Writing Effective Use Cases

Alistair Cockburn
Humans and Technology

in preparation for Addison-Wesley Longman, Q3 2000.

Prologue

There are still no trusted guides about how to write (or review) use cases, even though it is now nearly a decade since use cases have become the "norm" for writing functional requirements for object-oriented software systems, and are gaining acceptance for embedded software and business process reengineering work. I can say from firsthand attempts that is very difficult to articulate what makes a use case "good", let alone how to write them so they will come out being "good". This view is shared by many teachers of use case writing. The problem is that writing use cases is fundamentally an exercise in writing natural language essays, with all the difficulties in articulating "good" that comes with natural language prose writing in general.

These are the guidelines I use in writing and coaching. The ideas came from listening to my on the fly inventions, describing to beginners what is going wrong, how to think and observe to get the writing to improve. This book has those guidelines, examples of use cases, variations that make sense - - and best of all, the reassurance 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

This book is predominantly aimed at the professional practitioners who read and study alone. For these people, the book is organized as a self-study guide. There are introductory, intermediate and advanced concepts, reminders and rules, examples, test questions with answers, and a set of discussions around frequently asked questions.

The second group of people for whom this book is intended are consultants and coaches looking for explanations and sample use cases to show their project teams ways, to help them get better at their writing.

The third group are instructors and their students. The instructors can build course material around the topics and exercises in the book, and issue reading assignments as needed. However, as I include answers to most exercises, they will have to construct their own exam material :-).

Organization

The book is written in 16 parts, to address the different needs of the various readers:

1. **The One-Page Summary.** Here it is in condensed form: what really matters, boiled down to one sheet you can photocopy and crib from. If you can understand and practice these sentence, I claim you can write as good a use case as you need at the time (see "Diminishing Returns").
2. **Base Concepts**. The elements of use cases, from actors, through scope and goal levels, to postconditions and failures, sub-use cases. This section of the book provides the ABC's of use cases.
3. **Related concepts.** Building on the base concepts, what are different ways to write, how does UML work, what about quality, tools, project planning, and the information that didn't go into the use cases.
4. **Reminders**. Key phrases to remind yourself with. Assuming you know the ABCs of use cases, you will still need constant reminding about the Do's and Don'ts of good use case

writing, little rules showing what is better, and what is worse. What strikes me as remarkable, writing these down, is how very many of them there are - which somewhat explains why it has taken us so long to articulate them. Hopefully, someone will come up with a shorter adequate list in the future.

5. **Writing samples**. The best way to learn is to see "good" examples. They answer many little questions and gives us templates to build upon.
6. **Mistakes fixed**. We also learn from comparing the "worse" with the "better". These show us the gradients the writing lives along, how to improve from wherever we start.
7. **Answers to exercises**. For the self-studying soul. Here are the answers I gave to the questions. I reserve the right to find ways to improve them, and hope you are exacting enough to find things wrong with them, also.
8. **Process matters**. There are several process matters to take of, although not all of them matter terribly. I reprint the excellent experience report of Andy Kraus, and give my own, personal approach to the people and process of writing use cases. There are certainly other, valid ways of working, but it is useful to see at least one.
9. **Other voices**. Even people who run the same philosophy as I, do things a little bit differently. From Bruce Anderson's "requirements prolog" and "business process documentation", to Susan Lilly's description of how she manages use cases, these are the voices of people who do things a little differently, but still well.
10. **Project standards.** Understanding everything in the book, it is still useful to have some ready-made use case templates for the project - slightly different for different sorts of projects.
11. **Email answers**. Over the years, people have written to me, asking for answers to very specific questions, such as how many angels dance in a properly formatted use case. Seeing these questions and answers may help you to pinpoint an issue of interest, or work though your own situation.
12. **Related work on use cases.** Where else can you look for information?
13. **Glossary.** Definitions of terms.
14. **Where do I find answers on ?** Multiple indexing is part of the value of this book. Look up answers by topic.

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 way people were working. I took Ivar's course on use cases in the early 1990's, as part of evaluating Ivar's Objectory methodology for the IBM Consulting Group. While neither Ivar nor his team used teh words "goal" and "goal failure" at that time, it is clear in retrospect, from the coaching they gave me, that these notions were operating in the subtext.

I formulated the Actors & Goals model of use cases in 1994, while writing the use case guides for the IBM Consulting Group. I needed the model, because I still couldn't understand what Ivar was saying. This model explained a lot of what he was doing, and gave me good guidance as to how to structure, limit, and write use cases. The intent of the model was to match what Ivar was doing, not invent a new form. In several comparisons, he and I found that although my terms are a

little different, there are no basic contradictions between his and my advice. The model has circulated informally from my web site (http://members.aol.com/acockburn, now moved to www.usecases.org) since 1995, and it finally appeared in JOOP in 1997, as "Structuring use cases with goals".

From 1994 to 1999, I found nothing more to say about use cases that wasn't already in the article. Finally, while teaching and coaching, I started to understand 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 course with Ivar!). This, plus a few small 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. My interest with the UML standardizing group was just to see that they did not mess up a lovely idea, cluttering up a fundamentally textual description with too much graphics and strange ideas emerging from a committee. Fortunately, a former colleague of Ivar's was responsible for the use case section. Although he somehow felt bound to emphasize the graphics component of use cases for the UML committee, he understood that the life of a use case is in its text, not in the graphics. He and Ivar 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.

On the place of this book in the "Crystal" collection

"Crystal" is a collection of books centered around lightweight, high-tolerance, human-powered methodologies. The collection consists of books that stand on their own, but complement each other. Some books address team collaboration issues, some address the skills needed by various roles on the project. This book is a *technique* guide, sharing details of how to write use cases.

All of the books work from the view that software development is fundamentally a cooperative game of group invention and communication, that it comes from people working well alone and working well together. To improve software development, we improve people's personal skills and the team's collaboration effectiveness.

The second cornerstone of Crystal is that different projects have different needs, working from different group values and priorities, with different numbers of people on systems with wildly different characteristics. It cannot be possible to describe the one, best way of producing software. This view is reflected throughout the Crystal collection, and also in this book. This book describes the core techniques of use case writing, but you will find that the template recommendations, in particular, vary according to project needs.

The base book for the collection is Software Development as a Cooperative Game. That book describes the idea of multiple methodologies, software development as a cooperative game, and separating out the various aspects of methodologies for different roles. The essence of the discussion, as needed for use cases, is in this book under *Precision, Tolerance, Hardness*.

On the samples in this book

I should like to claim that the samples of use cases I include are "good" ones, worthy of copying. That is a difficult claim to make. I quite expect that some alert reader will apply some of my own rules to my samples, and find ways to improve them. That sort of thing happens to me all the time in my courses. But improving one's writing is a never-ending task, and the main point is to communicate with the readers of the use cases. So I accept the challenge and any criticism.

I have taken use cases from actual projects wherever possible, covering system details and names for anonymity, but keeping the writing style. I hope that by doing this, you will see how other people are actually writing use cases on live projects, and what I consider viable alternative writing styles.

Acknowledgements

Thanks to lots of people

I wish this book were shorter. I didn't suspect, when I started, that there were a dozen base concepts to cover, two dozen related concepts and so many related topics. Still, you will run into situations I didn't cover.

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. Adding those topics made the book longer, but hopefully it will help settle some of the rampant confusion about the subject.

Special thanks to Pete McBreen for being the first to try out the Stakeholders & Interests model, and for adding his usual common sense, practiced eye, and suggestions for improvement. 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 for the use case samples. Thanks to the Silicon Valley Patterns Group for their careful reading and educated commentary on various papers and ideas.

High-Level Table of Contents

Detailed Table of Contents

The one page Summary

Work top-down: Scope, Actors, Goals, Main story, Alternative conditions, Alternative paths.
Work middle-out: Start at user goals, go up to strategic level, then down to subfunctions.
Get clear about the scope of the system you are describing.
Brainstorm all the actors, human and non, who have an operational goal against the system.
Brainstorm all their goals against the system, over the entire lifetime of the system.
Double check for time-based and other events that cause the system to react.
Write the actor-goal list, double check it, prioritize, merge, reconsider goals.
Recognize that the actors are only there to help you find all your use cases.
Work the scope outward: define the system that contains the system under discussion.
Extend each goal outward to find out *who really cares* about achieving that goal.
Draw the primary actors and their goals against the outermost containing system.
Add those to the list of use cases to write.
Review all the above, prioritize and reconsider.
Pick a use case to write.
Brainstorm the stakeholders.
Write how their interests are satisfied at the successful conclusion of the use case.
Write what interests must be protected in case there is failure of the use case.
Write the precondition: what the system has ensured is certainly true.
Write what event or thought triggers the main success scenario.
Write the main success scenario.
Write each sentence as a goal succeeding, distinctly moving the process forward.
Write: "At some time, ActorA kicks ActorB with some information", or
"System verifies validation conditions are met", or
"System updates its state", or
"ActorA has ActorB kick ActorC".
Show the intent of the actor, what it/they want and get accomplished in that step.
Avoid user interface descriptions.
Manage the level of goal accomplishment so that the scenario is between 3 and 11 steps long.
Verify that the interests of all stakeholders are fully met.
Check that the sequencing requirements in the steps - or lack thereof - are clear.
Brainstorm the failures and alternative paths.
Include only the failures the system must detect and handle.
Write the failure or alternative condition as a condition phrase or sentence.
Write a scenario fragment showing how the alternative course leads to goal failure or success.
Write using the same rules as for the main success scenario.
Update the main success scenario with the new validations found while writing the extensions.
Put into a sub use case any sub-goals that got too complex to fit neatly in this use case.
Let the goals-becoming-use cases be your unfolding story.
Remember, even a fairly bad use case is still a good use case, so don't worry.

Preliminaries

This section contains some preliminary information to get you warmed up for use cases, such as: What does a use case look like? How do we warm up for writing use cases? Where do they fit into the overall requirements gathering work?

Why would different projects need to write them differently?

It also contains some theory you can skip over, and come back to when you are up to it. Feel free to bounce between this chapter and *Basic Concepts*, picking up this background information as you need.

1. What does a use case look like?

Here are two sample use cases, written at different levels, for different needs.

Use Case MO1. Register arrival of a box
RA means "Receiving Agent".
RO means "Registration Operator"
DS means "Department Supervisor"

Primary Actor: RA
System: Nightime Receiving Registry Software
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.

Use case example 1. A system use case, user-goal level.

Use Case: Get paid for car accident
Design Scope: The insurance company ("MyInsCo")
Goal Level: Strategic
Primary Actor: The claimant
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. Agent verifies all details are within policy guidelines
5. Insurance company pays claimant

Extensions:
1a. Submitted data is incomplete:
1a1. Insurance company requests missing information
1a2. Claimant supplies missing information
2a. Claimant does not own a valid policy:
2a1. Insurance company 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.

Use case example 2. A business use case, strategic goal.

More examples of use cases, written for different purposes and in different styles, are shown in the section, *Writing Samples*.

Different kinds of use cases

"Use case" is merely a form of writing. That form of writing can be put to use in different situations: to describe a business' work process, to serve as a basis for discussion about upcoming software system requirements (but not be the requirements), to be the actual functional requirements for a system, or to document a design.

In these various situations, slightly different templates and styles might be used, and yet, there is a common thread running through all of them, that of describing actors achieving sub-goals.

Because there are so many situations in which use cases can be used, it become difficult to describe any part of all of them in any one phrase or chapter. I list the different sorts of use cases below. In the rest of the book, I'll mostly say what they all have in common, but when I show an example, I have to name what kind it is.

The dimensions of difference are:

- casual / dressed

- business / system
- corporate / computer system / system innards
- white-box / black-box
- strategic / user-goal / subfunction

These will all be explained over time. The thing to see is that there are 72 combinations of those characteristics. So three of you comparing notes could each have come from reading or writing

casual business corporate white-box strategic
dressed system corporate black-box strategic
dressed system computer-system black-box user-goal.

So, of course you could disagree on some matter of writing.

In the two examples of use cases above, I picked two fairly typical use case forms. Both were "dressed", i.e., used a sentence numbering scheme and named fields in the template (as opposed to "casual", which short-circuits some of the numbering and fields).

The first is intended as software functional requirements, and so is a system use case written at computer system scope, although the writer put in some contextual activities, just to ease understanding. So that use case is:

dressed, system, computer-system, black-box, user-goal.

