itself, kick it to someone else, clean off the mud. About half of the time, the step ends with another actor having the ball. Ask yourself, "At the end of the sentence, now who has the ball?"

It should always be clear in the writing, "Who has the ball."

Guideline 3: It is written from a bird's eye point of view

Beginning use case writers, particularly programmers who have been given the assignment of writing the use case document, often write the scenario as seen by the system, looking out at the world, and talking to itself. The sentences have the appearance, "Get ATM card and PIN number. Deduct amount from account balance."

Write the use case from a bird's eye point of view.

> The customer puts in the ATM card and PIN.
> The system deducts the amount from the account balance.

Some people like to use the other style that has this quality: a play, describing actors performing their parts.

> Customer: puts in the ATM card and PIN.
> System: deducts the amount from the account balance.

Note that the information content the same in both styles.

Guideline 4: It shows the process moving distinctly forward

The amount of progress made in one step is related to how high or low the use case goal is. In a summary or white use case, the step probably moves forward an entire user goal. In a subfunction use case, it moves forward a much smaller amount. If we see the step "User hits the tab key", either

we are looking at a deep indigo (or black) use case, or the writer has simply chosen too small of an action to describe.

The mistake of choosing very small steps shows up in the length of the use case. If a use case has 13 or 17 steps, it is quite likely that the sentences are not moving the goal forward very much. The use case is easier to read and clearer and contains the same essential information when we merge those small steps to get one that moves the process distinctly forward. I rarely encounter a well-written use case with more than 9 steps in the main success scenario.

To find the slightly higher-level goal for a step, ask, "Why is the actor doing that?" (as described in “Merge steps, keep asking "why"” on page 76). The answer to that question is probably the goal you need for the step, although you may have to ask the question several times to get the answer you want. Here is an example of raising the goal level by asking:

> User hits tab key.
>
> Why is the user hitting the tab key? To get to the address field.
>
> Why is she trying to get to the address field? Because she has to enter her name and address before the system will do anything.
>
> Oh! She wants to get the system to do something (probably the use case itself), and to get it to do anything, she has to enter her name and address.
>
> So the action sentence that moves the process distinctly forward is
>
> User enters name and address.

Guideline 5: It shows the actor’s intent, not movements.

Describing the user’s movements in operating the system’s user interface is one of the more common and severe mistakes, and is related to writing goals at too low a level. I call these descriptions *dialog descriptions*. Dialog descriptions makes the requirements document worse in three ways: longer, brittle, and overconstrained.

* Longer documents are harder to read, and more expensive to maintain.
* The dialog being described is probably not really a requirement, it is probably just how the writer imagines the user interface design at that moment.
* The dialog is brittle, in the sense that small changes in the design of the system will invalidate the writing.

It is user interface designer’s job to invent a user interface that is effective and permits the user to achieve the intent that the use case describes. The description of particular movements belongs in that design task, not in the functional requirements document.

In the requirements document, we are interested in is the *semantic* description of the interface, one that announces the intent of the user is, and gives just a summary of the information that is passed from one actor to another. Larry Constantine and Lucy Lockwood devote a portion of their

book, Software for Use, to just this topic. They use the term *essential use cases* to designate the use cases they are interested in: sea-level system use cases written with semantics interface descriptions.

Typically, all the data that passes in one direction gets collected into just one action step. Here is a common piece of faulty writing fixed:

Before:

1. System asks for name.
2. User enters name.
3. System prompts for address.
4. User enters address.
5. User clicks 'OK'.
6. System presents user's profile.

After:

1. User enters name and address.
2. System presents user's profile.

If there are more three data items being passed, you may prefer to put the different items on separate lines in a tabbed list. Either of the following will work well. The first works better when you are first drafting the use cases, since it is faster to write and read. The second works better you want enhanced accuracy for traceability or testing.

Acceptable variant 1:

3. Customer enters name, address, phone number, secret information, emergency contact phone number.

Acceptable variant 2:

3. Customer enters
   - name
   - address
   - phone number
   - secret information
   - emergency contact phone number

Guideline 6: It contain a 'reasonable' set of actions.

Ivar Jacobson has described a step in a use case as representing a *transaction*. With this phrasing, he captures four pieces of a compound interaction (see Figure 16.):

11 The primary actor sends request and data to the system.

12 The system validates the request and the data.

13 The system alters its internal state.

14 The system replies to the actor with the result.

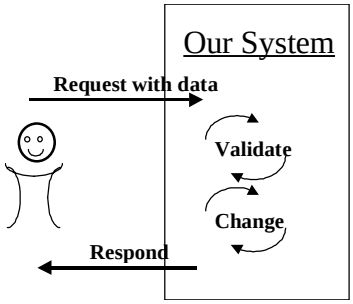


Figure 16. A transaction has four parts.

You can write each piece as a separate action step, or combine them in various ways, up to putting all four into a single action step. Which is best depends on how complicated each separate piece is and where the natural breaks occur in the processing.

As an example, here are five variations for you to consider. None is "wrong", although I consider version 1 too complicated to read easily. I like version 2 when the pieces are simple. I find them a bit too long to work in this instance. Version 3 is personal preference for this example. Version 4 is also good. I find the action steps in version 5 a bit too small, making the scenario too long and unwieldy for my taste. Version 5 does have the advantage, though, of having steps that are separately testable units, possibly suited for a more formal development situation.

Version 1.

1. The customer enters the order number. The system detects that it matches the winning number of the month, registers the user and order number as this month's winner, sends an email to the sales manager, congratulates the customer and gives them instructions on how to collect the prize.

Version 2.

1. The customer enters the order number.
2. The system detects that it matches the winning number of the month, registers the user and order number as this month's winner, sends an email to the sales manager, congratulates the customer and gives them instructions on how to collect the prize.

Version 3.

1. The customer enters the order number.
2. The system detects that it matches the winning number of the month.
3. The system registers the user and order number as this month's winner, sends an email to the sales manager, congratulates the customer and gives them instructions on how to collect the prize.

Version 4.

1. The customer enters the order number.
2. The system detects that it matches the winning number of the month.
3. The system registers the user and order number as this month's winner, and sends an email to the sales manager.
4. The system congratulates the customer and gives them instructions on how to collect the prize.

Version 5.

1. The customer enters the order number.
2. The system detects that it matches the winning number of the month.
3. The system registers the user and order number as this month's winner.
4. The system sends an email to the sales manager.
5. The system congratulates the customer and gives them instructions on how to collect the prize.

Guideline 7: It doesn't "check whether", it "validates"

One of the three kinds of action steps is that the system verifies that some business rule is satisfied. Often people write that the system *checks* the condition. This is not a good action verb. It does not move the process distinctly forward, it is not really the goal, and it leaves open what the result of the check is. You immediately have to write "If the check passes..." and "If the check fails".

Let's use the Ask Why technique to find a better phrase. Why is the system *checking* the condition? Answer: It is *establishing* or *validating* or *ensuring* something. Those are good goal-achieving action verbs to use. So replace "The system checks whether the password is correct" with

> The system verifies that the password is correct.

Let the presence of the word *if* trigger you to recall this guideline. Any time you see "if the (condition)... then...", look at the sentence before it. It is likely to say *checks*. Replace the first sentence with *validates*, and make the second sentence a simple action with no *if*. Here is an example.

Before:

2. The system checks whether the password is correct
3. If it is, the system presents the available actions for the user.

After:

2. The system validate that the password is correct
3. The system presents the available actions for the user.

Notice that the writing in the second case describes the scenario succeeding. It also triggers the reader to ask at step 2, "But what if the password is not valid?" The reader will turn to the extensions section and look for the extension starting with "Password is not valid". It gives the use case a consistent rhythm that makes the use case easy to read and review.

Guideline 8: It optionally mentions the timing

Most steps follow directly from the previous. Occasionally, you will need to say something like

> At any time between steps 3 and 5, the user will ...

or

> As soon as the user has ..., the system will ...

Feel free to put in the timing, but only when you need to. Usually the timing is obvious and need not be mentioned.

Guideline 9: Idiom: "User has System A kick System B"

Here is a situation you might encounter. You want the system under design to fetch information from B, or otherwise run an interaction with system B. It should only do so when the primary actor indicates that it is time to do so. We cannot write, "User hits FETCH button, at which time the system fetches the data from system B." That would have us describing the user interface details.

We can use two steps:

4. User signals to the system to fetch data from system B.
5. The system fetches the background data from system B."

While acceptable, that is awkward and redundant. Better is to write:

4. User has the system fetch the z data from system B.

With this small shift in writing, we indicate that the user controls the timing, that the ball passes from the user to the SuD to system B, and show the responsibilities of all three systems. The details of how the user initiates the action is unspecified, as it should be.

Guideline 10: Idiom: "Do steps x-y until condition"

On occasion, we want to mark that some steps can be repeated. Here again, we are lucky that we are writing in plain prose, as opposed to using a programming formalism. Just write that the step or steps will be repeated!

If there is only one step being repeated, you can put the repetition right into the step:

> "The user selects one or more products."
> "The user searches through various product catalogs until he finds the one he wants to use."

If several steps are being repeated, you can write the repetition before or after the repeating steps. I write the repetition after the steps, to make the scenario a little easier to read. However, either way will work.

1. Customer supplies either account identifier or name and address.
2. System brings up the customer's preference information.
3. User selects a item to buy, marks it for purchase.
4. System adds the item to the customer's "shopping cart".

Customer repeats steps 3-4 until indicating that he/she is done.
5. Customer *purchases the items in the shopping cart* (see use case xxx).

Notice that we need not number the statement about repetition, and need not have a statement that opens the repetition. Both of those clutter up the writing, making the scenario harder to read, not easier.

A variant of "Do steps x-y until condition" is "Steps x-y can happen in any order." I find it works well to put this before the steps affected.

1. Customer logs on
2. System presents available products and services.

Steps 2-4 can happen in any order.
2. User *selects products to buy.*
3. User *specifies preferred form of payment.*
4. User *gives destination address.*

5. User indicates shopping spree is complete.
6. System initiates order, with selected products to be charged against the form of payment and to be sent to the destination address.

To number or not to number

Step numbers clarify the steps and give places to refer to in the extensions section. They are, however, harder to maintain. Without good tools to automatically renumber when we insert or delete steps, it becomes tedious renumbering steps *yet again* for the same use case. You can write good use cases with or without numbering, so it comes down to preference, to be settled on a project-wide basis.

The teams I have visited who have tried both consistently selected numbered sentences over paragraphs. So numbered sentences have become my standard way of writing. Other people are simply happy writing in paragraph form and have no interest in switching.

I include in this book templates for both casual and fully dressed use cases, to emphasize the equivalence and personal nature of this choice.

Paragraphs are the default suggestion in both the casual template and the Rational Unified Process template (used in Use Case 32:“Manage Reports” on page 146). You can still number the steps if you wish, as that one does. I almost always use numbers, even if everything else about the use case is casual. I just find it easier to examine the behavior when the steps are numbered.

The fully dressed form requires the numbers.

Action Step Exercises

Exercise 26* Write a scenario for one of your use cases two ways: once using dialog description, and again using semantic description. Discuss the differences between the two.

Exercise 27* Write the main success scenario for the task-level use case "Withdraw money using FASTCASH option".

Exercise 28 Write the main success scenario for one strategic use case and one task use case for the PAF system.

Exercise 29 Fix faulty 'Login' Your colleague sends you the following. Using the reminders you have learned so far, send back a critique and corrections.

Use Case: LOGIN
This use case describes the process by which uses log into the order-processing system. It also sets up access permissions for various categories of users.
Flow of Events
Basic Path
1. The use case starts when the user starts the application.
2. The system will display the Login screen.
3. The user enters a username and password.
4. The system will verify the information.
5. The system will set access permissions.
6. The system will display the Main screen.
7. The user will select a function.
8. While the user does not select Exit loop
9. If the user selects Place Order then Use Place Order.
10. If the user selects Return Product then Use Return Product.
11. If the user selects Cancel Order then Use Cancel Order.
12. If the user selects Get Status on Order then Use Get Status.
13. If the user selects Send Catalog then Use Send Catalog.
14. If the user selects Register Complaint then Use Register Complaint.
15. If the user selects Run Sales Report then Use Run Sales Report.
end if
16. The user will select a function.
end loop
17. The use case ends.

8. EXTENSIONS

A scenario is a sequence of steps. We know that a use case should contain all of the scenarios, both success and failure. We know how to write the main success scenario. Now we need a way to add all the other scenarios

We could write every scenario individually. That would fit the striped trousers metaphor, as illustrated in Figure 3. on page 38. This approach gets advocated from time to time, and some tools force the writer to work this way. However, as described in the section on the striped trousers metaphor, it is a maintenance nightmare. Each change to a scenario has to be copied to all the other scenarios that contain the same text.

A second alternative is to write *if* statements throughout the text: "If the password is good, the system does ..., otherwise the system does.... ". This is perfectly legal, and some people write use cases this way. However, readers have a hard time with the *if* conditions, especially once there is an *if* inside an *if*. They lose track of the behavior after just two *if* branches, and most use cases contain many branching points.

A third way to arrange the text is to write the main success scenario as a simple sequence running from trigger to completion, and then write a scenario *extension* for each branch point. This seems to be the best choice of the three.

Extensions work this way: Below the main success scenario, for every place where the behavior can branch due to a particular condition, write down the condition, and then write the steps that handles it. Many extensions end by simply merging back in with the main success scenario.

The extension really is a stripped down use case. It starts with a condition, the condition that causes it to be relevant. It contains a sequence of action steps describing what happens under those conditions. It ends with delivery or abandonment of whatever goal the extension is about. There might be extensions to the extensions to handle the *if*s and *but*s that are encountered along the way.

These extensions are where the most interesting system requirements reside. The main success scenario often is known to the people on the team. Failure handling often uses business rules that the developers do not know. The requirements writers frequently have to do some research to determine the correct system response. Quite often, that investigation introduces a new actor, new use case, or new extension condition.

Here is a fairly typical discussion to illustrate.

"Suppose there is a sudden network failure. What do we do?"

"System logs it."

"Got it. Hey, if the network goes down, mmm, what's supposed to happen when it comes up again?"

"Oh. I guess we have to add a new use case for *System restarts after network failure*. The system will come back up, get the log, and either complete or abort the transaction then."

"Yes, but what if the log is corrupt? Then what's the system supposed to do?"

"Gee, I don't know. Let's write it as an Open Issue and continue."

In this conversation, the people discovered both a new use case and the need to research a business policy. Don't be fooled into thinking that these are unnecessary extensions to discuss. Some programmer on the project will run into the same conditions - while programming. That is an expensive time to discover new business rules that need researching. During requirements gathering is a good time to discover them.

I recommend working in three phases:

15 Brainstorm and include every possibility you and your colleagues can conjure up.

16 Evaluate, eliminate, merge, the ideas according to the guidelines below.

17 Then sit down and work your way through how the system should handle each of these conditions.

Here are the pieces and the steps.

8.1 The Extension Conditions

Extension conditions are the conditions under which the system takes a different behavior. We say *extension* as opposed to *failure* or *exception* so that we can include alternative success as well as failure conditions.

Samples

Here are two snippets to illustrate success and failure paths in extensions.

Example 1.

...
4. User has the system save the work so far.
...
Extensions;
4a. System autodetects the need for an intermediate save:
    4a1. ...
4b. Save fails:
    4b1. ...

The way to read the above is, "Instead of the user saving the work so far, the system might have autodetected the need for an intermediate save. In this case, ... During the save (coming from the user's request or from the autosave), the save may fail. In this case, ..."

Example 2.

...
3. The system goes through the document, checking every word against its spelling dictionary.
4. The system detects a spelling mistake, highlights the word and presents a choice of alternatives to the user.
5. The user selects one of the choices for replacement. The system replaces the highlighted word with the user's replacement choice.
...
Extensions;
4a. The system detects no more misspellings through the end of the document:
    4a1. The system notifies the user, terminates use case.
5a. User elects to keep the original spelling:
    5a1. The system leaves the word alone and continues.
5b. User types in a new spelling, not on the list:
    5b1. The system revalidates the new spelling, returns to step 3.
...

Brainstorm all conceivable failures and alternative courses.

I have found it important to brainstorm and get good coverage of the extension conditions before sitting down to write the handling of the extension conditions. Brainstorming the extension conditions is tiring, and so is documenting the handling of the extensions. Thinking up just one extension and correctly working out how the system deals with it costs a lot of energy. You are likely to feel drained after working your way through three or four extensions. This means that you will not think up the next set of extension conditions that you should.

If, on the other hand, you brainstorm all the alternative success and failure situations, you will have a list that will act as the scaffolding for your work for the next several hours or days. You can walk away from it at lunchtime or overnight, come back, and take up where you left off.

In the brainstorming phase, brainstorm all the possible ways in which the scenario could fail, or alternate ways it can succeed. Be sure to consider all of these:

* Alternate success path ("Clerk uses a shortcut code").
* The primary actor behaves incorrectly ("Invalid password").
* Inaction by the primary actor ("Time-out waiting for password").
* Every occurrence of the phase "the system validates" implies there will be an extension to handle failure of the validation ("Invalid account number").
* Inappropriate or lack of response from a supporting actor ("Time-out waiting for

response").

* Internal failure within the system under design, which must be detected and handled as part of normal business ("Cash dispenser jams").
* Unexpected and abnormal internal failure, which must be handled and will have an externally visible consequence ("Corrupt transaction log discovered").
* Critical performance failures of the system that you must detect. ("Response not calculated within 5 seconds").

Many people find that it works well to brainstorm the steps from the beginning of the scenario to the end, to ensure the best coverage of the conditions. When you do this, you will be amazed at the number of things you can think of that can go wrong. Exercise 30 gives you your chance to try your hand at brainstorming a set of failures. The answer to that exercise at the back of the book can serve as a check on your thoroughness. John Collaizi and Allen Maxwell named almost three times as many failures as I have in my answer to the exercise. How well can you do?

Guideline 11: The condition says what was detected.

Write down *what the system detects*, not just what *happened*. Don't write "Customer forgets PIN." The system can't detect that they forgot their PIN. Perhaps they walked away, had a heart attack, or are busy quieting a crying baby. What does the system detect in this case? Inaction, which really means it detects that a time limit was exceeded. Write, "Time limit exceeded waiting for PIN to be entered," or "PIN entry time-out."

The condition is often just a phrase describing what was detected. Sometimes a sentence is appropriate. I like to put a colon ('**:**') after the condition to make sure the reader doesn't accidently think it is an action step. This little convention saves many mistakes. Here are samples:

Invalid PIN:
Network is down:
The customer walked away (time-out):
Cash did not eject properly:

If you are using numbered steps, give the step number where the condition would be detected, and put a letter after it (e.g., 4a). There is no sequence associated with the letters, so there is no implication that 4b follows 4a. This allows us to attach as many extension conditions as we like to any one action step.

2a. Insufficient funds:
2b. Network down:

If the condition can occur on several steps and you feel it important t indicate that, simply list the steps where it can occur:

2-5a. User quit suddenly:

If the condition can occur at any time, use an asterisk ("*****") instead of the step number. List the asterisk conditions before the numbered conditions.

*a. Network goes down:
*b. User walked away without notice (time-out):
2a. Insufficient funds:
2b. Network down:

Don't fuss about whether the failure occurs on the step when the user enters some data or step after, when the system validates the data. One could argue that the error happens at either place, but the argument is not worth the time. I usually put it with the validation step if there is one.

When writing in straight paragraphs with no step numbers, you cannot refer to the specific step where the condition occurs. Therefore, make the condition sufficiently descriptive that the reader will know when the condition might occur. Put a blank line or space before each, and put the condition in some emphasis, such as *italics*, so that it stands out. See Use Case 32:“Manage Reports” on page 146 for a sample.

A small, true, sad story.

The opposite of good extension brainstorming happened on an important, large project.

Like many developers, we didn't want to consider what would happen in case the program encountered bad data in the database. We each hoped the other team would take care of it. Can you guess what happened? A week after the first delivery of the first increment, a senior vice-president decided to see how his favorite customer was using the new sales devices. He fired up his brand-new system and inquired about this large customer. The system replied, "*No Data Found*". One way to describe his reaction would be "excited" (but not in the positive sense).

It wasn't very many hours before the entire senior staff was gathered in an emergency meeting to decide what to do about database errors. We found that there was only one bad data cell in the database. The error message should have read, "Some data missing." But more importantly, we had missed that *how the system reacts when detecting bad internal data* is really part of its external requirements.

We redesigned the system to do the best it could with partial data, and to pass along both its best available results and the message that some data was missing.

The lesson we learned was, consider internal errors, such as discovering missing data.

Note: *About your brainstormed list.*

Your list will contain more ideas than you will finally use. That is all right. The point of the exercise is to try to capture all the situations that the system will ever encounter in this use case. You will reduce the list later.

Your list is probably still incomplete. You are likely to think of a new failure condition while writing the extension scenarios, or when adding a new validation step somewhere inside the use case. Don't worry about this. Do the best you can during this brainstorming stage, and add to the list over time.

Rationalize the extensions list.

The purpose of the rationalizing activity is to reduce the extensions list to the shortest possible. The ideal extension conditions list shows *all* the situations the system must handle, and *only* the ones that it must handle. Recall, always, a long requirements document is hard to read, and redundant descriptions hard to maintain. By merging extension conditions, you shorten the writing, and your readers' reading.

After brainstorming, you should have a short and simple main success scenario, and a long list of conditions to consider. Go through the list carefully, weeding out the ones the system need not handle, and merging those that have the same net effect on the system. Use these two criteria:

* The system must be able to detect the condition.
* The system must be obliged to handle detecting the condition.

Try converting undetectable conditions into detectable conditions before deleting them. If the condition you have is "Customer forgot ATM card," then eliminate it, because there is no equivalent condition the system can detect. If the condition is "Customer forgot PIN code," don't eliminate it. Convert it to "Time-out entering PIN code," which the system can detect.

Next, merge equivalent conditions. You might have written these three conditions: Card is scratched; Card reader is malfunctioning; Card is not even an ATM card. From a requirements point of view, the ATM's behavior is the same: Return the card and notify the customer. You would therefore try to merge these conditions. If you can't find a meaningful single phrasing for all of them, just make a list:

> Card unreadable or non-ATM card

Roll up failures

As part of merging conditions that produce the same net effect, merge failures from lower-level use cases that have the same net effect on the higher-level use case. This *rolling up* of lower-level failures is one of the ways we avoid having an explosion of extension conditions at the higher levels of use cases.

Consider, as an example, that you are working on our Personal Advisor / Finance package. You are writing the user-goal use case *Update Investment*. Let's suppose one of the last steps says,

> **Use Case:** *Update Investment*
> ...

7. User has PAF *save the work.*
8. ...

This reference calls the use case *Save Work*. *Save Work* will contain conditions of the following sort:

Use Case: *Save Work*
...
Extensions:
3a. File already exists (user doesn't want to overwrite): ...
3b. Directory not found: ...
4a. Out of disk space: ...
4b. File write-protected: ...
... and so on...

Save Work ends with success or failure, putting the execution back at the end of step 7 of *Update Investment*. On success, execution continues with step 8. But what if the save failed? What should we write for extension 7a? The reader of *Update Investment* doesn't care *why* the save failed. All failures have the same net effect. So in *Update Investment*, write just the one extension, describing what happens when the save fails.

Use Case: *Update Investment*
...
7. User has PAF *save the work.*
8. ...
Extensions:
7a. Save fails:
    7a1. ...whatever should happen next...

The best part about this rolling up of failures is that even at the highest-level use case, failures are reported in vocabulary appropriate for the level. Even busy executives can take the time to read them, because the failure reporting in a very high level use case is at a similarly high level.

Extension Condition Exercises

Exercise 30 * Brainstorm and list the things that could go wrong during operation of an ATM.

Exercise 31 Brainstorm and list the things that could go wrong with the first user-goal use case for the PAF system (PAF system described in Exercise 15 on page 68).

8.2 Extension Handling

In the simplest case situation, there is just a simple sequence of steps to deal with the condition. In the general case, though, the extension is a stripped-down use case. The trigger is the extension

condition. The goal is either to complete the use case goal, or to recover from whatever failure was just encountered. The body is a set of action steps, and possibly extensions to those action steps. The extension can end with delivery or abandonment of its goal, just as a use case. This similarity is not accidental, and it proves to be very handy in streamlining a complicated extension.

Start writing the handling of a condition with the action step that follows the condition being detected. You needn't repeat that the condition was detected. Continue the story in exactly the same way as when writing the main success scenario. Use all the guidelines about goal level, verb style, and sentences discussed earlier. Keep writing until you reach a place where the main scenario can be rejoined or the use case fails.

Typically, the scenario fragment ends in one of just these ways:

- The step that branched to the extension has been fixed and replaced. At the end of the extension handling, the situation is as though the step had succeeded.

  ...
  3. User activates web site URL.
  4. ...
  Extensions:
  3a. No URL available:
      3a1. User *searches for web site.*
  3b. ...

- The system gives the actor another chance. At the end of the extension handling, the story is back at the beginning of the same step. Notice that the system will revalidate the password in the following example.

  ...
  3. User enters password.
  4. System validates password
  5....
  Extensions:
  4a. Invalid password:
      3a1. System notifies user, requests password again.
      3a2. User reenters password.
  4b. ...

- The use case ends, due to total failure.

  ...
  3. User enters password.
  4. System validates password
  5....
  Extensions:
  ...
  4c. Invalid password entered too many times:

3a1. System notifies user, terminates user session.
5a. ...

- The behavior follows a completely different path to success.

...
3. User does ...
4. User does...
5....
Extensions:
3a. User runs personal macro to complete processing:
3a1. Use case ends.

.In the first two cases, it is not necessary to say what happens next in the extension, because it is obvious to the reader that the step will restart or continue. In the last two cases, it is generally not necessary to say more than "fail!" or "the use case ends," because the steps show the system setting the stakeholders' interests into place.

Most extensions do not say where the story goes back to after the extension. Usually, it is obvious, and writing "go to step N" after every extension makes the overall text harder to read. On the rare occasion that it is not obvious that the story jumps to some other part of the main success scenario, the final step may say, "go to step N".

Examples of all of these situations can be found in the various writing samples in the book.

Guideline 12: Condition handling is indented.

When using the numbering style as shown in this book, indent the action steps that show how the condition is handled, and start the numbering again at 1, after the letter. The action steps follow all the style guidelines given earlier.

Extensions
2a. Insufficient funds:
2a1. System notifies customer, asks for a new amount.
2a2. Customer enters new amount.

When using the straight prose (unnumbered) style, either indent or start a new paragraph for the action steps. The Rational Unified Process template has a special heading level for the extension condition, with the action steps being the text under that heading.

Failures within failures

Inside the extension handling scenario fragment, you may find yourself facing a new branching condition, probably a failure. If you are using the indentation writing style as shown in this book, simply indent again, and continue with the condition naming and scenario writing as before.

At some point, your indentation and numbering will become so complex that you will decide to break the extension out into another use case entirely. Most of the people who have written to me agree that this happens at about the third level of indentation.

Here is an example, taken from Use Case 22:“Register Loss” on page 83.

> 6a. Clerk decides to exit without completing minimum information:
> 6a1. System warns Clerk it cannot exit and finalize the form without date, name or policy number, or adjuster name provided.
> 6a1a. Clerk chooses to continue entering loss
> 6a1b. Clerk saves as "intermediate" report, and exits.
> 6a1c. Clerk insists on exiting without entering minimum information:
> System discards any intermediate saved versions, and exits.

In this example, notice that the writer did not even put a number on the last line. Faced with numbering it 6a1c1, she decided that the extension was already cluttered enough, and that a short piece of straight text would be more readable.

In general, the cost of creating a new use case is large enough that people delay breaking an extension out into its own use case for as long as possible. The consensus seems to be that the above is as far as it makes sense to indent before doing so.

Creating a new use case from an extension

To break an extension out into a new, sub-use case, simply decide what the primary actor’s goal is, give the use case that name, name it at its new level (probably *subfunction*), open up the template for a new use case, and fill in the details that you pulled out of the calling use case.

Use Case 32:“Manage Reports” on page 146 illustrates just this case. *Manage Reports* once contained a step that said

> User can Save or Print report at any time

It had a set of extensions describing the various alternatives and failure situations. But that list of extensions kept growing: unnamed report, preexisting name (do or don’t overwrite), user cancels save in the middle, and so on. Finally, the writers decided to put *Save Report* in its own use case.

In the original use case, you must still deal with the fact that the new sub-use case might fail, so your writing is likely to show both success and failure conditions.

From both a theoretical and effort point of view, it is a simple matter to move an extension into its own use case or back again. The use case model permits us to consider this a minor decision. It was no trouble to move the *Save Reports* extensions out, and similarly, it would only be a few minutes with the text editor to move them back into *Manage Reports*.

However, the cost of creating a use case is not in the mechanical effort needed for the typing. The new use case must be labeled, tracked, scheduled, tested and maintained. These are expensive operations for the project team.

Keeping an extension inside the use case generally makes better sense economically. Two situations will drive you to create a new use case for the extension:

- The extension is used in several places. Putting the extension into its own use case means that it can be tracked and maintained in one place. Ideally, this is the only reason you ever create a use case below sea level.
- The extension makes the use case really hard to read. I find the limit of readability at around 2 pages of use case text, and three levels of indentation. (My use cases are shorter than most people's, so your page length may be greater.)

Extension Handling Exercises

Exercise 32 Write the "Withdraw Cash" use case, containing failure handling, using "if" statements. Write it again, this time using scenario extensions. Compare the two.

Exercise 33 Find a requirements file written in a different form than with use cases. How does it capture the failure conditions? What do you prefer about each way of working, and can you capitalize on those observations?

Exercise 34 * PAF. Write the full use case using the PAF system, filling out the failure repair steps (PAF system described in Exercise 15 on page 68).