The second is intended as a business process description, written prior to software functional requirements. Hence, the computer does not show up in the use case. Unlike some business process descriptions, this one treats the entire insurance company as a single entity (other business use cases show *how* the organization reacts to the external requests - they are "white-box"). So this one is

dressed, system, corporate, black-box, strategic

Trying to find a way to talk about use cases in general, while allowing the 72 different particular use cases to be written will plague us throughout the book. The best I can do is outline it now, and let the examples speak for themselves.

2. What is a use case?

A use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.

This is the first, rough definition, which will serve us for a long time. It is missing the fact that the system has responsibilities toward the stakeholders of the system, and capturing those responsibilities is part of the requirements-writing work. However, for the moment, we can trust that our use case writers know that and will take care of capturing those items without special prompting.

The working definition tells us we need to elaborate on what we mean by "description", "sequence of interaction", "system under discussion", "actor", and "goal". The definition does not say anything about just *when* the use cases are written, what the size of the system might be, or the format used.

In general, this book is concerned with *how* the use cases are written, and not *when*. As a writing technique and a thought-capturing format, use cases are suited to describing the behavior

of *any* system that interacts purposefully with its external environment, and can be written at any time.

- They have been successfully used to capture the business process design of companies and organizations (see, for example, Ivar Jacobson's Business Process Reengineering with Use Cases, the two use cases shown as examples above, and Writing Sample 1). In such a case, the external actors are the customers and vendors of the company, and the processes of the organization are being *documented* or *redesigned*.
- Use cases have been successfully used to *document* the behavior of software systems, often just before a new round of maintenance work is done.
- They have been successfully used to specify the requirements of software systems. In some companies that is done before design work is started, but I have been shown situations when the requirements were so inextricably tied to implementability, that the use cases more "documented" the agreed-upon to-be-designed behavior than "specified" what the designers were supposed to figure out how to build.
- They have been successfully used to specify the requirements of combined hardware-software systems, from very large military complexes, to small, embedded real-time controllers.
- They have been successfully used as *documentation* of the behavior of internal subsystems (see Writing Sample 4).

What all these uses of the use case idea have in common is that they show how the system responds to the outside world, the responsibilities and behavior of the system, without revealing how the internal parts are constructed. The nice thing is that "writing a use case" is common across all of the above situations: in all cases, we want to write as little as possible, as clearly as possible, showing the ways in which the system reacts to various situations. Therefore, in this book I do not separate out how to write use cases for the different systems and situations. The examples in the book show different types and sizes of use cases.

The next thing to get comfortable with is that use cases are fundamentally and historically a text-based documentation form. They can be written using Petri nets, flow charts, sequence charts, pseudo-code, or simple text. Under normal circumstances, their main purpose in life is communicate from one *person* to another. Often the two people have no special training in informatics, and so textual use cases come out as the best choice - they can written and read by anyone. Text has strong consequences for the tool set used: text is much harder for the tool to manipulate than drawings, while drawings are harder for the people to work with. The tool issue is covered in the tools section. For now, it is sufficient to be aware that use cases are mostly text.

A use case is a description of the interactions and responsibilities of a system, the "system under discussion" or "system under design", with external agents, or actors. An actor may be a person, a group of people, or a computer system. The use case is associated with the goal of one particular actor, who is called primary actor for that use case. The use case describes the various sets of interactions that can occur between the various external agents, or actors, while the primary actor is in pursuit of that goal. It also describes the responsibilities of the system under design, without getting into implementation techniques or system components. Each possible sequence of

interactions is called a scenario. The use case collects together all the scenarios related to that goal of that primary actor, including both those in which the goal is achieved, and those in which the goal must be abandoned. I expand on each of these ideas in the relevant section.

3. The System-in-use story

A *system-in-use story* (or *user story*, as it is sometimes called) is a situated example of the use case in operation - a single, highly specific, example story of an actor using the system. It is not a use case, and in most projects it does survive into the official requirements document. However, it is a very useful device, worth my describing, and worth your writing.

When getting started on a new project, you as the writer of the use case may have either little experience with use case writing or may not have thought through the operation of the system. To get comfortable and familiar with the material, sketch out a *vignette*, a few moments in the day of the life of one of the actors, a system-in-use story.

In this short story, 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. The writer simply writes "how the world works, in this particular case", from the start of the situation to the end.

The system-in-use story anchors the use case. The use case itself is a dried-out form of the system-in-use story, a formula, with generic actor name instead of the actual name used in the system-in-use story.

The place of the system-in-use story is to envision the system in use. You might also use it to "warm up", before writing a use case, or to work through the details of action that you will capture in the use case. In a few organizations I have visited, system-in-use storys are presented at the beginning of the use case chapter, or just before the specific use cases they illustrate. The people who have done this have been happy with their choice. The system-in-use storys take little energy to write, little space, and lead the reader into the use case itself easily and gently. The system-in-use story is not the requirements, rather, it sets the stage for more detailed and generalized descriptions of the requirements.

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 system-in-use story.

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.

Example of a "system-in-use story".

On rare occasion, a (very!) low-ceremony project will actually use system-in-use storys as their requirements. The characteristic of such a project is that 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. The system-in-use story acts as a "promissory note" for a future conversation, in which the requirements owner will elaborate the details and boundary conditions of the requirements for that story. This can work well (indeed, it is formal part of the Extreme Programming methodology, see Kent Beck's Extreme Programming Explained), if the conditions for being able to fulfill on the promissory note are met. In most projects they are not met, and so the system-in-use story acts as a warm-up exercise at the start of a use case writing session.

Exercise 1. Write two system-in-use storys 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 2. Write a system-in-use story for a person going into a brand new video rental store, interested in renting the original version of "The Parent Trap".

Exercise 3. Write a system-in-use story for your current project. Get another person to write a system-in-use story 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?

4. Requirements files vs. use cases

The use cases form part of the requirements file, they are not the requirements file itself. The requirements file contains several parts, and may be organized as a single paper document or as a subdirectory with files. These days, it tends to be the latter, and so I refer to the "requirements file".

The requirements file is likely to contain approximately the following sections. Each organization, even project, may have slightly different contents to fit their needs. The following, sample requirements file outline is one I adapted 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). I found the published template fantastically complete but intimidating in its completeness, so I cut it down to the following form. This is still too large for most of the projects I encounter, and so we tend to cut it down further as needed on projects. However, this template, as the original from the Atlantic Systems Guild, 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".

It is not the role of this book to standardize your requirements file, but I have run into so many people who have never seen a requirements file outline, that I pass along this outline for your consideration. The main purpose in including it is to illustrate the place of use cases in the overall

requirements file, and to make the point that use cases will not hold the entire requirements for the project, but only the behavioral portion, the "required function." For more detail on the subject, read the Robertsons' book.

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 use cases
2a. The primary actors and their general goals
2b. The business use cases
2c. The system use cases

Chapter 3. The terms used / Glossary

Chapter 4. The technology to be used
Q1. What technology requirements are there for this system?
Q2. 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 requirements are there on the development process (e.g. testing, installation)?
Q6. What dependencies does the project operate under?
5b. Business rules
5c. Operations, security, documentation
5d. Use and usability
5e. Unresolved or deferred
5f. Maintenance and portability

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, what dependencies are there on the human environment?

Sample requirements file outline.

The thing to note is that use cases only occupy chapter 2 of the requirements file. Most people coming to learn how to write use cases somehow have formed the impression that use cases *are* the requirements. However, use cases cover only the "behavioral" aspects of the system, i.e., the functional 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.

Having said that, the *names* of use cases provide a terrific framework on which to hang many other project details, such as priority, teaming, release dates, status, performance requirements and interface channels. They also link together bits of information that would otherwise be hard to connect together, such as user characteristics, business rules and data format requirements. While these are not *in* the use cases directly, they are associated with use cases. The use cases act as the hub of a wheel, and the other, different sorts of information act as spokes leading in different directions. It is for these reasons that people often consider use cases as the central element of the requirements file.

Exercise 1. Discuss with another person which sections of the requirements file outline are sensitive to use cases, and which are not.

Exercise 2. Design another plausible requirements file outline. Organize it for web access, complete with subdirectory structure and date-stamping conventions (why will you need date-stamping conventions?).

5. Precision, Harness, Tolerance

This is the section of the book with the hardest theory. Come back to it when you need it. However, you should understand how to manage your energy, and why other people have different needs than you do. The essence of the story is that high precision is rarely needed and *Your use case is not my use case.*

Manage Precision and Energy

Save your energy. Or at least, manage it. If you try to write down 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 will

- 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.

When people 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. Saying *pi* is 4.141592 is speaking it with great precision, but simply wrong, inaccurate. Saying *pi* is "about 3" is not very precise - there aren't very many digits showing - but it is, at least, accurate for as much as was said. The same is true for use cases.

You will eventually add details to each use case, adding precision. But if you happened to be wrong (*inaccurate*) with your original, low-precision statement of goals, then the energy put into the high-precision description was 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, matching levels 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, or 1-bit, of precision.
2. *Main Success Scenarios*. For the use cases you have selected to pursue, write the stakeholders, trigger and main success scenario. These you can review in draft form to make sure that the system really is delivering the interests of the stakeholders you care about. This is the 2nd level of precision on the functional requirements. It is fairly easy material to draft, unlike the next two steps, which take a lot of energy.
3. *Failure conditions*. Brainstorm all the failures that could occur. Draft this list completely before working out how the system must handle them all. Once again, filling in the next step, failure handling, will take very much more energy than simply listing the failures. People who start writing the failure handling immediately often run out of the energy needed to finish 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 is therefore a priority in how you work.

In Surviving Object-Oriented Projects, I introduced the notions of *precision, accuracy, relevance, tolerance, scale*. The above list talks about precision (amount of detail) and a little about accuracy (correctness). Later in the book I shall cover scale (hiding and combining to cover more ground in a small space). Which leaves us with tolerance (how much variation is permitted from use case to use case), and a related topic, new to me in the recent years, but relevant to use case writing, *hardness*.

Tolerance and hardness

On a given project, not all use cases need to written exactly the same way. I rather doubt that you will be able to get all the writers to use the same grammar style, indentation, punctuation, naming conventions, page breaks, and so on. Nor should you. Certain projects can *tolerate* more variation in writing style from one use case to another. How much can your project tolerate?

What would it mean if two of your people chose significantly different templates for their use cases? It would probably be disturbing, at the very least. On a large project, it could cost quite a lot of time, money, energy. If you can withstand different templates being used, then your project can handle a wide range of variations, or large tolerance.

Suppose that although using the same template, one person always writes from the point of view of a bird on the wire, sitting above the action and seeing all actors, while another person always writes from the eye-view of the system under design, describing only the outsiders' actions. Or, suppose that the same person sometimes writes one way, and sometimes the other way.

Suppose that although using the same template, some people fill it out completely and some people don't.

What is the correct response to all those different situations? The answer is that there is no fixed answer. Not all projects need the same amount of precision, and not all projects need the use case templates filled out the same way.

In UML Distilled, Martin Fowler introduces the marvelous notion of *high-ceremony* vs. *low-ceremony* practices. A low-ceremony practice is more casual, a high-ceremony is more rigorous, careful and less tolerant of variation. A low-ceremony practice relies on person-to-person communication to cover missing information and variation across practitioners, a high-ceremony practice relies on uniformity in the work-products and formalized activities.

In Software Development as a Cooperative Game, I characterize the *weight* of methodologies according to their *size* and *density*, keeping with the mechanical vocabulary. *Size* has to do with the number of elements in the methodology. *Density*, or *hardness*, as I use for a synonym, has to do with the ceremony, effort and rigor involved. *Hardness* and *ceremony* allow me to say what I need to say next.

Your use case is not my use case

On a large, life-critical project, such an atomic power plant, it may be appropriate to use a *hardened* form, a larger, fancier template for the use cases (a *high-density* use case form). Further, it may be appropriate to encourage or require people to adhere to that template closely, and to follow conventions of naming and grammar closely (*low tolerance*). Then, it may be appropriate to have a number of formal reviews of the requirements (*larger* methodology, with more *ceremony*). On this project, the use cases are high-density (or hardened), low-tolerance, the process is high-ceremony. Each of those characteristics costs time, money, energy, and, significantly, the people paying for the project feel they are an appropriate use of time, money and energy, given the characteristics of the project - coordinating lots of people on a life-critical system.

On a small project, four to six people building a system whose worst damage is the loss of a bit of comfort, easily remedied with a phone call, the above work would be considered an inappropriate use of time, energy and money. For such a project, a much simpler, casual (*low density, softer)* template would be chosen. In that template, certain information would be optional (*higher tolerance*) or left out entirely. The writing style might or might not be legislated (*higher tolerance*). As Jim Sawyer said in an email exchange,

> "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."

Neither style of writing is wrong. The selection of what to use 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. Not that we haven't been saying, "One size doesn't fit all" for years, but just how to translate that into concrete advice has remained a mystery and a trap for methodologists.

For this reason, I have slowly come to the conclusion that it is incorrect to publish "a use case template". There must be at least *two* use case templates, a "casual" template for low-ceremony projects that cannot even use the simplest variant of the full template, and a "fully dressed" template that can be adapted to the needs of the range of higher-ceremony projects, and. The two boxes below show examples of the same use case written in the two styles.

Use Case 1: Buy something

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.

Example of the "casual" template applied to a "white-box" business use case.

Use Case 1: Buy something

Context of use: Requestor buys something through the system, gets it.

Scope: Corporate - The overall purchasing mechanism, electronic and non-electronic, as seen by the people in the company.

Level: Summary

Preconditions: none

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

Failed End Protection: Either order not sent or goods not being billed for.

Primary Actor: Requestor

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.
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.*

Deferred Variations
none

Project Information

Priority	Release Due	Response time	Freq of use	
Various	Several		Various	3/day

Calling Use Case: none
Subordinate Use Cases: see text
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?

Example of the "fully dressed" template applied to a "white-box" business use case.

Neither use case template is wrong. It is for the project team to read this book and come to a decision as to whether to adapt the casual template, or adopt a convention as to how to use the **fully dressed** template. They must also decide how much variation to tolerate across the use cases, and how much or how little ceremony to use in the reviewing process. This is a subject of *project standards*, which must be selected on a project-by-project basis.

Saying that "use cases will be used on the project" is therefore an insufficient phrase, and any recommendation or process definition that simply says, "use 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 is permitted.

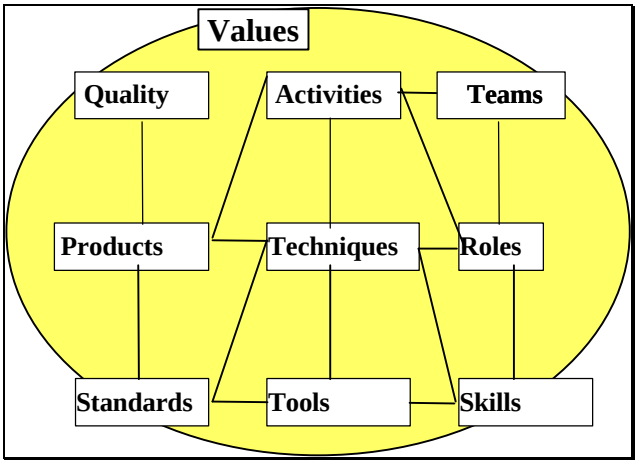


Figure "Methodology Parts".

The full discussion of tolerance, variation across projects, and how to choose for yourself, is given in Software Development as a Cooperative Game. We don't need that full discussion in order to learn how to write use cases. We do need to understand the place of writing use cases in the overall work of one project, and how the recommendations shift from one project to the next. Here is the briefest review of "Methodology" and how the contents of this book fit in.

A methodology has on the order of ten basic parts, as shown in Figure "Methodology Parts". The three parts most relevant for us now are "products", "standards", and "techniques".

The "product" that we are discussing in this book is "functional requirements". The first standard being applied is "functional requirements are captured in use cases", which makes use cases our work product of concern.

Three things attach to the work product called use cases.

- "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 describe work at a moment-to-moment basis, and so are to a large extent independent of the size of the project. Therefore, a book on technique can be useful to both large and small projects, and a person skilled in the 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. In this book, I describe the best way of writing I have seen, for each part of the use case template, and across the use cases as a set. In the end, though, the way you evaluate the quality of your use cases depends on the tolerance, density and ceremony you adopt in the project. The Figure "Project Grid" shows one useful way to characterize projects:
- the number of people being coordinated,
- the potential for damage on the project, and
- the key priorities for development.

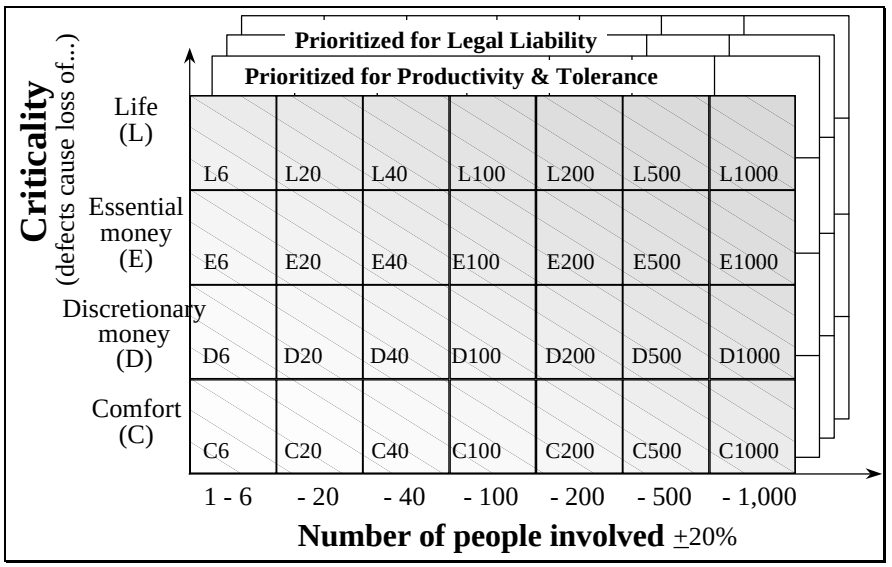


Figure "Project Grid".

On a project with low potential for damage, you can afford to save time, effort and money by working with low ceremony, using a casual use case template. Perhaps you still want people to write the same way (lower tolerance), otherwise the miscommunication will cost a lot in rework.

On a project with either greater criticality, I would expect you to want lower tolerance and more ceremony in the review process. You may or may not need to use the fully dressed template. On a project with many people, you will need to standardize carefully on the template you use,

whether casual or fully dressed, because however much you try, there will already be a lot of variation in the way so many writers write, and you don't want to confuse the many readers.

Therefore in a high-tolerance situation, you will accept the quality of a use case that will not pass the quality tests in a low-tolerance, hardened, high-ceremony situation. And so there is no way this book can legislate for you just what quality measures you should adopt, beyond some fairly simple recommendations.

Base Concepts

1. Actors & stakeholders

Actors

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.

A more correct word would be "role". A "role" would mean the role that a particular person is playing when they use the system, e.g., "invoice creator" or "order taker", or "manager". The word "actor" seems to imply to most people a particular person, when it really means "someone / thing playing an active part in the operation of a system." Ideally we would speak of "actor type" or "role", to indicate that we mean a category of person, perhaps a job description ("clerk") or relationship ("customer"). However, "actor" is the word the industry has accepted, and it works quite adequately.

Actors come in four flavors:

The system under discussion, itself. The system under discussion itself is an actor, a special one. We usually refer to it by name, or call it "system under discussion", "system under design", or SuD. It will not be a primary actor, or a secondary actor. It is just an actor, which we can take advantage of, at certain times.

The subsystems of the SuD, as *internal* actors. An individual object in the software is also an internal actor.

Normally, we deliberately write use cases so that the SuD is an unopened (black) box, which we cannot peek inside. Internal actors are carefully not mentioned, since normally they are not known at the time the requirements are being written, or we wish to specifically exclude the details of the design.

However, there are times when we open up the SuD, and use the use case writing style to describe how the pieces of the SuD work together to deliver the externally visible behavior. I shall call these "white box" use cases. Everything I cover 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.

Business use cases, in particular, tend to be written white-box style, to include internal actors (see the use case samples above, and Writing Sample 1). For computer systems, it is rare that we open up the black box.

The *primary* actor of a use case. The one and only primary actor of a use case is the external actor whose goal the use case is trying to satisfy, the actor whose action initiates the use case activity.

Careful readers will detect that the two halves of that sentence might lead to different results. We might detect that the person initiating the action is not the person with the goal.

Strictly speaking, I like to think of the primary actor as the stakeholder whose goal is the theme of the use case. The operator, in a sense, is a technological convenience for that stakeholder (I don't mean that in a derogatory way). As we shift technologies over the life of the system, it becomes more likely that that stakeholder will undertake the action herself or himself.

An example of a stakeholder non-initiator might be a customer calling and speaking to a clerk. Once we implement voice recognition, touch tone commands, or a web interface, the customer will trigger the use case directly, the clerk might disappear (more likely, the other technologies will augment the clerk). A second example is the Marketing Department of a company, or the Auditing Division. These departments might insist on the presence of some use cases, to be initiated by a clerk. It is not really the goal of the clerk to have the use case run, it just happens that they are the technological convenience for the managers of the departments.

Which is really the "primary actor"? To the first approximation, it doesn't matter. If you write Clerk, or Marketing Department, you will be fine. Strictly speaking, I prefer to write Marketing Department, because then I know *to whom* the use case brings value.

A less obvious example of a non-operator primary actor is time. Some use cases are triggered by time, say, midnight, or the end of the month. There is no operator initiating the action. To find the primary actor, we find out who cares that the use case actually runs at that time. Again, it helps to understand to whom the use case brings value. However, as one student astutely asked, "How much damage is there if we get this wrong?" The answer is, "Not much."

Below, I discuss how primary actors can be important and unimportant at the same time.

The *secondary* actor of a use case. A secondary actor of a use case is an external actor which provides a service to the SuD. We collect secondary actors to identify the external interfaces the system will use, and the protocol that crosses that interface. An external actor might be a secondary actor on one use case and a primary actor on another use case.

A secondary actor 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 (e.g., the coroner's office provides the insurance company with affirmation of a person's death).

"Secondary" does not mean "unimportant", any more than "primary" means "most important". On occasion, a person will call secondary actors primary actors, "because they are so important, I don't want them forgotten." They will not be forgotten, they will be listed as secondary actors, and their interfaces will be collected.

Why actors are unimportant (and important)

Actors are important at two points in the development of the system, 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:

- Their first value at the beginning of requirements gathering is to help us get our arms (or minds) around the entire system for one brief moment (it will escape us soon enough). We brainstorm all the actors we can in order to brainstorm all the goals we can. It is really the goals we are interested in, but if we brainstorm only the goals, we should miss too many. Brainstorming the actors first gives a working structure to traverse, to try to get all the key goals. Finding that several actors have the same goal does not hurt, since it is goal coverage we want. Similarly, discovering a little later that we have missed a goal during this part of the exercise should not depress us - we wouldn't have found that concealed goal anyway at an earlier time! Other parts of the use case exercise will flush out the hard-to-find goals.

- Their next value during requirements gathering is to focus our minds on the *capabilities* of 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, and just as importantly, the sort of background and expertise they have, so the system designers can match the system to their expertise.
- Their final value during requirements gathering is to prepare the Actor-Goal list, which will be used to prioritize and partition the development work.

Once we start developing the use cases in detail, the primary actors become almost unimportant. That announcement may come as a surprise, initially. Over time, though, the use case writers discover that a use case can be used by multiple types of actors. Everyone higher than a clerk can answer the phone and talk to a customer. So, they name the primary actor in an increasingly generic way, using terms such as "loss taker", "order taker", "invoice producer", etc. This results in use cases that say, "The invoice producer produces the invoice..." (not terribly enlightening, is it?).

An alternative is to say, somewhere in the front of the use case section, "The Manager can do any use case the Clerk can, plus others, and the Regional Manager can do any the Manager does, plus others. Therefore, wherever we write that the use case is Clerk (e.g.), 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." In UML, we would mark the Manager as a "subtype" of the Clerk, meaning that anything the Clerk can do, the Manager can do, also. This works in the case of Clerk and Manager, but doesn't work in the case that Sales Clerks and Auditor Clerks have overlapping sets of use cases they can run.

As a result, the "Primary Actor" field of the use case template becomes devalued over time. This is all right, so you needn't worry about it. It is really the goals we are after, and how they are delivered or abandoned.

At the end of system development, just before delivery, the primary actors become important again. We need the list of all the *people* who are use portions of the system, and which use cases they will have access to. We need these:

- to partition the system into delivery packages that will be loaded onto the various users' machines;
- to set the security levels for the various use cases (casual user, internal use, supervisor, etc.);
- to establish a training curriculum for the various users, according to job classification, expertise, seniority, or however it might be done.

And so, at the beginning, we act terribly concerned over the completeness and correctness of the primary list. The potential damage of missing an actor is to leave out a section of the requirements entirely, with a cost of late discovery that is much higher. The potential damage of having too many actors is merely a bit of extra work in the early stages until the unnecessary actors are eliminated.