9. TECHNOLOGY & DATA VARIATIONS

Extensions serve to express that *what* the system does is different, but occasionally, you want to express that "there are several different ways this can be done". *What* is happening is the same, but *how* it is done might vary. Almost always, this is because there are some technology variations you need to capture, or some differences in the data captured. Write these into the Technology and Data Variations List section of the use case, not in the Extensions section.

Example 1.

Your system must credit a customer for returned goods. You write the action step,

7. Repay customer for returned goods.

They can be paid by check, by electronic funds transfer, or by credit against the next purchase. So you add:

Technology & Data Variations List:
7a. Repay by check, EFTS, or credit against future purchases.

Example 2.

You are specifying a new ATM. Technology has advanced to the point that customers can be identified by bank card, eye scan or fingerprints. You write:

Main Success Scenario:
...
2. User identifies him/herself, bank, and account #.
...
Technology & Data Variations List:
2a. Use magnetic bank card, eye scan, or fingerprints."

These points of variation are not extensions to *this* use case. Each would unfold into its own extension, at some lower level of use case, which you might never write. Each of these variations has a noticeable impact on your cost and work plan, so you need to capture and track them. You mark the possibilities using the Technology and Data Variations List.

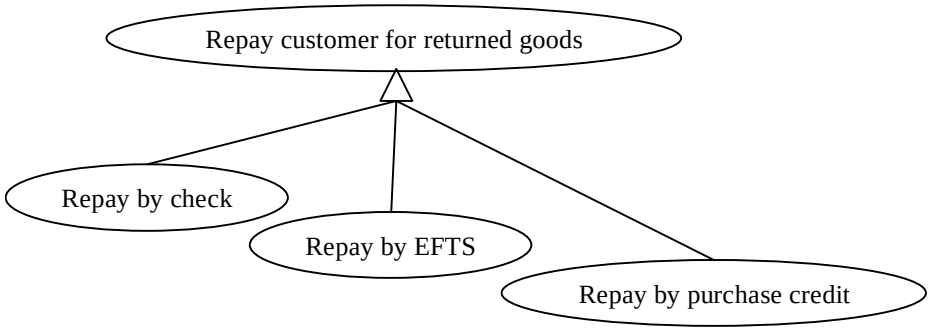
The technology or deferred variations list is used quite rarely, and contains only a list, not action steps. If you find yourself putting conditions and action steps there, then you are using the section incorrectly.

The Technology and Data Variations section is used in Use Case 13:“Serialize access to a resource” on page 55.

Note: *For those determined to use UML use case diagrams*

Create an empty generic base use case for the basic step, and create a specializing use case for each variation. The empty generic base use case names the *what* but not any *how.* Each specialized use case gets to define its own steps explaining *how* it is done. The UML notation is explained in Appendix A.

Figure 17. Technology variations using specialization. .



10. Linking Use Cases

10.1 Sub use cases

An action step can be a simple step or the name of another use case. The step,

> User saves the report

with no emphasis or annotation indicates the step is a simple, atomic step. Writing either of

> User saves the report
> User saves the report (UC 35 Save a Report)

indicates there is a use case called *Save a Report*. This is so natural that it scarcely needs more explanation. Even casual use case readers understand the idea, once they are told that underlined or *italicized* action indicates that there is another use case being called[1], which expands the action mentioned. It is extremely pleasant to be able to write use cases this way, rolling up or unrolling the action as needed.

That is the easiest way of connecting use cases. It takes little time to learn and little space to describe.

10.2 Extension use cases

On occasion, you need a different linkage between use cases, one closely modeled after the *extension* mechanism. Consider this example.

> You are designing a new word processing program, call it *Wapp*. The user's main activity is typing. However, the user might suddenly decide to change the zoom factor or the font size, run the spell checker, or do any of a dozen different of things, not directly connected to typing. In fact, you want the typing activity to remain ignorant of what else might happen to the document.
>
> Even more importantly, you want different software development teams to come up with new ideas for services, and not force them all to update the base use case for each new service. You want to be able to extend the requirements document without trauma.

The characteristics of such a situation are

* There is a main activity, which can be interrupted.
* It can be interrupted in a number of ways, without the main activity being in control

1.------In UML vocabulary, "another use case is *included*." I find that beginning writers and casual readers are much happier saying that one use case *refers to* or *calls* another. Which term you use is up to you.

of the interruptions.

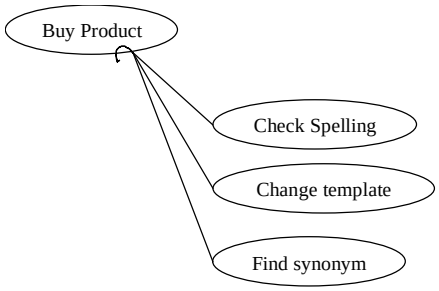
This is different from having a main menu that lists the systems services for the user to select. The main menu is in control of the user's choices. In the word processor example above, the main activity is not in control. It is simply interrupted by another activity.

In these sorts of instances, you do not want the base use case to explicitly name all the interrupting use cases. Doing so produces a maintenance headache, since every person or team adding a new interrupting use case has to edit the base use case. Every time it gets edited, it might get corrupted, it needs to be versioned, reviewed, etc.

In this situation, use the same mechanism as described for scenario extensions, but create a new use case. The new use case is called an *extension use case*, and is really identical a scenario extension, except that it occupies its own use case. Just as a scenario extension, it starts with a condition, referencing a situation in the base use case where the condition might occur. Put all that into the Trigger section of the template.

Here are some extracts for the word processor *Wapp*, to illustrate. Figure 18. shows the *Wapp* situation in UML diagram form. I carefully use a special style of connector, a *hook*, to show a use case extending (hooking into) another. See Appendix A for details.

Figure 18. UML diagram of extension use cases.



Use Case: *Edit a document*
Primary actor: user
Scope: Wapp
Level: user goal
Trigger: User opens the application.
Precondition: none
Main success scenario:
1. User *opens a document to edit.*
2. User *enters and modifies text.*
...
... User *saves document* and exits application.

Use Case: *Check spelling*
Primary actor: user
Scope: Wapp
Level: **subfunction!**
Precondition A document is open
Trigger: Anytime in *edit a Document* that the document is open and the user selects to run the spell checker

Main success scenario:
...etc....

Use Case: *Find synonym*
Primary actor: user
Scope: Wapp
Level: **subfunction!**
Precondition A document is open
Trigger: Anytime in *edit a Document* that the cursor is in a word and the user selects to run the thesaurus.
Main success scenario:
...etc....

Use Case: *Change document template*
Primary actor: user
Scope: Wapp
Level: **subfunction!**
Precondition A document is open
Trigger: Anytime in *edit a Document* that the document is open and the user selects this function.
Main success scenario:
...etc....

When to use extension use cases

Create extension use cases only when you need to. They are harder for people understand and maintain. Two situations call for extension use cases.

The most common use is when there are many asynchronous or interrupting services the user might use, which should not disturb the base use case. Often, they will be developed by different teams. These situations show up with shrink-wrapped software such as word processors, as illustrated above.

The second situation is when you are writing additions to a locked requirements document. Susan Lilly writes,

> "You're working on a project with an iterative process and multiple drops. You have baselined requirements for a drop. In a subsequent drop, you *extend* a baselined use case with new or additional functionality. You do *not* touch the baselined use case."

If the base use case is not locked, then the extension is fragile: changing the base use case can mess up the condition mentioned in the extending use case. Be wary about using extension use cases is such situations. Exercise 36, below, gives you a chance to experiment with this sort of extension use case.

Alan Williams offers a useful principle to help with the decision of whether to have a use case call to a sub use case or have the second use case extend the base one:

> If the trigger involves things for which the base use case is responsible, that is, the base use case knows when/where/why the second use case should be followed, then the base use case *include / calls / references* the other.
>
> If the trigger involves things for the second use case is responsible, that is, the second use case knows when/where/why it should be followed) then the second use case *extends* the base use case.

Note that when the base use case *calls* a second use case, the base use case names the other, saying when and where it executes. The referenced use case does not contain the name of the base use case. When the second use case *extends* a base use case, the base use case does not name, or indeed, know anything about the extending (interrupting) use case. The extending use case names the use case it interrupts, and under what circumstances it executes.

Use Case Linkage Exercises

Exercise 35 Find a condition in a user-goal use case for the PAF system that requires a sub-use case to be broken out (PAF system described in Exercise 15 on page 68). Write that sub-use case, and sew it, both success and failure, back into your user-goal use case.

Exercise 36 Consider the ATM situation in which you are not at your home bank, but are a guest at another bank. Write the sub-use case for a bank-to-bank withdrawal request, and connect it into your previous use case(s) about withdrawing money. Do this in two ways: as a sub use case that your base use case references, and as an extension use case. Discuss with a colleague which you prefer, and why.

11. Use Case Formats

11.1 Formats to choose from

Fully dressed form

Most of the examples in this book are in my preferred style, which is the fully dressed form:

* one column of text (not a table),
* numbered steps,
* no *if* statements,
* the numbering convention in the extensions sections involving combinations of digits and letters (e.g., 2a, 2a1, 2a2 and so on).

The alternate forms that compete best with this are the casual form, the 2-column style and the Rational Unified Process template (all described below). However, when I show a team the same use case in multiple styles, including those two, they almost always select the one-column, numbered-step version. So I continue to use and recommend it. Here is the basic template, which project teams around the world have put into Lotus Notes, DOORS, Word, Access, and various other text tools.

Use Case 24: Fully Dressed Use Case Template <name>

<the name should be the goal as a short active verb phrase>

Context of use: <a longer statement of the goal, if needed, its normal occurrence conditions>

Scope: <design scope, what system is being considered black-box under design>

Level: <one of: Summary, User-goal, Subfunction>

Primary Actor: <a role name for the primary actor, or description>

Stakeholders & Interests: <list of stakeholders and key interests in the use case>

Precondition: <what we expect is already the state of the world>

Minimal Guarantees: <how the interests are protected under all exits>

Success Guarantees: <the state of the world if goal succeeds>

Trigger: <what starts the use case, may be time event>

Main Success Scenario:

<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>

<step #> <action description>

Extensions

<put here there extensions, one at a time, each referring to the step of the main scenario>

<step altered> <condition>: <action or sub-use case>

<step altered> <condition>: <action or sub-use case>

Technology and Data Variations List

<put here the variations that will cause eventual bifurcation in the scenario>

<step or variation # > <list of variations>

<step or variation # > <list of variations>

Related Information

<whatever your project needs for additional information>

Casual form

Contrast the fully dressed form with the following, an example of a casual use case taken from OO Design with Applications, 3rd edition (by Booch, Martin, Newkirk). I have added the primary actor, scope and level to make a fully formed example. Notice that the extensions are still present in the second paragraph.

USE CASE 25: ACTUALLY LOGIN (CASUAL VERSION)

Primary actor: User Scope: ApplicationLevel: Subfunction

Upon presenting themselves, the user is asked to enter a username and password. The system verifies that a submitter exists for that username, and that the password corresponds to that submitter. The user is then given access to all the other submitter commands.

If the username corresponds to a submitter which is marked as an administrator then the user is given access to all the submitter and administrator commands. If the username does not exist, or if the password does not match the username, then the user is rejected.

One-column table

Some people like to put the steps of the scenarios into a table. Over the years, I have found that the lines on the table obscure the actual writing. However, that is merely personal preference, and others often choose the table style. Here is the template

USE CASE #	< the name is the goal as a short active verb phrase>	
Context of use	<a longer statement of the context of use if needed>	
Scope	<what system is being considered black box under design>	
Level	<one of : Summary, Primary Task, Subfunction>	
Primary actor	<a role name for the primary actor, or description>	
Stakeholder & Interests	Stakeholder	Interest
	<stakeholder name>	<put here the interest of the stakeholder>
	<stakeholder name>	<put here the interest of the stakeholder>
Preconditions	<what we expect is already the state of the world>	
Minimal Guarantees	<the interests as protected on any exit>	
Success Guarantees	<the interests as satisfied on successful ending>	
Trigger	<the action upon the system that starts the use case>	
DESCRIPTION	**Step**	Action
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>

	2	<...>
	3	
EXTENSIONS	Step	Branching Action
	1a	<condition causing branching> : <action or name of sub-use case>
TECHNOLOGY & DATA VARIA-TIONS		
	1	<list of variation s>

Two-column table

Rebecca Wirfs-Brock invented the idea of a *conversation*, whose distinguishing visual characteristic is the use of two columns. The primary actor's actions are in the left-hand column, the system's actions are in the right-hand column. Conversations are most often written in preparation to designing the user interface, so they may pay contain more detail on the user's movements.

You can write a use case using the two-column table form. The result is clear, but often quite long, even exceeding three pages (see, for example Use Case 36:"Research a solution - Before" on page 190). Usually, by the time we revise the text to fit into 3-9 steps at appropriate goal levels, the writing is so simple and clear that people no longer find the need for the two columns.

Constantine and Lockwood adopt the format of the conversation in their *essential use cases*, as described in their book, Software for Use. The difference is that in an *essential use case*, all of the user movements (*dialog description*) are left out of the writing, so the result is very short, exactly as described in this book.

The one difficulty with using the two-column format to capture behavioral requirements (once you deal with the length) is that there is no place to write about the supporting actors. One could add a third column for them, but I have never heard it suggested, nor seen it done. I think this is because *conversations* and *essential use cases* are aimed at capture user interface requirements rather overall system behavioral requirements.

All of the above taken into account, many people do find the two-column form attractive while they are learning about use cases and want to make the actions clear, or when they are analyzing and partitioning the requirements use cases. Experiment with it, if you like, and read what Wirfs-Brock and Constantine are doing with it. Here is a scenario fragment in two-column style:

Customer	System
Enters order number.	
	Detects that the order number matches the winning number of the month.
	Registers the user and order number as this month's winner.
	Sends an email to the sales manager.
	Congratulates the customer and gives them instructions on how to collect the prize.
Exits the system.	

RUP style

The Rational Unified Process uses a template fairly similar to the fully dressed template. Numbering steps is optional. Extensions are given their own heading sections and called *alternate flows*. Everything I in this book works nicely with this template, which, although a bit cluttered with heading numbers, is attractive and easy to follow. Here is the basic template.

1. Use Case Name

1.1 Brief Description

...text...

1.2 Actors

...text...

1.3 Triggers

...text...

2. Flow of Events

2.1 Basic Flow

...text...

2.2 Alternative Flows

2.2.1 Condition 1

...text...

2.2.2 Condition 2

...text...

2.2.3 ...

3. Special Requirements

3.1 Platform

...text...

3.2 ...

4. Pre-Conditions

...text...

5. Post-Conditions

...text...

6. Extension Points

...text...

Rational Software Corporation sent the following as a sample. Normally, it would be accompanied in the tool set by a use case diagram and other work artifacts. I find the use case quite self-explanatory, and think you will, too. Note that both simple paragraphs and numbered steps are used, as the writer felt best suited the presentation. I added the two graphical icons to the title just to be consistent with the examples in this book, but did not add any fields to their template.

Use Case 32:“Manage Reports” on page 146 also uses the RUP template.

USE CASE 26: REGISTER FOR COURSES

1. Use Case Name: Register for Courses

1.1 Brief Description

This use case allows a Student to register for course offerings in the current semester. The Student can also modify or delete course selections if changes are made within the add/drop period at the beginning of the semester. The Course Catalog System provides a list of all the course offerings for the current semester.

The main actor of this use case is the Student. The Course Catalog System is an actor within the use case.

2. Flow of Events

The use case begins when the Student selects the "maintain schedule" activity from the Main Form. [Refer to user-interface prototype for screen layout and fields]

2.1 Basic Flow

2.1.1 Create a Schedule

2.1.1.1 The Student selects "create schedule."

2.1.1.2 The system displays a blank schedule form. *[Refer to user-interface prototype for screen layout and to the domain model for required fields]*

2.1.1.3 The system retrieves a list of available course offerings from the Course Catalog System. *[How is this selected and displayed? Text? Drop-down lists?]*

2.1.1.4 The Student selects 4 primary course offerings and 2 alternate course offerings from the list of available offerings. Once the selections are complete the Student selects "submit." *[Define "primary course offerings" and "alternative course offerings" in project glossary. Must exactly 4 and 2 selections be made? Or "up to 4…", etc.]*

2.1.1.5 The "Add Course Offering" sub-flow is performed at this step for each selected course offering.

2.1.1.6 The system saves the schedule. *[When is the master schedule updated? Immediately? Nightly (batch)?]*

2.2 Alternative Flows

2.2.1 Modify a Schedule

2.2.1.1 The Student selects "modify schedule."

2.2.1.2 The system retrieves and displays the Student's current schedule (e.g., the schedule for the current semester). *[Is this only available for the current semester?]*

2.2.1.3 The system retrieves a list of all the course offerings available for the current semester from the Course Catalog System. The system displays the list to the Student.

2.2.1.4 The Student can then modify the course selections by deleting and adding new courses. The Student selects the courses to add from the list of available courses. The Student also selects any course offerings to delete from the existing schedule. Once the edits are complete the Student selects "submit".

2.2.1.5 The "Add Course Offering" sub-flow is performed at this step for each selected course offering.

2.2.1.6 The system saves the schedule.

2.2.2 Delete a Schedule

2.2.2.1 The Student selects the "delete schedule" activity.

2.2.2.2 The system retrieves and displays the Student current schedule.

2.2.2.3 The Student selects "delete."

2.2.2.4 The system prompts the Student to verify the deletion.

2.2.2.5 The Student verifies the deletion.

2.2.2.6 The system deletes the schedule. *[At what point are the student slots freed up?]*

2.2.3 Save a Schedule

At any point, the Student may choose to save a schedule without submitting it by selecting "save". The current schedule is saved, but the student is not added to any of the selected course offerings. The course offerings are marked as "selected" in the schedule.

2.2.4 Add Course Offering

The system verifies that the Student has the necessary prerequisites and that the course offering is open. The system then adds the Student to the selected course offering. The course offering is marked as "enrolled in" in the schedule.

2.2.5 Unfulfilled Prerequisites or Course Full

If in the "Add Course" sub-flow the system determines that the Student has not satisfied the necessary prerequisites or that the selected course offering is full, an error message is displayed. The Student can either select a different course offering or cancel the operation, at which point the use case is restarted.

2.2.6 No Schedule Found

If in the "Modify a Schedule" or "Delete a Schedule" sub-flows the system is unable to retrieve the Student's schedule, an error message is displayed. The

Student acknowledges the error and the use case is restarted.

2.2.7 Course Catalog System Unavailable

If the system is unable to communicate with the Course Catalog System after a specified number of tries, the system will display an error message to the Student. The Student acknowledges the error message and the use case terminates.

2.2.8 Course Registration Closed

If, when the student selects "maintain schedule", registration for the current semester has been closed, a message is displayed to the Student and the use case terminates. Students cannot register for courses after registration for the current semester has been closed.

3. Special Requirements

No special requirements have been specified for this use case at this time.

4. Pre-Conditions

4.1 Login

Before this use case begins the Student has logged onto the system.

5. Post-Conditions

There are no post-conditions associated with this use case.

6. Extension Points

There are no extension points associated with this use case.

If-statement style

Programmers inevitably want to write *if* statements in the text. After all, it is easier to write,

> If the order matches the winning number, then <all the winning number business>, otherwise tell the customer that it is not a winning number.

than it is to learn about how to write extensions.

If there were only one *if* statement in the use case, I would agree. Indeed, there is nothing in the use case model that precludes "if ... then ... else". However, once there are even two *if* statements, then the writing becomes much harder to understand. There is almost certainly a second if statement, and a third, and a fourth. There is probably even an "if" statement inside the "if" statement.

When people insist they really want to write with *if* statements, I invite them to do so, and to report back on what they experienced. Every one who has done that has concluded within a short time that the *if* statements made the use case hard to read, and has gone back to the extensions style of writing. Therefore, a strong stylistic suggestion is, "Don't write *if* statements in your scenario".

Exercise 37 Rewrite the following use case, getting rid of the "if" statements, and using goal phrases at the appropriate levels and alternate scenarios or extensions

"Perform clean spark plugs service"

Conditions: plugs are dirty or customer asks for service.

1. open hood.
2. locate spark plugs.
3. cover fender with protective materials.
4. remove plugs.
5. if plugs are cracked or worn out, replace them.
6. clean the plugs.
7. clean gap in each plug.
8. adjust gap as necessary.
9. test the plug.
10. replace the plugs.
11. connect ignition wires to appropriate plugs.

12 check engine performance.

13. if ok, go to step 15.
14. if not ok, take designated steps.
15. clean tools and equipment.
16. clean any grease from car.
17. complete required paper work.

Outcome: engine runs smoothly.

_____

OCCAM style

If you are really determined to construct a formal writing model for use cases, look first to the Occam language, invented by Tony Hoare. Occam lets you annotate the alternate, parallel, and optional sequencing you will need easier than any other language I know. I don't know how OCCAM handles exceptions, which is necessary for the extension-style of writing.

You write:

ALT

alternative 1

alternative 2

TLA(this ends the alternatives)
PAR
parallel action 1
parallel action 2
RAP(this ends the parallel choices)
OPT
optional action
TPO

However, if you do decide to create or use a formal language for use cases, make Use Case 22:“Register Loss” on page 83 your first test case. It has parallel, asynchronous, exceptional, co-processing activities. I think is shows well natural language deals with that in a way still quite easy to understand.

Diagram style

A use case details the interactions and internal actions of actors, interacting to achieve a goal. A number of diagram notations can express these things: sequence charts, collaboration diagrams, activity diagrams, and Petri nets. If you choose to use one of these notations, you can still use most of the ideas in this book to inform your writing and drawing.

The graphical notations suffer from two usability problems. The first is that end users and business executives are not likely to be familiar with the notations, and have little patience to learn. Using graphical notations means you are cutting off valuable readers.

The second problem is that the diagrams do not show all that you need to write. The few CASE tools I have seen that implement use cases through interaction diagrams, force the writer to hide the text of the steps behind a pop-up dialog box attached to the interaction arrows. This make the use case impractical to scan - the reader has to double click on each arrow to see what is hidden behind it. In the "bake offs" I have held, the use case writers and readers uniformly chose no tool support and simple word processing documents over CASE tool support in diagram form.

One particular diagramming style that is not suitable is...

The UML use case diagram

The use case diagram, consisting of ellipses, arrows and stick figures, is *not* a notation for capturing use cases. The ellipses and arrows show the packaging and decomposition of the use cases, not their content.

Recall that a use case names a goal, it consists of scenarios, a scenario consists of action steps, and each action step is phrased as a goal, and so can be unfolded to become its own use case. It is

possible to put the use case goal as an ellipse, to break out every action step as an ellipse, and to draw and arrow from the use case to the action step ellipse, labeling it *includes*. It is possible to continue with this decomposition from the highest to the lowest level use case, producing a monster diagram that shows the entire decomposition of behavior.

However, the ellipse diagram is missing essential information such as which actor is doing each step and notes about the ordering of the steps. It is useful as a table of contents, and should be saved for that purpose. See Reminder 24.“The Great Drawing Hoax” on page 218 and Appendix 23.1“Ellipses and Stick Figures” on page 224.

The point of this section is to prevent you from trying to replace the text of the use cases with ellipses. One student in a lecture asked me,

> "When do you start writing text? At the leaf level of the ellipse decomposition?"

The answer is that the use case lives in the text, and all or any drawings are only an illustration to help the reader locate the text they need to read.

Many people find the topmost use case diagram useful, the one showing the external actors and user-goal use cases. That provides a *context diagram*, similar to other context diagrams that people have been drawing for years. The value of use case diagrams drops rapidly from there. I discuss this more in “Appendix A: Use Cases in UML” .

11.2 Forces affecting Use Case Writing Styles

At the 1998 OOPSLA conference, 12 experienced use case writers and teachers gathered to discuss common points of confusion or difficulty with use cases, and the forces that drive people to write use cases differently. Paul Bramble organized the workshop and put together the following categorization of the items collected. If you feel overwhelmed at all the different situations in which use cases are used, feel comforted by the fact that we were, too!

We are lucky that is a consistent answer to the question: "How does one write readable use cases?" Nonetheless, you may find yourself in a situation with some combination of the issues listed below that obliges you to work differently than you expect. Be patient, be tolerant, and write use cases to suit the purpose that you have at hand.

Countervailing Forces: Business Setting, Social Interaction, Conflicting Cultures

You want to introduce use cases, but run into the following situation / argument (I won't try to fix the argument, but you may enjoy recognizing you are not alone!):

> "We've always done it this other way..."

With multiple cultures:

There is prejudice across teams,

There are different work cultures, and people there simply "do things differently",

The people writing the use cases use a different vocabulary than the people who will read the use cases.

Level of Understanding

Understanding is different at different times and places and among different people. You might choose to shift the recommended writing style due to:

How much you know now ...

... about the domain

.. about use cases in general

Where in life cycle do you know it?

Do you need to establish Content, or Cost;

Do you need the Breadth view now, or the Depth view now

Clandestine Analysis

Creeping Analysis

Watch out, people tend to stress the things they know!

Scheduling vs. depth of knowledge vs. domain knowledge

Stakeholder needs

What is the Viewpoint you are after?

Customer? This a reader, the use case consumer, happy with a high-level description.

Corporate / IT? This is a writer, or an implementer, interested in a detailed description.

Several? Wanting to represent multiple viewpoints, for use Cases across several service groups.

Wanting a Complete Model versus Incomplete Model (See cooperation between teams)

Are there, or what are, the different readers involved?

Experience versus Formality

Experience: every use case team includes people new to use cases, but they soon become "experienced" writers. Experienced people know some short cuts, new people want clear directions and consistency in the instructions.

Formality: perhaps the leader, or perhaps the departmental methodology dictates a formal (or informal!) writing style, despite any experience of lack thereof.

Coverage

Breadth of coverage depends on the team composition, on the skill in writing, on their communication, how badly they need to cover the whole problem vs. the need to communicate information to the readers

Coverage of Problem may vary based on:

- The subject matter experts (they may focus narrowly)
- Number of writers
- Number of readers
- Number of implementers involved
- Business people don't know what they want
- Everyone decides they need to work along common model
- Group may be geographically dispersed

Consistency

Consistency of Content vs. Conflicting Customer Requirements vs. users (owners of requirements) often disagree.

Requirements Volatility

Consistency of Format.

Forces of Complexity

Use Case Complexity

- Achieving Completeness
- People want to describe full problem domain.
- Representing multiple viewpoints raises use case complexity
- Want simplified view of a system.
- Simplicity of expression.
- Detailed expression. Design free is easy to understand
- Narrow versus broad view.
- Problem Complexity
- People like to add technical details to use cases, especially when they have a difficult problem

System Complexity

- Analysis paralysis – complexity of system overwhelms analyst.
- Number of actor profiles
- Number of function points
- Kind of system

Simple user system

Real Time System

Embedded System (Must be error resistant)

Conflict

Resolve customer conflict

ambiguity masks conflict

Completeness

Requirements incomplete for re-engineer.

Don’t have access to users (users are not your customers)

Goals versus Tasks - i.e. what to accomplish versus how to accomplish it

Users often specify requirements rather than usage.

Context versus usage

Activities and tasks describe what is happening in a system, not why it is happening.

Resources

It requires time to write good use cases, but project time is critical.

Need Management buy-in, else management wants code, not use cases.

Other factors

Tool Requirements/support

The objective is sometimes not even known!

Need to partition description for subsequent analysis.

Don’t constrain design vs. level of design to do.

Clean design vs. understandable

Abstract or concrete use cases?

Traceability

Corporate Agility.

Whew! That was quite the list. Even though most of this book applies to all situations, you might reflect on that list to decide whether to use more formality / less formality, or whether to do less now and more later, and similarly, how much to write or how to stage the writing, or how much breadth or how much precision to get before getting some depth.

11.3 Standards for five project types

You are on the project. You and the others have read this book, so you know the ideas. The question at hand, now, is, "What standards are *we* going to adopt?" The answer depends on who you are, what your skills are, what your objective is at this moment. Compare to the list of forces just given. In this section, nominate writing standards for five particular situations. You will notice that the basic choice in each standard is between casual and fully dressed use cases. The five situations are:

1 Eliciting requirements, even if use cases will not be the final form

2 Modeling the business process

3 Drafting / sizing system requirements

4 Writing functional requirements on a short, high-pressure project

5 Writing detailed functional requirements at the start of an increment, on a longer or larger project

You should find it practical to use these standards *as is*. After some consideration, you may decide to tune them to your corporate needs, or needs of the moment, according to the principles given in the book.

In the following, I use the example of a company, MyCo, about to develop a new system, Acura, to replace an old system, BSSO. I do this to remind you not to write the words *corporation* and *system*, but to write their names.

For requirements elicitation

USE CASE 27: ELICITATION TEMPLATE - OBLE A NEW BISCUM

Scope: *Acura* **Level:** *Sea level*
Context: The quibquig needs to oble a new biscum once a dorstyp gets nagled (Text about the goal in operational context.)
Primary actor: A quibquig (or whoever the primary actor is)
Stakeholders & Interests: Qubquig, MyCo, ...whomever & whatever is appropriate.
Preconditions: what must be true before this use case can be triggered.
Triggers: The quibquig selects the obling function (whatever it may be).
Main Success Scenario:
... A paragraph of stuffstuff describing the quibquig successfully obling a biscum within Acura the success scenario ... actors do this, that, and the other.
... Another paragraph of stuffstuff describing conditions and alternate paths in trying to oble the biscum or failing ... actors do this, that, and the other.
Frequency of occurrence: yay many times a day
Open Issues: ...a good thing to fill in at this point...

This template is for when your ultimate purpose is to *discover* the requirements (reread “Steve Adolph: "Discovering" Requirements in new Territory” on page 25). Bear in mind that your requirements may get written in another form than use cases. The game, therefore, is to move quickly through the use cases, drafting them in a lively work session.

The template is the casual template. Keep the stakeholders and interests in the template, to help remind everyone about their requirements, but don't include system guarantees.