Then, after brainstorming the goals, the primary actors become reasonably unimportant. Fussing over the exact and correct name for the primary actor adds little to the value of the behavior description, and will be corrected in the last stage anyway.

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 have no special use cases for themselves. They may never interact directly with the system at all, but still have a right to care how the system behaves. Examples include the owners of the system or company, and regulatory bodies. The business rules reflect the interests of the different stakeholders in the use cases.

The board of directors of a bank do not interact directly with the ATM in the wall, except as ordinary customers. The Internal Revenue Service does not type anything special into the ATM, just as the Dept. of Insurance of the federal government does not type anything special into the insurance company computer. Again, beginning use case writers like to include the "owner of the coffee vending machine" as a primary actor, when they mean that the owner is an important stakeholder.

Stakeholders do not appear directly in the text of a use case. A use case is a description of behavior, and the stakeholder does not act in the use case, except in their role as a primary or secondary actor. The interests of the stakeholders are captured in the behavior of the system, the checks and validations it performs, the logs it creates, and the actions it performs.

All that being said, stakeholders are important to the requirements of the system. The use case is a form of contract, in which the possibly conflicting interests of all the stakeholders are reconciled. The specified behavior of the system under discussion must capture a mutually acceptable set their peculiar interests.

The section *Refined Model of Use Cases* discusses this further. For most writers, however, it is sufficient to master two ideas:

- A stakeholder is not a primary actor, unless that stakeholder/actor interacts directly with the system in that capacity.
- The behavior of the system under design must protect the interests of the stakeholders, who are not present to defend their interests themselves.

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

Exercise: We are hired to design an ATM (automated bank teller machine). Identify which of the following are actors in our use cases, saying whether each: is a primary actor, is a secondary actor, is the system under design, or is not an actor for our purposes (or is multiple of the above).

The ATM
The customer
The bank teller
The ATM card
The bank
The front panel

The bank robber
The bank owner
The serviceman
The printer
The mainframe
The network link

Exercise: The ATM is contained as part of a larger system. In fact, it is part of several larger systems, one of which is the bank to which it belongs, complete with buildings, people, machinery, and so on. Repeat the previous exercise on this larger system.

Exercise: Personal Advisors, Inc. (a hypothetical company) is coming out with a new product, which will allow a person 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 scenarios to learn how to optimize their 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 the actors, primary actors, secondary actors, system under design, containing system of PAF.

2. The Actor List

A relatively valuable product of the requirements gathering work is the "Actor List". The Actor List has two parts:

NAME	CHARACTERIZATION
The name of the role, or the job title, or the actor type	A characterization of the skills and possibly goals of that actor type.
(e.g.) Customer	Person walking in 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.
(e.g.) Returned Goods Clerk	Person working with this software continuously. Touch-types, sophisticated user. May want to customize UI.
(e.g.) Manager	Occasional user. Used to GUIs but will not be familiar with any particular software function. Impatient.

I say the Actor List is relatively valuable, but relatively few project bother to produce this list, so I have to rate it as "relatively valuable, but not crucial". The people I have interviewed who

have created the Actor List claim to have a better control overview of how their software is shaping up, and the likelihood of it meeting the needs of the end users (because they have thought about the characteristics of their end users, and have those characterizations in front of them during development).

3. Design Scope.

Scope is the word we use to indicate the extent of what we consider "to be designed by us", as opposed to "already existing" or "someone else's design job". There are two kinds of "scope": functional scope and design scope.

Functional scope

Functional scope is what functions you choose to deliver or not deliver. The way to manage functional scope is with a Goal List. You simply mark each goal as "In scope", or "Out of scope" (possibly, you might add a third category, "In Next Release"). In this book I shall not refer to functional scope again, because it is so straightforward and causes little confusion. When I write about "scope" I intend design scope.

Rob Thomsett introduced me to an excellent discussion-management technique, the "in/out list", a generalization of the goal list. You 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 remarkable thing, as Rob related and I have seen, is that is it completely clear to each person whether the topic is in or out... they just have opposite views. Rob relates that sometimes it takes an escalation up to the steering board as to whether some particular topic really is inside the scope of work or not, and that the difference is sometimes many staff-months of work.

Here is a sample in/out list we produced for a purchase request tracking system. Try this little techique out on your next projects, or perhaps your next meeting!

Item	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	

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 the boundary of the system. If we are charged with designing an ATM, we are charged with producing hardware and software that sits in a box. The box and everything in it is ours to design. The computer network and networked computers the box will talk to are not ours to design. They are "out of the design scope of the ATM we are charged with designing."

From now on, I shall simply say "scope" to mean "design scope", because design scope is a non-standard term. I just need a word to distinguish it from functional scope.

This "Scope" or "design scope" is incredibly important in the use case. Systems come in all sizes, and use cases are used to describe the behavior of any of them, from corporations down to object-oriented frameworks. When your readers start reading a use case, they must be very clear as to what you intend is *inside vs. not inside* the system.

Whether the networked computer is inside or outside the scope of a use case about the ATM makes an enormous difference. Suppose we are bidding on the design of the ATM, and guess wrong as to whether the network computers are in scope. Our bid will be off course by a factor of 2 or more, with disastrous results for the outcome of the contract.

A small, true story:

We were about half-way through constructing the bid of a large system, a fixed-time, fixed-cost bid. To help with the sizing, we walked through some sample designs, using CRC card and interaction diagram techniques. At one point, I picked up the printer and spoke its responsibility and function. The IS person on staff laughed out loud and said, "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 bundle!"

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

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

We quickly corrected course, and found that there was a complicated two-day interface to the printing system. On day one, our system prepared a magnetic tape with the things to be printed. At some time, the printing system read the tape and printed whatever it could. It prepared a tape containing the results of the printing job, complete with error records for anything it couldn't print. On day two, our system read the results back in, and discovered what had been printed correctly and what not. The design job for interfacing to that tape was significant, and completely different from what we had been expecting.

The printing system was a "secondary actor", out of design scope. Had we not detected that, we would have written the use case to include it in scope, and set to work building the wrong system.

Typically, when one person writes a use case, they know, in their head, what the design scope of the system is, what is in and what is out. It is so obvious that they don't mention it. However, once there are multiple writers and multiple readers, then scope is not at all obvious. One writer is thinking of the entire corporation as the design scope, 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 "requirements" document.

What can we do to clear up these misunderstanding?

The answer I have found is to give names to the different design scopes, and to *label each and every use case with its design scope*. Here are the names I use. You might think up slightly better names for your situation. Just be clear to broadcast your definition.

- "**Corporate**" (or "**organization**") scope means that the entire company is being covered in the discussion. We are discussing how the corporation behaves to delivery the goal of an external actor. A smaller scope one might occasionally use is "department", when we are discussing how one department gives value to actors outside the department. *Business use cases* typically are written at corporate or departmental scope.
  Exercise: Describe the difference between "*corporate-scope white-box business use cases"* and "*corporate-scope black-box business use cases"*.
- "**System**" (or better yet, put the system's name in here) scope 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 you are to interface with.
- "**Subsystem**" (or better yet, put the subsystem's name in here) scope means that you have opened up the system that is the main one being designed or discussed, for whatever reason, and are about to talk about how a piece of it works. I am not sure why you would do this, but if you do, please label it.

I always recommend writing at least one corporate-scope use case. This use case shows the value of the use case to some actor outside the company, and typically ties together many of the system-scope, lower-level use cases (more on this when we get to goal levels). In general, I advocate that use case writers find the *outermost primary actor* for every use case, and write a high-level use case showing how the SuD satisfies the goals of that actor.

There tend to be very few of these high-level, wide-scope use cases, and they are interesting. We often find that the IT department has the computer security use cases, the marketing department has the advertising use cases, and the customer has the main system function use cases.

Example of design scopes.

Let us take a fictitious but typical example. A telephone company, call it MyTelCo, is designing a new, 3-tier computer system to take orders for services and upgrades. The clerk will use a workstation. The workstation is connected to a server, a Unix box, which is connected to a mainframe. Currently the clerk uses a terminal attached to the mainframe directly. The new system, consisting of the workstation and Unix box, is called "Acura". The older system, consisting of the mainframe, is called "BSSO".

We start by listing the primary actors for Acura. The primary actors include the customer, clerk, various managers, and BSSO, the mainframe system. We are clear that BSSO is not the system under design. In fact, we are not allowed to touch BSSO, we can only use its existing interfaces.

The most obvious goal is "Add new service". We decide the customer is the primary actor (or maybe we don't, maybe we decide the clerk is the primary actor!). We sit down to write the use case.

What design scope shall we describe? The novice writer often will write the use case for Acura's workstation component, but that is not what is really of interest to us. Two scopes are of interest to us:

- What does Acura provide for service at the moment of the request, as Acura interfaces to the clerk on one side, and to the BSSO system on the other side? This is the "system scope", which we should call "Acura". It includes both the workstation and server, since they are both part of the new design.
- What does Acura's service look like to the customer, even abstracting away the clerk, and showing the entire new service from initial request to implementation and delivery? This is "corporate" scope, which we should call "MyTelCo". It includes the entire company as a single black box, and shows how the entire request cycle appears to the customer and suppliers.

The design scopes "Acura's workstation" and "Acura's server" are typically not of interest to us. What I should say is they are not of interest to us *now*. When they are of interest, probably a different notation other than use cases will be used to document their interfaces. Use cases could be used, in which case some use cases will be written with the labels "Scope: Acura's workstation", and "Scope: Acura's server". My experience is that, typically, these use cases are never written, but a command-response protocol is documented in text, instead.

It is not for this book to say what use cases you should write. But whatever you write, make sure you label what design scope you are writing about.

Exercise: You are standing in front of an ATM attached to your bank. It is dark. You have entered your PIN, and are looking for the "Enter" button. Name the scope of the system under discussion. In fact, name 5 different possible systems you could be discussing or designing at this moment.

Exercise: What system are you busy 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. Draw a picture of the multiple scopes in action for an ATM, including hardware and software.

Exercise. Draw a picture of the multiple scopes in action for the PAF system.

Exercise. Draw a picture of the multiple scopes in action for a web - workstation - server - mainframe system attaching to a legacy system.

4. The Actors & Goals model

The basic model of use cases is that of actors having goals against other actors. This is the model I documented in the original article "Structuring use case with goals", in 1995, which is most familiar to use case writers. It is essentially what was used by Ivar Jacobson since the late 1960s, although he did not use the same words. The Actors & Goals model is simple and usable, but contains a small flaw, which I correct in *The Refined Model: Stakeholders & Interests*.

However, people write quite adequate use cases using this simpler model, and so I present it first, with the other as an extension to it.

So far, our definition of a use case is: *A use case is a description of the possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.* It is time to refine that.

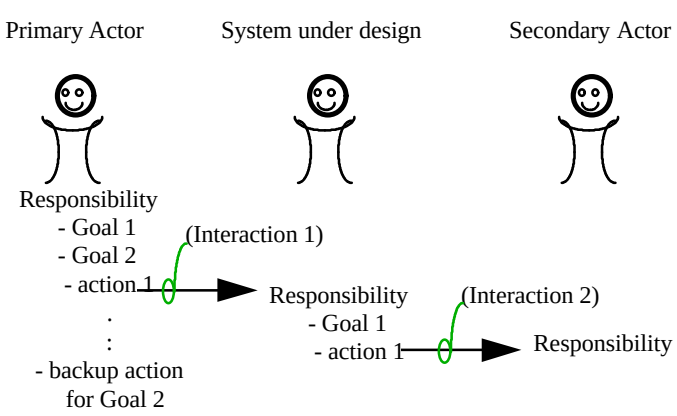
Actors have goals

In the figure below, Actor 1, the primary actor, has a job responsibility, perhaps to take service requests over the phone. A call comes in, and so now Actor 1 has, as a consequence of the job responsibility, a goal: get the computer to register the request and get it started. Actor 1 could, of course be any actor, whether a human, organization or computer.

The second actor, the system under design or system under discussion, could also be any actor, whether a human, an organization, or a computer system. Actor 2 also has a job responsibility, in our example, to register and initiate the service request. Astute readers will notice that the responsibility can be thought of as a promise to deliver a service; the use case or goal is the primary actor calling upon that service promise. Responsibility and goal often are two grammatically different versions of the same phrase.

To carry out its job responsibility, the SuD formulates a set of goals. Some it carries out internally. In some cases it needs the help of a third, external actor. This is the secondary actor, and again it may be any form of actor, from a printing subsystem to the coroner's office.

Figure 1. Communication Model



The significant thing about *goals* is that they can fail! It might happen that the computer system is down. What is the clerk with the customer on the phone supposed to do? The answer is that if the SuD cannot deliver its service promise, the clerk must invent a *backup* goal - in this case, probably pencil and paper. The primary actor (clerk) is still responsible for the main job responsibility, and must have a plan in case the SuD fails.

Similarly, and more importantly for our use cases, the SuD might discover a goal failure. It might be that the primary actor sent in bad data, it might be an internal failure, or it might be the secondary failed to deliver it's promised service.

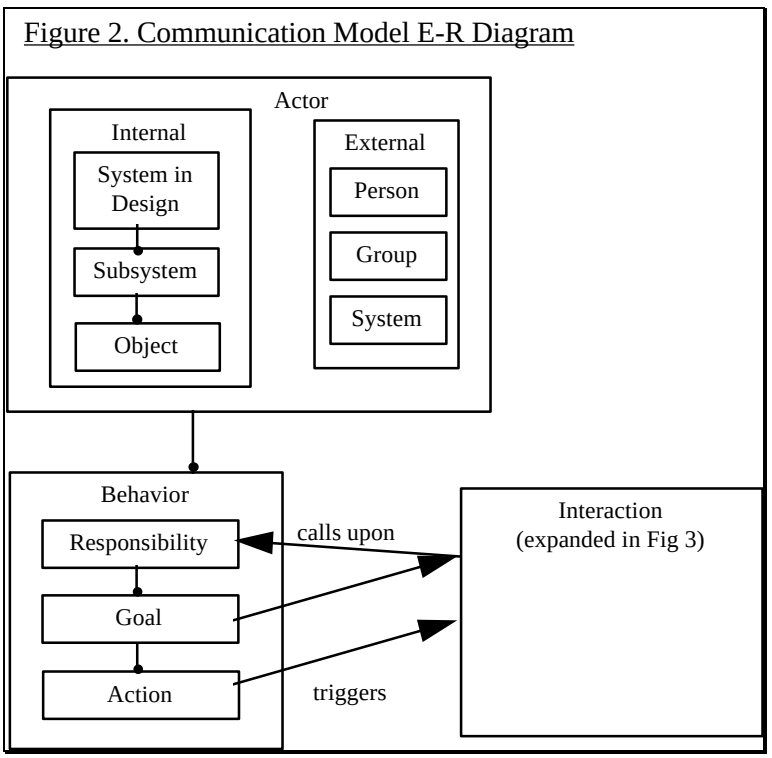
How is the SuD supposed to behave? *That* is part of the requirements. The functional requirements document needs to state how the SuD behaves in the case of intermediate goal failure. Those goal-repairing actions are the interesting part of the behavior of the SuD, the rest are typically pretty obvious.

Beyond goal repair, goals can actually fail. 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. There are times when the goal will simply fail. Noting those times is also a critical part of the functional requirements for the system.

Highlighting goal failures and failure responses are two of the many reasons why use cases make excellent 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.

For those who like entity-relationship models and meta-models, the figure below shows the model of use cases as we have it so far. The figure shows the different kinds of actors, that they have behavior, and that one actor's action triggers an interaction with another actor, the interaction being one actor's goal calling upon the other actor's responsibility.

Figure 2. Communication Model E-R Diagram



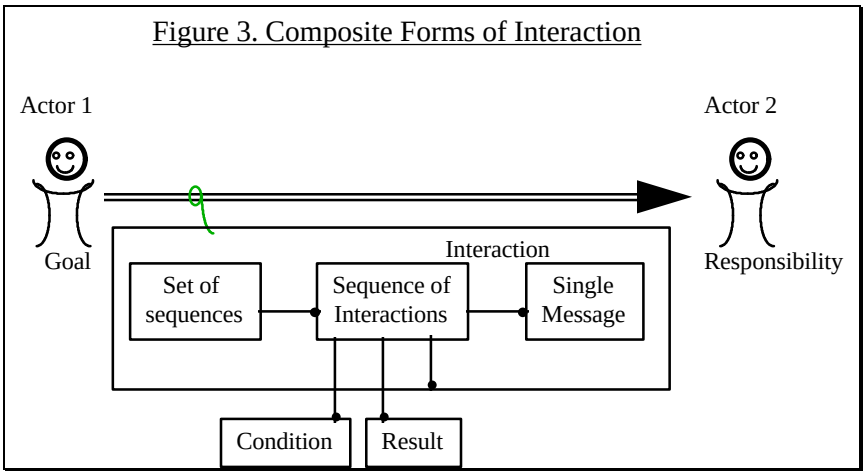
Two notes on the model.

Goals are a normal part of our programming work, but no methodology or language yet tracks them formally. The developer writes the responsibility in the function name and comments, writes a comment that states the goal informally, e.g., “Find the path with the least cost”, then writes the code that gives the actions. In terms of the above model, our development work goes straight from Responsibility to Actions, bypassing Goal. On the basis of the communication model, I hope that one day we shall have more formal tracking of goals and backup goals.

Some readers may notice the recursion already present in the figure. Responsibilities contain goals, which call upon responsibilities. True enough, this recursion produces the wonder and the pain of *goal levels*, discussed in a later section, *Goal Levels*.

Interactions are compound

Figure 3. Composite Forms of Interaction



In the simplest case, an interaction is simply the sending of a message. If I pass Jean in the hall and say, “Hi, Jean”, that is a simple interaction. In software, a function call such as “Print(value)” is a simple interaction. It is simply a single message.

An interaction also could be a *sequence of interactions*. We go to the Coke machine and put in two quarters, only to be told we need exact change. Our "interaction" with the machine was the sequence:

- buyer inserts quarter
- buyer inserts quarter
- buyer presses Coke
- machine says "Exact change required"
- buyer curses, pushes Coin Return
- machine returns two quarters
- buyer takes quarters

A simple sequence of actions consists of simple interactions, or single messages.

However, it could also consist of sequences folded together. The entire above sequence can be folded together into one phrase, "I tried to buy a Coke from the machine (but it needed exact change)." This folding together of sequences is useful for shortening a description of a long interaction. To the answer, "How did you get that Coke?", I might answer:

- I went to the coin change machine and got change for a dollar.
- I tried to buy a Coke from the machine (but it needed exact change).
- So I walked down to the cafeteria and bought one there.

It is clear that each step of that sequence could be unfolded into a sequence of smaller steps, as we had earlier. It should also be clear that the sequences can be unfolded into sequences of smaller and smaller steps. This folding up into larger steps is one of the advantages of use cases, while the potential for unfolding sequences into miniscule steps is both an advantage and a hazards, as we shall see. Since each step in a scenario can be unfolded into its own use case, I often refer to the set of use cases as "an ever-unfolding story". Our task it to write this "ever-unfolding" story in such a way that the reader can read, understand, and move around in it.

Sequences of interactions are good for describing interactions in the past, because the past is fully determined. What happened, happened. No "if" statements are necessary. However, to describe interactions in the future, we need to use a set of possible sequences, one for each possible condition of the world in the future.

If we describe asking the boss for a raise in the past, we would say:

I had a serious interaction with the boss today:
- I said, " ... "
- So he/she said, " ... "
- So then I said, " ... "
- and so on.

But if we describe the same thing talking about the future we 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 describe *how* to buy a Coke from the machine, we would have to say:

- first get your money ready.
- if you have exact change, that's great. 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 ...
- and so on.

So to describe an interaction in the future, we have to deal with conditions. For each condition, we can say what the sequence will be, and what the outcome will be (goal success or goal failure). Part of what characterized a sequence of interactions are

- the conditions under which the sequence occurs, and
- the outcome (goal success or goal failure)

In case you haven't guessed it already, the "sequence of interactions" is going to be what is called a "scenario" in the use cases, and the "set of possible sequences" is going to be a "use case"

However, we are not done with the model. It turns out that we can also fold up a "set of sequences" (the interaction in the future) into a single step, as in, "Go and buy a Coke from the machine", or "Ask your boss for a raise." When we do this, we can talk about the future interactions of a pair of actors at any level of brevity or detail.

Example.

> "I'm going to buy a house. Here is how it will work.
>
> First, I'm going to ask my boss for a raise. If she gives it to me, I'll stick all my new wages into a savings account, ... (etc.). If she doesn't, then I'll ... (etc.)."

Notice that the real goal of asking for a raise is to buy a house. Asking for a raise is a sub-goal, which might succeed or fail. To make sure of buying the house, the speaker arranges for backup goals, and so on.

This model of interactions is very powerful, because it is so natural, and because it lets us fold or unfold future behaviors to high-level, or arbitrarily detailed, low-level descriptions. Whatever level is appropriate for the moment, we can produce.

The down side of the model is that you, the use case writer, have to choose the right level for your reader, you have to be consistent about the level at which you are writing, and you have to make sure your reader knows at which level you are writing. This is the single hardest thing about learning to write effective use cases.

Before going on, I have to answer the objection that I used the term “sequence” rather loosely above. In many cases, the interactions in the future do not have to occur in a single sequence. In the simplest case of buying a Coke that costs 35 cents (remember those days?), I could have put in 7 nickels, or a quarter and a dime, or ... (you can fill in the list). But it didn't matter if I put in the quarter first, or the dime first.

People who have tried to create formal languages for use cases tend to get into difficulty at this point. They encounter all the ways something can happen and either force the writer to list them all, or invent complex notations to permit the arbitrary ordering to be described. The way out is to

use ordinary natural language. We simply write, "Buyer puts in 35 cents, in nickels, dimes or quarter, in any order."

Officially, then, "sequence" is not the right phrase. The mathematically correct phrase is "partial ordering". However, "sequence" is very close, shorter, and much more easily understood by the people about to write the use cases. When people ask, "What about messages that can happen in parallel?", the appropriate response is to say, "Fine, write a little about that." For brevity, I shall continue to say "sequence". Examples of non-sequential use cases are shown in Examples.

Use cases contain Scenarios for Goal Achievement

So far, we have that a use case is a set of possible scenarios of interaction for achieving a goal. We need to add two clauses:

- All the interactions relate to the same goal.
- Interactions start at the triggering event and end when the goal is delivered or abandoned, and the system's completes its responsibilities with respect to the interaction.

Now we are ready for the definition of use case in the Actors & Goals model.

Use Case. A use case is the statement of the goal the primary actor has toward the system's declared responsibilities, and the collection of possible scenarios between the system under discussion and external actors, showing how the primary actor's goal might be delivered or might fail.

Scenario. A scenario is a sequence of interactions that happens under certain conditions, with the intent to achieve the primary actor's goal, and having a particular result with respect to that goal. The interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction.

The alert reader or experienced use case writer will notice two small, niggling problems about these definitions.

- The definitions don't say anything about having to write down what the system does internally, such as checking for valid security clearance or sufficient balance for a cash withdrawal. While that is a valid criticism of the definition, in point of practicality, use case writers understand that they have to write down what the system is actually supposed to *do*. Ivar Jacobson got around this difficulty by using the phrase "transaction with the actor", and included all the system's checks and actions in the definition of the word, "transaction." I have never found this to be a problem with people actually writing use cases, but it is a flaw in the model, and is corrected in the Stakeholders & Interactions refined model.
- The second definition introduces the notion of the system having a responsibility with respect to the interaction. Indeed, many systems have to log every interaction with a client (e.g., banking systems). In some software systems, the software must release a resource (e.g., a file handle). If the definition did not include this requirement, the

scenario would end without that responsibility being taken care of. Once again, this is obvious to most use case writers, and is corrected by the refined model. For the time being, it is enough just to make a mental note.

Putting the definition to work, we get the following, fairly high-level description about the interaction between an insurance policy-holder and an insurance company ("MyInsCo"). The primary actor is the customer, the goal is to get paid for the damage. I give the main success scenario first, with small scenario fragments attached, describing what happens under various, different conditions.

Use Case: Get paid for car accident
Design Scope: The insurance company ("MyInsCo")
Primary Actor: The claimant
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. Agent verifies all details are within policy guidelines
5. Insurance company pays claimant

Extensions:
1a. Submitted data is incomplete:
1a1. Insurance company requests missing information
1a2. Claimant supplies missing information
2a. Claimant does not own a valid policy:
2a1. Insurance company 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.

This use case seems short and simple, but it obviously can be unfolded to a great number of lower levels. The value provided by having an interaction described at this level is that it is easily reviewed and can be expanded when needed. Steps 2 and 4 lead down into the details of the insurance company's design.

5. The Stakeholders & Interests model

The Actors & Goals model has held up well over the years. However, as a model, it comes up short when people ask, "Why isn't the owner of the coffee vending machine an actor?", and "Why do I write the system's internal activities, and not just the external parts?" The answer to the first has always been, "Because there are no special operational requirements on the system's design for the owner". The answer to the second has been, "Because these constitute the functional requirements of the system under design, and if we don't describe it's internal responsibilities, then we haven't done our job as requirements writers."