Your use cases will generally be black-box, most of them at user-goal level. You may generate higher level use cases for context. You shouldn’t go below sea level very often.

For business process modeling

USE CASE 28: BUSINESS PROCESS TEMPLATE - SYMP A CARSTROMMING

Scope: *MyCo operations* **Level:** *Summary*
Context: Text about the goal in operational context.
Primary actor: whoever the primary actor is.
Stakeholders & Interests: whomever & whatever is appropriate.
Minimal Guarantees: whatever they are.
Success Guarantees: whatever they are.
Preconditions: what must be true before this use case can be triggered.
Triggers: whatever it may be.
Main Success Scenario:
1. ...action steps...
2. ...
Extensions:
1a. ...extension conditions:
    1a1. ...action steps...
Frequency of occurrence: whatever
Open Issues: ...a good thing to fill in at this point...

This template is for redesigning the business or the process to handle new software. The people reading these use cases will be senior line staff, department managers, and senior executives, so keep them easy to read, and reduce emphasis on data details. Number the steps to make the sequencing stand out. Be sure do describe failure handling in the extensions, as they reveal important business rules.

The top-level, outermost use cases will be black-box , showing the company interacting with external partners. These are used either as specifications against which the business process will be measured, or to set context for the white-box use cases. The white-box use cases such as MyCo Operations, show the organization in action, with people and departments working together to deliver the organization's responses. You will use goal levels from cloud to sea level. I selected a kite-level summary goal for the example in the template.

For sizing the requirements

USE CASE 29: SIZING TEMPLATE: BURBLE THE TRAMLING

Scope: Acura **Level:** blue
Context: put here preconditions or conditions of normal usage
Primary actor: whomever
Put here a few sentences describing the actors successfully freeing the necessary fadnap in the main success scenario ...
Put here a few sentences mentioning some of the alternate paths and the handling ...
Frequency of occurrence: how often
Open Issues: ...always a good idea to mark...

This template is for when you are drafting the system requirements to estimate the size and shape of the system. Later, you may detail them further, into fully dressed requirements. You might choose to design directly from the casual use cases if your project fits the profile (see the discussion around “A sample of use case briefs” on page 47 and Reminder 19.“Know the cost of mistakes” on page 215).

The template is casual, as befits early work at medium precision. The system under discussion can be a system or the business. The goals may be at any level, including subfunctions, since project effort depends largely on the complexity of the subfunction use cases. In the example for the template, I use Acura and user goal. See also Use Case 25:“Actually Login (casual version)” on page 121.

For a short, high-pressure project

USE CASE 30: HIGH-PRESSURE TEMPLATE: KREE A RANFATH

Scope: Acura **Level:** User goal
Context:
Primary actor:
Stakeholders & Interests:
Minimal and Success Guarantees:
Preconditions:
Triggers:
Main Success Scenario:
... A paragraph of text describing the actors achieving success in the main success scenario ...
Extensions:
... A paragraph of text mentioning *all* the alternate paths and the handling ...
Frequency of occurrence:
Open Issues: ...

Use this template when you need written requirements, but the project is short and under heavy time pressure. For time and economic reasons, you prefer to avoid the overhead of numbers and full template and therefore use the casual form. Still, capture preconditions, guarantees and extensions. I assume that you will work carefully on improving project internal communications, as described in Reminder 19.“Know the cost of mistakes” on page 215.

For detailed functional requirements

USE CASE 31: USE CASE NAME: NATHORIZE A PERMION

Scope: *Acura*
Level: *User goal*
Context of use:
Primary actor:
Stakeholders & Interests:
Minimal Guarantees:
Success Guarantees:
Preconditions:
Triggers:
Main Success Scenario:
1. ...
2. ...
Extensions:
1a. ...
    1a1. ...
Frequency of occurrence:
Open Issues: ...

Use this template when your purpose is to collect behavioral requirements using all of the features of fully dressed use cases. This could be for a larger or critical cost project, fixed-price bid, a geographically distributed team, at the start of an increment when it is time to expand and examine the sizing use cases drafted earlier, or because it is your culture to do so

The system under design may be anything, the actors and goals, similarly anything. I used Acura and user-goal level in this sample template.

11.4 Conclusion about formats

All of the above different formats for a use case express approximately the same basic information. The recommendations and guidelines of this book apply to each format. Therefore, do not fuss too much about which format you are obliged to use on your project, but select one that the writers and the readers can all be comfortable with.

Chapter 11. Use Case Formats
Conclusion about formats - Page 140

PART 2
FREQUENTLY ASKED QUESTIONS

12. When are we done?

You are "done" when

- You have named all the primary actors and all the user goals with respect to the system.
- You can captured every trigger condition to the system either as a use case trigger or an extension condition.
- You have written all the user-goal use cases, along with the summary and subfunction use cases needed to support them.
- Each use case is clearly enough written that
  - the sponsors agree they will be able to tell whether or not it is actually delivered.
  - the users agree that is what they want or can accept as the system's behavior.
  - the developers agree they can actually develop that functionality.
- The sponsors agree that the use case set covers all they want (for now).

All the primary actors and their user goals defines the boundary of what the system must accomplish. Since there is no other source of this information to compare this list against, just the minds of the people who have to accept the system, you cannot know you are done with this list, you can only suspect you are done with this list. Therefore, it is worthwhile going over this list in brainstorming mode several times.

All the trigger conditions is the fine-tuning of the boundary. The system will have to react to every trigger. In the use cases, some of those triggering events will show up as use case triggers. Examples might be User puts card into slot, Customer calls to add/remove a clause to their insurance policy or User selects to install software upgrade. Other triggers are taken care of in the scenario extensions. Examples are User hits cancel button or System detects power drop.

One way to reexamine the set of triggers to the system is to identify all the elements that have a lifecycle, and then review each of their lifecycles. Look for all the events that cause something to change its state in its lifecycle.

Summary and subfunction use cases. The summary use cases create the context for the user-goal use cases. They answer the question people often ask, "But how do all these (user-goal) use cases fit together?" I like to make sure that every use case sits inside a higher-level one, up to a single root. That root use case is only table of contents with not much storyline, but new readers

find it useful to have a single starting point from which they can start accessing every use case in the system.

Subfunction use cases support the user-goal use cases. They are only needed only are called from several by other use cases or isolate a complicated piece of behavior.

Agreement on the use cases. The use cases are only done when the both the sponsors and usage experts can read and agree with them and say that's all they want, *and* the developers can read them and agree they can build a system to these specifications. That is a difficult challenge. It is the challenge of requirements writing.

On *being done*. Uttering the phrase *Being Done* manages to give an impression that one should sit down and write all the use cases, beginning to end, before starting on design tasks. I hope you are aware that this is not the case. See Chapter 17.1"Use Cases In Project Organization" on page 164 for a discussion of developing a project plan with partial release of use cases. Read Surviving Object-Oriented Projects for a longer description of incremental development. Read the article "VW-Staging" online at http://members.aol.com/acockburn/papers/vwstage.htm for a short and dedicated discussion of incremental and iterative development.

Different project teams use different strategies, depending on their situation. Some draft all the use cases right away, perhaps to prepare a bid for a fixed-price contract. These teams need to be aware that the use cases will need fine tuning over the course of the project. Other teams only draft all the actors and user goals, delaying use case elaboration until the appropriate increment. Others will create the use cases for each 6-9 months worth of work, deferring all other requirements discussion until that work is almost finished. Still others will write use cases just barely before starting on a round of work. Each of these strategies its place and its advocates.

13. Scaling up to Many Use Cases

There are two ways to deal with large numbers of use cases: say less about each one, or group them into separable groups. You should use both techniques.

Say less about each one (low precision representation)

Just the use case name alone is useful. That's why it counts as the first level of precision. The collection of use case names is an excellent working medium for manipulating the full set of use cases, particularly for estimating, planning and tracking. Put the list of use case names into a spreadsheet and use the spreadsheet's capabilities to sort, order and summarize the various qualities of interest about the use case. See Section 1.5"Manage Your Energy" on page 29, and Chapter 17.1"Use Cases In Project Organization" on page 164.

The second level of precision is the use case brief, a 2-3 sentence summary of the use case. This also can be put into a spreadsheet or table form, and reviewed. See "A sample of use case briefs" on page 47. This is also useful for getting an overview of the system and organizing the work.

Saying less about each use case is valuable when you want to scan the full set of use cases at one time. There are times, though, when you need to collect them into separate clusters.

Create clusters of use cases

If you are working from text-based tools such as Lotus Notes or spreadsheets and word processors, you can form *use case clusters* using ordinary labeling techniques. If you are using UML tools, you will call them *packages*. There are three common and effective clustering techniques.

By actor. The most obvious way to cluster use cases is by primary actor. At some point, perhaps around 80 or 100, that loses effectiveness. You will have too many use cases per primary actors, too many primary actors, or there will be too much overlap of primary actors to use cases.

By summary use case. You will find that some sets of use cases naturally cluster by their lifecycle, or on larger projects, by their place in the lifecycle. These related use cases show up in a summary use case. If you do not write summary use cases, you may still want to cluster the use cases to create, update and delete certain kinds of information will naturally cluster. On one project. One system maintained a mock checkbook for customers. We referred to all of the checkbook altering use cases together, as "the checkbook use cases". This cluster was developed by

the same development team, and progressed together in a way that was easy for the project managers to handle.

By development team and release. Clustering use cases by which team will develop the design and release number simplifies tracking the work. It becomes natural to ask questions like, "Is the user profile cluster going to make it on time?" This clustering holds even for larger projects.

By subject area. For projects with over 100 use cases, people will automatically separate them by subject areas in their speaking. It is usually quite easy to name natural subject areas. One project used customer information, promotions, invoicing and advertising. Another used booking, routing, tracking, delivery, billing. Often there are subprojects around the different subject areas. Each subject area might contain 20-100 use cases.

Tracking 240 use cases is difficult. Tracking 15-20 clusters is quite reasonable. On a large project, I would cluster first by *subject area* to get 3-6 clusters of 20-100 use cases each, and then by *release and development team*. Depending on the number of use cases in these clusters, I would summarize work progress using clusters of related or summary use cases.

14. Two Special Use Cases

14.1 CRUD use cases

There is not yet a consensus on how to organize all those little use cases of the sort, *Create a Frizzle*, *Retrieve a Frizzle*, *Update a Frizzle*, *Delete a Frizzle*. These are known as CRUD use cases, from the Create, Retrieve, Update, Delete operations on databases. The question is, are they all part of one bigger use case, *Manage Frizzles*, or are they separate use cases?

In principle, they are three use cases, because each is a separate goal, possibly carried out by a different person with a different security level. However, they clutter up the use case set and can triple the number of items to track.

Opinion is split as to the best way to deal with them. Susan Lilly advocates keeping them separate in order to keep track of which primary actors have security access to the different functions. I tend to start with just one, *Manage Frizzles* to get the advantage of less clutter. If the writing gets complex, I break out that one part, as described in “Creating a new use case from an extension” on page 112. I track user access to system data and functions using a separate worksheets. Neither way is wrong, and I have not seen enough evidence to form a rule one way or the other.

The following is a use case written by John Collaizi and Allen Maxwell. They started writing both ways. They decided to merge the use cases into summary level *Manage* use case, and eventually broke out the *Save* sub use case to deal with the complexity. I include the use case also to show one way to fit a personal writing style with a different template. They used the Rational Unified Process use case template, and numbered the steps and extensions.

Use Case 32: Manage Reports

1. Brief Description

This Use Case describes and directs operations for Creating, Saving, Deleting, Printing, exiting and Displaying Reports. This particular use case is at a very low level of precision and utilizes other use cases to meet its goal(s). These other use cases can be found in the documents listed in the “Special Requirements” Section.

1.1 Actors

User (Primary)·

File System: typical PC file system or network file system with access by user. (Secondary)

1.2 Triggers

User Selects operations explicitly using the Explorer interface.

1.3 Flow of Events

1.3.1 Basic Flow – Open, Edit, Print, Save, and Exit report

a. User selects Report by clicking report in Explorer and selects open (open also triggered by Double clicking on a report in the Explorer).
b. System displays report to screen.
c. User sets report layout etc. using use case: “Specify Report Specs”.
System displays altered report
d. Steps c and d repeat until user is satisfied
e. User Exits report using use case: “Exit Report”
f. User can Save or Print report at any time after step c using use case: “Save Report” or the “Print Report” Alternate Flow listed below.

1.3.2 Alternative Flows

1.3.2.1 Create New Report

a. User selects “Create New Report” from Explorer by right clicking and selecting option from popup menu.
System creates New Report with Default Name and sets report status for name as “unset”, status as “modified”.
b. Use case flow continues with Basic flow at step b.

1.3.2.2 Delete Report

a. User selects Report by clicking report in Explorer and selects Delete.
b. System opens report (or makes it current if it is already open) and requests validation from user for deleting report.
c. Report is closed and resources cleaned up
d. System removes report entry from report list and report data is removed from storage medium

1.3.2.3 Print Report

a. User selects Report by clicking report in Explorer and selects Print OR user selects print option of current report (a report being edited/displayed in Basic Flow of this use case).
b. User selects printer to send report to and printing options specific to printer (print dialog etc. controlled by operating system) OR user selects to Print Report to File…
c. System loads report and formats. System sends report job to operating system or prints report to designated report file. System closes report.

1.3.2.3 Copy Report

a. User selects Report by clicking report in Explorer and selects Copy.
b. System Prompts for new report name and validates that name doesn’t exist yet
c. System repeats b until user enters a valid (non-existent) name, opts to save over existing

report, or cancels copy operation altogether.

d. System saves report with designated name as a new report.

e. If copy is replacing an existing report, existing report is removed.

1.3.2.4 Rename Report

a. User selects Report by clicking report in Explorer and selects Rename.

b. User enters new name, system validates that name is valid (not the same as it was previously, doesn't exist already, valid characters etc.)

c. System repeats step b until valid name accepted or user cancels use case operation with "cancel" selection.

d. System updates Report List with new name for Selected Report

1.3.3 Special Requirements

1.3.3.1 Platform

The platform type must be known for control of the report display operations and other UI considerations.

1.3.4 Pre-Conditions·

A data element exists on the machine and has been selected as the current element.

1.3.5 Post-Conditions

1.3.5.1 Success Post-Condition(s) [note: this is the Success Guarantee]

System waiting for user interaction. Report may be loaded and displayed, or user may have exited (closed) the report. All changes have been saved as requested by user, hard copy has been produced as requested, and report list has been properly updated as needed.

1.3.3.3.2 Failure Post-Condition(s) [note: this is the Minimal Guarantee]

System waiting for user. The following lists some state possibilities:·

Report may be loaded·

Report list still valid

1.3.4 Extension Points

None

_____

USE CASE 33: SAVE REPORT

1. Brief Description

This Use Case describes the Save report process. This use case is called from the use case: "Manage Reports" and from the use case: "Exit Report".

1.1 Actors

User (Primary)·

File System: typical PC file system or network file system with access by user. (Secondary)

1.2 Triggers

User Selects operations through tasks in the “Manage Reports” Use Case or “Exit Report” Use Case (which is included in “Manage Reports” Use Case) calls this use case.

1.3 Flow of Events

1.3.1 Basic Flow– Save New Report

a. Use case begins when user selects Save Report.
b. System detects that report name status is “not set” and prompts for new report name. User chooses report name, system validates that the report name doesn’t exist in Report List yet. Adds entry to Report List.
1. User cancels save operation… Use case ends.
d. System updates Report List with Report information. System creates unique report file name if not already set, and saves report specs to file system.
e. Report is set as “unmodified” and name status set to “set”
f. Use Case ends with report displayed.

1.3.2 Alternative Flows

1.3.2.1 Alternative Sub Flow – Report name exists - overwrite

a. System finds name in list, prompts user for overwrite. User elects to overwrite. System uses existing report filename and Report List entry. Use case continues with step d of Basic Flow.

1.3.2.2 Alternative Sub Flow – Report name exists - cancel

b. System finds name in list, prompts user for overwrite. User elects to cancel. Use case ends with report displayed.

1.3.2.3 Alternative Flow – Save Report As...

a. User selects Save Report As…
b. User enters new report name, system checks for existence of name in Report List. Name does not exist yet. System finds name in list, prompts user for overwrite. User elects to NOT overwrite. Use case continues at step b
c. Use case continues with step d of Basic Flow.

1.3.2.4 Alternative Sub Flow – Report name exists - overwrite

c. System finds name in list, prompts user for overwrite. User elects to overwrite. System uses existing report filename and Report List entry. Use case continues with step d of Basic Flow.

1.3.2.5 Alternative Sub Flow – Report name exists - cancel

d. System finds name in list, prompts user for overwrite. User elects to cancel. Use case ends with report displayed.

1.3.2.6 Alternative Flow – Save Existing Report

a. User Selects Save Report for Current Report (where Current Report has been saved before and exists in the Report List).

b. System locates entry in Report List, update List information as needed, saves report specs to report file.
c. Report is set as “unmodified”
d. Use Case ends with report displayed

1.3.3 Special Requirements

None

1.3.4 Pre-Conditions·

A data element exists on the machine and has been selected as the “Current element”.·
A report is currently displayed and set as the “Current Report”.·
Report status is “modified”

1.3.5 Post-Conditions

1.3.5.1 Success Post-Condition(s) [note: this is the Success Guarantee]

System waiting for user interaction. Report loaded and displayed. Report List is updated with report name etc. as required by specific Save operation. Report status is “unmodified”, Report Name Status is “Set”.

1.3.3.3.2 Failure Post-Condition(s) [note: this is the Minimal Guarantee]

System waiting for user.·
Report loaded and displayed ·
Report status is “modified”, Report name status same as at start of Use Case.·
Report list still valid – (Report list cleaned up when save fails as necessary)

1.3.4 Extension Points

None

————————————————

14.2 Parameterized use cases

We are occasionally faced with writing a series of use cases that are almost the same. The most common examples are Find a Customer, Find a Product, Find a Promotion, etc. It is probable that just one development team will create the generic searching mechanism, and other teams will make use of that mechanism.

Writing half-a-dozen similar use cases is not much of a problem with casual use cases. However, writing six similar fully dressed use cases is a chore, and it won’t take the writers long to want a short cut. I’ll describe that short cut using the *Find a Whatever* example first mentioned in Use Case 23:“Find a Whatever (problem statement)” on page 86.

We first observed that

* naming a goal in a step is very like a subroutine call in programming, and
* use cases will be read by people, not computers.

Next we noted that finding a thing, whatever thing it might be, must use basically the same logic:

1. User specifies the thing to be found.
2. System searches, brings up a list of possible matches.
3. User selects, perhaps resorts the list, perhaps changes search
4. In the end system finds the thing (or doesn't).

What differs, from use to use is

* the name of the thing to be found
* the searchable qualities (search fields) of the thing to be found
* what to display about the thing to be found (the display values, in sequence)
* how to sort the choices (the sort criteria).

We created a parameterized use case, one that works with a nickname for each of those items. In this case, we need a word that indicates "whatever we are looking for", and also for its search fields, display fields, and sort criteria. We decided to call the use case *Find a Whatever.*

We decided that with a little bit of coaching the reader could safely recognize that the phrase "clerk finds a customer" means call *Find a Whatever* looking for a customer. Readers are surprisingly intelligent and make this jump with little trouble.

We did the same for all the nicknames in the parameterized sub-use case: search fields, display values, and sort criteria. We just had to decide where the writer specified the details.

For the data values, we defined three levels of precision. The first was the phrase mentioned in the use case text, the *data nickname*, such as *Customer Profile*. The second was the *field lists* associated with the data nickname, naming all the information collected under that nickname, e.g., *Name, Address, Day Phone, Night Phone*, etc. The third level of precision was a precise field definition, listing field lengths, validation criteria and the like.

Only the first level of precision was put into the use case. The data descriptions and the search and sort criteria were all put onto a separate page that was hyperlinked into the use case step.

The result was an action step that looks like this:

> Clerk finds a customer using customer search details.

The page *Customer search details* specifies the search fields, display fields in sequence, and sort criteria. The reader of the use case would clicks on the underlined phrase to see it. This entire mechanism was easy, quickly understood by readers, writers, and implementers.

Find a Whatever now starts to look like this:

1. The user identifies the searchable qualities of the whatever thing.
2. The system finds all matching whatevers and displays their display values in a list.
3. The user can resort them according to the sort criteria.
4. The user selects the one of interest.

Using this neat trick, the calling use cases remain uncluttered by the gory (and lower-level) details of searching, sorting, resorting, and the like. The common searching behavior gets localized, written only once. Consistency across finders is assured, the people who really need to know the details for programming purposes can find their details. Indeed, in the case I witnessed, the implementation team was delighted to be given just *one* specification for their search mechanism, so they didn't have to wonder whether all the searches were really the same.

Those are enough hints. Now go away and finish Exercise 19“Find a Whatever.” on page 86. Worry, in particular, about the guarantees and the extensions, because they are both important to the callers.

15. Business Process Modeling

Everything in this book applies to business processes as well as to software systems design. Any system, including a business, that offers a set of services to outside actors while protecting the interests of the other stakeholders can be described with use cases. In the case of businesses, the readability of use cases is quite helpful.

There are examples of business use cases in other parts of this book, specifically:

* Use Case 2:“Get paid for car accident” on page 18
* Use Case 5:“Buy Something (Fully dressed version)” on page 22
* Use Case 18:“Operate an Insurance Policy” on page 72
* Use Case 19:“Handle Claim (business)” on page 78
* Use Case 20:“Evaluate Work Comp Claim” on page 79

Modeling versus designing

Saying, "We are using use cases for business process reengineering" may mean any of:

* "We use them to document the old process before we redesign it."
* "We use them to create outer behavioral requirements for the design to meet."
* "We will use them to document the new process, after it gets redesigned."

All of the those are valid and interesting. I ask that you understand which one you intend.

I carefully say business process *modeling* or *documentation* instead of business process *reengineering* or *design* when talking about use cases. A use case only documents a process, it doesn’t reengineer or redesign it. In creating the design, there is a leap of invention made by the designers. The use cases do not tell the designers how to make that leap of invention. Each level of documentation serves as a behavioral requirement that the next level of design must meet (indeed, we say "this design *meets* these behavioral requirements").

Introducing new technology often changes the business’ processes. You can work to realign them from the core business toward technology, from the new process to the technology, or from the technology directly (deriving the process concurrently). Any of these ways can work.

Working from the core business

In this top-down way of working, you start by carefully identifying the organization's core business, as described in Hammer's Reinventing the Organization. At the end of the exercise you will know the

Stakeholders in the organization's behavior,

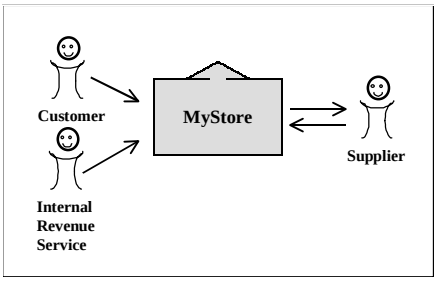
External primary actors having goals you propose the organization satisfy,

Services the business offers, with the success outcomes for the stakeholders, and

Triggering events that the organization must respond to.

Notice that, without saying *how* your organization will work, you now have the information that sets boundary conditions for its behavior. Not accidently, this is also the bounding information for a use case: stakeholders and interests, primary actors with goals, success guarantees.

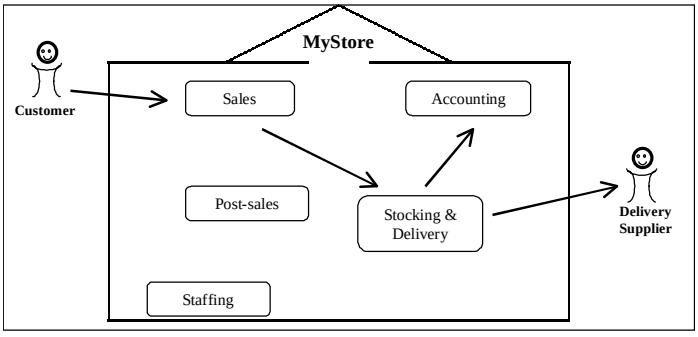
Figure 19. Core business black box.



The context for the business process design can be documented using business black-box use cases, having the company or organization as the system under design (Figure 19.).

At this point, you invent new groupings of your resources and new processes to make the best use of current technology. These days, computer systems serve as active memories and active communication conduits for the organization. Hammer gives many examples in his book of how different acts of invention lead to different business designs, and their relative effectiveness. The result is a new corporate or organizational design (Figure 20.).

Figure 20. New business design in white box.



The result of the process re-invention gets then be documented using white-box use cases, showing people and departments (and maybe computers) interacting to deliver the organization's externally visible behavior.

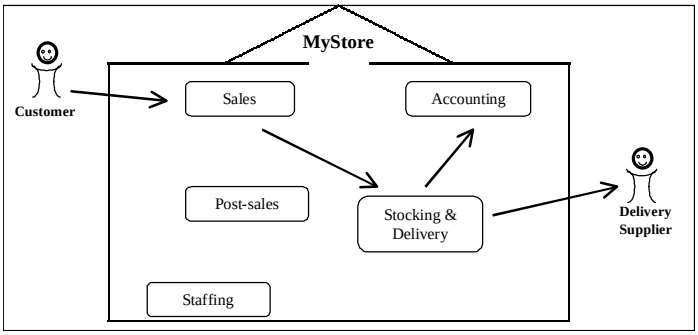
The fully developed white-box use cases must show the organization's handling of all failure and exceptional conditions, just as would any complete set of use cases or any complete business process model. You may choose to name the technology in the use cases or not, as suites your purpose.

Work from business process to technology.

With this intermediate starting point, you are not questioning the organization's core purpose, but rather, defining the new business that the new technology will fit into. You probably have already nominated some new technology, perhaps a new set of software applications, or mobile devices. You want to set boundary conditions for the technologists' invention.

You therefore write white-box business use cases that document the proposed new business processes *without* mentioning the new technology (Figure 21.). Mentioning the new technology in this situation is as inappropriate as describing user interface technology in a system use case. An example is given in Use Case 21:"Handle a Claim (systems)" on page 80.

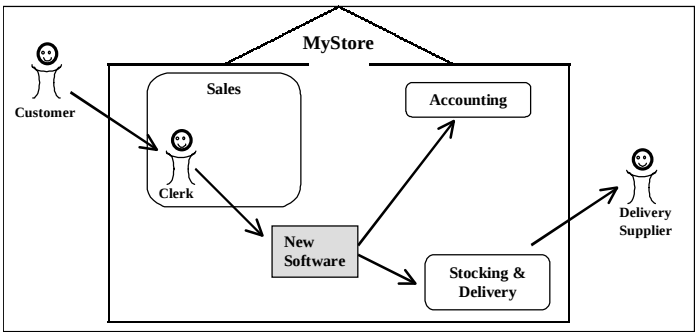
Figure 21. New business design in white box (again).



In principle, the computer can be replaced in the descriptions by baskets of paper shipped from person to person. Your team's task will be to invent how active conduits such as computers or a mob of palm computers and radios can improve the process.

The designers now know what process their invention must support. After inventing, they will write black-box *system* use cases that show the new system fitting into the technology-free white-box business use cases. The system use cases will be the requirements used for the system design. See Figure 22..

Figure 22. New business process in black-box system use cases.



While this looks quite wonderful on paper, it costs a lot of money and is time consuming. Technology moves so fast and creates such a pressure that often there is no time to work this way. I have several times found that *usage experts,* people in the business who are experts in their working areas, can do the new business process modeling in their heads, allowing you to save time and money. You would then work in the third way...

Work from technology to business process.

First, collect some experienced usage experts, people in the business who are experts in their working areas, who probably have been eager to improve their groups' technology and work habits for some time. Make sure you have two representatives for each part of the business that your system will affect.

In negotiation with the technology experts, they will nominate system capabilities that will improve the processes. Be prepared. They will nominate more than your development team can deliver (that's all right - technologists need to be stretched!).

Have these usage experts write system black-box use cases for the system they imagine. They will not mention the user interface in these use cases. The use cases describe what the system will *do* to support the primary actors in their business goals as effectively as possible. The extensions sections of the use cases will involve all sorts of critical or seldomly discussed business rules. The usage experts may have to talk to their colleagues to clarify fine points of system behavior. As part of writing the system use cases, they will, of course, write a few summary level and business use cases showing context, and the linkage of goals over time.

In other words, a light business process documentation will be generated as a normal part of the system requirements exercise. The usage experts will have done the new process modeling in their heads, while arguing about how the actors and new system should behave under various circumstances. I have found this to be quite an effective way of working.

* Use Case 3:"Register arrival of a box" on page 19 illustrates how documenting the system behavior can end up describing a fragment of the business process, complete with exceptional conditions.

* Use Case 21:“Handle a Claim (systems)” on page 80 is a summary (kite-level) use case that shows the business process context for the system.

Linking business- and system use cases

A business use case has the same appearance as a system use cases, so all the training in writing and reviewing use cases can be applied to both business use cases and system use cases. The ever-unfolding story starts in the business and unfolds into the system use cases. That is the synergy that business and system use cases offer.

The bad news is that writers and readers will accidentally mix them. They are likely to put system behavior into the business use cases and business operations into the system use cases. That could useful, if done deliberately, but often the writers and readers don't realize they doing so. A reader expecting a system use case will criticize reading a business use case for being too high level, not noticing that it is not intended to provide system behavior detail. A writer of a business use case might accidentally includes a great detail of system behavior, with the result that the business executives lose interest while reading these overly detailed documents.

To help reduce the confusion, *always* write the scope and level in the use case template. Train your writers to write them, train your readers to read them at the start of every reading. Use graphic icons if you can. Use slightly different templates for the two. Number them with entirely different use case numbers (one group started the business use cases at 1,000 and the system use cases at 1). Concoct something *immediate* and *visual*. Then you can take advantage of the synergy available, without people getting lost.