Experienced modelers and serious students of use cases have never been quite satisfied with these answers, although they were clearly true and sufficient. Busy practitioners have never cared about the distinctions, since they knew they had better capture the system's validations and internal changes of state anyway.

There is a slightly better model that builds on Actors & Goals.

Recall all the stakeholders that we decided to ignore, back in section *Actors & Stakeholders*? It is time to bring them forward again.

The reason the system does anything that it does, is that it has to satisfy the interests of some stakeholder. Why does the ATM write a log of all interactions? So that the bank owners and the federal auditors can check that the behavior was legal. Why does the ATM and banking system check that you have sufficient funds before giving you cash? The bank owners want to make sure the ATM only gives out money that customers really have in the bank.

This gives us two new ways to speak the model for use cases.

- The first is that a use case represents an agreement between the stakeholders in a system about its behavior.
- The second is that a use case shows the different ways in which the actors act to either achieve the interests of each stakeholder, or fail in an agreeable fashion, under a specific triggering condition.

The first thing to notice about the Stakeholders & Interests model is that it does not invalidate the Actors & Goals model. The primary actor is still a stakeholder, and still triggers most use cases (remember, time can trigger use cases, too). The interest of the primary actor is to achieve the announced goal. So, everything said about actors, interactions, scenarios, etc., above, is still true. We are adding the other stakeholders to the story.

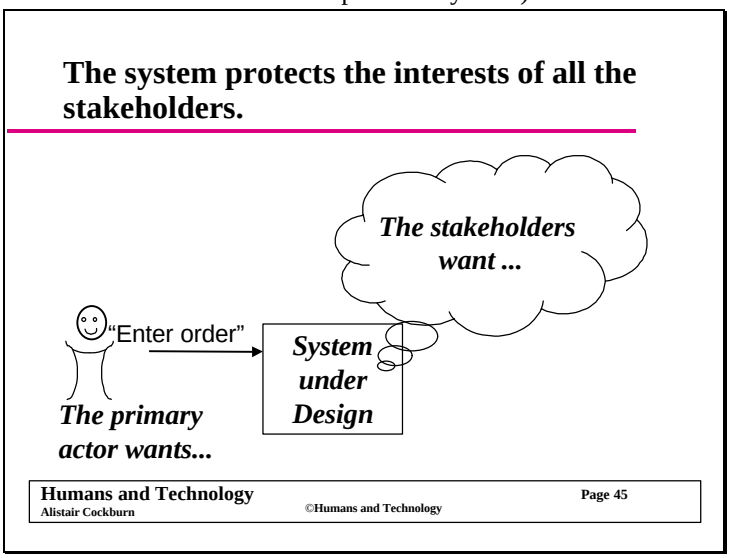
The consequence of using the Stakeholders & Interests model is that we shall list the other stakeholders and their interests in the use case, and check that every stakeholder's interests are protected by the use case.

Every action of the system should be to further or protect the interests of a stakeholder, and every interest should be protected, and ideally, delivered at the successful completion of the use case.

The attractions of the Stakeholder & Interests model are:

- it answers the questions about non-actor stakeholders and system responsibilities,
- it gives a more precise answer as to what to include and exclude in the writing of a use case, and

- using it has resulted in more accurate use cases. (One company found their all their first year's software change requests in the stakeholder lists when they applied the Stakeholder & Interests model to their previous system.)



Here is how the S&I model makes the use case appear:

1. A scenario consists of steps, each step is an action.
2. An action is of one of three sorts:
   1. an interaction between two actors
   2. a validation
   3. an internal state change
3. In an interaction, one actor will notify or make a request of another, passing along some information.
4. A validation is to protect the interests of one of the stakeholders.
5. An internal state change is to protect the interests of one of the stakeholders.
6. The SuD has the responsibility to preferably satisfy, and certainly protect, the agreed-upon interests of the agreed-upon stakeholders.
7. If the scenario is a "success" scenario, then all the (agreed-upon) interests of the stakeholders are satisfied for the service it has responsibility to honor.
8. If the scenario is a "failure" scenario, the all the (agreed-upon) interests are protected for failure to achieve the declared service.
9. The scenario writing ends when all of the interests of the stakeholders are satisfied or protected. The goal of the primary actor is one of the interests of one of the stakeholders (the primary actor is a stakeholder).

10. The use case is an agreement between the stakeholders about the behavior of the SuD upon request for a goal's delivery. The "primary actor" is the stakeholder who wants the goal delivered.
11. Triggers that request a goal's delivery may include the primary actor initiating an interaction with the SuD, the primary actor using an intermediary to initiate that interaction, or a time- or state-based trigger.

The key points to remember are simply that:

- the primary actor is just one of the stakeholders,
- the use case shows how all the interests of the stakeholders are satisfied or protected,
- every step in the scenario has to do with a stakeholder's interest.

The new model only makes a small change in the overall procedure in writing a use case: you will list the stakeholders and their interests, briefly, and then use that list as a double check to make sure that in your writing, you have not inadvertently left out one of them. The rest of the writing is the same as with the Actors & Goals model.

6. Goal Levels.

We have seen that the interactions in a scenario can be unfolded into smaller and smaller interactions (recall "buy a Coke" and "put coin into machine"). Each of these interactions represents an actor attempting to achieve a goal. So it is natural to see that goals also can be unfolded into smaller and smaller goals. This is normal, and we are used to it in everyday life, but it often causes confusion when writing a use case. Every writer is faced with, "What level of goal shall I describe, now?" And is faced with that question on every sentence in the use case.

Let us look at two examples.

1. My current goal is to get a sales contract.
2. So I decide my immediate goal is to take the other party out to lunch.
3. So my more immediate goal is to get some cash.
4. So my even more immediate goal is to withdraw some cash from this ATM.
5. So my even more immediate goal is to get it to accept my identity.
6. So my even more immediate goal is to get it to read my ATM card.
7. So my even more immediate goal is to find the place where I put the card in.

1. I want to use an insurance policy for my car.
2. So I want to buy one.
3. So I want to get a quote.
4. So I want to enter my characteristic information into the program.
5. So I want to enter my address into the program.
6. So I want to get the cursor over to the address field.
7. So I want to find the tab key.

These examples are not completely silly. We could write a use case for any one of those goals. We are likely to find ourselves writing one of those phrases in our scenario. Which one should we write?

Since goal levels can go all the way up to "achieve peace on earth", and down to "find the tab key", we need a way to talk about the level of any particular goal. I have found three goal levels sufficient for use case work:

User-goal level ("blue", "sea level").

The level of greatest interest is the *user goal*, the goal of the primary actor trying to get work done. This might also be called “user's task”, and it corresponds to “elementary business process” in the business process engineering literature.

A user goal addresses the question: “Does your job performance depend on how many of these you do today?” “Log on” does not generally count as a user goal as it fails this test - logging on 43 times in a row does not (usually) satisfy the user’s job responsibilities. On the other hand, “Register a new customer” is likely to be a meaningful user goal. Registering 43 new customers has some significance.

The tests for a user-goal usually are:

- Is it done by one person, in one place, at one time (2-20 minutes)?
- Can I go to lunch as soon as this goal is completed?
- Can I ask for a raise if I do many of these?

Recently, because of confusion about what constitutes a task, and overlap of terms in use, I introduced the use of color in describing goal levels.

The reference color is "blue" (we'll get to white and indigo).

The definition of a "blue" goal or blue use case is this:

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

<X> is a "blue" level goal, or the "user goal". You accomplished a "blue" level use case.

In accomplishing goal <X>, you had lots of lower-level goals. We'll color those "indigo". The person on the phone probably had a higher-level goal in mind, and accomplishing <X> was only one step in that. We'll color those higher level goals "white".

The "blue" level goals, the user goals, are incredibly important, so it is worth a large amount of effort to understand and internalize just what constitutes a blue level goal. In the requirements file for the system under design, you will justify the existence of the system by the goals it supports of various primary actors. That list will consist of "blue" level goals. The shortest summary of the function of a system is the list of blue level goals it supports. That list is the basis for prioritization, delivery, team division, estimation and development. It is for that reason that we use so many names, metaphors and tests for the level.

Given the significance of the user-goal level, I often use one more term, "sea level". 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. That visual image is appropriate for use cases. The system function is justified by the highly distinguished level of "user goal", "blue", "sea level" goal.

Goals and use cases can and will be written at an indeterminate number of levels 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", or below sea level ("underwater" was an appropriate phrase to describe them. That requirements document did not serve either its writers or readers. The sender later sent me the six user-goal use cases that had replaced them, and said everyone found them easier to understand and work with.

Goals and use cases also occur at an indeterminate number of levels above sea level. This is appropriate, and some of these "white" or "strategic" use cases should be written. Which gets us to...

Strategic ("white") use cases

Strategic-level goals involve multiple user-task level goals. They serve three purposes in the describing the SuD:

- showing the context in which the system services must function,
- showing life-cycle sequencing of related goals,
- providing a table of contents for the lower-level use cases (both lower-level white use cases and blue use cases).

In color terms, we can call strategic level goals or use cases "white". 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."

Strategic-level use cases typically take many hours, days, weeks, months, or years to complete. Here is an example of a the main scenario from a long-running use case, whose purpose is to tie together blue use cases scattered over years.

Goal: Operate an insurance policy.
Primary Actor: The customer
Scope: The insurance company ("MyInsCo")
Level: Strategic ("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.

Each of those sentences refers to a user-goal use case, a "blue" use case. Those use cases are called,

"Get a quote for a policy", "Buy a policy", "Make a claim against a policy", "Close a policy".

Writing high-level use cases

Earlier, I made the recommendation that you write a few high-level use cases for whatever system you are designing. Now that the terms "design scope" and "goal level" are in place, I can finally say what those use cases are.

1. Look at the blue-level goals for your SuD.
2. Ask, "what -preferably external- primary actor does this goal really serve?"
3. Find the most outer-level design scope that still has that primary actor outside, name that design scope, and consider writing a use case for that actor with that goal against that system scope (but don't write it yet).
4. Find all the blue goals with that ultimate primary actor and that outermost design scope.
5. Now see if you can name the strategic or white goal that primary actor has against that system.
6. Write a white use case for that goal of that actor against that system. This use case will come to tie together a great many of your blue use cases, provide an excellent context for future readers, and will be easy for your sponsoring executives to review.

Ivar Jacobson has said many times that people write too many use cases, at too low a level. He has said, approximately, "A really large system might have seven use cases." I think that he is referring to these white use cases I just described. Certainly, in my experience, I have typically found three outermost design scopes and primary actors (the customer to the company, the marketing department to the software system, the security department to the software system), and only about 5 of these topmost-level use cases, even in the largest systems I have helped build. However useful these use cases are, I would still not like to trust them as the functional requirements for the system to be built. For me, still, those reside in the blue use cases.

Subfunction ("indigo"/"black") use cases

Subfunction-level goals are those needed to carry out user-task goals. They are only included as needed. They are needed on occasion for readability, and more often because many other goals involve them. Examples of indigo use cases are "Find a product", "Find a Customer", "Save as a file."

We use the color "black" to mean, "this is so low level, please don't even expand it into a use case". It is handy to have such a level, because then when someone writes such a use case, you can mark it as "black", and signify that its contents ought to rolled up into a higher-level

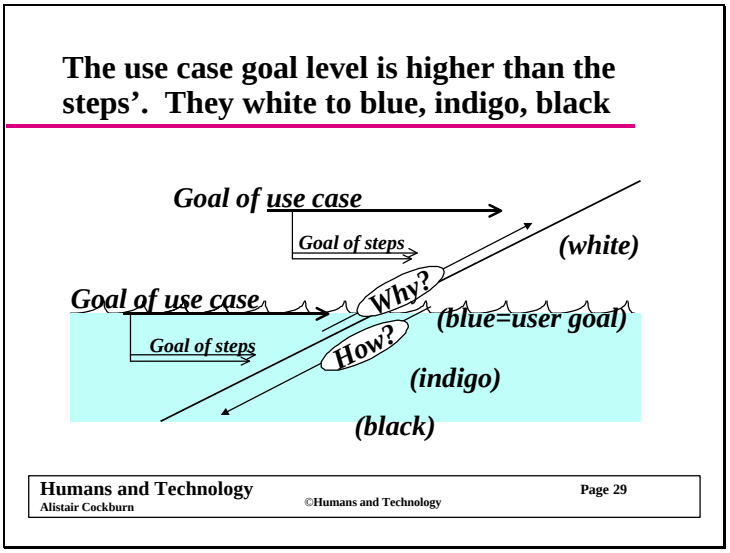


Figure "The use case gradient". (Needs the "why"/how arrows)

Rather by definition, each of the goals/steps in a blue-level use case will be indigo. The above figure shows this. A white use case can contain steps that are either white or blue. A blue use case contains indigo steps (there is a small exception to this, which I discuss in *Goals and Level Confusion*). An indigo use case contains indigo and black steps. The line at which the colors change is "sea level" or "blue".