The other bad news is that it rarely is worth the effort to completely and properly link the business and the system use cases. Usually, the people writing the business use cases describe the business process down to, but not including the use of the system. They run out of time, money, energy and motivation before writing how the new system is used in the daily life of the business people. The people writing the system use cases sometimes add a sentence or two of business process for context, but have no motivation to rewrite the business use cases to include the new system’s functionality.

The result is that there is this little gap between business and system use cases. Rusty Walters comments on this:.

> I have yet to experience to my satisfaction a full fledged story of business use cases that unfold into system use cases. In my experience, it is quite common to have three levels of business use cases. A few black-box, cloud level business uses cases to get started. These quickly turn into white-box, cloud level business use cases, that unfold, meaning they name a white-box, kite level business use case.
>
> However, I have not seen a clean connection from the business use case to system use cases.

This is bad news for those looking for some sort of automatic way to derive system requirements from business processes. I don't think that automatic derivation is possible, as I described in "Modeling versus designing" on page 153.

Some people find this troubling. I don't. Most of the people I deal with in organizations are quite capable of making the mental leap to link the lowest *business* use case with the kite or sea-level *system* use cases, once they know to do that. Furthermore, I have not yet seen that writing that final set of use cases, which completely links the business to the system use cases, is worth the time and money it would cost to do so. The two efforts run from separate budgets, and each activity appropriately ends when its goal is satisfied.

Reexamine the use cases starting with Use Case 19:"Handle Claim (business)" on page 78. The system use cases are, indeed mentioned in the business ones, but they were written specifically to provide context for the system requirements group, not at the start of a separate business process reengineering activity.

Rusty Walters of Firepond write of his experiences with business process use cases.

RUSTY WALTERS: BUSINESS MODELING AND SYSTEM REQUIREMENTS

Having the benefit of reading your book early, I've been able to rationalize problem areas with previous attempts, and utilize my new found knowledge.

Analyzing my pre-book experiences after reading the book

Prior to reading the book, I helped document functional requirements for several applications in a product suite.

For one application, we developed *system* use cases at summary, user, and subfunctional level. We concentrated totally on the system functionality. We were pleased with the outcome of the use case model, as it read quite well. We found no need to develop any business use cases to show context; the system summary level use cases were all that we needed.

On another application within the suite the story was quite different, even though the same group was responsible for this use case model as the previous. Looking back, I now can see the crux of the problem was different people on the team approaching the problem from different perspectives. I was working from business process to technology. Some other people were working from technology to the business process. Needless to say, the design scope for each use case wasn't clear between the two groups.

The business-process-to-technology group never got to writing the system use cases, and the technology-to-business-process group never got to writing the business use cases. Instead, they sort of hit each other in a head-on collision, with each group trying to coerce the others as being their business/or system use case. Not having the insight or the necessary understanding at the time to label the use cases correctly for scope and level, the use case model became quite a mess. In the end, the team was never really happy

with the use case model, they knew it didn't "smell" right, but didn't know what exactly was wrong.

My Post-Book Experience

Working from the core business seems to cause the least confusion, as I discovered in a group whose purpose was to understand and document their processes.

It was clear to everyone that we were gathered to talk about their processes and the way they work within the business, and not about software/hardware systems. The areas of confusion that did come up were related to *business* versus *department* scope.

We started with business, very-summary (cloud) *black*-box use cases -- this was very clear to everyone, even though the group quite often wanted to dive down into lower level steps. We quickly moved into writing very-summary (cloud) *white*-box use cases, as you describe. When we decided to talk about the next lower-level use cases, confusion arose quickly about the design scope -- were we talking about the business or about a particular department? This also went hand-in-hand with what constituted success for the use case. In one particular case, we ended up removing the last two steps after we realized those steps were really done in the calling use case and were out of success scope for the current one. Currently the group has no intention to ever unfold the business use cases into system use case requirements.

It was much easier to understand the problem areas after the meeting, although it was difficult to notice and correct course during the meeting. In documenting the outcome, I used the design scope context diagrams, labeling the design scope and level of each use case with graphic icons as you suggested. As simple as the graphics may seem, it has quite an impact upon reading the use cases, and helps greatly in keeping them straight in everyone's mind.

---

16. The Missing Requirements

It is all very well to give the advice, "Write the *intent* of the actor, just use a *nickname* for the data that gets passed," as in "Customer supplies name and address." However, it is clear to every programmer that this is not sufficient to design to. The programmer and the user interface designer need to know what exactly is meant by address, which fields it contains, the lengths of the fields, the validation rules for addresses, zip codes, phone numbers, and the like. All of this information belongs in the requirements somewhere - and not in the use case!

Use cases are only "chapter three" of the requirements document, the behavioral requirements. They do not contain performance requirements, business rules, user interface design, data descriptions, finite state machine behavior, priority, and probably some other information.

"Well, where are those requirements?!" the system developers cry! It is all very well to say the use cases shouldn't contain those, but they have to get documented sometime.

Some of the information can, in fact, be attached to each use case as associated information. This might include

* use case priority
* expected frequency of occurrence
* performance needs
* delivery date
* list of secondary actors
* business rules (possibly)
* open issues

Different projects adjust this collection of information to contain whatever they consider important.

In many cases, a simple spreadsheet or table captures the information well. Many people use a spreadsheet at the start of the project to get an overview of the use case information. With the use case name in the leftmost column, they fill the other columns with:

* primary actor
* trigger
* delivery priority

* estimated complexity
* probable release
* performance requirement
* state of completion
* ...and whatever else you need.

That still leaves out the section the programmers need just as much as they need the behavioral requirements - the data requirements, including field checks to perform.

Precision in data requirements

Collecting data requirements is subject to the same discussion of managing energy with "precision" as every other work product (Section 1.5“Manage Your Energy” on page 29). I find it useful to divide the data requirements into three levels of precision:

* Information nicknames
* Field lists, or data descriptions
* Field details & checks

Information nicknames. We write "Clerk collects customer information" or "Clerk collects customer's name, address and phone number." We expect to expand on the description of each of *name*, *address*, and *phone number* - in some other place.

The nicknames are appropriate within a use case. To write more would slow down the requirements gathering, make the use cases much longer and harder to read, and make them also more brittle (sensitive to changes in the data requirements). It is also likely that many use cases will reference the same information nicknames.

Therefore, break the details of the data requirements out of the use case, and link from the use case to the relevant *field list*. This can be done with a hyperlink in many tools, or with a requirements number in tools which support cross-linking numbered elements.

Field lists. At some moment, the requirements writers will have to negotiate over just what each information nickname means. Does "customer's name", consist of two parts, first and last names, or three parts (or more)? What exactly is needed for "address"? Addresses around the world have so many different formats. This is the appropriate place for the writers to expand the data descriptions to the second level of precision. It might be done gently in parallel with writing the use cases, or it might be done afterward, perhaps working alongside the user interface designer.

There are many strategies for dealing with the second level of precision. I’ll name two for you, you can consult Constantine and Lockwood's Software for Use and Luke Hohmann's GUIs with Glue for other ideas, or you may have experience in this area yourself.

- The first strategy is to have a separate entry in your tool for each nicknamed item. Under "customer name", you identify that there are the three fields: the customer's first name, middle name, last name. That's all. Over time, you will add more precision to this entry, adding field details and checks, until it contains all the details about those fields, as described in *Field details & field checks*, next.
- The second strategy is to notice that you wrote "name, address and phone number" together in a single use case step. That you did so means that it is significant to you that these three parcels of information arrive together. This is useful information for the user interface designer, since it is quite likely that these three parcels of information will show up together. The UI designer may design a sub-screen, or field cluster, to support the fact that these three parcels show up together in different places. Therefore, you create single entry in your tool for "name, address, and phone number". There you will list what fields are required for name, what fields are required for the phone number, and what fields are required for the address, but you don't expand those lists further.

The difference between the two strategies is that in the second strategy, you put clusters of nicknamed information onto each field list page. When you expand to more precision, you will not expand in that entry, but will create a separate entry for each field.

Whichever strategy you choose, you can expect the information at the second level of precision to change as the project moves forward, and the team learns more about the specifics of the data. You may also have different people to define the second and third levels of precision for the data. It will probably turn out handy to keep the second level of data precision separate from the use case itself.

Field details & field checks. What the programmers and database designers really need to know is, "How many characters long can the customer's name be?" and "What restrictions are there on Date of Loss?" These are field types, field lengths, and field checks.

Some project teams put these items into a requirements chapter called Data Requirements or External Data Formats. Some, who use a hyperlinked medium or database, put them into separate entries classified under Field Definitions, and others put UI data details directly into the user interface requirements and design document.

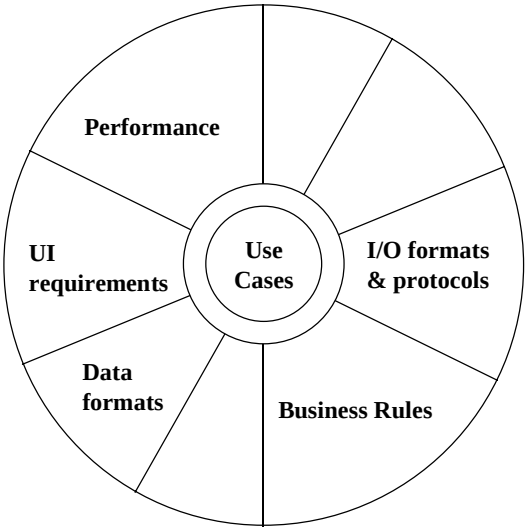
Whatever you decide, note that:

- You do need to expand the field details and checks to the third level of precision.
- The use case is not the place to do that expansion.
- The use case should link to that expansion.
- The field details are likely to change over time, independently from the use case details.

Cross-linking from use cases to other requirements

Data formats are not part of the use case, but the use case names the need for the data, and so we can hyperlink from the use case to the data descriptions. Complex business rules do not fit well into the use case narrative, but again, we can link to entries containing them. This sort of hub-and-spoke linking makes the use cases the center of the requirements document, even for many of the non-functional requirements (see Figure 23.).

Figure 23. Recap of Figure 1.“"Hub-and-spoke" model of requirements”.



Just be careful not to force into use cases those requirements that don't fit well into the use case form. Use cases are best suited to capture interactions between actors.

Occasionally, I hear someone complain that it is hard to describe the requirements for a tape merge operation or a compiler using use cases. Indeed, I wholeheartedly agree. Those are best described using algebraic or tabular forms, not use cases!

Only a fraction of the entire requirements set is suited to the use case form. The other portion must be written using other forms. It just happens that the interaction part is central, and connects many other requirements.

17. Use Cases in the Overall Process

17.1 Use Cases In Project Organization

The use cases give the management team a handle on usable function being delivered to the users. The title of each use case names a goal that a primary actor will find supported by the system. The body of each use case announces just what the system will require and provide.

Organize by use case titles

During early project planning, create a table with the use case and primary actor names in the left two columns. In the next column, have the business sponsors put the priority or value to the business of each use case. In the column after that, have the development team estimate the complexity or difficulty of delivering that function. This is a natural evolution of the Actor-Goal list work product (see “The Actor-Goal List” on page 45).

In a nice variation on this theme, Kent Beck, in his "Planning Game"[1] has the developers estimate the development cost while the business sponsors decide the priority of each use case. The business sponsors have the opportunity to change the priorities after seeing the work estimates, but may not change the work estimates. In the light of this idea, you may wish to fill the columns of the planning table in two passes.

In the other columns, put the development priority of the use case, the release it will first show up in, and the team that will develop it. You can view and manipulate this list with ease over the course of the project.

Figure 24. Sample planning framework.

Actor	Task-level Goal	Business Need	Technical Difficulty	Priority	UC #
Any	Check on requests	Top	Large job in general case	1	2
Authorizor	Change authorizations	High	Simple	2	3

1. Beck, K., Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

Figure 24. Sample planning framework.

Actor	Task-level Goal	Business Need	Technical Difficulty	Priority	UC #
Buyer	Change vendor contacts	Medium	Simple	3	4
Requestor	Initiate an request	Top	Medium	1	5
	Change a request	Top	Simple	1	6
	Cancel a request	Low	Simple	4	7
	Mark request delivered	Low	Simple	4	8
	Refuse delivered goods	Low	Unknown	4	9
Approver	Complete request for submission	High	Complex	2	10
Buyer	Complete request for ordering	Top	Complex	1	11
	Initiate PO with vendor	Top	Complex	1	12
	Alert of non-delivery	Low	Medium	4	13
Authorizer	Validate Approver's signature	Medium	Hard	3	14
Receiver	Register delivery	Top	Medium	1	15

Over time, complete the estimates for each use case, assign them to teams, and track the work per use case per release. Here is a short true story about using this planning table to gauge, estimate, prioritize and reduce the possible working set of use cases to an appropriate set.

A small, true story.

> A developer was given the assignment to decide what business processes to support in the next releases of the software. She came up with a 7- page actor-goal list! She estimated value, complexity, trigger, and so on. She and the executive sponsor trimmed it to half a page. She wrote the main success scenario for those business processes, and with the executive sponsor trimmed the list of steps to consider to about half a page of system-level goals. Having that information, she toured the branch offices. She came back with a clear picture of which business steps could be addressed to most benefit the branch office workers. She and the sponsor identified the four use cases to develop over the next six months.

The benefits of working this way:

* The use case list clearly shows the system's value to the business.
* The list of names provides a structure to work with development priority and

timeframe for the use cases.

Use cases cross releases

It would be soothing to say that a tidy set of complete use cases map to each release. Except that they don't.

A use case such as *Order Product* calls out all sorts of special handling, which will be delivered over time. A typical staging strategy is

* Deliver the simple case in release 1
* Add high-risk handling in release 2
* Put preferred customer handling in release 3

Either you write one use case and deliver a fraction of it in each release, or you write three use cases. Each way will work, and each way will hurt.

Some teams choose to split the use cases into units that get delivered *completely* on releases. They write *Order Product (Base Case)*, followed by *Order Product (High-Risk Handling)* and *Order Product (Preferred Customer Addition)*. They either repeat the use case body, adding the new elements in italics, or they write the second as extension use case on the first and the third as an extension use case on the second. Splitting the use cases simplifies tracking. On the other hand, there are three times as many use cases to track, and it can become tricky to write the add-on use cases.

Others (me, for instance) like the use cases to be as readable as possible and can live with the fact that use cases will be released portions at a time. They highlight (in yellow or italics) the parts that will be released on the first release, and refer to "the highlighted portions of use case 5". This is the way I have seen most projects work, and it works passably well. But it is not as tidy as we might like.

A possible middle-road strategy is to start by writing the full, multi-release use case. Then, at the start of an increment, the team writes a version of it, isolating just the functionality they plan to deliver in that increment. On the next increment, the team writes in plain text for the parts of the use case that have already been delivered, and **in bold text** for the parts that are new in the upcoming release. With this strategy, the number of use cases still multiplies, but in a more controlled way. If there are 80 use cases to start with, the team's sensation is of managing 80 use cases plus however many are being delivered in the specific increment. The use case set expands, but in a localized and controlled fashion.

You will have to choose which way to work, knowing that each has drawbacks.

Note that all approaches presuppose the organization is using incremental staging and delivery of the system, with increments of about four months or shorter. Incremental development has

become a standard recommendation on modern software projects and is discussed in detail in the book, Surviving Object Oriented Projects, and the article "VW Staging'[1].

Deliver complete scenarios

A short true story about integration

> The test and integration manager of a very large project asked me about the hazards of incremental development. It turned out that the team was developing and delivering to him to test, the application in increments of "features", rather than by use case. His team could not test the pieces given to him, since there was no "story" that they supported, just a heap of mechanisms. The project leaders eventually rebuilt the project plan so that each increment's work produced a usable storyline.

It often happens that not all of a use case gets delivered at one time, but each delivery must delivers a full scenarios from a use case. The use case says what the user wants to accomplish, and enumerates the many scenarios involved in accomplishing that. When you deliver the software, you must deliver some of those scenarios *top to bottom*, or your application is not delivering usable function.

Planning and design must coincide to produce end-user usable collections of functions. Those are full scenarios from the use cases. Functional testers will test for use case compatibility. Deployment can only deploy in full usable threads of the use cases.

This seems trivial, but overlooking it can hurt, as the story illustrates.

17.2 Use Cases to Task or Feature Lists

Development teams with good communications work out their design tasks from the use cases. This is good, because otherwise there awaits a small project management nightmare in mapping the use cases to design tasks, and keeping that mapping current.

The following email from a colleague illustrates ...

> Two of us visited ____ the last two weeks to relay the requirements and to establish a working relationship with the developers. We had been focusing on the use cases and felt they were 90%-95% precise enough for design and development; so we were confident. There was an scope document created with the intent of describing what features or feature sets were in or out, very similar to the sample scope section you made available. This document was originally very small, short and sweet, but we kept getting requests for a "traditional" requirements document. So, we had someone expand it a little bit but tried to keep the level of detail to a minimum.

1.---- Online at http://members.aol.com/acockburn/papers/vwstage.htm

> Low and behold, in our first meeting with the developers, they wanted to review the "requirements". The "requirements" to them were represented in this scope document. We saw how the development team was latching on to this document, and knew it lacked the detail we wanted because we were hoping to focus around the use cases. We spent the next three days developing the newly revised scope document that you will see attached. I like to refer to it as "spoon feeding" the use cases to them. Hoping to make an impact on the development culture and to help the company make a transition to use case based requirements, we essentially used the name of each use case as a feature set, and each step within the scenarios as features. We were literally copying the steps from the use case documents and pasting them into the scope document underneath their appropriate use case heading.
>
> The problem we faced, and ultimately caused us to do this double duty maintenance, is the exact text from the scenarios doesn't stand on its own very well as a line-item feature. Even though we copied the text from the scenarios, we would constantly reword a little, or add some context around it.

This is the start. The designers want to work from numbered requirements paragraphs, or feature lists, much as just described. In other but similar versions of the above story, the "detailed requirements document" or numbered paragraph document was written first. Someone then decided to write use cases from them (this seems a little backwards to me, but that's what happens).

There is a fundamental tension between the flow of the system's behavior as described in use cases, and the design tasks an individual designer is given. A designer works on a single line item or feature buried in one use case, or perhaps cross linked across many. That might be the Undo mechanism, the Transaction Logging mechanism, or the Search Screen Framework. The designers cannot describe their work in use case language, because they are not working with the flow of the system. The best they can say is, "I'm working on the Search Screen part of use case 5."

At the same time, the sponsors of the software want to see the application in full flow, delivering value to the users. The individual design tasks are not individually interesting.

It is time-consuming and tiring work to keep the use case document and the design task list in sync, requiring the work described in the email above, repeated many times on the project. I view this as work best avoided. So far, I have not yet had to break apart the use cases into line items on projects up to 50 people in size. Perhaps that is because I stress personal communication and cooperation on my projects, and have been fortunate with the cultures I have worked in. We have been able to break apart the line items in our heads, or using a yellow highlighter, and write the key ones down into the task list for scheduling without much overhead.

The other alternative is to generate two documents, and work hard to keep them up-to-date. If you decide to follow this strategy, break the use case text into pieces that can be allocated to single developers or single development teams. Each piece becomes a program feature, mechanism or design task that will be assigned, tracked, and checked off. The detailed estimate for the software

development is the sum of all the design task estimates. Project tracking consists of noting the starts and completions of each of these design tasks.

The following is an example of converting a use case to a work list. Here is the use case.

USE CASE 34: CAPTURE TRADE-IN

Goal in Context: The Shopper has a shopping cart containing products and wants to add a trade-in to see how it will affect costs.
Scope: Commerce software system
Level: Subfunction
Preconditions: The shopping cart must contain product(s).
Success Guarantees: The trade-in has been valued, added to the shopping cart, and reduced the cost of the items contained in the shopping cart.
Minimal Guarantee: If not finalized, the trade-in is not captured or added to the shopping cart.
Primary Actor: Shopper (any arbitrary web surfer)
Trigger: Shopper selects to capture a trade-in.
Main Success Scenario:
1. Shopper selects to capture a trade-in.
2. System determines the value of trade-in by presenting information to the Shopper, and asking a series of questions to determine the value of the trade-in. The series of questions and information presented depend on the answers the Shopper gives along the way. The path of questions and information is pre-determined, in order to highlight probable business practices around trade-ins.
3. The system logs the navigation and trade-in information along the way.
4. Shopper reviews the trade-in summary and value, considers it.
5. Shopper adds it to the shopping cart.
6. System adds to the shopping cart the trade-in and the navigation information.
7. The System presents a view of the shopping cart with all the products and trade-ins contained in it, as well as re-calculating the total cost taking into consideration the trade-in(s).

The Shopper repeats the above steps as many times as desired, to capture, and value different trade-ins, adding them to the shopping cart as desired.

Extensions:
2a. At any time prior to adding the trade-in to the cart, the shopper can go back and modify any previous answer.
5a. Shopper decides not to add the trade-in to the shopping cart: System retains the navigation information for later.
5b. Shopper wants the trade-in to be applied to a specific item in the shopping cart: Shopper specifies a product contained in the shopping cart they wish the trade-in to be applied against.

Here is the generated work list:

FEATURE LIST FOR *CAPTURE TRADE-IN*

	Feature	**Business Need**	**Ver**
EC10	**Capture Trade-in**	**Must have**	**1.0**
EC10.1	Provide the ability for the shopper to enter a trade-in into the shopping cart.	Must have	1.0
EC10.2	Provide the ability determine the value of the trade-in by presenting and navigating through generated UI forms (based on templates) to gather the shopper's trade-in information to determine its value.	Must have	1.0
EC10.3	Provide the ability to go to an external trade-in system (or site) to determine trade-in value. The shopper's related trade-in information would be passed to the external site and the external site would evaluate the trade-in and return its value and important characteristics.	Must have	1.0
EC10.4	Provide the ability for the shopper to be present a trade-in summary including its value.	Must have	1.0
EC10.5	Provide the ability for the shopper to add or discard the trade-in. Upon adding the trade-in to the shopping cart the shopper can associate it to an individual product or all products in the shopping cart.	Must have	1.0
EC10.6	Provide the ability to re-calculate the total cost of the contents of the shopping cart taking into consideration of the trade-in(s).	Must have	1.0
EC10.7	Provide the ability to edit an existing trade-in by returning to the trade-in question/answer process for editing.	Must have	1.0
EC10.8	Provide the ability to delete an existing trade-in from the shopping cart and re-calculate its total cost.	Must have	1.0
EC10.9	Provide the ability to log any trade-in information or steps based on pre-configured triggers.	Must have	1.0

17.3 Use Cases to Design

Use cases provide *all and only* black-box behavioral requirements for the design to meet. The requirements are to name everything that the system must do, and not usurp any design freedom the designers should have. It is for the designers to use their craft to produce a "good" design that meets the requirements. The two are not supposed to meet more than that.

There are several things to say about transitioning from use cases to design, some good news and some bad news. The bad news is that

* design doesn't cluster by use case, and
* blindly following the use case structure leads to functional decomposition designs (this is really of concern to object-oriented and component design teams).

The good news is that

* some design techniques can take advantage of all those the scenarios, and
* use cases name the concepts needed in domain modeling.

Let's look at the bad news first.

Design doesn't cluster by use case. The design tasks do not map themselves tidily to use case units. A design task results in a business object or a behavioral mechanism that will be used in several use cases. One of the use cases scheduled for a later release is likely to contain important information for a design task done in an earlier release. That means that the designers will have to change their designs in the later release, when they encounter that information.

There are three ways to handle this. The first is to have the designers scan all the use cases to collect key information that might apply to their design task. This can clearly only be done on smaller projects. If you can manage to do this, then you will be ahead.

The second approach is to scan all the use cases looking for *high-risk* items, key functions that are likely to have a strong effect on the design. The team creates a design to fit these key functions, hoping that the remaining functions will not disturb the design too much.

The third alternative, which is my preference, is to recognize that the software will change over its lifetime, and to relax about that. The team designs each release as well as practical, recognizing that sometime in the next year, new requirements are likely to surface that will cause a change.

This third alternative may cause some of your team discomfort, particularly those coming from a database design culture. In many of those environments, it is expensive to add new fields to an table and reoptimize the database. The economics indicate they should first identify all the attributes that will *ever* be referenced. They build and release software that can then be called 20% complete, 40% complete, up to 100% complete, with respect to the total set of attributes.

In most modern environments using incremental development, however, adding an attribute to a class or table is a minor operation. It is cheap enough that developers only define those parts of a class or entity that are needed immediately. The consequence is that the classes and components are never "complete". They are "complete with respect to a given set of functions". As new functions are delivered, the notion of "complete" changes.

A short, true story.

> This matter came to a head on one project, when one team lead complained that he had not been allowed to "complete" the classes past the 20% completion mark, even though the delivered application did everything the customer wanted! It took us a long time to sort out that he was speaking from the database design culture, but working on a project using the incremental development culture.

Be prepared for this discussion. Unless there are extremely strong economics penalties, I suggest that your team work to the model of "completeness with respect to a named function set."

Use cases and functional decomposition If you are using structured decomposition techniques, then the function decomposition in the use cases is probably useful to your work. However, if you are doing object-oriented design, then there are some special notes to take.

A SPECIAL NOTE TO OBJECT-ORIENTED DESIGNERS

Use cases can be said to form a functional decomposition, or function hierarchy. This has shown itself to be useful for the purposes of constructing and communicating behavioral requirements. The writing is easy to understand, and the behavior rolls up neatly into higher and higher levels. It makes life easier on the people who have to agree on *what* the system must *do*.

Such functional decomposition may be good for requirements, but that does not imply it is good for software design. Co-encapsulating data and behavior has shown itself to be useful for simplifying maintenance and evolution of software designs. There is, however, no evidence that it makes a good structure for gathering or communicating requirements. My experience is that it is *not* as good as the function-based structure. In other words, gathering requirements benefits from functional decomposition *at the same time* as software design benefits from data+behavior componentization.

Designers have to read the use cases, think and discuss for a while, and then come up with useful abstractions. It is their job. It is not the user’s job to do this.

One hazard is that the inexperienced or unwary designer creates classes that mirror the functional decomposition of the requirements document, simply casting each use case into a class/object/component. Over the years, this has shown itself to be a poor strategy, and many OO experts explicitly warn against it.

One can probably defend casting a user-goal use case as its own class, since a user-goal use case captures a full transaction, with coherent commit and rollback semantics. It is quite possibly a good candidate for encapsulation. However, subfunction use cases rarely have those characteristics. They are usually partial mechanisms, belonging piecewise in different classes.

The opposite hazard lies with the OO designer who wants to model the domain directly, without worrying about the functions it needs to support. These designers miss out on the contribution of the functional requirements. The use cases tell the domain modeler *which aspects of the domain are interesting to consider*. Without that information, the domain modeler is likely to spend excessive time modeling parts of the domain not relevant to the system at hand. The use cases provide boundary conditions for sound domain modeling. Read more on this in the article "An Open Letter to Newcomers to OO"[1].

In all cases, it is clear that use cases partition the world along a different organizing scheme than do objects. This means that you will have to think while designing.

Now let's look at the good news.

Design makes use of scenarios The use cases serve as handy design scenarios when it comes time to design the program. They are particularly useful when used with *Responsibility Based Design*[2]., which is based on designing while stepping through scenarios. The use cases also serve well with other design techniques, showing when the design is complete and handles all situations (the extensions).

Use cases name domain concepts The use cases fairly shout out the names of the domain objects involved. Consider the use case phrase:

> "System produces an *invoice*, fills the *invoice line items* with their costs, adds tax and shipping costs, and produces the total. The *invoice footer* states terms of delivery."

It does not take a great leap of imagination to see Invoice, InvoiceLineItem, InvoiceFooter with attributes cost, tax, shipping, total. These are not necessarily your final design items, but they certainly are a good starter set of business objects. I have seen project teams go directly from the concepts in the use cases to a draft design. They tighten and refine the design from there.

1.http://members.aol.com/humansandt/papers/oonewcomers.htm

2.Read the original article: Beck, K., Cunningham, W., "A laboratory for object-oriented thinking", ACM SIGPLAN 24(10):1-7, 1989, or the book: Wirfs-Brock, R., Wilkerson, B., Wiener, L., Designing Object-Oriented Software, Prentice-Hall, 1990. Online, visit http://c2.com/cgi/wiki?CrcCards or http://members.aol.com/humansandt/papers/crc.htm.

17.4 Use Cases to UI Design

Larry Constantine and Lucy Lockwood, in Software for Use, and Luke Hohmann, in GUIs with Glue, have written better than I can about designing the user interface. However, most project teams ask, during use case writing, "How do we transition from UI-free use cases to actual UI design?"

You have some people on staff with the assignment, and hopefully the skill, to invent a pleasant-to-use user interface. Those people will read the use cases and invent a presentation that preserves the steps of the use case while minimizing the effort required of the user. Their UI design will *satisfy the requirements given by the use cases*. The design will be reviewed by users and programmers for that.

Use case writers can find it helpful to pretend they are typing into a data-capturing screen, or filling out a paper form, to discover what information has to be entered and whether there are any sequencing constraints on entering that information. Pay attention that these forms are interpreted as indications of how the usage experts view their task, not part of the requirements.

While it is not for the requirements document to describe the UI design, it is useful to *augment* the requirements document with samples of the UI design as it evolves. This design information can add to the readability of the requirements document, since it give both textual (abstract) and visual (concrete) renditions of the system's behavior.

The UI design has three levels of precision, low, medium and high:

- The *low-precision* description of the user interface is a screen-navigation diagram, drawn as a finite state machine or statechart. Each state is the name of a screen the user will encounter. The finite state machine shows what user events cause movement from one screen to another.
- The *medium-precision* description is a drawing or reduced size snapshot of the screen. Place this at the end of the use case, so that readers can both see and read what design is being nominated.
- The *high-precision* description lists all the field types, lengths and validation checks of each screen, and does not belong in the requirements document at all!

17.5 Use Cases to Test Cases

Use cases provide a ready-made functional test description for the system. Most test groups positively salivate at the opportunity to work with use cases. It is often the first time they are ever given something so easy to work. Even better, they are given this test suite right at requirements time! Best of all is when they get to help write the use cases!

In a formal development group, the test team will have to break the use cases up into numbered tests and write a test plan that identifies all the individual test settings that will trigger all of the different paths. They will then construct all the test cases that set up and exercise those settings. In addition, they will exercise all the different data settings needed to test the various data combinations, and will design performance and load tests for the system. These last two are not derived from the use cases.

All of this should be business as usual for the test team. Here is a small example provided by Pete McBreen[1]. First comes the use case, then comes the set of Acceptance Tests. I leave it for your test team to work from this example, and map it to their own, particular, work habits.

Notice Pete's use of stakeholders and interests to help identify the test cases, and how his test cases contain specific test values.

USE CASE 35: ORDER GOODS, GENERATE INVOICE (TESTING EXAMPLE)

Context: Customer places order for goods, an invoice is generated and sent out with the ordered items.

Minimal Guarantees

In case of failure, goods will not be allocated to the Customer, Customer account information will remain unchanged, and the transaction attempt will have been logged.

Success Guarantees

Goods will have been allocated to the Customer
Invoice will have been created (Customer Invoicing Rule applies)
Picking list will have been sent to distribution

Main Success Scenario

1.Customer selects items and quantities
2.System allocates required quantities to customer
3.System obtains authenticated invoicing authorization
4.Customer specifies shipping destination
5.System send picking instructions to distribution

Extensions

2a Insufficient stock to meet required quantity for item:
2a1 Customer cancels order
2b Out of stock on item:
2b1 Customer cancels order
3a Customer is bad credit risk (link to acceptance test case for this exception)
4a Invalid shipping destination: ??

ACCEPTANCE TEST CASES

(At least 1 test case is needed for every extension listed above. For complete coverage you need more test cases, to test the data values. The Main Success Scenario test case should come first since it is nice to show how the system works in the high volume case.

1.http://www.cadvision.com/roshi/papers.html

Often this test case can be generated before all of the extension conditions and recovery paths are known.)

Main Success Scenario Tests (Good Credit Risk)

Initial system state/ inputs	Customer Pete, a Good Credit risk, orders 1 of item#1 price $10.00 Quantity on hand for item#1 is 10
Expected system state/ outputs	Quantity on hand for item#1 is 9 Delivery instructions generated Invoice generated for Customer Pete for 1 of Item#1 price $10.00 Transaction logged

Bad Credit Risk

Initial system state/ inputs	Customer Joe, a Bad Credit risk, orders 1 of item#1 price $10.00 Quantity on hand for item#1 is 10
Expected system state/ outputs	Quantity on hand for item#1 is 9 Delivery instructions specify Cash on Delivery Transaction logged

17.6 The Actual Writing

You group will have to form and sort out a set of working habits. In the next section, I give my preferred way of working, a branch-and-join process. In the following section, Andy Kraus describes with wonderful clarity his group's experiences coordinating a large, diverse user group. You should get some useful insights from his report.

A branch-and-join process

A team does two things better as a group: brainstorming, and forming consensus (aligning). They produce more bulk text when they split up. Therefore, my favorite process is to have people work in a full group when it is necessary to align or brainstorm, and spend the rest of their time working in ones or twos. Here is the process, first in outline, then in more detail.

1 Produce a low-precision view of the system's function

* Align on usage narrative (group)

- Align on scope, brainstorm actors and goals (group)
- Write narratives (separately)
- Collect the narratives (group)

2 Produce the high-precision view, the use cases

- Brainstorm the use cases to write (group)
- Align on use case form (group)
- Write the use cases (separately)
- Review the use cases (separately)
- Review the use cases (group)

Stage one - Produce a low-precision view of the system's function

Stage one is done in four rounds.

Round 1.1: Align on what a usage narrative looks like (group). The group spends time together to understand what a narrative is and looks like (review Section 1.6"Warm up with a Usage Narrative" on page 30). Each person writes one, perhaps all on the same story, perhaps not. The group reads and discusses the writing, to generate a common idea of what a decent narrative looks like, its length, what details it does and doesn't contain. This can take a few hours. At the end of this time, the group has a concrete idea of (some part of) what is being built

Round 1.2: Align on scope, brainstorm actors and goals (group). The group spends as much time as needed to work out the overall purpose, scope and primary actors of the system. They create a vision statement, an in/out list, a design scope diagram, the list of primary actors and stakeholders, and the most important initial set of user goals. Each of these items involves the others, so discussing one shifts the understanding of the others. Therefore, the items all get built at the same time. If the group thinks they know what they are going to build, this could take several hours to a day. If the group has not yet thought clearly through it, this can take several days. At the end, there is consensus of what is within scope of the discussion, what is being built, and who the key primary actors are.

Round 1.3: Write narratives (separately). The people split up to write usage narratives for selected functions of the proposed system. They write individually, then trade with a partner or circulate their writing in a small group. They send the results to the entire group.

Round 1.4: Collate the narratives (group). The group gets together to discuss the content (not the writing style) of the narratives. They are addressing the question: "Is this a sample of what we really want to build?" There may be more discussion about the nature of the system, there may be

another writing cycle (or perhaps not), until the people decide they it portrays how they want the system to look.

At this point, the first phase of work is done. The group has a packet that can be distributed to their sponsoring groups. The packet shows a (low precision) draft view of the new system:

* A system vision statement
* A list of what is in and what is out of scope (both function and design scope)
* A drawing of the system in its environment
* A list of the key primary actors
* A list of the stakeholders in the system, their main interests
* A list of the most important user goals
* A set of narratives (each less than half a page long).

Stage two - Produce the high-precision view, the use cases

Stage two is done in five rounds.

Round 2.1: Brainstorm the use cases to write (group). This first round produces a more exact list of use cases to write. We uses facilitated brainstorming techniques to review, brainstorm and list *all* the primary actors the system will encounter in its life. We brainstorm and list *all* the user goals we can imagine, for all primary actors. We may split into subgroups for this activity.

A useful technique for dealing with large, heterogeneous groups is to split into working groups of 3-5 people. Usually, there are several domains or interest groups, whose knowledge needs to be combined. We form working groups with one person from each domain. Each small group contains all the knowledge needed to resolve discussion, and each person can more easily be heard. The small group can move more quickly than the full team. Having the team split into several such groups means that they cover more territory in the same time.

If the full list of primary actors and user goals is constructed using subgroups, they get together again to pool their results. They do a group review to complete and accept the list. At the end of this period, the team has what allegedly will be the full set of user goal use cases to write. They are, of course, almost certain to discover new user goals over time.

The team publishes the list of primary actors and user goals. There may be additional discussion at this point about what development priorities to give them, how to split into releases, estimates of complexity and development time, etc.

Round 2.2: Align on use case form (group). This round starts with the team writing a use case as a group (or individually, then bringing the individual versions to the group). They discuss the level and style of writing, the template, the stakeholders and interests, minimal guarantees, and so on. At the end of this session, they have an initial standard for their writing.

Round 2.3: Write the use cases (separately). Organize groups by specialty, probably 2-4 people per groups. Select use cases for each specialty group and then split up.

Over the next days or weeks, the people write use cases, individually or in pairs (I don't find larger writing partnerships to be effective). They circulate for comment, and improve their drafts within their speciality groups until the writing is "correct". They write the summary use cases. They will almost certainly split some use cases, create subfunction use cases, add some primary actors, some new goals, and so on.

It is useful to have two people associated with each use case, even if one is designated primary writer. Many questions will surface about the rules of the business, what is really the requirement versus what is merely a holdover characteristic from the old days. It helps to have someone to ask about the business rules. The second person can double check that the GUI has not crept in, and that the goals are at the right levels.

Round 2.4: Review the use cases (separately). The writers circulate the drafts, either electronically or on paper. It is interesting to see a peculiar advantage to paper. A paper copy collects everyone's comments, so the writer can make one editing pass through the use case, merging all the suggestions. One team said that when they had tried online suggestions, they found they were doing much more revising, at one moment editing to fit one person's suggestion, then taking that change out to meet another person's suggestion. In all cases, a peer group should check the level of writing and the business rules inside the use cases.

The writers send the use cases for review to both a system developer and an expert on system use. The technical person ensures that the use case contains enough detail for implementation (excluding the data descriptions and UI design). The usage expert makes sure all the requirements are true requirements and that the business really works that way.

Round 2.5: Review the use cases (group). Eventually there is a group review in which software designers, business experts, usage experts, and UI designers are present. What actually happens after the writing teams have created their best draft depends upon the project and its policy of review and review mechanism. The writers need to make sure the steps are understandable, correct, and detailed enough to implement. This may be done by official reviews, unofficial reviews, user reviews, or developer reviews.

A use case reaches its first official baseline once the draft has passed user and technical scrutiny. Start design then, and change the use case only to fix mistakes, not to just change wording.

You will quickly see the difference between drafting a use case and completing it. In completing a use case, the writers must:

* Name all the extension conditions.
* Think through the business policies connected with failure handling.

* Verify that the interests of the stakeholders are protected.
* Verify that the use case names all and only actual requirements,
* Ensure each use case is readable by the users or usage experts, and clear enough that the developers know what to implement.

Time required per use case.

I find that it takes a several hours to draft a use case, and days to chase down the extension handling. One team of 10 people produced 140 use case briefs in a week (2.8 use case briefs per person per day) and then spent the next four weeks working on them and adding the other requirements, making 2 work-weeks per use case plus its associated requirements. A team on a different project spent an average of 3-5 work weeks per use case to get extremely high-quality use cases.

Collecting use cases from large groups

On occasion, you will find yourself working with a large, diverse, non-technical group of usage experts. This is very challenging. I excerpt below from an illuminating report Andy Kraus wrote on his successful experience facilitating use case sessions with up to 25 people of different specialties. From Object Magazine, May, 1996, courtesy of SIGS publication.

ANDY KRAUS: COLLECTING USE CASES FROM A LARGE, DIVERSE LAY GROUP

Don't skimp on conference facilities. you'll be living in them for weeks, and you need to "own" them for the duration of your sessions... We faced significant logistical problems moving from one conference room to another. Try to stay in one room.

You can't elicit the right use cases without the right people in the room. Better too many people and points of view than not enough... The people whose ideas and experience we needed were the real life analysts, officers, detectives, data entry operators, and their supervisors from the twenty-three departments and numerous "Ex Officio" members. Without knowing in advance what the *real* system actors were, it was impossible for us to predict which users would need to come to which sessions, a real problem when trying to get a piece of so many people's time. We solved the problem by staging a "Use case Kick-Off" with representatives from all the user areas at which we jointly determined a tentative schedule for future sessions.

Large groups are more difficult to facilitate than small ones. You'll have to cope as best you can. Above all, be organized. As things actually evolved we found ourselves forced to conduct sessions with groups ranging in size from eight to twenty-five people with all the problems we had been told to expect that could derive from working with such large groups. Only by having representation from the diverse membership in the room at the same time could we flush out nuances to the requirements caused by the differing philosophies and operating procedures among the members.

Don't spend more than half of each work day in session with the users. Our sessions taught us that we had been too ambitious in our estimates of the amount of material that could be covered in a session and or our ability to process that material before and after the sessions. It will take you every bit as long to process the raw material gained in the sessions as it did to elicit it. You'll have plenty of housekeeping chores, administrative work, planning and preparation for the next session to do in that half day.

Get management into the boat with you. -- The Administrator, the Project Manager were all present during various parts of the elicitation process.

Those responsible for architecting the system should be present during use case elicitation. -- The Architect brought expertise in the development process... A "SME" (Subject Matter Expert) provided the domain expertise to jump start the process and keep it on track once it's moving.

You've got a better chance of attaining your goal if you get people who support the application being replaced into the sessions with you. -- A number of participants from organization, and DIS, the Department of Information Services, participated in the sessions. They provided insights into the emerging requirements, particularly with respect to the external interfaces, as well as with the history of the current system.

There's no substitute for getting the "True" users involved in use case solicitation. We were able to secure the participation of actual officers, analysts, investigators, data entry Operators, and Supervisors for those groups of people, in the sessions.

Use a scribe. The importance of fast, accurate scribes cannot be over emphasized... We had several scribes working the early sessions who proved invaluable...

Displaying the cumulative use case output can facilitate the process. The use cases were developed interactively, recorded on flip chart paper by a facilitator and in a word processor by the scribe. The flip charts were then posted to the conference room walls. Unwieldy as it was to deal with use cases on flip charts, we discovered some unexpected benefits accrued from posting the use cases onto the conference room walls. We were unintentionally sloppy in hanging the use cases not completely in sequence. As new use cases were developed, this lack of sequencing forced participants to scan the previously developed use case. Such iteration seemed to have the effect of helping people become more familiar with existing use cases, developing new ones that were "like another use case", as well as helping participants to develop new ones faster.

A job title may not an *actor* make. It was extremely difficult for our users to accept the notion that an actor *role* could be different than a job title. At the end of a difficult day's discussion, we had been able to derive a tentative actor list, but people did not seem comfortable with it. How could we help them be more comfortable?

The application doesn't care who you are; it care. about the hat you wear. Struck by the notion that playing an actor *role* with a computer system is similar to "wearing a hat", we bought a set of baseball caps and embroidered the actor names, one per cap. The next day, when the participants arrived at the session, the caps were arrayed in a row on the facilitator's table at the front of the room. And as soon as use case elicitation began, the

facilitator took one of the hats, "Query(or)", and put it on his head. The results were very gratifying. We had been able to help the users understand that no matter what their job title, when they were using the system, they had to wear a certain "hat" (play a certain role), when doing so.

Expect to miss some actors in the initial formulation of actors. ...It wasn't until several weeks into the elicitation process that we (facilitators) realized that there appeared to be some use cases missing; use cases dealing with the supervision of users of the system. Despite all our efforts to assure a broad range of participation, no supervisory personnel had been part of the sessions, and interestingly enough, the people that worked for them did not conceive of any supervisory use cases.

"Daily work use cases" can facilitate use case brainstorming. [*Narratives* by another name - Alistair] ...our users seemed to lack a "context" for the use cases. We decided to have them write (at a very high level) the steps they followed in their daily activities to perform some work task, e.g., making a stop, At some point in the making of that step the system would be used, and this is the way we were able to free them to think of "uses of the system". A day spent developing these daily work use cases on day three yielded twenty use cases on day four.

Don't expect "use cases by the pound". Like any creative activity use case elicitation has its peaks and valleys. Trying to rush people in the creative process is counter productive.

Expect to get stuck; react with catalysts. "prompting". i.e., the use of overheads and handouts with topic and/or issue bullets related to the system uses under discussion proved to be effective catalysts. Intentionally we sometimes introduced controversial topics and view points as discussion generators. We found that people, when confronted by a viewpoint they could not support, would be able to express their own viewpoints more quickly and clearly.

Eliciting use cases is a social activity. Feelings were hurt, ideas bashed, participants sided against the facilitator only to defend him later in the same session. A few after-hours mixers served as social lubricants and kept us all friends. Ultimately, we all bonded, having come to respect and support each other in the task of bringing the ideas from the use case sessions to the project's decision makers.

Standard "descriptors" help facilitate the process...Standard descriptors held attributes for the new system divided along certain lines, e.g., People, Places, Locations. The descriptor sets provided a pathway to a consistent presentation of information... The sets were named, cataloged and evolved to allow us to use them generically in session discussions as well as subsequent use case refinement, Similarly, standard System Responsibilities, Success and Failure Scenarios allowed us to focus on the exceptions rather than redundantly copying from one use case to another,

Build, maintain and display an assumptions list. During certain periods of the work we found it necessary to start sessions with a "reading of the assumptions". That reading tended to minimize arguments over points already considered.

Be a minimalist. Keep your use case template as slim as possible.

18. Use Cases Briefs and eXtremeProgramming

The ultralight methodology, Extreme Programming, or XP, uses an even lighter form of behavioral requirements than the ones I show in this book (see Extreme Programming Explained, by Kent Beck, Addison-Wesley, 1999). In XP, the usage and business experts sit right with the developers. Since the experts are right there, the team does not write down detailed requirements for the software, but writes *user stories* as a sort of promissory note to have a further discussion about the requirements around a small piece of functionality.

An XP user story, in its brevity, may look either like the *use case briefs* described in “A sample of use case briefs” on page 47, or a system *feature* as described in “Feature list for Capture Trade-in” on page 170.

Each XP user story needs to be just detailed enough that both the business and technical people understand what it means and can estimate how long it will take. It must be a small enough piece of work that the developers can design, code, test and deploy it in three weeks or less. Once meeting those criteria, it can be as brief and casual as the team can get manage. It often gets written just on an index card.

When the time comes to start working on a user story, the designer simply takes the card over to the business expert and asks for more explanation. Since the business expert is always available, the conversation continues, as needed, until the functionality is shipped.

On rare occasion, a small, well-knit development team with full-time users on board will take usage narratives or the use case briefs as their requirements. This only happens when the people who own the requirements sit very close to the people designing the system. The designers collaborate directly with the requirements owners during design of the system. Just as with XP’s user stories, this can work *if* the conditions for being able to fulfill on the promissory note are met. In most projects they are not met, and so it is best to keep the usage narrative acts as a warm-up exercise at the start of a use case writing session, and the use case brief as part of the project overview.

19. Mistakes Fixed

The most common mistakes in writing are leaving out the subjects of sentences, making assumptions about the user interface design, and using goal levels that are too low. Here are some examples. The purpose of this section is not to quiz you, but to sharpen your visual reflexes.

The first examples are short; the last one is a long example from a real project. Practice on the small ones, then tackle the last one.

19.1 No system

Before:

Use Case: *Withdraw Cash*
Scope: ATM
Level: User goal
Primary Actor: Customer
1. Customer enters card and PIN.
2. Customer enters "Withdrawal" and amount.
3. Customer takes cash, card and receipt.
4. Customer leaves.

Working notes:

This use case shows everything the primary actor does, but does not show the system's behavior. It is surprising how often people write this sort of use case. The response of the reviewer is, "I see that the system doesn't actually have to do anything. We can sure design that in a hurry."

The fix is exactly as before: name all the actors with their actions.

After:

You should be able to write an ATM use case in your sleep by now.

Use Case: *Withdraw Cash*
Scope: ATM
Level: User goal
Primary Actor: Account Holder
1. Customer runs ATM card through the card reader.
2. ATM reads the bank id, account number, encrypted PIN from the card, validates the bank id and account number with the main banking system.
3. Customer enters PIN. The ATM validates it against the encrypted PIN read from the card.
4. Customer selects FASTCASH and withdrawal amount, a multiple of $5.

5. ATM notifies main banking system of customer account, amount being withdrawn, and receives back acknowledgement plus the new balance.
6. ATM delivers the cash, card and a receipt showing the new balance.
7. ATM logs the transaction.

19.2 No primary actor

Before:

Here is a fragment of use case for withdrawing money from an ATM:

Use Case: *Withdraw Cash*
Scope: ATM
Level: User goal
Primary Actor: Customer
1. Collects ATM card, PIN.
2. Collects transaction type as "Withdrawal"
3. Collects amount desired.
4. Validates that account has sufficient funds.
5. Dispenses money, receipt, card
6. Resets

Working notes:

This use case is written strictly from the system’s viewpoint It shows everything the ATM does, but does not show the primary actor's behavior. This sort of writing is hard to understand, verify and correct. In some cases, critical information is omitted about the actor's behavior, often having to do with sequencing.

The fix is straightforward. Name every actor and action.

After:

Same as before.

19.3 Too many user interface details

Before:

Use Case: *Buy Something*
Scope: Purchasing application
Level: User goal
Primary Actor: Customer
1. System presents ID and Password screen.
2. Customer types id and password into system, clicks OK.
3. System validates user id and password, displays Personal Information Screen
4. Customer types in first and last names, street address, city, state, zip code, phone number, and click OK.
5. System validates that user is a known user.
6. System presents available product list.
7. Customer clicks on pictures of items to be purchased, types in quantity next to each, clicks on DONE when finished.
8. System validates with the warehouse storage system that sufficient quantity of the requested product is in stock.
...etc.

Working notes:

This mistake is perhaps the most common one. The writer describes much too much about the user interface. It is not really a requirements document, but a user manual. The extra UI detail adds nothing to the story, but clutter the reading and make the requirements brittle.

The fix is to find a way to describe the intentions of the user without actually nominating a specific solution. This sometimes takes a little creativity with the wording.

After:

Use Case: *Buy Something*
Scope: Purchasing application
Level: User goal
Primary Actor: Customer
1. Customer accesses system with id and password.
2. System validates user.
3. Customer provides name, address, telephone number.
4. System validates that Customer is a known Customer.
4. Customer selects products and quantity.
5. System validates with the warehouse storage system that sufficient quantity of the requested product is in stock.
... etc.

19.4 Very low goal levels

Before:

Use Case: *Buy Something*
Scope: Purchasing application
Level: User goal
Primary Actor: Customer user
1. User accesses system with id and password.
2. System validates user.
3. User provides name.
4. User provides address.
5. User provides telephone number.
6. User selects product
7. User identifies quantity.
7. System validates that user is a known customer.
8. System opens a connection to warehouse system.
9. System requests current stock levels from warehouse system.
10. Warehouse storage system returns current stock levels.
11. System validates that requested quantity is in stock.
... etc.

Working notes:

It is clear that this is going to be a long and dull use case. We can't criticize the steps on the grounds that they describe the user interface too closely, but we definitely want to shorten the writing and make it clearer what is going on.

To shorten the text:

- Merge the data items (steps 3-5). The writer uses separate steps to collect each data item. If we ask, "What is the user trying to provide, in general?", we get, "personal information," a good nickname for all of the pieces of information that will be collected about the person. I find the nickname alone to be too vague, so I'll hint at the field list. That field list will be expanded elsewhere without affecting this use case.
- Put all the information going in the same direction into one step (steps 3-7). This is not always the best thing to do. Sometimes "providing personal information" is considerably different from "selecting product and quantity", so that the writer prefers to place them on separate lines. This is a matter of taste I like collecting all the information going in one direction. If it looks too cluttered, or it the extensions need them separated, then I separate them again.
- Look for a slightly higher-level goal (steps 8-11). Asking, "Why is the system doing all these things in steps 8-11?", I get, "It is trying to validate with the warehouse storage system that

sufficient quantity of the requested product is in stock." That slightly higher level goal captures the requirements as clearly as before and is much shorter

After:

Use Case: *Buy Something*
Scope: Purchasing application
Level: User goal
Primary Actor: Customer user
1. User accesses system with id and password.
2. System validates user.
3. User provides personal information (name, address, telephone number), selects product and quantity.
4. System validates that Customer is a known Customer.
5. System validates with the warehouse storage system that sufficient quantity of the requested product is in stock.
... etc.

19.5 Purpose and content not aligned

This is a reminder for you to do Exercise 29“Fix faulty ’Login’” on page 102.

In case you haven’t done it yet, please do, and look for three sorts of mistakes.

* The body of the use case does not match the intent described in the name and description. In fact, it is at least two use cases rolled together.
* It describes user interface details.
* It uses programming constructs in the text, rather than the language of ordinary people.

If you are determined to avoid doing the work, turn to Answers: “Exercise 37 on page 128:” on page 241 to see the discussion and resolution.

19.6 Advanced example of too much UI

FirePond Corporation kindly let me use the following *before* and *after* example. The *before* version covered 8 pages, 6 of which were used for main success scenario and alternatives. The *after* version is a third as long, holding the same basic information, but without constraining the user interface.

Read the main success scenario in detail, and ask yourself how you might make this long use case more palatable without losing content? Notice, particularly, the UI design that shows up in the text. You should look at some of the extensions, but it is not important that you read all of them in

detail. I removed some of the extensions, but left enough bulk so you can appreciate the difficulty of working with such a long use case. How would *you* shorten it?

A note about the title of the use case. In the language of information technology marketing, it is considered insufficient to say that a shopper merely "selects a product." Faced with a complex product set, the shopper "researches a solution" to their "situation". I might like to retitle this use case *Select a Product*, but it is not my place to do so. This title is considered correct in the world in which it was written and read. So the title stays *Research a Solution*.

Thanks to Dave Scott and Russell Walters of FirePond Corporation.

Before:

USE CASE 36:RESEARCH A SOLUTION - BEFORE

Scope: Our web system
Level: User goal
Primary actor: Shopper - a consumer or agent wanting to research products they may purchase.
Main success scenario:

Actor Action	System Response
1. This use case begins when the Shopper visits the e-commerce web-site.	2. The system may receive information about the type of Shopper visiting the web-site. 3. System will require establishing the identity of the Shopper, *Initiate Establish Identity.* If the system doesn't establish the identity here, it must be established prior to saving a solution. 4. The system will provide the Shopper with the following options: Create a new solution, recall a saved solution
5. Shopper selects create a new solution.	6. System will present the first question to begin determining the Shopper's needs and interests.
7. Shopper can repeat adding Product Selections to shopping cart: 8. While questions exist to determine needs and interest: 9. Shopper will answer questions	10. System will prompt with a varying number and type of questions based on previous answers to determine the Shopper's needs and interests along with presenting pertinent information such as production information, features & benefits, and comparison information.

11. Shopper answers last question	12. At the last question about needs and interest, the system will present product line recommendations and pertinent information such as production information, features & benefits, comparison information, and pricing.
13. Shopper will select a product line	14. System will present the first question to begin determining the product model needs.
15. While questions exist to determine Product Model recommendations: 16. Shopper will answer questions	17. System will prompt with questions that vary based on previous answers to determine the Shopper's needs and interests related to product models, along with pertinent information such as production information, features & benefits, comparison information, and pricing.
18. Shopper answers last question	19. At the last question about product model needs, the system will present product model recommendations and pertinent information such as production information, features & benefits, comparison information, and pricing.
20. Shopper will select a product model	21. System will determine standard product model options, and then present the first question about determining major product options.
22. While questions exist to determine Product Option recommendations: 23. Shopper will answer questions	24. System will prompt with questions that vary based on previous answers to determine the Shopper's needs and interests related to major product options, along with pertinent information such as production information, features & benefits, comparison information, and pricing.
25. Shopper answers last question	26. At the last question about major product option desires, the system will present the selected model and selected options for Shopper validation.
27. Shopper reviews their product selection, determines they like it, and chooses to add the product selection to their shopping cart.	28. System will add product selection and storyboard information (navigation and answers) to the shopping cart. 29. The system presents a view of the shopping cart and all of the product selections within it.
30. End repeat steps of adding to shopping cart.	31.

32. Shopper will request a personalized proposal on the items in their shopping cart.	33. System will present the first question to begin determining what content should be used in the proposal.
34. While questions exist to determine proposal content: 35. Shopper will answer questions	36. System will prompt with questions that vary based on previous answers to determine the proposal content, along with pertinent information such as production information, features & benefits, comparison information, and pricing.
37. Shopper answers the last question	38. At the last question about proposal content, the system will generate and present the proposal.
39. Shopper will review the proposal and choose to print it.	40. System will print the proposal.
41. Shopper will request to save their solution.	42. If the Shopper identity hasn't been established yet, *initiate Establish Identity* 43. System will prompt the user for solution identification information.
44. Shopper will enter solution identification information and save the solution.	45. System will save the solution and associate with the Shopper.

Extensions:

*a.At any point during the Research Solution process, if the Shopper hasn't had any activity by a pre-determined time-out period, the system will prompt the Shopper about no activity, and request whether they want to continue. If the Shopper doesn't respond to the continue request within a reasonable amount of time (30 seconds) the use case ends, otherwise the Shopper will continue through the process.

*b. At any point during the question/answer sequences, the Shopper can select any question to go back to, and modify their answer, and continue through the sequence.

*c. At any point after a product recommendation has been presented the Shopper can view performance calculation information as it pertains to their needs. The system will perform the calculation and present the information. The Shopper will continue with the Research Solution process from where they left off.

*d. At any point during the question/answer sequences, the system may interface with a Parts Inventory System to retrieve part(s) availability and/or a Process & Planning System to retrieve the build schedule. The parts availability and schedule information can be utilized to filter what product selection information is shown, or be used to show availability to the Shopper during the research solution process. Initiate Retrieve Part Availability, and Retrieve Build Schedule use cases.

*e. At any point during the question/answer sequences, the system is presenting pertinent information, of which industry related links are a part. The Shopper selects the related link. The system may pass product related information or other solution information to this link to drive to

the best location or to present the appropriate content. The Shopper when finished at the industry web-site, will return to the point at which they left, possibly returning product requirements that the system will validate. *Initiate View Product Information*

*f. At any point during the research process, the Shopper may request to be contacted:

Initialize Request For Contact Use Case.

*g. At any point during the question/answer sequences, the system may have established capture market data trigger points, in which the system will capture navigational, product selection, and questions & answers to be utilized for market analysis externally from this system.

*h. At any pre-determined point in the research process, the system may generate a lead providing the solution information captured up to that point. Initialize Generate Lead Use Case.

*i. At any point the Shopper can choose to exit the e-commerce application:

If a new solution has been created or the current one has been changed since last saved, the system will prompt the Shopper if they want to save the solution. *Initiate Save Solution*

*j. At any point after a new solution has been created or the current has been changed, the Shopper can request to save the solution. *Initiate Save Solution*

1a. A Shopper has been visiting a related-product vendor's web-site and has established product requirements. The vendor's web-site allows launching to this e-commerce system to further research a solution:

1a1. Shopper launches to e-commerce system with product requirements, and possibly identification information.

1a2. System receives the product requirements, and potentially user identification.

1a3. System will validate where the Shopper it at in the research process and establish a starting point of questions to continue with the research process.

1a4. Based on established starting point, we may continue at step 5, 12, or 18.