Two levels of blue use cases

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

Suppose I am walking past the video rental store, doing some errands. I think, "I might as 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. On another day, I go in with my membership card and "*Rent a Video"*. Two user goals on two days.

However, in a different version of the story, I walk into the video store and want to "*Rent a Video"*. The clerk asks, "Are you a member?" I say, no, and so the clerk has me "*Set up a Membership"*. In this case, *Set up a Membership* is a single step inside *Rent a Video*. Both are blue goals at different times.

This "register a person in passing" is the only time I can recall seeing a blue use case inside a blue use case. To keep with the waterline metaphor, I say that *Rent a Video* sits at the top of a wave (see the waterline picture, above) and *Set up a Membership* sits in the trough of the wave, but both are at sea level :-).

Working the goal levels

Section *Goals and Level Confusion* talks more about handling the different levels of goals and resolving the confusion that can arise. For now, three points are important.

- Put a lot of energy into detecting and listing the "blue", "sea-level", "user-goal" use cases. These are the important ones.
- Plan on writing a few highest-level, outermost use cases, to serve as context and index for the blue ones.

and most importantly,

- Don't make a big fuss over whether your favorite phrase among the system requirements sentences "makes it" as a use cases title.

Being a use case title does not mean "most important requirement", and not being a use case title does not mean "unimportant". I keep seeing people upset because something the system is supposed to do is "merely" a step in a use case, and did not get "promoted" to being a use case that can be tracked on its own. One of the benefits of the goal model of use cases is that it is a "small" change to break out a complex chunk of scenario into its own use case, or to fold a trivial use case back into a higher-level use case. Every sentence will be written as a goal, and every goal could 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.

How to phrase a goal

Write the goal as an active goal phrase: <verb> <direct object>.

There are two schools of thought about just what kind of verb form to use, present tense or gerund. Frankly, it really does not matter which you use.

I happen to like the present tense, which produces use case titles such as, "Get a quote", "Buy a policy". I find them short, direct, and easy to incorporate into scenario writing: "The customer gets a quote", "The customer buys a policy".

Other people advocate using a verb's gerund form, which produces use case title such as "Getting a quote", "Buying a policy". The net result in writing the scenario is the same, "The customer gets a quote", "The customer buys a policy".

Simply choose a form for your project. You will not lose any requirements if you choose one form instead of the other. Simply choose one, and spend the rest of your energy on something more critical to your project. In this book, I stick to my preferred style of writing, the present tense form.

Exercise: You are standing in front of an ATM attached to your bank. It is dark. You have entered your PIN, and are looking for the "Enter" button. Give examples of white, blue, indigo and black-level goals that you might have at this moment.

Exercise: Name at least ten goal various primary actors will have with respect to the ATM, and name their goal levels.

Exercise: Name the strategic and task-level goals of all the primary actors for the PAF software package. Identify the highest-level, outermost actor - scope - goal use combinations.

Using graphic icons to indicate design scope and goal level

Consider attaching different graphics to the top of a use case, to distinguish the dominant distinctions:

Design scope differences: business - system. The "business" use case has the entire company or organization as the design scope. The "system" use case has a computer system / hardware / or software as its design scope.

Put the scope logo (preferably) at the very top-left of the entire use cases, to signal the design scope even before the reader starts reading. I offer the following possible icons (see figure):

- Business use cases use the 'house" logo
- System use cases use a "person sitting at a computer" logo,
- If you are in the rare situation that you are describing system internal design with use cases (as in *Documenting a design framework*), use the "donut" logo.

Goal level differences: strategic - user-goal - subfunction.

Put the goal-level logo immediately before the goal phrase in the title, or in the "goal level" field:

- Strategic or "white" (anything above "blue") use cases use a cloud,
- User-goal or "blue" use cases use a person at a screen,
- Subfunction or "indigo" use cases use a pitchfork.

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 the difference, and when your use case template does contain scope and level fields (shame on your template provider).

Design Scope	Goal Level
Organization: (building)	Strategic: (cloud)
Organization: (building)	
System: (black box)	User-goal: (person at computer)
System: (white box)	
Component: (donut)	Subfunction: (pitchfork)

Icons for use case scopes and levels.

Here are some common combinations and variations:

	high-level use case about the business (as a black box)
	high-level use case about business operations (a white box use case)
	a user-goal level use case about the business (white box)
	a high-level use case about a computer system.
	a user-goal use case about a computer system

	a white-box user-goal use case about a computer system
	a subfunction on the computer system
	a user-goal use case on a component of the system
	a subfunction on a component of the system

I add these two icons useful as reminders about casual vs. fully dressed use cases, but they wouldn't be needed on every use case:

 = a fully dressed use case

 = a casual use case

7. The Actor-Goal list

One of the crucial products of the requirements gathering work is the "Actor-Goal List". This is the list of user-goal or "blue" use cases, and the primary actor(s) with the interest in the goal. The list has at least two columns:

ACTOR	GOAL
The primary actors(s) having this goal.	The goal, written as a short verb phrase.
(e.g.) Customer	Buy an insurance policy
etc.	

You may want to add some more columns to this list. Often, people will have column for "trigger", to show which ones are time-based triggers vs. actor-initiated trigger. Or, people will have a column for "priority", to show how essential this is to deliver. Or, they will have a column for "complexity", to estimate how much internal complexity they think this use case contains (either interface complexity or algorithmic complexity).

Personally, I typically start with three columns, using "priority" as the third column. Here is an actor-goal list for a purchase-request tracking system:

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

I cannot overstate the importance of producing this list. This list is the negotiating point between the end-user representative, the financial sponsor, and the development group. It is short, easy to read, easy to review, and it carries the arguments that are crucial to the overall layout and ultimate success of the project. It is essential that the goals listed be blue-level goals.

8. Precondition, Success End Condition, Failure Protection

Precondition

Sometimes the use case will only be triggered if the user is already logged on and validated, or some other condition is known to be true. Writing some condition of the world into the Precondition means that that condition is known to be true, and need not be checked again.

Writing style: The 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.

Unsuitable preconditions: It is common for people to write into the precondition things that are often true, but are not necessarily true.

Suppose we are writing the use case, "Request benefits summary". In the normal course of events, we would expect a person has taken some prior action, such as submitting a claim, before asking for the benefits summary. However, from the perspective of the system developers, that is not necessarily the case, and they cannot ensure it. So it is inappropriate to write, in the Preconditions, "The person has submitted an form before asking what their benefits are." That is not a true precondition. In fact, the software will almost certainly have to handle the situation that the benefits statement is empty, as one of its normal cases.

So only put into the Precondition things that the system can guarantee are true.

Pass/Fail test for a Precondition: Here is a really elaborate way to say what to put in the Precondition. I have made it so elaborate for emphasis.

"The precondition is something that you want the programmers to ensure has happened before this use case starts. The use case can't possibly start unless these preconditions are true - you want the programmers to go to extra lengths to have the system guarantee that this use case absolutely won't start unless these conditions are met. Therefore, you will not check these conditions again during the use case."

Note again, the conditions mentioned in the Precondition will not get tested or revalidated inside the use case, because they are *known* to be true - the system has ensured that.

Where to put the "almost" preconditions: So where do we put the fact that usually a person has submitted a form before requesting their benefits statement? The answer is: into the "**Context of use**" section. That is the place to talk about normal occurrence of this use case, how this use case fits into the overall operation of the system and the containing system.

Success End Condition

The Success End Condition states what interests of the stakeholders have been satisfied at the end of the use case. Usually at least two stakeholders are represented, the system user, and the company providing the system under discussion. There may be a regulatory body also with interests

Writing style: The Success End Condition is written as simple assertions about the state of the world at the moment of successful completion of the use cases, often using the present or past particle of the verb ("has been done", "is accomplished"). It should show the interests of each stakeholder being satisfied. Suitable examples are:

Success End Condition: The claimant has been paid the agreed-upon amount, the claim is closed, the settlement is logged.
Success End Condition: The file is saved.
Success End Condition: The system has initiated an order for the customer, payment information has been received, the request for the order has been logged.

Pass/Fail test for the Success End Condition: Each of the stakeholders in the use case would agree, from reading the success end condition, that their interests had been satisfied for this goal of the primary actor.

Exercise: Write the success end condition for withdrawing money from the ATM.
Exercise: Write the success end condition for the PAF system's main use case.
Exercise: Write the success end condition for a blue use case for your current system. Show it to a colleague and have them analyze it with respect to the interests of the stakeholders.

Failure Protection Condition

There are dozens of ways to fail, and often little in common between them. However, sometimes there are some special interests that must be protected under failure conditions, and those get written in the Failure Protection Condition. Most typical is that the system has logged how far it got before failure, so that the transaction can be recovered, and the interests of the stakeholders protected. On occasion, the Failure Protection Condition is written as a conditional statement, such as "If the ATM did not dispense the cash, the account is not debited."

Very often, the Failure Protection Condition is left blank, because there is nothing interesting to say - e.g., if a customer calls to place an order, and hangs up in the middle, it is rather obvious that nothing will be sent and no payment deducted. The dedicated writer may choose to write this down, but most people leave the trivial conditions blank.

Failure logging, however, is not a trivial condition. I often find that a log needs to be created, but is not mentioned in the main success scenario. The need for it shows up when the interests of the stakeholders are examined while filling out the Failure Protection Condition, and the writer discovers that on failure, the system has to have a log or marker about how far it got. Rereading the use case text, the writer discovers that creating a log or marker was never mentioned, and adds it in.

Writing style: The Failure Protection Condition is written as simple assertions about the state of the world when the primary actor's goal has to be abandoned. Suitable examples are:

Protected Protection Condition: No order has been initiated if no payment was received.

Protected Protection Condition: If the minimum information was captured (see business rules), then the partial claim has been saved and logged. If the minimum information was not captured, the partial claim has been discarded and no log made of the call.

Pass/Fail test for the Failure Protection Condition: Each of the stakeholders in the use case would agree, from reading it, that their interests had been protected under failure conditions for this goal.

Exercise: Write the failure protection condition for withdrawing money from the ATM.

Exercise: Write the failure protection condition for the PAF system's main use case.

Exercise: Write the failure protection condition for a blue use case for your current system. Show it to a colleague and have them analyze it with respect to the interests of the stakeholders.

9. Main success scenario

A *scenario* is written as a sequence of goal-achieving actions by the primary actor, secondary actors and the system under design. Recall that "sequence" is a handy euphemism for "partial ordering", meaning than we can indicate that steps repeat, can be taken in a different order, even are optional (although I tend to discourage this).

The *main success scenario* is the most common, or perhaps easiest to understand, scenario. It may not be the only success scenario - but we shall choose to construct the other success scenarios as extensions off this one. It may, actually, not even be the shortest success scenario. It is simply the one that you choose as a good basis to built from. It should be fairly typical, easy to understand, and it definitely will contain no failures.

One of the wonderful things about use cases is that they capitalize on our natural cognitive ability to start from a simple story, and add complexity to it. We naturally explain things to other people by giving a short, easy to understand description, and then saying, "Well, actually, there is a little complication. When such-and-such occurs, what really happens is ... " People do very well with this form of explanation, and it is exactly this form of explanation for system behavior that

Ivar Jacobson invented when he invented use cases. It is not at all an obvious way to describe system behavior, but it has shown itself to be a remarkably effective way.

The main success scenario consists of actions, as discussed in the *Stakeholders & Interests Model*. There are three kinds of actions: an interaction, a validation, or a state change. Once you master the writing of the three kinds of actions, you are pretty well set for your writing style. The same writing style is used in the main success scenario, the alternative scenarios or scenario extensions, for business use cases, high-level use cases, low-level use cases, for every active part of any use case.

Even with all the recommendations for writing in this book, there are a large number of reasonable and different ways to organize your writing. Different organizations, different teachers, and different individuals have formed their separate and distinct writing styles and formats. We cannot get past the main success scenario until we look at some of the valid and popular alternative writing styles and formats.

10. Overall Writing Style

Core form

At the core of the writing is the form:

"At time z, actor y does x to actor w with data v"

Using the verb "verb" to indicate some active, goal-forwarding verb, a interaction sentence typically looks like:

"Actor1 verbs actor2 with some data."