3a. Shopper wants to work with a previously saved solution:

3a1. Shopper selects to recall a solution

3a2. System presents a list of saved solutions for this Shopper

3a3. Shopper selects the solution they wish to recall

3a4. System recalls the selected solution.

3a5. Continue at step 26

23a. Shopper wants to change some of the recommended options:

{create a select options use case because there are alternatives to the normal flow: System maybe setup to show all options, even in-compatible ones, if the Shopper selects an in-compatible one, the system will present a message and possibly how to get the product configured so the option is compatible.}

...while Shopper needs to change options:

23a1. Shopper selects an option they want to change

23a2. System presents the compatible options available

23a3. Shopper selects desired option

26a. Shopper wants to change quantity of product selections in shopping cart:

26a1. Shopper selects a product selection in the shopping cart and modifies the quantity.

26a2. System will re-calculate the price taking into consideration discounts, taxes, fees, and special pricing calculations based on the Shopper and their related Shopper information, along with their answers to questions.

26b. Shopper wants to add a product trade-in to the shopping cart:

26b1. *see section Trade-In*
26c. Shopper wants to recall a saved solution:
26c1. System presents a list of saved solutions for this Shopper
26c2. Shopper selects the solution they wish to recall
26c3. System recalls the selected solution.
26c4. Continue at step 26
26d. Shopper wants to finance products in the shopping cart with available Finance Plans:
26d1. Shopper chooses to finance products in the shopping cart
26d2. System will present a series of questions that are dependent on previous answers to determine finance plan recommendations.
System interfaces with Finance System to obtain credit rating approval. *Initiate Obtain Finance Rating.*
26d3. Shopper will select a finance plan
26d4. System will present a series of questions based on previous answers to determine details of the selected finance plan.
26d5. Shopper will view financial plan details and chooses to go with the plan.
26d6. System will place the finance plan order with the Finance System *initiate Place Finance order.*

Working notes:

The writers had selected the two-column format to separate the actors' actions, as described by Wirfs-Brock, Constantine and Lockwood. They did not visualize any other user interface design than the question-and-answer model, so they described questions and answers in the use case.

My first action was to get rid of the columns and create a simple story in one-column format. I wanted to see the storyline easily and without turning pages.

Looking at the result, I hunted for assumptions about the user interface design, and for goals that could be raised a level. The key is in the phrase, "System will prompt with a varying number and type of questions based on previous answers." While this does not mention anything so obvious as mouse clicks, it does assume a user interface based on the user typing answers to questions. I wanted to imagine a completely different user interface to see how that might affect the writing, so I conjured up a design in which there would be no typing at all. The user would only click on pictures. I was then able to capture the user's intent and remove UI dependency. As you will see, this also shortened the writing enormously.

I rechecked the goal level in each statement, to see if perhaps the goal of the step was too low level, and could be raised.

One of the open questions at the start of this work was whether the one-column format would still show the system's responsibility clearly to the designers. Rusty's evaluation was that it does. In fact, he was happier because:

* It is shorter and easier to read.

* All the real requirements are still there, clearly marked.
* The design is less constrained.

After:

USE CASE 37:RESEARCH POSSIBLE SOLUTIONS - AFTER

Scope: web software system
Level: User goal
Preconditions: none
Minimal Guarantee: no action, no purchases
Success Guarantee: Shopper has zero or more product(s) ready to purchase, the system has a log of the product selection(s), navigation moves and characteristics of the shopper are noted.
Primary Actor: Shopper (any arbitrary web surfer)
Trigger: Shopper selects to research a solution.

Main Success Scenario

1. Shopper initiates the search for a new solution.
2. The **system** helps the shopper select a product line, model and model options, presenting information to the shopper, and asking a series of questions to determine the shopper's needs and interests. The series of questions and the screens presented depend on the answers the shopper gives along the way. The system chooses them according to programmed selection paths, in order to highlight and recommend what will be of probable interest to the shopper. The presented information contains production information, features, benefits, comparison information, etc.
2. The **system** logs the navigation information along the way.
3. Shopper selects a final product configuration.
4. Shopper adds it to the shopping cart.
5. The **system** adds to the shopping cart the selected product, and the navigation information.
6. The **system** presents a view of the shopping cart and all the products in it.

The shopper repeats the above steps as many times as desired, to navigate to, and select, different tailored products, adding them to the shopping cart as desired.

Extensions

*a. At any time, the shopper can request contact.
1a. Due to an agreement between this web site owner and the owner of the sending computer, the sending computer may include information about the type of Shopper along with the request:
1a.1. **System** extracts from the web request any and all user and navigation information, adds it to the logged information, and starts from some advanced point in the question/answer series.
1a.1a. Information coming from the other site is invalid or incomprehensible:
System does the best it can, logs all the incoming information, continues as it can.
1b. Shopper wants to continue a previous, saved, partial solution:

1b.1. Shopper establishes identity and saved solution.

1b.2. The **system** recalls the solution and returns the shopper to where shopper left the system upon saving the solution.

2a. Shopper chooses to bypass any or all of the question series:

2a.1. Shopper is presented and selects from product recommendations based upon limited (or no) personal characteristics.

2a.2. The **system** logs that choice.

2b. *At any time prior to adding the product to the cart*, the shopper can go back and modify any previous answer for new product recommendations and/or product selection.

2c. *At any time prior to adding the product to the cart*, the shopper can ask to view an available test/performance calculation (e.g., can this configuration tow a trailer with this weight):

System carries out the calculation and presents the answer.

2d. The shopper passes a point that the web site owner has predetermined to generate sales leads (dynamic business rule):

System generates sales lead.

2e. System has been setup to require the Shopper to identify themselves:

Shopper establishes identity

2f. System is setup to interact with known other systems (parts inventory, process & planning) that will affect product availability and selection:

2f.1. **System** interacts with known other systems (parts inventory, process & planning) to get the needed information. (Retrieve Part Availability, Retrieve Build Schedule).

2f.2. **System** uses the results to filter or show availability of product and/or options(parts).

2g. Shopper was presented and selects a link to an Industry related web-site:

Shopper views other web-site.

2h. System is setup to interact with known Customer Information System:

2h.1. **System** retrieves customer information from Customer Information System

2h.2. **System** uses results to start from some advanced point in the question/answer series.

2i. Shopper wants to know finance credit rating because it will influence the product selection process:

Shopper obtains finance credit rating.

2j. Shopper indicates they have purchased product before and the system is setup to interact with a known Financial Accounting System:

2j.1. The **System** retrieves billing history.

2j.2. The **System** utilizes the shopper's billing history to influence the product selection process.

3a. Shopper decides to change some of the recommended options:

System allows the Shopper to change as many options as they wish, presenting valid ones along the way, and warning of incompatibilities of others.

4a. Shopper decides not to add the product to the shopping cart:

System retains the navigation information for later.

6a. Shopper wants to change the contents of the shopping cart:

System permits shopper to change quantities, remove items, or go back to an earlier point in the selection process.

6b. The shopper asks to save the contents of the shopping cart:
6b.1. The **system** prompts the shopper for name and id to save it under, and saves it.
6b.1a. The Shopper's identity has not been determined:
Shopper establishes identity
6c. The shopper has a trade-in to offer
System captures trade-In.
6d. Shopper decides to finance items in shopping cart:
Shopper obtains finance plan
6e. Shopper leaves the E-Commerce System when shopping cart has not been saved:
6e.1. **System** prompts the shopper for name and id to save it under, saves it.
6e.1.a Shopper decides to exit without saving:
System logs navigational information and ends the shopper's session.
6f. Shopper selects an item in shopping cart and wishes to see availability of matching product (new or used) from inventory:
6f.1. Shopper requests to locate matching product from inventory.
6f.2. Shopper exchanges item in shopping cart with matching product from inventory.
6f.2a. Shopper doesn't want the matching inventory item:
System leaves the original shopping cart item the shopper was interested in matching against inventory alone.
6f.3. **System** ensures there is one item in shopping cart configured to inventory item.

Calling Use Cases: Shop over the web, Sell Related Product
Open Issues
Extension 2c. What questions are legal? What other systems are hooked in? What are the requirements for the interfaces/

Chapter 19. Mistakes Fixed
Advanced example of too much UI - Page 198

PART 3
REMINDERS FOR THE BUSY

20. Each Use Case

Reminder 1. A use case is a prose essay

Recall from the preface, *"writing use cases is fundamentally an exercise in prose essays, with all the difficulties in articulating good that comes with prose writing in general."*

Russell Walters of Firepond Corporation wrote,

> I think the above statement clearly nails the problem right on. This is the most misunderstood problem, and probably the biggest enlightenment for the practicing use case writer. However, I'm not sure the practitioner can come to this enlightenment on their own, well, at least until this book is published. :-) I did not understand this as the fundamental problem, and I had been working with the concept of use cases for 4 years, until I had the opportunity to work alongside you. And even then, it wasn't until I had a chance to analyze and review the *before* and *after* versions of the use case you assisted with re-writing [Use Case 36: on page 190] when the light bulb came on. Four-plus years is long time to wait for this enlightenment! So, if there is only one thing the readers of this book walk away understanding, I hope it is the realization of the fundamental problem with writing effective use cases.

Use this reminder from Rusty to help keep your eyes on the text, not the diagrams, and be aware of the writing styles you will encounter.

Reminder 2. Make the use case easy to read.

You want your requirements document short, clear, easy to read.

I sometimes feel like an 8th grade English teacher walking around, saying,

> "Use an active verb in the present tense. Don't use the passive voice, use the active voice. Where's the subject of the sentence? Say what is really a requirement, don't mention it if it is not a requirement."

Those are the things that make your requirements document short, clear, and easy to read. Here are a few habits to build to make your use cases short, clear, and easy to read:

1 Keep matters short and too the point. Long use cases make for long requirements, which few people enjoy reading.

2 Start from the top and create a coherent story line. The top will be a strategic use case. The user goal, and eventually, subfunction-level use cases branch off from here.

3 Name the use cases with short verb phrases that announce the goal to be achieved.

4 Start from the trigger, continue until the goal is delivered or abandoned, and the system has done

any bookkeeping it needs to, with respect to the transaction.

5 Write full sentences with active verb phrases that describe the subgoals getting completed.

6 Make sure the actor is visible in each step.

7 Make the failure conditions stand out, and their recovery actions readable. Let it be clear what happens next, preferably without having to name step numbers.

8 Put alternative behaviors in the extensions, rather than in *if* statements in the main body.

9 Create extension use cases only under very selected circumstances.

Reminder 3. Just one sentence form

There is only one form of sentence used in writing action steps in the use case:

* a sentence in the present tense,
* with an active verb in the active voice, describing
* an actor successfully achieving a goal that moves the process forward.

Examples are,

"Customer enters card and PIN."
"System validates that customer is authorized and has sufficient funds."
"PAF intercepts responses from the web site, and updates the user's portfolio."
"Clerk *finds a loss* using search details for "loss"."

Use this sentence form in business use cases, system use cases, summary, user, subfunction use cases, with the fully dressed or the casual template. It is the same in the main success scenario and in the extension scenario fragments. Master this sentence style.

It is useful to have a different grammatical form for condition part of an extension, so it doesn't get confused with the action steps. Use a sentence fragment (possibly a full sentence, preferably (but not always) in the past tense). End the condition with a colon ('**:**') instead of period.

"Time-out waiting for PIN entry:"
"Bad password:"
"File not found:"
"User exited in the middle:"
"Data submitted is not complete:"

Reminder 4. Include sub use cases

What you would do quite naturally if no one told you to do otherwise, is to write a step that calls out the name of a lower-level goal or use case, as in:

"Clerk *finds a loss* using search details for "loss"."

In the terms of the Unified Modeling Language, the calling use case just *included* the sub use case. It is so much the most obvious thing to do, that it would not even deserve mention if there weren't writers and teachers encouraging people to use the UML *extends* and *specializes* relations (for my views, see “Appendix A: Use Cases in UML”).

As a first rule of thumb, always use the *includes* relation between use cases. People who follow this rule report that they and their readers simply have less confusion with their writing than people who mix *includes* with *extends* and *specializes*. For the other occasions, see“When to use extension use cases” on page 118.

Reminder 5. Who has the ball?

Sometimes people write in the passive voice or from the point of view of the system itself, looking out at the world. This produces sentences like: "Credit limit gets entered." This sentence doesn’t mention who it is that enters the credit limit.

Write from the point of view of a bird up above, watching and recording the scene. Or write in the form of a play, announcing which actor is about to act. Or pretend for a moment that you are describing a soccer game, in which actor1 has the ball, dribbles it, then kicks it to actor2. Actor2 passes it to actor3, and so on.

Let the first or second word in the sentence be the name of the actor who owns the action. Whatever happens, make sure it is always clear who has the ball.

Reminder 6. Get the goal level right

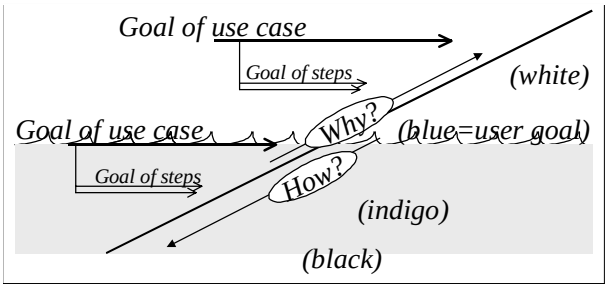
- Review 5.5“Finding the right goal level” on page 75 for the full discussion.
- Make sure the use case is correctly labeled with its goal level: summary, user, or subfunction.
- Periodically review to make sure you know where "sea level" is for your goals, and how far below (or above) sea level the steps are. Recall the tests for sea level goals:
  - It is done by one person, in one place, at one time (2-20 minutes).
  - The actor can go away happily as soon as this goal is completed.
  - The actor (if an employee) can ask for a raise after doing many of these.
- Recall that most use cases have 3-9 steps in the main success scenario, and that the goal level of a step is typically just below the goal level of the use case. If you have more than 9 steps, look for steps to merge in
  - places where the same actor has the ball for several steps in a row, and
  - places where the user’s movements are described. They are typically user interface movements, violating Guideline 5:“It shows the actor’s intent, not movements.” on

page 96

* places where there is a lot of simple back-and-forth between two actors. Ask if they aren't really trying to accomplish something one level higher with all that back-and-forth.

- Ask, "why is the user/actor doing this action?" The answer you get is the next higher goal level. You may be able to use this goal to merge several steps. Review the diagram below to see how goal of steps fit within goals of use cases at different levels.

(Recap of Figure 15."Ask "why" to shift levels" on page 76.)



Reminder 7. Keep the GUI out

Verify that the step you just wrote captures the real intent of the actor, not the just movements in manipulating the user interface. This advice applies when you are writing functional requirements, since, clearly, you can write use cases to document the user interface itself.

In a requirements document, describing the movements of the user in manipulating the interface has three drawbacks:

* the document is needlessly longer;
* the requirements become brittle, meaning that small changes in the user interface design cause a change in the "requirements" (which weren't requirements after all);
* it steals the work of the UI designer, whom you should trust will do a good job.

Most of the use cases in this book should serve as good examples for you. I select this extract of Use Case 20:"Evaluate Work Comp Claim" on page 79 as a representative example:

1. Adjuster reviews and evaluates the reports, ...
2. Adjuster rates the disability using ...
3. Adjuster sums the disability owed, ...
4. Adjuster determines the final settlement range.

I select this as an example of what not to do:

2. The system displays the Login screen with fields for username and password.
3. The user enters username and password and clicks 'OK'.
4. The system verifies name and password.

5. The system displays the Main screen, containing function choices.
6. The user selects a function and clicks 'OK'.

It is very easy to slip into describing the user interface movements, so be on your guard.

Reminder 8. Two endings

Every use case has two possible endings, success and failure.

When an action step calls a sub use case, bear in mind that the called use case could succeed or fail. If the call is in the main success scenario, then the failure is handled in an extension. If it is called from an extension, describe both success and failure handling in the same extension (see, for example, Use Case 22:"Register Loss" on page 83).

You actually have two responsibilities with respect to goal success and failure. The first is to make sure that you deal with failure of every called use case. The second is to make sure that your use case satisfies the interests of every stakeholder, particularly in case the goal fails.

Reminder 9. Stakeholders need guarantees

A use case does *not* only record the publicly visible interactions between the primary actor and the system. If it did only that, it would not make acceptable behavioral requirements. It would only document the user interface.

The system enforces a contractual agreement between stakeholders, one of whom is the primary actor, the others of whom are not there to protect themselves. The use case describes how the system protects all their interests under different circumstances, with the user driving the scenario. It describes the guarantees the system makes to them.

Take the time to name the stakeholders and their interests in each use case. You should find 2-5 stakeholders: the primary actor, the owner of the company, perhaps a regulatory agency, and perhaps someone else. Perhaps it is the testing or maintenance staff who has an interest in the operation of the use case.

Usually, the stakeholders are the same for most use cases, and usually their interests are very much the same across use cases. It soon takes little effort to list their names and interests. Typically, these are the sorts of interests:

* The primary actor's interest is the use case name, it usually is to get something.
* The company's interest is usually to ensure that they don't away with something for free, or that they pay for what they get.
* The regulatory agency's interest usually is to make sure that the company can demonstrate that they followed guidelines, usually that some sort of a log is kept.
* One of the stakeholders typically has an interest in being able to recover from failure in the middle of the transaction, i.e., more logging sorts of interests.

See that the main success scenario and extensions address the stakeholders' interests. This takes very little effort. Read the text of the use case, starting with the main success scenario, and see whether those interests are present. You will be surprised by how often one is missing. Very often, the writer has not thought about the fact that a failure can occur in the middle of the main success scenario, leaving no log or recovery information. Check that all failure handling protects *all* the interests of all the stakeholders.

Often, a new extension condition reveals a missing validation in the main success scenario. On occasion, there are so many validations being performed that the writer moves the set of checks out into a separate writing area, perhaps creating a *business rules* section.

Pete McBreen wrote me on the first time his group listed the stakeholders' interests. They chose a system they had already delivered. They discovered, in that list, all the change requests for the first year of operation of their software. They had successfully built and delivered the system without satisfying certain needs of certain stakeholders. The stakeholders figured this out, of course, and so the change requests came in. What excited this team was that, had they written down the stakeholders and interests early on, they could have avoided (some number of) those change requests. As a result, Pete is a strong advocate of capturing the stakeholders' interests in the use cases. Performing this check takes very little time, and is very revealing for the time spent.

The guarantees section of the template documents how the use case satisfied these interests. You might skimp on writing the guarantees on a less critical, low ceremony project on which the team has good personal communications. You will pay more attention to documenting the guarantees on more critical projects, where potential for damage or cost of misunderstanding is higher. However, in both cases, your team should at least go through the mental exercise of checking both exits of the use case, the success and failure exits.

It is a good strategy to write the guarantees *before* writing the main success scenario, because then you will think of the necessary validations on the first pass, instead of discovering them later and having to go back and change the text.

Read 2.2"Contract between Stakeholders with Interests" on page 40 and 6.2"Minimal Guarantees" on page 89 for more details on these topics.

Reminder 10. Preconditions

The preconditions section of the use case declares its valid operating conditions. The precondition must be something the system can ensure will be true. You document the preconditions because you will not check those conditions again in the use case.

The most common preconditions is that the user is logged on and validated. The other time a precondition is needed is when a second use case picks up the thread of activity part-way through a first use case. The first use case sets up a particular condition that the second relies upon. An

example is when the user selects a product or other partial choice in the first use case, and the second one uses knowledge of that product or choice in its processing.

Whenever I see a precondition, I know there is a higher-level use case in which the precondition gets established.

Reminder 11. Pass/Fail tests for one use case

It is nice when we can find simple pass/fail tests to let us know when we have filled in a part of the use case correctly. Here are the few I have found. All of them should produce a "yes" answer.

Table 20-1: Pass/fail tests for one use case

Field	Question
Use case title.	**1** Is the name an active-verb goal phrase, the goal of the primary actor?
	2 Can the system deliver that goal?
Scope and Level:	**3** Are the scope and level fields filled in?
Scope.	**4** Does the use case treat the system mentioned in the Scope as a black box? (The answer may be 'No' if the use case is a white-box business use case, but must be 'Yes' if it is a system requirements document).
	5 If the Scope is the actual system being designed, do the designers have to design everything in the Scope, and nothing outside it?
Level.	**6** Does the use case content match the goal level stated in Level?
	7 Is the goal really at the level mentioned?
Primary actor.	**8** Does the named primary actor have behavior?
	9 Does it have a goal against the SuD that is a service promise of the system?
Preconditions.	**10** Are they mandatory, and can they be ensured by the SuD?
	11 Is it true that they are never checked in the use case?
Stakeholders and interests.	**12** Are they mentioned? (Usage varies by formality and tolerance)
Minimal guarantees.	**13** If present, are all the stakeholders' interests protected?
Success guarantees.	**14** Are all stakeholders interests satisfied?

Table 20-1: Pass/fail tests for one use case

Field	Question
Main success scenario.	**15** Does it run from trigger to delivery of the success end condition?
	16 Is the sequence of steps right (does it permit the right variation in sequence)?
	17 Does it have 3 - 9 steps?
Each step in any scenario.	**18** Is it phrased as an goal that succeeds?
	19 Does the process move distinctly forward after successful completion of the step?
	20 Is it clear which actor is operating the goal (who is "kicking the ball?)
	21 Is the intent of the actor clear?
	22 Is the goal level of the step lower than the goal level of the overall use case? Is it, preferably, just a bit below the use case goal level?
	23 Are you sure the step does not describe the user interface design of the system?
	24 Is it clear what information is being passed?
	25 Does the step "validate", as opposed to "checking" a condition?
Extension condition.	**26** Can and must the system detect it?
	27 Must the system handle it?
Technology or Data Variation List.	**28** Are you sure this is not an ordinary behavioral extension to the main success scenario?
Overall use case content.	**29** To the sponsors and users: "Is this what you want?"
	30 To the sponsors and users: "Will you be able to tell, upon delivery, whether you got this?"
	31 To the developers: "Can you implement this?"

21. The Use Case Set

Reminder 12. An ever-unfolding story

For a development project, there is one use case at the top of the stack, called something like "Use the ZZZ system". This use case is little more than a table of contents that names the different outermost actors and their highest-level goals. It serves as a designated starting point for anyone looking at the system for the first time. It is optional, since it doesn't have much of a storyline, but most people like to see just one starting place for their reading.

It calls out the *outermost use cases*, which show the summary goals of the outermost primary actors of the system. For a corporate information system, there is typically an external customer, the marketing department, and the IT or security department. These use cases show the interrelationships of the sea-level use cases that define the system. For most readers, the "story" starts with one of these use cases.

The outermost use cases unfold into user-goal or sea level use cases. In the user-goal use cases, the design scope is the system being designed. The steps show the actors and system interacting to deliver the user's immediate goal.

A step in a sea-level use case unfolds into an underwater (indigo, or subfunction) use case if the sub use case is complicated or is used in several places. Subfunction use cases are expensive to maintain, so only use them when you have to. Usually, you will have to create subfunction-level use cases for *Find a Customer*, *Find a Product* and so on.

On occasion, a step in an indigo use case unfolds to another, deeper indigo use case.

The value of viewing the use case set as an ever-unfolding story is that it becomes a "minor" operation to move a complicated section of writing into its own use case, or to fold a simple sub use case back into its calling use case. Each action step can, in principle, be unfolded to become a use case in its own right.

See 10.1“Sub use cases” on page 116.

Reminder 13. Corporate scope and system scope

Design scope can cause confusion. People have different ideas of where, exactly, the boundaries of the system are. In particular, be very clear whether you are writing a *business use case* or a *system use case*.

A business use case is one in which the design scope is business operations. The use case is about an actor outside the organization achieving a goal with respect to the organization. The

business use case often contains no mention of technology, since it is concerned with how the business operates.

A system use case is one in which the design scope is the computer system about to be designed. The use case is about an actor achieving a goal against the computer system. It is a use case about technology.

The business use case is often written in white-box form, describing the interactions between the people and departments in the company, while the system use case is almost always written in black-box form. This is usually appropriate because the purpose of most business use cases is to describe how the present or future design of the company works, while the purpose of the system use case is to create requirements for new design. The business use case described the inside of the business, the system use case describes the outside of the computer system.

If you are designing a computer system, you should have both business and system use cases in your collection. The business use case show the context for the system's function, the place of the system in the business.

To reduce confusion, always label the scope of the use case. Consider creating a graphic icon to pictorially illustrate whether it is a business or system use case (see "Using graphical icons to highlight the design scope" on page 49). Consider placing a picture of the system inside its containing system, within the use case itself, to illustrate the scope pictorially (see Use Case 8:"Enter and Update Requests (Joint System) ." on page 52).

Reminder 14. Core values & variations

People keep inventing new use case formats. Experienced writers seem to be coming to a consensus on core values for them. Two papers in 1999 conference [Firesmith99], [Lilly99] described a top dozen or so mistakes made in writing use cases. The mistakes and fixes described in those articles echo the core values.

Core values

Goal-based. Use cases are centered around the goals of the primary actors, and the subgoals of the various actors, including the SuD, in achieving that goal. Each sentence describes a subgoal getting achieved.

Bird's eye view. The use case is written describing the actions as seen by a bird above the scene, or as a play, naming the actors. It is not written from the "inside looking out".

Readable. The ultimate purpose of a use case, or any specification, is to be read by people. If they cannot easily understand it, it does not serve its core purpose. You can increase readability by sacrificing some amount of precision and even accuracy, and make up for the lack with increased conversation. But once you sacrifice readability, your constituents won't read them.

Usable for several purposes. Use cases are a form of behavioral description that can be used for various purposes at various times in a project. They have been used:

* to provide black-box functional requirements.
* to provide requirements to an organization's business process redesign.
* to document an organization's business process (white box).
* to help elicit requirements assertions from users or stakeholders (being discarded afterward, as the team members write the final requirements in some other form).
* to specify the test cases that are to be constructed and run.
* to document the internals of a system (white box).
* to document the behavior of a design or design framework.

Black box requirements. When used as a functional specification technique, the SuD is always treated as a black box. Project teams who have tried writing white-box requirements (guessing what the insides of the system will look like) report that those use cases were hard to read, not very well received, and were brittle, changing as design proceeded.

Alternative paths after main success scenario. Jacobson's original idea of putting alternative courses after the main success scenario keeps showing up as the easiest to read. Putting the branching cases inside the main body of text seems to make the story too hard to read.

Not too much energy spent. Continued fiddling with the use cases does not keep increasing their value. The first draft of the use case brings perhaps half of the value of the use case. Adding to the extensions keeps adding value, but changing the wording of the sentences ceases to improve the communication after a short while. At that point, your energy should go to other things, such as checking the external interfaces, the business rules and so on, all part of the rest of the requirements. This comment about diminishing returns on writing varies, of course, with the criticality of the project.

Suitable variants

Even keeping to core values, a number of acceptable variants have been discovered.

Numbered steps vs. simple paragraphs. Some people number the steps, in order to be able to refer to them in the extensions section. Other people write in simple paragraphs and put the alternatives in similar paragraph form. Both seem to work quite well.

Casual vs. fully dressed. There are times when it is appropriate to allocate a lot of energy to detail the functional requirements, and other moments, even on the same project, when that is not a good use of energy. See 1.2“Your use case is not my use case” on page 20 and 1.5“Manage Your Energy” on page 29. This is true to such an extent that I don't even recommend just one template any more, but always show both the casual and fully dressed templates. Different writers

sometimes prefer one over the other. Each works in its own way. Compare Use Case 25:“Actually Login (casual version)” on page 121 with Use Case 5:“Buy Something (Fully dressed version)” on page 22.

Prior business modeling with vs. without use cases. Some teams like to document or revise the business process before writing the functional requirements for a system. Of those, some choose use cases to describe the business process, and some choose another business process modeling form. From the perspective of the system functional requirements, it does not seem to make much difference which business process modeling notation is chosen.

Use case diagrams vs. actor-goal lists. Some people like actor-goal lists to show the set of use cases being developed, while others prefer use case diagrams. The use case diagram, showing the primary actors and their user-goal use cases, can serve the same purpose as the actor-goal list.

White box use cases vs. collaboration diagrams. There is a near equivalence between white-box use cases and UML's collaboration diagrams. You can think of use cases as textual collaboration diagrams. The difference is that a collaboration diagram does not describe the components’ internal actions, which the use case might.

Unsuitable variants

"If" statements inside the main success scenario. If there were only one branching of behavior in a use case, then it would be simpler to put that branching within the main text. However, use cases have many branches, and people lose the thread of the story. Individuals who have used *if* statements report that they soon change to the form with main success scenario followed by extensions.

Sequence diagrams as replacement for use case text. Some software development tools claim to support use cases, because they supply sequence diagrams. While sequence diagrams also show interactions between actors,

* sequence diagrams do not capture internal actions (needed to show how the system protects the interests of the stakeholders);
* sequence diagrams are much harder to read (they are a specialized notation, and they take up a lot more space);
* it is nearly impossible to fit the needed amount of text on the arrows between actors;
* most tools force the writer to hide the text behind a pop-up dialog box, making the story line very hard to follow;
* most tools force the writer to write each alternate path independently, starting over each time from the beginning of the use case. This duplication of effort is tiring, error prone, and is also hard on the reader, who has to detect what difference of behavior is presented in each variation.

Sequence diagrams are not a good form for expressing use cases. People who insist on sequence diagrams, do so in order to get the tool benefit of automated services: cross-referencing, back-and-forth hyperlinks, ability to change names globally. While these services are nice (and lacking in the textual tools currently available), most writers and readers agree they are not sufficient reward for the sacrificed ease of writing and reading.

GUIs in the functional specs. There is a small art to writing the requirements so that the user interface is not specified along with the needed function. This art is not hard to learn, and is worth learning. There is strong consensus not to describe the user interface in the use cases. See 19.6“Advanced example of too much UI” on page 189, and the book by Constantine and Lockwood, Designing Software for Use.

Reminder 15. Quality questions across the use case set

I have only three quality questions that cross the use case set:

- Do the use cases form a story that unfolds from highest level goal to lowest?
- Is there a context-setting, highest level use case at the outermost design scope possible for each primary actor?
- To the sponsors and users: "Is this everything that needs to be developed?"

22. Working on the Use Cases

Reminder 16. It's just chapter 3 (where's chapter 4?)

Use cases are only a small part of the total requirements gathering effort, perhaps "chapter 3" of the requirements. They are a central part of that effort, they act as a core or hub that links together data definitions, business rules, user interface design, the business domain model, and so on. But they are not all of the requirements. They are the behavioral requirements.

This has to be mentioned over and over, because such an aura has grown around use cases that some teams try to fit every piece of requirements into a use case, somehow.

Reminder 17. Work breadth first

Work breadth first, not depth first, from lower precision to higher precision. This will help you manage your energy. See Section 1.5"Manage Your Energy" on page 29. Work in this order:

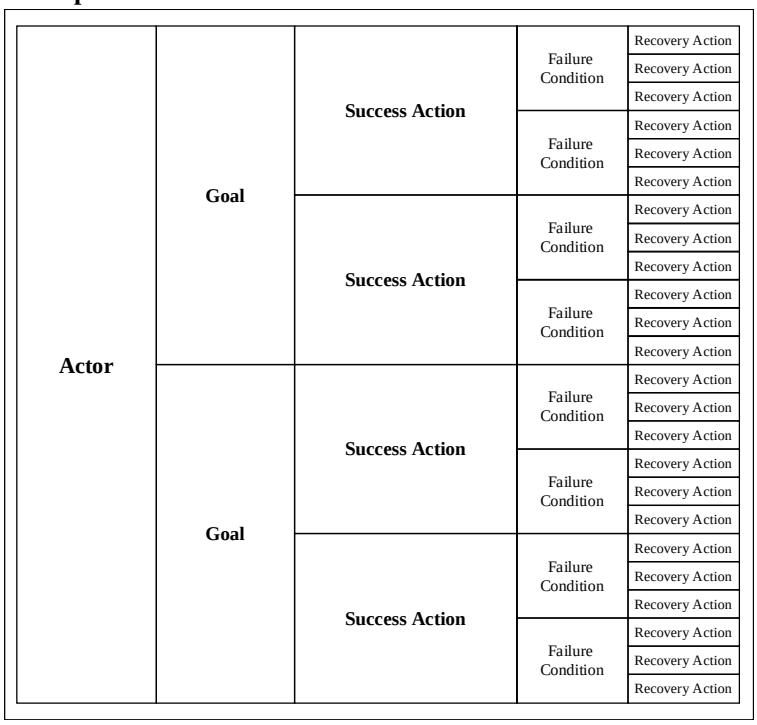
1 **Primary actors**. Collect all the primary actors as the first step in getting your arms around the entire system for a brief while. Most systems are so large that you will soon lose track of everything in it, so it is nice to have the whole system in one place for even a short time. Brainstorm these actors to help you get the most goals on the first round.

2 **Goals**. Listing all the goals of all the primary actors is perhaps the last chance you will have to capture the entire system in one view. Spend quite some time and energy getting this list as complete and correct as you can. The next steps will involve more people, and much more work. Review the list with the users, sponsors, and developers, so they all agree on the priority and understand the system being deployed.

3 **Main success scenario**. The main success scenario is typically short and fairly obvious. This tells the story of what the system delivers. Make sure the writers show how the system works once, before investigating all the ways it can fail.

4 **Failure / Extension conditions**. Capture all the extension conditions before worrying about how to handle them. This is a brainstorming activity, which is quite different than researching and writing the extension handling steps. Also, the list serves as a worksheet for the writers. They can write in spurts with breaks, without worrying about losing their place. People who try to fix each condition as they name them typically never complete the failure list. They run out of energy after writing a few failure scenarios.

Figure 25. Work expands with precision.



5 **Recovery steps**. These are built last of all the use case steps, but surprisingly, new user goals, new actors, and new failure conditions are often discovered during the writing of the recovery steps. Writing the recovery steps is the hardest part of writing use cases, because it forces the writer to confront business policy matters that often stay unmentioned. It is when I discover an obscure business policy, a new actor or use case while writing recovery steps, that I feel a vindication or payoff for the effort of writing them.

6 **Data-fields**. While formally outside the use case writing effort, often the same people have the assignment of expanding data names (such as "customer information") into lists of data fields (see 18.“Use Cases Briefs and eXtremeProgramming” on page 184).

7 **Data-field details & checks**. In some cases, different people write these details and checks at the same time the use case writers are reviewing the use cases. Often, it will be IT technical people who write the field details, while IT business analysts or even users write the use cases. This represents the data formats to the final level of precision. Again, they are outside the use cases proper, but have to be written eventually.

Reminder 18. The 12-step recipe

Step 1: Find the boundaries of the system (Context diagram, In/out list).

Step 2: Brainstorm and list the primary actors. (Actor List)

Step 3: Brainstorm and list the primary actors' goals against the system. (Actor-Goal List)

Step 4: Write the outermost summary level use cases covering all the above.

Step 5: Reconsider & revise the strategic use cases. Add, subtract, merge goals.

Step 6: Pick a use case to expand or write a narrative to get acquainted with the material.

Step 7: Fill in the stakeholders, interests, preconditions and guarantees. Double check them.

Step 8: Write the main success scenario. Check it against the interests and the guarantees.

Step 9: Brainstorm and list possible failure conditions and alternate success conditions.

Step 10: Write how the actors and system should behave in each extension.

Step 11: Break out any sub use case that needs its own space.

Step 12: Start from the top and readjust the use cases. Add, subtract, merge. Double check for completeness, readability, failure conditions.

Reminder 19. Know the cost of mistakes

The cost of lowered quality in the use case depends on your system and project. Some projects need next to no quality in the writing of the requirements document, because they have such good communications between users and developers:

> The Chrysler Comprehensive Compensation project team, building software to pay all of Chrysler's payroll using the "eXtreme Programming" methodology [Beck99], never went further than use case briefs. They wrote so little that they called them "stories" rather than use cases, and wrote each on an index card. Each was really a promise for a conversation between a requirements expert and a developer. Significantly, the team of 14 people sat in two (large) adjacent rooms, and had excellent in-team communications.

The better the internal communications are between your usage experts and developers, the lower the cost of omitting parts of the use case template. People will simply talk to each other and straighten matters out.

If you are working with a distributed development group, a multi-contractor development group, very large development group, or on life-critical systems, then the cost of quality failure is higher. If it is critical to get the system’s functionality correctly written down, then you need to pay close attention to the stakeholders and interests, the preconditions and the minimal guarantees.

Recognize where your project sits along this spectrum. Don't get too worked up over relatively small mistakes on a small, casual projects, but do get rigorous if the consequences of misunderstanding are great.

Reminder 20. Blue jeans preferred

Odd though it may sound, you will typically do less damage if you write too little, compared with too much. When in doubt, write less text, using higher level goals, with less precision, and write in plain story form. Then you will have a short, readable document - which means that people will bother to read it, and will then ask questions. From those questions, you can discover what information is missing.

The opposite strategy fails: if you write a hundred or so, low-level use cases, in great detail, then few or no people will bother to read the document, and you will shut communications on the team, instead of opening them. It is a common fault mistake programmers write at too low of a goal level, so this mistake happens quite often.

A small, true story.

> I helped on a successful 50-person, 15M$ project. We wrote only the main success scenario and a few failures, in a simple paragraph text form. This worked because we had excellent communications. Each requirements writer was teamed with 2-3 designer-programmers. They say next to each other or visited many times each day.

Actually, enhancing the quality of in-team communications helps every project. The teaming strategy described in the just-mentioned project, is the *Holistic Diversity* pattern from Surviving Object-Oriented Projects.

Reminder 21. Handle failures

One of the great values of use cases is naming all the extension conditions. It happens on many projects that there is a moment when the programmer has just written,

```
If <condition>
    then <do this>
    else ..?.
```

She or he stops. "else..?." she (he) muses. "I wonder what the system is supposed to do here? The requirements don't say anything about this condition. I don't have anyone to ask about this odd situation. Oh, well, ...", and then types something quick into the program,

```
    else <do that>
```

The "else" handling was something that should have been in the requirements document. Very often, it involves significant business rules. I often see usage experts huddling together or calling associates to straighten out, "just what should the system do under these circumstances?"

Finding the failure conditions and writing the failure handling often turns up new actors, new goals, and new business rules. Often, these are subtle and require some research, or change the complexity of the system.

If you are only used to writing the success scenario, try capturing the failures and failure handling on your next use cases. You are likely to be surprised and heartened by what you uncover.

Reminder 22. Job titles sooner and later

Job titles are important at the beginning and at the end of the project, but not in the middle. Pay attention to them early and pay attention to them later on, and don't worry about them in the midst of use case writing.

At the beginning of the project, you need to collect all the goals the system must satisfy, and put them into a readable structure. Focusing on the job titles or societal roles that will be affected by the system allows you to brainstorm effectively and make the best first pass at naming the goals. With a long list of goals in hand, the job titles also provide a clustering scheme to make it easier to review and prioritize the nominated goals.

Having the job titles in hand also allows you to characterize the skills and work styles of the different job titles. This information informs your design of the user interface.

Once people start developing the use cases, discussions will surface about how roles overlap. The role names used for primary actor become more generic (e.g., "Orderer") or more distant from the people who will actually use the system, until they are only place-holders to remind the writers that there really is an actor having a goal.

Once the system starts being deployed, the job titles become important again. The development team must

* assign permission levels to specific users, permission to update or perhaps just read each kind of information,
* prepare training materials on the new system, based on the skill sets of the people with those job titles, and which use cases each group will be using,
* package the system for deployment, packaging clusters of use case implementations together.

The job titles, important at the beginning and relatively insignificant in the middle, become important again at the end.

Reminder 23. Actors play roles

Actor means either the job title of the person using the system, or the role that person is playing at the moment of using the system (assuming for the moment it is a *person*). It is not really significant which way we use the term, so don't spend too much energy on the distinction.

The important part is the goal. That says what the system is going to do. Just exactly who calls upon that goal will be negotiated and rearranged quite a lot over the life of the system. When you discover that the Store Manager also can act in the capacity of a Sales Clerk, you can:

* Write in the use case, the primary actor is "Either Sales Clerk or Store Manager" (UML fans, draw the arrow from both actors to the ellipse).
* Write "The Store Manager might be acting in the role of Sales Clerk while executing this use case" (UML fans, draw a generalization arrow from Store Manager to point to Sales Clerk).
* Create an "Orderer" to be the primary actor. Write that a Sales Clerk or Store Manager is acting in the role of Orderer when running this use case (UML fans, draw "generalization" arrows from Sales Clerk and Store Manager to point to Orderer).

None of these is wrong, so you can choose whichever you find communicates to your audience.

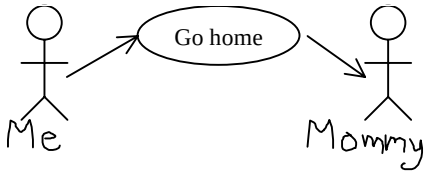
- Recognize that a person fills many roles, a person in a job is just a person filling a role, and that a person with a job title acts in many roles even when acting in the role of that job title.
- Recognize that the important part of the use case is not the primary actor's name, but the goal.
- Recognize that it is useful to settle a convention for your team to use, so that they can be consistent in their use.

Review “Why primary actors are unimportant (and important)” on page 63 and Reminder 22.“Job titles sooner and later” to see how actor names shift to role names and get mapped back to actor names again.

Reminder 24. The Great Drawing Hoax

For reasons that remain a mystery to me, many people have focused on the stick figures and ellipses in use case writing since Jacobson's first book came out, and neglected to notice that use cases are fundamentally a text form. The strong CASE tool industry, which already had graphical but not text modeling tools, seized upon this focus and presented tools that maximized the amount of drawing in use cases. This was not corrected in the OMG's Unified Modeling Language standard, which was written by people experienced in textual use cases. I suspect the strong CASE tool lobby affected OMG efforts. "UML is merely a tool interchange standard", is how it was explained to me on several occasions. Hence, the text that sits behind each ellipse somehow is not part of the standard, and is a local matter for each writer to decide.

Figure 26. "Mommy, I want to go home".



Whatever the causes, we now have a situation in which many people think that the ellipses *are* the use cases, even though the ellipses convey very little information. Experienced developers can be quite sarcastic about this. I thank Andy Hunt and Dave Thomas for this lighthearted spoof, mocking the cartoonish "requirements made easy" view of use cases. From The Pragmatic Programmer, 1999.

It is important to recognize that the ellipses cannot possibly replace the text. The use case diagram is (intentionally) lacking sequencing, data, and receiving actor. It is to be used

* as a table of contents to the use cases,
* as a context diagram for the system, showing the actors pursuing their various and overlapping goals, and perhaps the system reaching out to the secondary actors,
* as a "big picture", showing how higher-level use cases relate to lower-level use cases.

These are all fine, as described in Reminder 14.“Core values & variations” on page 209. Just remember that use cases are fundamentally a text form, so use the ellipses to augment, not replace the text. The following two figures below show two ways of presenting the context diagram.

Figure 27. Context diagram in ellipse figure form. (Adapted from Booch, Martin, Newkirk, Object-Oriented Design with Applications, Addison-Wesley, 2000.)

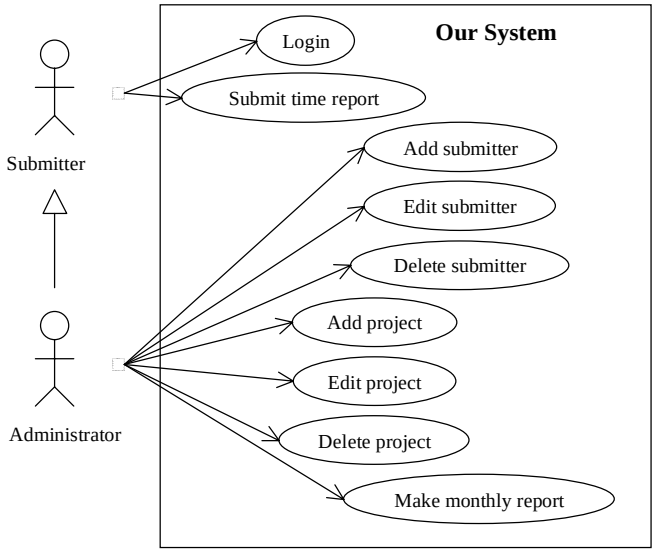


Figure 28. Context diagram in actor-goal format. Same actors and goals, common use cases repeated for clarity.

Table 22-1: Actor-Goal list for context diagram

Actor	Goal
Submitter	Log in
	Submit time report
Administrator	Log in
	Submit time report
	Add submitter
	Edit submitter
	Delete submitter
	Add project
	Edit project
	Delete project
	Make monthly report

Reminder 25. The great tool debate.

Sadly, use cases are not supported very well by any tools on the market (now, early 2000). Many companies claim to support them, either in text or in graphics. However, none of the tools contain a metamodel close to that described in *2.3"The Graphical Model" on page 41*, and most are quite hard to use. As a result, the use case writer is faced with an awkward choice.

Lotus Notes. Still my favorite, Lotus Notes has no metamodel of use cases, but supports cooperative work, hyperlinking, common template, document history, quick view-and-edit across the use case set, and easily constructed varieties of sortable views, all significant advantages. It allows the expanding data descriptions to be kept in the same database, but in different views. When you update the template, all use cases in the database get updated. It is fairly easy to set up, and extremely easy to use. I have used Lotus Notes to review over 200 use cases on a fixed-cost project bid, with the sponsoring customers.

The drawback of Lotus Notes, as of any of the plain text tools, is that renumbering steps and extensions while editing a use case soon becomes a nuisance. The hyperlinks eventually become out of date. Manually inserted backlinks become out of date very soon. There are no automated backlinks on the hyperlinks, so you can't tell which higher-level use cases invoke the use case you are looking at.

What makes Lotus Notes most attractive to me is the ease of use combined with the way the annotated actor-goal list is a dynamically generated view of the use case set. Just write a new use case, and the view immediately shows its presence. The view is simultaneously a hyperlinked table of contents, an actor-goal list, and a progress tracking table. I like to view the use cases either by priority, release, state of completion, and title, or by primary actor or subject area, level, and title.

Word processors with hyperlinks. With hyperlinking, word processors finally became viable for use cases. Put the use case template into a template file. Put each use case into its own text file using that template, and it becomes easy to create links across use cases. Just don't change the file's name! Writers are familiar with word processors, and are comfortable using them to write stories.

They have all the drawbacks of Lotus Notes. More significantly, there is no way to list all the use cases, sorted by release or status, and click them open. This means that a separate, overview list has to be constructed and maintained, which means it will soon be out of date. There is no global update mechanism for the template, and so multiple versions of the template tend to accrete over time.

Relational databases. I have seen and heard of several attempts to put the model of actors, goals, and steps, into a relational database such as Microsoft Access. While this is a natural idea, the resulting tools have been awkward enough to use that the use case writers went back to using their word processors.

Requirements management tools. Specialized requirements management tools, such as DOORS or Requisite Pro, are becoming more common. Such tools provide automated forward and

backward hyperlinks, and are intended for text-based requirements descriptions. On the minus side, none that I know of supports the model of main success scenario and extensions that is the heart of use cases. The few use cases I have seen from these sorts of tools are very lengthy, with a great deal of indenting, numbering, and lines, making the use case text hard to read (remember Reminder 2.“Make the use case easy to read.” on page 200, and Reminder 20.“Blue jeans preferred” on page 216). If you are using such a tool, find a way to make the story shine through.

CASE tools. On the plus side, CASE tools support global change to any entity in its metamodel, and automated back links for whatever it links. However, as described earlier, CASE tools tend to be built around boxes and arrows, doing poorly with text. Sequence diagrams are not an acceptable substitute for textual use cases, and most CASE tools offer little more than a dialog box for text entry. I have seen writing teams members mutiny, and revert to word processing, rather than use their CASE tool.

That leaves you with a less than pleasant choice to make. Good luck.

Reminder 26. Project planning using titles and briefs

Review 17.1“Use Cases In Project Organization” on page 164 for the pluses and minuses of using use cases to track the project’s progress, and an example of using the actor goal list as a project planning framework. Here are the reminders.

The use case planning table. Put the actors and goals in the left-most two columns of a table, and in the next columns record any of the following as you need to: business value, complexity, release, team, completeness, performance requirement, external interfaces used, and so on.

Using this table, your team can negotiate over the actual development priority of each use case. They will discuss business need versus technical difficulty, business dependencies and technical dependencies, and come up with a sequencing of development.

Delivering partial use cases As described in “Use cases cross releases” on page 166, you will quite often to decide to deliver only part of a use case in a particular release. Most teams simply use a yellow highlighter or bold text to indicate which portion of a use case is being delivered. You will want to write in the planning table the *first* release in which the use case shows up, and the *final* release, in which will deliver the use case in its entirety.

PART 4
END NOTES

How could I not discuss the Unified Modeling Language and its impact on use cases? UML actually impacts use case writing very little. Most of what I have to say about writing effective use cases fits inside one ellipse. Appendix A covers ellipses, stick figures, *includes*, *extends*, *generalizes*, the attendant hazards, and drawing guidelines.

Appendix B provides answers to selected exercises. I hope you do those exercises, and read the discussions provided with the answers.

Appendix C is a glossary of the key terms used in the book.

Appendix D is a list of the articles, books, and web pages I referred to along the way.

Appendix A: Use Cases in UML

The Unified Modeling Language defines graphical icons that people are determined to use. It does not address use case content or writing style, but it does provide lots of complexity for people to discuss. Spend your energy learning to write clear text instead. If you like diagrams, learn the basics of the relations, and then set a few, simple standards to keep the drawings clear.

23.1 Ellipses and Stick Figures

When you walk to the whiteboard and start drawing pictures of people using the system, it is very natural to draw a stick figure for the people, and ellipses or boxes for the use cases they are calling upon. Label the stick figure with the title of the actor and the ellipse with the title of the use case. The information is the same as the actor-goal list, but the presentation is different. The diagrams can be used as a table of contents.So far, all is all fine and normal.

The trouble starts when you or your readers believe that the diagrams define the functional requirements for the system. Some people get infatuated with the diagrams, thinking they will make a hard job simple (as in Figure 26.“"Mommy, I want to go home"” on page 219). They try to capture as much as possible in the diagram, hoping, perhaps, that text will never have to be written. Here are two typical events, symptoms of the situation.

> A person in my course recently unrolled a taped-together diagram several feet on a side, with ellipses and arrows going in all directions, *includes* and *extends* and *generalizes* all mixed around (distinguished, of course, only by the little text label on each arrow). He wanted coaching on whether their project was using all the relations correctly, and was unaware it was virtually impossible to understand what his system was supposed to *do*.
>
> Another showed with pride how he had "repaired" the evident defect of diagrams not showing the order in which sub use cases are called. He added yet more arrows to show which sub use case preceded which other, using the UML *preceeds* relation. The result, of course, was an immensely complicated drawing, that took up more space than the equivalent text, and was harder to read. To paraphrase the old saying, he could have put 1,000 readable words in the space of his one unreadable drawing.

Drawings are a two-dimensional mnemonic device that serve a cognitive purpose: to highlight relationships. Use the drawings for this purpose, not to replace the text.

With that purpose in hand, let us look at the individual relations in UML, their drawing and use.

23.2 UML's *Includes* Relation

A *base* use case *includes* an *included* use case if an action step in the base use cases calls out the included one's name. This is the normal and obvious relationship between a higher-level and a lower-level use case. The included use case describes a lower-level goal than the base use case.

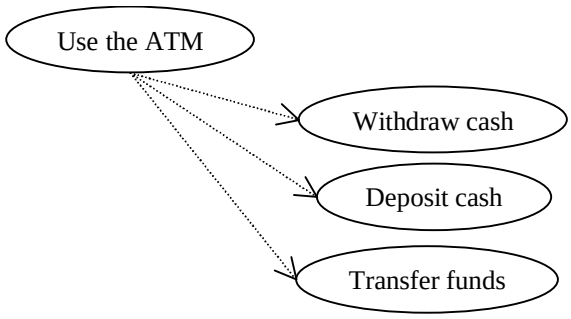
The verb phrase in an action step is potentially the name of a sub use case. If you never break that goal out into its own use case, then it is simply a step. If you do break that goal out into its own use case, then the step *calls* the sub use case (in my vocabulary), or it *includes the behavior of* the included use case, in UML 1.3 vocabulary. Prior to UML 1.3, it was said to *use* the lower level use case, but that phrase is now out of date.

A dashed arrow goes from the (higher-level) base use case to the included use case, signifying that the base use case "knows about" the included one, as illustrated in Figure 29..

Guideline 13: Draw higher goals higher

Always draw higher level goals higher up on the diagram than lower level goals. This helps reduce the goal-level confusion, and is intuitive to readers. When you do this, the arrow from a base use case to an *included* use case will *always* point down.

Figure 29. Drawing *Includes*.



UML permits you to change the pictorial representation of each of its elements. I find that most people drawing by hand simply draw a *solid* arrow from base to included use case (drawing dashed ones by hand is tedious). This is fine, and now you can justify it :-). When drawing with a graphics program, you will probably use the shape that comes with the program.

It should be evident to most programmers that the *includes* relation is the old subroutine call from programming languages. This is not a problem or a disgrace, rather, it is a natural use of a natural mechanism, which we use in our daily lives and also in programming. On occasion, it is appropriate to parameterize use cases, pass them function arguments, and even have them return values (see 14."Two Special Use Cases" on page 146). Keep in mind, though, that the purpose of a use case is to communicate with another person, not a CASE tool or a compiler.

23.3 UML's *Extends* Relation

An *extending* or *extension* use case *extends* a *base* use case if the extending one names the base one and under what circumstances it interrupts the base use case. The base use case does not name the extending one. This is useful if you want to have any number of use cases interrupt the base one, and don't want the maintenance nightmare of updating the higher level use case each time a new, interrupting use case is added. See Section 10.2"Extension use cases" on page 116.

Behaviorally, the extending use case specifies some internal condition in the course of the base use case, with a triggering condition. Behavior runs through the base use case until the condition occurs, at which point behavior continues in the extending use case. When the extending use case finishes, the behavior picks up in the base use case where it left off.

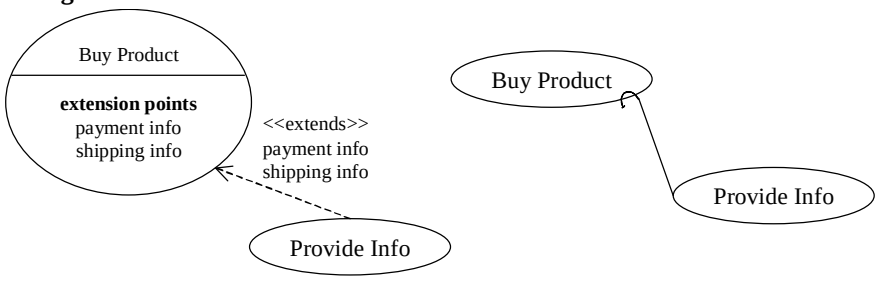
Rebecca Wirfs-Brock colorfully refers to the extending use case as a *patch* on the base use case (programmers should relate to the analogy of program patches!). Other programmers see it as a text version of the mock programming instruction, the *come-from* statement.

We use the extension form quite naturally when writing extension conditions within a use case. An *extension use case* is just the extension condition and handling pulled out and turned into a use case on its own (see*10.2"Extension use cases" on page 116*). Think of an extension use case as a scenario extension that outgrew its use case and was given its own space.

The default UML drawing for *extends* is a dashed arrow (the same as for *includes*) from extending to base use case, with the phrase <<extends>> set alongside it. I draw it with a hook from the extending back to the base use case, as shown in *Figure 30.*, to highlight the difference between *includes* and *extends* relations.

Figure 30.(a) shows the default UML way of drawing *extends* (example from UML Distilled. Figure 30. (b) shows the hook connector.

Figure 30. Drawing *Extends*.



Guideline 14: Draw extending use cases lower

An extension use case is generally at lower level than the use case it extends, and so it should similarly be placed lower on the diagram. In the *extends* relation, however, it is the lower use case that knows about the higher use case. Therefore, draw the arrow or hook *up* from the extending to the base use case symbol.

Guideline 15: Use different arrow shapes

UML deliberately leaves unresolved the shape of the arrows connecting use case symbols. Any relation can be drawn with an open-headed arrow and some small text that says what the relation is. The idea is that different tool vendors or project teams might want to customize the shapes of the arrows, and the UML standard should not prevent them.

The unfortunate consequence is that people simply use the undifferentiated arrows for all relations. This makes drawings hard to read. The reader must study the small text to detect which relations are intended. Later on, there are no simple visual clues to help remember the relations. This combines with the absence of other drawing conventions to make many use case diagrams truly incomprehensible.

Therefore, take the trouble to set up different arrow styles for the three relations.

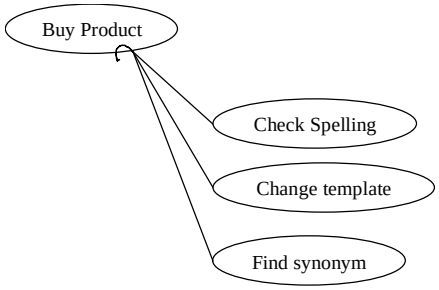
- The standard *generalizes* arrow in UML is the triangle-headed arrow. Use that.
- The default, open-headed arrow should be the frequently used one. Use it for *includes.*
- Create a different arrow for *extends*. I have started using a hook from extending to base use case. Readers like that it is immediately recognizable, doesn't conflict with any of the other UML symbols, and brings along its own metaphor, that an extending use case has its hooks in the base use case. Whatever you use, work to make the *extends* connector stand out from the other ones on the page.

Correct use of *extends*

"When to use extension use cases" on page 118 discusses the main occasions on which to create extension use cases. I repeat those comments here.

The most common is when there are many asynchronous services the user might activate, which should not disturb the base use case. Often, they are developed by different teams. These situations show up with shrink-wrapped software packages as illustrated in Figure 31..

Figure 31. Three interrupting use cases extending a base use case.



The second situation is when you are writing additions to a locked requirements document. In an incrementally staged system, you might lock the requirements after each delivery. You would then *extend* a locked use case with one that adds function.

Extension points

The circumstance that caused *extends* to be invented in the first place was the practice of never touching the requirements file of a previous system. In the original telephony systems where these were developed, the business often added asynchronous services, and so the *extends* relation was practical, as just described. The new team could build on the safely locked requirements document, adding the requirements for a new, asynchronous service at whatever point in the base use case was appropriate, without touching a line of the original system requirements.

But referencing behavior in another use case is problematic. If no line numbers are used, how should we refer to the point at which the extension behavior picks up? And if line numbers are used, what happens if the base use case gets edited and the line numbers change?

Recall, if you will, that the line numbers are really line labels. They don't have to be numeric, and they don't have to be sequential. They are just there for ease of reading and so the extension conditions have a place to refer to. Usually, however, they are numbers, and they are sequential. Which means that they will change over time.

Extension points were introduced to fix these issues. An *extension point* is a publicly visible label in the base use case that identifies a moment in the use case's behavior by nickname (technically, it can refer to set of places, but let us leave that aside for the moment).

Publicly visible extension points introduce a new problem. The writers of a base use cases are charged with knowing where it can get extended. They must go back and modify it whenever someone thinks up a new place to extend it. Recall that the original purpose of *extends* was to avoid having to modify the base use case.

You will have to deal with one of these problems. Personally, I find publicly declared extension points more trouble than they are worth. I prefer just describing, textually, where in the base use case the extending use case picks up, ignoring nicknames, as in the example below.

If you do use extension points, don't show them on the diagram. The extension points take up most of the space in the ellipse, dominating the reader's view and obscuring the much more important goal name (see Figure 30.). The behavior they refer to does not show up on the diagram. They cause yet more clutter.

There is one more fine point about extension points. An extension point name is permitted to call out not just *one* place in the base use case, but as many as you wish, places where the extending use cases needs to add behavior. You would want this in the case of the ATM, when adding the extension use case *Use ATM of Another Bank*. The extending use case needs to say,

> "Before accepting to perform the transaction, the system gets permission from the customer to charge the additional service fee.
> ...
> After completing the requested transaction, the system charges the customer's account the additional service fee."

Of course, you could just say that.

23.4 UML's *Generalizes* Relations

A use case may *specialize* a more general one (and vice versa, the general one *generalizes* the specific one). The (specializing) child should be of a "similar species" to the (general) parent. More exactly, UML 1.3 says, "a generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior and extension points defined in the parent use case, and participates in all the relationships of the parent use case".

Correct use of *generalizes*

A good test phrase is *generic*, using the phrase "some kind of". Be alert for when find yourself saying, "the user does some kind of this action", or saying, "the user can do one of several kinds of things here". Then you have a candidate for *generalizes*.

Here is a fragment of the *Use the ATM* use case.

---

1. Customer enters card and PIN.
2. ATM validates customer's account and PIN.
3. Customer does a transaction, one of:
   - Withdraw cash
   - Deposit cash

- Transfer money
- Check balance

Customer does transactions until selecting to quit
4. ATM returns card.

What is it the customer does in step 3? Generically speaking, "a transaction". There are four *kinds of* transactions the customer can do. *Generic* and *kinds of* tip us off to the presence of the generic or generalized goal, "Do a transaction". In the plain text version, we don't notice that we are using the *generalizes* relation between use cases, we simply list the kinds of operations or transactions the user can do and keep going. For UML mavens, though, this is the signal to drag out the *generalization* arrow.

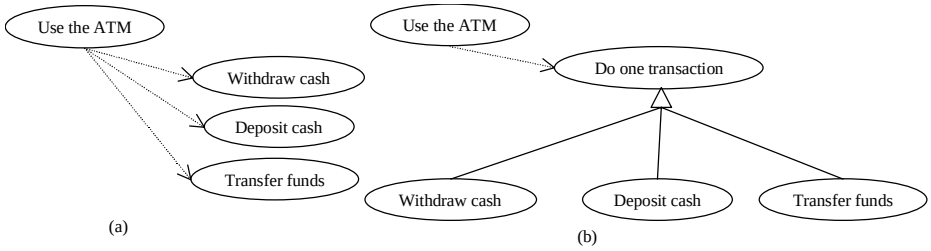
Actually, we have two choices. We can ignore the whole generalizes business, and just *include* the specific operations, as shown in Figure 32.(a). Or, we can create a *general* use case for "Do one ATM transaction", and show the specific operations as specializations of it, as in Figure 32.(b).

Use whichever you prefer. Working in prose, I don't create generalized use cases. There is rarely any text to put into the generic goal, so there is no need to create a new use case page for it. Graphically, however, there is no way to express "does one of the following transactions", so you have to find and name the generalizing goal.

Guideline 16: Draw generalized goals higher

Always draw the generalized goal higher on the diagram. Draw the arrowhead pointing up into the bottom of the generalizing use case, not into the sides. See Figure 32. and Figure 34. for examples.

Figure 32. Drawing *Generalizes.* Converting a set of *included* use cases into specializations of a generic action.



Hazards of *generalizes*

Watch out when combining specialization of actors with specialization of use cases. The key idiom to avoid is that of a *specialized actor using a specialized use case*, as illustrated in Figure 33."Hazardous generalization, closing a big deal".

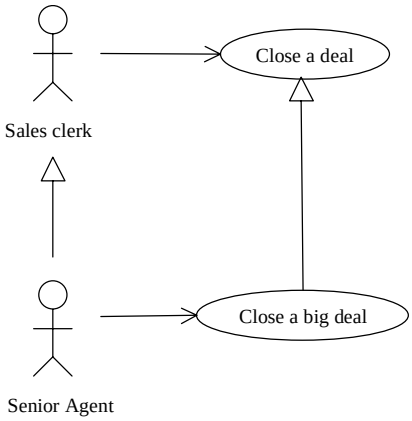


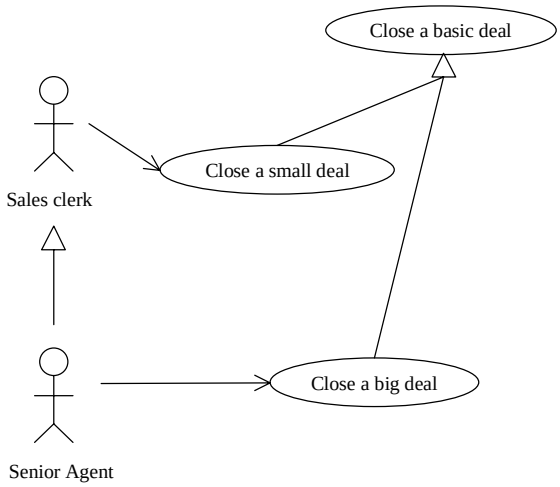
Figure 33. Hazardous generalization, closing a big deal.

Figure 33. is trying to express the fairly normal idea that a Sales Clerk can close any deal, but it takes a special kind of sales clerk, a Senior Agent, to close a deal above a certain limit. Let's watch how the drawing actually expresses the opposite of what is intended.

From Section 4.2"The primary actor of a use case" , we recall that the specialized actor can do every use case the general actor can do. So the Sales Clerk is a generalized Senior Agent. To many people, this seems counterintuitive, but it is official and correct.

The other specialization seems quite natural: Closing a Big Deal is a special case of closing an ordinary deal. However, the UML rule is, "A *specialized use case can be substituted wherever a general use case is mentioned*". Therefore, the drawing says that an ordinary Sales Clerk can close a Big Deal!

Figure 34. Correctly closing a big deal.



The corrected drawing is shown in Figure 34."Correctly closing a big deal" . You might look at this drawing and ask, does closing a small deal really *specialize* closing a basic deal, or does it *extend* it? Since working with text use cases will not put you in this sort of puzzling and economically wasteful quandary, I leave that question as an exercise to the interested reader.

In general, the critique I have of the generalizes relation is that the professional community has not yet reached an understanding of what it means to subtype and specialize behavior, what properties and options are implied. Since use cases are descriptions of behavior, there can be no standard understanding of what it means to specialize use cases.

If you do use the generalizes relation, my suggestion is to make the generalized use case empty, as in *Do a transaction*, above. Then the specializing use case will supply *all* the behavior, and you only have to worry about the one trap described above.

23.5 *Subordinate* vs. sub use cases

In the extended text section of UML specification 1.3, the UML authors describe a little-known pair of relations between use cases, one that has no drawing counterpart, is not specified in the object constraint language, but is simply written into the explanatory text. The relations are *subordinate use case*, and its inverse, *superordinate use case*.

The intent of these relations is to let you show how the use cases of *components* work together to deliver the use case of a larger system. In an odd turn, the components themselves are not shown. The use cases of the components just sit in empty space, on their own. It is as though you were to draw an anonymous collaboration diagram, a special sort of functional decomposition, that you are later supposed to explain with a proper collaboration diagram.

> "A use case specifying one model element is then refined into a set of smaller use case, each specifying a service of a model element contained in the first one. ... Note though, that the structure of the container element is not revealed by the use cases, since they only specify the functionality offered by the elements. The subordinate use cases of a specific superordinate use case cooperate to perform the superordinate one. Their cooperation is specified by collaborations and may be presented in collaboration diagrams." (UML 1.3 specification)

The purpose of introducing these peculiar relations in the explanatory text of the use case specification is unclear. I don't propose to explain them. The reason that I bring up the matter is because I use the term "sub use case" in this book, and someone will get around to asking, "What is the relation between Cockburn's sub use case and the UML subordinate use case?"

I intend sub use case to refer to a goal at a lower goal level. In general, the higher level use case will call (*include*) the sub use case. Formerly, I said "subordinate" and "superordinate" for higher and lower level use cases. Since UML 1.3 has taken those words, I have shifted vocabulary. My experience is that people do not find anything odd to notice about the terms "calling use case" and "sub use case". These notions are clear to even the novice writer and reader.

23.6 Drawing Use Case Diagrams

When you choose to draw use case diagrams with stick figures and ellipses, or just with rectangles and arrows, you will find that the ability of the diagram to communicate easily to your readers is enhanced if you set up and follow a few simple diagramming conventions. Please don't hand your readers a rat's next of arrows, and then expect them to trace out your meaning. The guidelines mentioned above, for the different use case relations, will help. There are two more drawing guidelines that can help.

Guideline 17: User goals in a context diagram

On the main, context diagram, do not show any use cases lower than user-goal level. The purpose of the diagram is, after all to provide context, to give a table of contents for the system being designed. If you decompose use cases in diagram form, put the decompositions on separate pages.

Guideline 18: Supporting actors on the right

I find it helpful to place *all* the primary actors on the left of the system box, leaving room on the right for the supporting (secondary) actors. This reduces confusion about primary versus secondary actors.

Some people never draw supporting actors on their diagrams. This frees up the right side of the box so that primary actors can be placed on both sides.

23.7 Write text-based use cases instead

If you spend very much time studying and worrying about the graphics and the relations, then you are expending energy in the wrong place. Put your energy into writing easy-to-read prose. In prose, the relations between use cases are straightforward, and you won't understand why other people are getting tied up in knots about them.

This is a view shared by many use case experts. It is somewhat self-serving to relate the following event, but I wish to emphasize the seriousness of the suggestion. My thanks to Bruce Anderson of IBM's European Object Technology Practice for the comment he made during a panel on use cases at OOPSLA '98. A series of questions revolved around the difference between *includes* and *extends* and the trouble with the exploding number of scenarios and ellipses. Bruce responded that his groups don’t run into scenario explosion and don’t get confused. The next questioner asked why everyone else was concerned about "scenario explosion and how to use *extends"*, but he wasn't. Bruce's answer was, "I just do what Alistair said to do." His teams spend time writing clear text, staying away from *extends*, and not worrying about diagrams.

People who write good text-based use cases simply do not run into the problems of people who fiddle with the stick figures, ellipses and arrows of UML. The relations come naturally when you write an unfolding story. They become an issue only if you dwell on them. As more consultants gain experience both ways, an increasing number reduce emphasis on ellipses and arrows, and recommend against using the *extends* relation.

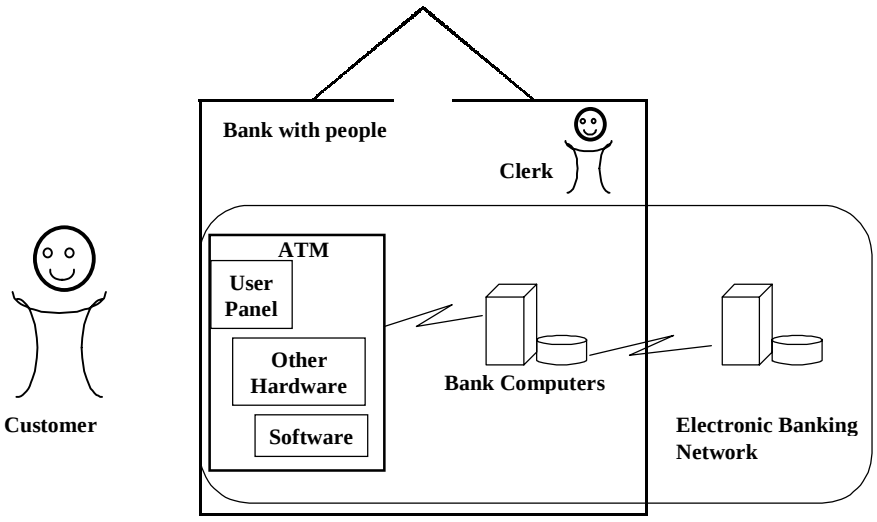
Appendix B: Answers to (some) Exercises

Exercise 6 on page 58

We could be describing our **neighborhood**, or the set of **electronically connected industries**. On a smaller scale, we could be designing the **bank building and lighting system**. We could be designing a new **bank computer system and ATM**. Or just the **ATM**. Or we could be discussing a new **key panel design**. Or we could be talking about the design for a new **Enter key**. There is no way to tell, from this fragment of the story, which system is being discussed.

Exercise 7 on page 58

Figure 35. Design scopes for the ATM.



Again, note, from the user story fragment, we cannot tell which of the systems we are discussing. This becomes relevant for Exercise 13 and Exercise 14.

Exercise 13 on page 68

Recall the pass/fail tests. An actor must be able to execute an *if* statement's worth of behavior. A primary actor has a goal, calling upon a system's promised services.

The ATM. The SuD.

The customer. A primary actor and stakeholder.

The ATM card. Not an actor. It does not have sufficient behavior (note: this refers to "dead iron filings" cards; "smart cards" with embedded chips may qualify). The ATM card is really just a data envelope, serving as no more than fast, fixed typing on the customer's part.

The bank. Not an actor for our purposes. It is a system containing the ATM.

The front panel. Not an actor for our purposes. It is a component of the SuD.

The bank owner A stakeholder. Probably not a primary actor**.**

The serviceman. A primary actor.

The printer. Not an actor for our purposes. It is a component of the SuD.

The main bank computer system. A secondary actor. It might be a primary actor, if you can think of a situation in which it initiates a conversation with the ATM.

The bank teller. Depends on the job assignments. Who empties and refills the cash? If you said, "Refiller" or "Service staff", then perhaps you will never create a use case with the bank teller as primary actor. If you answer, "The bank teller does", then bank teller is a primary actor.

The bank robber. Depends on the design scope and your creativity. I could never think of a decent use case for the bank robber that wasn't just an extension condition of a customer's use case, until someone suggested, "Steal the ATM!" That brings up the idea of a movement detector. Depending on how we phrase the goal, we could either end up with the robber having a use case (whose goal never succeeds!), or just more extension conditions in the customer's use case.

Exercise 14 on page 68

The answers depend on which containing system you choose (see Figure 35.).

The ATM. Not an actor for our purposes. It is a now a component of the SuD.

The customer. Still a primary actor and stakeholder.

The ATM card. Not an actor, for the same reasons (with the same disclaimers).

The bank. Look at Figure 35.. If you chose "Bank with people" as the containing system, then this is your SuD. If you chose "Electronic Banking System", then this is probably an actor (depending on whether you can justify a service of the Electronic Banking System that it calls upon).

The front panel. Not an actor for our purposes. It is a component.

The bank owner Depends on which containing system you chose and what sorts of service goals you come up with. Could end up either as a component of the Bank and hence not a primary actor, or as primary actor to the Bank. Probably not a primary actor of the Electronic Banking System.

The serviceman. A primary actor if a hired outside serviceman. A component if an employee of the bank and you chose the Bank as the SuD.

The printer. Not an actor for our purposes. It is a component.

The main bank computer system. Now a component of either containing system.

The bank teller. Either a component (of the Bank), or possibly a primary actor of the Electronic Banking System.

The bank robber. Same discussion as before.

Exercise 16 on page 77

Summary (white): Take someone out for dinner. 

Summary (white): Use the ATM 

User goal (blue): Get money from the ATM  .

Subfunction (indigo): Enter PIN 

Subfunction (black): Find the Enter button. 

Exercise 17 on page 77

Table 24-1:

Actor	Goal	Level
Serviceman	Put ATM into working order	summary
	Run ATM self-test	user goal
Bank Clerk	Restock money	user goal
	Refill supplies	user goal
Customer	Use the ATM	summary
	Withdraw cash	user goal
	Deposit money	user goal
	Transfer money	user goal
	Check balance	user goal

Exercise 20 on page 89

The easiest way to find the minimal guarantee is to ask, "What would make a stakeholder unhappy?" The stakeholders are the customers, the bank, and the banking overseer agency.

The customers will be unhappy if they don't get their cash, but that is not expected in the minimal guarantee. Let's assume they don't get their cash. Then they'll be unhappy if they get debited for the transaction. In fact, they'll be unhappy anytime they get debited more than they got cash. They also want a log of all transactions, so they can defend themselves against fraud.

The bank will be unhappy if the customer gets more cash than they get debited. They also want a log to protect themselves. They probably also want a special kind of log that says how far the transaction got in case of catastrophic failure, so they can sort out any errors.

The overseer agency wants to see that guidelines are being followed, so they are mostly interested that a log of all transactions gets produced.

As a result, we have a minimal guarantee that the amount debited equals the amount dispensed, with a micro-log of how far the transaction handling got in case of catastrophic failure. And each transaction is logged.

Exercise 23 on page 90

The success guarantee is that the account is debited the amount *dispensed* (not the amount *requested* - check failure conditions!), the card is returned, the machine is reset, and a log was made of the transaction.

Exercise 26 on page 102

Here is the dialog description for withdrawing cash from an ATM. Sending out 100 use cases like this will make for some unhappy readers. See the next answer for a semantic description.

1. Customer runs ATM card through the card reader.
2. ATM reads the bank id and account number.
3. ATM asks customer whether to proceed in Spanish or English.
4. Customer selects English.
5. ATM asks for PIN number and to press Enter.
6. Customer enters PIN number, presses Enter.
7. ATM presents list of activities for the Customer to perform.
8. Customer selects "withdraw cash".
9. ATM asks customer to say how much to withdraw, in multiples of $5, and to press Enter.
10. Customer enters an amount, a multiple of $5, presses Enter.
11. ATM notifies main banking system of customer account, amount being withdrawn.
12. Main banking system accepts the withdrawal, tells ATM new balance.
13. ATM delivers the cash.

14. ATM asks whether customer would like a receipt.
15. Customer replies, yes.
16. ATM issues receipt showing new balance.
17. ATM logs the transaction.

Exercise 27 on page 102

Here is the streamlined version of FASTCASH, showing the actors' intents.

1. Customer runs ATM card through the card reader.
2. ATM reads the bank id and account number from the card, validates them with the main computer.
3. Customer enters PIN. ATM validates PIN.
4. Customer selects FASTCASH and withdrawal amount, a multiple of $5.
5. ATM notifies main banking system of customer account, amount being withdrawn, and receives back acknowledgement plus the new balance.
6. ATM delivers the cash, card and a receipt showing the new balance.
7. ATM logs the transaction.

Exercise 29"Fix faulty 'Login'"

From page 102.

The sample contains three kinds of mistakes. The first thing to catch is that the use case is not about logging in, never mind what the use case name and description say. It is about using the order processing system. The real use case here is a summary use case at kite level. The first six steps are about logging in, but that is at a different level of goal entirely and should be separated out. Once we do that, we'll notice that the user logs in but never logs out of this system!

"While the user does not select Exit loop", "end if" and "end loop" are programmer constructs that will not make sense to the users reviewing the use case. The continual "if" statements clutter the writing. The steps describe the user interface design. All these should be fixed.

"The use case starts when..." and "The use case ends when..." are stylistic conventions suggested by some teachers. There is nothing particularly wrong with them, they are simply ornamentation, which I don't find necessary. Most people assume the use case starts with step 1 and ends when the writing stops.

The other style to note is the phrasing, "User...then Use Place Order". The "use" in that phrase refers to the *includes* relation of UML (formerly called the *uses* relation!). I find it clutters rather than clears the writing, and so I prefer to write "User...places the order". You will probably follow whatever convention your project team sets up for referring to other use cases.

In the end, we find two use cases to pull apart, the kite use case *Use the order processing system*, and the subfunction *Log in*. *Log in* you can derive on your own. Note that the links to other use cases are written in underline.

USE CASE 38: USE THE ORDER PROCESSING SYSTEM

Main success scenario:

1. User logs in.
2. System presents the available functions. User selects and does one:
   - Place Order
   - Cancel Order
   - Get Status
   - Send Catalog
   - Register Complaint
   - Run Sales Report
3. This repeats until the user selects to exit.
4. System logs user out when user selects to exit.

Exercise 30 on page 109

Here are a sampling of failure conditions. Typically, my classes produce a list 2-3 times this long. Notice that all conditions are detectable and must be handled. How did you do?

Card reader broken or card scratched
Card for an ineligible bank
Incorrect PIN
Customer does not enter PIN in time
ATM is down
Host computer is down, or network is down
Insufficient money in account
Customer does not enter amount in time
Not a multiple of $5
Amount requested is too large
Network or host goes down during transaction
Insufficient cash in dispenser
Cash gets jammed during dispensing
Receipt paper runs out, or gets jammed
Customer does not take the money from the dispenser

Exercise 34 on page 113

USE CASE 39:BUY STOCKS OVER THE WEB

Primary Actor: Purchaser / user
Scope: PAF Level: User goal
Precondition: User already has PAF open.
Minimal guarantees: sufficient log information exists that PAF can detect that something went wrong and can ask the user to provide the details.
Success guarantees: remote web site has acknowledged the purchase, PAF logs and the user's portfolio are updated.
Main success scenario:
1. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwabb, etc.)
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site, and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.
Extensions:
2a. User wants a web site PAF does not support:
    2a1. System gets new suggestion from user, with option to cancel use case.
3a. Web failure of any sort during setup:
    3a1. System reports failure to user with advice, backs up to previous step.
    3a2. User either backs out of this use case, or tries again.
4a. Computer crashes or gets 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 user about the outcome.
    4b2. (see use case *Update questioned purchase*)
5a. Web site does not return the needed information from the purchase:
    5a1. PAF logs the lack of information, has the user *Update questioned purchase*.
5b. Disk crash or disk full during portfolio update operation:
    5b1. On restart, PAF detects the log inconsistency, and asks the user to *Update questioned purchase*.

Exercise 37 on page 128:

USE CASE 40:PERFORM CLEAN SPARK PLUGS SERVICE

Precondition: car taken to garage, engine runs
Minimal guarantee: customer notified of larger problem, car not fixed.
Success guarantee: engine runs smoothly
Main Success Scenario:
1. open hood and cover fender with protective materials.
2.remove spark plugs.
3. wipe grease off spark plugs
4. clean and adjust gaps
5. test and verify plugs work
6. replace plugs
7. connect ignition wires to appropriate plugs.
8. test and verify that engine runs smoothly.
9. clean tools, equipment.
10. remove protective materials from fenders, clean any grease from car.

Extensions:
4a. Plug is cracked or worn out: Replace it with a new plug
8a. Engine still does not run smoothly:
8a1.*Diagnose rough engine* (UC 23)
8a2. *Notify customer of larger problem with car* (UC 41)

25. Appendix C: Glossary

Main terms

Use case. A use case expresses a contract between the stakeholders of a system about its behavior. It describes the system's behavior and interactions under various conditions as it responds to a request on behalf of the stakeholders, the *primary actor*, showing how the primary actor's goal gets delivered or fails. The use case collects together the scenarios related to the primary actor's goal.

Scenario. A scenario is a sequence of action and interactions that occurs under certain conditions, expressed without *if*s or branching.

A ***concrete scenario*** is a scenario in which all the specifics are named: the actor names and the values involved. It is equivalent to describing a story in the past tense, with all details named.

A ***usage narrative***, or just ***narrative***, is a concrete scenario that reveals motivations and intentions of various actors. It is used as a warm-up activity to reading or writing use cases.

In requirements writing, scenarios are written using placeholder terms like "customer" and "address" for actors and data values. When it is necessary to distinguish these from *concrete scenarios* they can be called ***general scenarios***.

Path *through a use case* and ***course*** *of a use case* are synonyms for *general scenario*.

The ***main success scenario*** is the one written in full, from trigger to completion, including goal delivery and any bookkeeping that happens after. It is a typical and illustrative success scenario, even though it may not be the only success path.

An ***alternate course*** is any other scenario or scenario fragment written as an extension to the main success scenario.

An ***action step*** is the unit of writing in a scenario. Typically one sentence, usually describes behavior of only one actor.

Scenario extension. A scenario fragment that starts upon a particular condition in another scenario.

The ***extension condition*** names the circumstances under which the different behavior occurs.

An ***extension use case*** is use case that interrupts another use case, starting upon a particular

condition. The use case that gets interrupted is called the ***base use case.***

An ***extension point*** is a tag or nickname for a place in a base use case where an extension use case can interrupt it. An extension point may actually name a set of places in the base use case, so that the extension use case can collect together all the related extension behaviors that interrupt the base use case for one set of conditions.

A ***sub use case*** is a use case called out in a step of a scenario. In UML, the calling use case is said to ***include*** *the behavior of* the sub use case.

Interaction. A message, a sequence of interactions, or a set of interaction sequences.

Actor. Something with behavior (able to execute an *if* statement). It might be a mechanical system, computer system, a person, an organization or some combination.

An ***external actor*** is an actor outside the system under discussion.

A ***stakeholder*** is an external actor which is entitled to have its interests protected by the system, and satisfying whose interests requires the system to take specific actions. Different use cases can have different stakeholders.

A ***primary actor*** is a stakeholder who requests the system to deliver a goal. Typically but not always, the primary actor initiates the interaction with the system. The primary actor may have an intermediary initiate the interaction with the system, or may have the interaction triggered automatically on some event.

A ***supporting*** or ***secondary*** actor is a system against which the SuD has a goal.

An ***off-stage*** or ***tertiary*** actor is a stakeholder of a use case who is not the primary actor.

An ***internal actor*** is either the system under discussion (SuD) itself, a subsystem of the SuD, or an active component of the SuD.

Types of use cases

A use case ***brief*** is a one-paragraph synopsis of the use case.

A ***casual use case*** is one written in simple, paragraph, prose style. It is likely to be missing project information associated with the use case, and is likely to be less rigorous in its description than a fully dressed use case.

A ***fully dressed use case*** is written with one of the full templates, identifying actors, scope, level, trigger condition, precondition, and all the rest of the template header information, plus project annotation information.

A ***black-box use case*** does not mention any components inside the SuD. Typically used in the system requirements document.

A ***white-box use case*** mentions the behavior of the components of the SuD in the description. Typically used in business process modeling.

A ***summary-level use case*** is one that takes multiple user-goal sessions to complete, possibly weeks, months or years. Sub use cases can be any level of use case. Marked graphically with a cloud  or a kite  . The cloud is used for use cases that contain steps at cloud or kite level. The kite is used for use cases that contain user-goal steps.

A ***user-goal use case*** satisfies a particular and immediate goal of value to the primary actor. Typically performed by one primary actor in one sitting of 2-20 minutes (less if the primary actor is a computer), after which they can leave and proceed with other things. Steps are user-goal or lower. Marked graphically with waves  .

A ***subfunction use case*** is one satisfying a partial goal of a user-goal use case or of another subfunction. Steps are lower-level subfunctions. Marked graphically with a fish  or a clam  . Using the clam signifies that the use case is too low level and should not be written at all.

The phrase ***business use case*** is a short-cut phrase indicating that the use case puts the emphasis on the operation of the business rather than the operation of a computer system. It is possible to write a business use case at any goal level, but only at *enterprise* or *organization* scope.

The phrase ***system use case*** is a short-cut phrase indicating that the use case puts the emphasis on the computer or mechanical system rather than the operation of a business. It is possible to write a system use case at any goal level and at with any scope, including *enterprise* scope. A system use case written at enterprise scope highlights the effect of the SuD on the behavior of the enterprise.

Enterprise scope means the SuD is an organization or enterprise. Labeled on the use case with the name of the organization, business or enterprise. Marked graphically with a building in gray  or white  depending on whether the use case is black- or white-box.

System scope means the SuD is a mechanical/ hardware/ software system or application. Labeled on the use case with the name of the system. Marked graphically with a box in gray  or white  depending on whether the use case is black- or white-box.

Subsystem scope means the SuD in this use case is a portion of an application, perhaps a sub-system or framework. Labeled on the use case with the name of the subsystem, and marked graphically with a threaded bolt  .

Diagrams

Use case diagram. In UML, the diagram showing the external actors, the system boundarry, the use cases as ellipses, and arrows connecting actors to ellipses or ellipses to ellipses. Primarily useful as a context diagram and table of contents.

Sequence diagram. In UML, the diagram showing actors across the top, owning columns of space, and interactions as arrows between columns, with time flowing down the page. Useful for showing one scenario graphically.

Collaboration diagram. In UML, a diagram showing the same information as the sequence diagram but in a different form. The actors are placed around the diagram, and interactions are shown as numbered arrows between actors. Time is shown only by numbering the arrows.

26. Appendix D: Reading

Books referenced in the text.

Beck, K., Extreme Programming Explained, Addison-Wesley, 1999.

Cockburn, A., Surviving Object-Oriented Projects, Addison-Wesley, 1998.

Cockburn, A., Software Development as a Cooperative Game, Addison-Wesley, due 2001.

Constantine, L., and Lockwood, L., Software for Use, Addison-Wesley, 1999.

Hohmann, L., GUIs with Glue, in preparation as of 2000.

Roberson, S. and Robertson, R., Managing Requirements, Addison-Wesley, 1999.

Wirfs-Brock, R., Wilkerson, B., Wiener, L., Designing Object-Oriented Software, Prentice-Hall, 1990.

Articles referenced in the text.

Beck, K., Cunningham, W., "A laboratory for object-oriented thinking", ACM SIGPLAN 24(10):1-7, 1989.

Cockburn, A., "VW-Staging", http://members.aol.com/acockburn/papers/vwstage.htm

Cockburn, A., "An Open Letter to Newcomers to OO", http://members.aol.com/humansandt/papers/oonewcomers.htm

Cockburn, A., "CRC Cards", http://members.aol.com/humansandt/papers/crc.htm

Cunningham, W., CrcCards", http://c2.com/cgi/wiki?CrcCards

McBreen, P., "Test cases from use cases", http://www.cadvision.com/roshi/papers.html

Online resources useful to your quest.

The web has huge amounts of information. Here are a few starting points.

http://www.usecases.org

http://members.aol.com/acockburn

http://www.foruse.com

http://www.pols.co.uk/usecasezone/