For the other two kinds of actions, the validation and the state change, the core sentence forms are:

"The system verifies that the something meets some criteria."

"The system changes the something to reflect something."

Key characteristics of an action step

There are many valid, viable, usable ways to write the action sentences in use cases, all minor variations on the core forms. However you choose to actually write, here are some crucial characteristics to preserve.

The sentence shows the process moving distinctly forward.

The amount of forward progress is related to how high or low the use case goal is. For a white or strategic use case, the process probably moves forward by an entire user-task level goal. For a subfunction or indigo use case, the process moves forward a much smaller amount. While it is conceivable to have a step, "User hits the tab key", moving the process so little forward indicates that we have either chosen the wrong action to present, or we are in a deep indigo (probably black) use case.

For some reason, I have never seen a well-written use case with more than 11 steps in the main success scenario. Eleven certainly isn't a magic number. It just seems that when we look at a use case with, say, 13 or 17 steps in it, we find that the sentences are not moving the goal forward very much. When we combine steps (see *Goal Levels*), we always find that the text is clearer, easier to read, and has not lost any essential information.

Whenever you feel that perhaps the action is not moving sufficiently forward, ask the question: "Why is this actor doing that?" The answer to your question is *the higher-level goal of that actor*, the goal you probably should be writing in the sentence.

For example, you see the sentence, "User hits tab key".

You ask, Why is the user hitting the tab key? Answer: to get to the address field.

Why is she trying to get to the address field? Answer: because she has to enter her name and address before the system will do anything.

Oh! So she wants to get the system to do something (probably the use case name, 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".

Be clear, "Who's got the ball"

A useful visual image is that of friends battering a soccer ball around in the back yard. Sometimes person 1 kicks it to person 2, and then person 2 kicks it on their foot for 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", so you can ask, at each step, "Who's got the ball?". That actor is going to be the subject of the sentence, the first actor named, probably 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, check it for 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's got the ball."

Use 8th grade grammar.

The sentence structure should be absurdly simple:

> "Noun... verb... direct object... prepositional phrase."

That's all there is to it. I mention this simple matter because many people leave off the first noun, when they start writing use cases, write grammatically incorrect sentences. If you leave off the first noun, it is no longer clear who has the ball (see, however, the "play" style of writing, below). If your sentence is incorrect, the story gets hard to follow. A good use case is be easy to read and follow.

Write 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 will write the entire use case 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."

Instead, write the use case from a bird's eye point of view, write neutrally, with the actions of all actors described with the same sorts of verbs. "The customer puts in the ATM card and PIN. The system deducts the amount from the account balance."

I said a use case in an "ever-unfolding" story. I should have say, "play", because it resembles a play, with all the actors performing their parts. A play, like a use case, is written from a bird's eye point of view. Another way of thinking about how to write is to pretend that you are writing a play (see *Variations of Writing Style*, below).

"Verify", don't "check"

In the second type of action step, the system verifies that some condition is met. "Verifies" / "validates" / "ensures" / "establishes" are good, goal-achieving action verbs. "Checks" is not. After you write "checks", you must immediately write, "If the check passed...", and, of course, "If the check failed..." The "check... if..." phrases are not moving the process distinctly forward. It is not really the goal to "check" something; it is the goal to establish something (quick test: why is the system checking the condition? Answer: to establish something.).

Therefore, avoid the verbs "checks" and "sees whether". Use instead one of the other goal-achieving action verbs for a validation step. Let the presence of the word "if" trigger your thinking.

The sentence shows the intent of the actor, not their particular movements in accomplishing that intent (semantics, not dialog).

A *dialog* description of the interface shows the step-by-step activity of the actor while operating the user interface. The alternative we are interested in is the *semantic* description of the interface, which tells what the intent of the user is, and what information must be provided.

During requirements, we are not interested in the dialog description. A dialog description makes the requirements longer, brittle, and overconstrained.

- It makes the document longer, harder to read, and more expensive to maintain.
- It is probably not really a requirement, it is probably just how the writer imagines the most obvious user interface design to look.
- It is "brittle", in the sense that small changes in the actual design of the system will invalidate the writing.
- There will be a UI designer on the project, whose job it is 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.

Therefore, in each step, capture what the *intent* of the actor, what they want to accomplish. If there is data being passed in the step, describe it either as a comma-separated list, or a tabbed list.

Larry Constantine and Lucy Lockwood devote a large portion of their book, Software for Use (Addison-Wesley, 1999), to just this topic. They refer to user-task level use cases that capture the semantics of the interface, as "essential use cases". Essential use cases are the most important ones for the system design.

Exercise: Write a simple scenario for one of your use cases. Write it using a dialog description. Write the same scenario using semantic description. Discuss the differences between the two.

Mentioning the timing is optional

Most steps follow directly from the previous. Occasionally, you will need to say, "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 don't feel obliged to. Usually the timing is obvious and need not be mentioned.

Idiom: "User has System A kick System B"

On occasion, you want the system under design to undertake some specific action with respect to a secondary actor, perhaps system B. However, it should only do so when the primary actor indicates that it is time.

Recall that we are not going to describe the user interface design, so we cannot write, "User hits the FETCH button, at which time the system fetches the z data from system B."

We could write it in two steps: "User signals to the system to fetch data from system B. The system fetches the z data from system B." But that is awkward and redundant.

A handy idiom to use in this situation is: "User has the system fetch the z data from system B." This indicates that the user controls the timing, that the ball passes from the user to the SuD to system B, and shows the responsibilities of all three systems in this situation. Just how the user initiates that action is unspecified, as it should be.

Idiom: "Do steps x-y until condition"

On occasion, we need to mark that one or more steps can be repeated over and over. Here again, we are aided by the fact that we are writing in plain prose, as opposed to using a programming formalism. So we simply write that the step(s) will be repeated.

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

> "User selects all the products they want to order."
>
> "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, it works well to write the repetition after the steps:

1. Customer supplies name and address or account identifier.
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. System helps customer *purchase the items in the shopping cart*.

In the above, notice that we need not number the statement about repetition, and need not have a statement that opens the repetition. These would clutter up the writing, making the scenario harder to read, not easier. The step numbers are there to help us, not clutter the writing (see *Numbering*, below)

Notice also how step 5 refers to another use case, "Purchase items in shopping cart".

How much fits into one step?

One of the points of variation between writers - and one that is permitted, because it is not significant to the overall quality of the use case - is how much to put into one step.

Ivar Jacobson has describe a step in a use case as representing "a transaction". With this phrase, he captures four parts of a compound interaction:

- The primary actor sending in request and data to the system.
- The system performing some validations on the data.
- The system altering its internal state.
- The system replying to the actor with the result.

So, the first valid style of writing is to put all four parts of a transaction into one step. This style is rarely used in practice, because the sentence typically becomes long and unwieldy. It is handy to know about this variation, though. From time to time, the entire round trip description is short and not worth breaking into pieces. You might think to use this style when might find that you have two, three, or four short sentences, so short that they are actually harder to read as separate sentences than if you combined them. The idiom, "User has system A kick system B" is an example of such a combination.

A second valid style of writing is to put "Actor verbs the system" into the first sentence, and everything the system does after that in the next sentence. The third and fourth styles are to break out the validations, and then the responses.

I have seen multiple of these sentence styles used, even within the same use case, to good effect. How much you choose to put into one sentence is a function of what you have to say, and is up to you.

Here is an example of variations of opening and combining parts and steps (see also *The 3 Bears Pattern*).

Version 1.

1. The customer enters their 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.

Version 2.

1. The customer enters their 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 3.

1. The customer enters their 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 4.

1. The customer enters their 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.

Which do you prefer? None of them is wrong. All of them work. At a personal preference level, I would vote most against version 2. I find it a bit too complicate to understand. Version 1 is a bit too broken up for easy reading. Either of versions 3 or 4 would satisfy me. I have a slight preference for version 4. It separates out the validation, and the actions internal to the company from those that go out to the primary actor. But in a use case review, or even as a coach to the writer, I would defer to the use case writer's preference between versions 1, 3 and 4.

Numbering and writing notes

Whether to number the steps or not is another personal preference. Ideally, this matter is settled on a project-wide basis, but there is very little damage if some people prefer to write with step numbers and some prefer to write without.

Important note: The numbering serves only so that we can refer to a step, not because the step occurs in that sequence. Recall that the "sequence" of steps is really a "partial ordering".

The fact that we number steps does not mean we need to number every line. We can freely write notes to our readers between steps, without having to number them. See the example above, in *Idiom: "Do steps x-y until condition"*. Another example of writing a note to the reader is to say,

1. ...

2. ...

   Steps 3-5 can happen in any order.

3. ...

4. ...
5. ...
6. ...

So why use numbers? Using numbers has a cost and a benefit.

The purpose of the numbers is to allow us to refer to the step when filling in the Alternative Courses or Extensions section. We are going to brainstorm things that could go wrong, and it is helpful to be able to refer to which action step we are thinking could go wrong. Since the failures only occur on action steps, there is little value in numbering the notes to the readers (although doing that would, again, not be considered "wrong").

If the value of the numbers is clarity of reference, the cost is energy. It takes more time and energy to think through all the failure conditions and write down where they all occur. We might find ourselves on a project or in a situation where writing a highly precise and accurate use cases is actually not necessary (for example, there is a user on the project team full-time, or it is a low-damage system and the project team is all located in the same room). Or, we might be writing for business people in the field, who would be more comfortable reading a paragraph style or writing. Or, we might simply select to write in paragraph style out of personal preference, and will specify the failure conditions adequately without line numbers (the Rational Unified Process uses a such a template).

Whatever the reason, neither style is wrong, and it comes down to personal preference. Personally, I always use numbers - that is my personal writing style. In the use case "bake offs" I periodically hold, people tend to prefer the numbered style. However, a number of excellent writers choose to avoid numbers, I have seen many fine use cases written in paragraph style, and there are times when the writers simply don't want to take the time and energy to get fancy about the use case they are writing. So they use the paragraph style.

As a result, I include in this book two (three?) templates (see "Use Case 1" in *Precision and Tolerance*, and the section *Use Case Templates*).

The numbered version I call the "fully dressed template". It should be used on large or critical projects to enhance consistency and to simplify checking for writing quality. It can be used by personal preference of the writer (for example, I always use it, even if I discard a number of the associated fields at the end of the template).

The unnumbered version I call the "casual" template. It can be used on smaller projects just whenever the writers choose to. It contains the same basic information, and the writing should follow the recommendations in this book. The use case just will not use line numbers.

Sample step styles

The step shows the actor about to "kick the ball", the verb, and the data being passed. There are several ways to organize the sentence. Again none are wrong, they each follow the basic guidelines in this book.

The first point of variation is whether to write the step as a full prose sentence, or to write the step in the style of a play, with the actor's name followed by a colon. I have seen both done well.

Style A: The step is written as a full sentence.

"3. The customer enters "withdrawal", and the withdrawal amount."

Style B: The step is written as Actor name, colon, goal phrase.

"3. Customer: enters transaction type for withdrawal, and withdrawal amount."

Style C: As style B, but each actor is given an abbreviation, to shorten the writing.

(Somewhere near the top of the use case it says:

Actors: S: system
C: customer
B: bank's networked computer)

"3. C: enters transaction type for withdrawal, and withdrawal amount."

The second point of variation is whether to write the data list directly into the sentence, or break it out into a tabbed list. While it is generally inappropriate in a use case to list the detailed characteristics of each date element (see section *Data descriptions*), it is sometimes useful to break out the list of items being passed between actors.

Style D: The step is written as a full sentence with a comma list for the data items.

"3. The customer enters "withdrawal", and the withdrawal amount."

Style E: The step is written as a sentence with a tabbed list for all the data items.

"3. Customer enters
transaction type withdrawal
withdrawal amount."

Often, it is attractive to use the comma-separated list during early drafts of the use case, because they are easier to write and read. At some point, precision becomes an issue, or the list becomes long, and the tabbed list becomes more attractive.

For all those valid styles, there are some styles to avoid:

Unacceptable:

3. Enter "withdrawal" and amount

(This sentence is missing the actor).

3. The customer checks to see whether they have enough money in their account.

(This sentence does not show the real goal of the customer.)

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

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

Exercise: Critique and repair Exhibit 6-8, page 77, in Applying Use Cases.

Exercise: Critique and repair the "Log in" use case, page 79, in Applying Use Cases.

Various sample formats

Cockburn style

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

- one column of plain prose,
- 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 Rebecca Wirfs-Brock's 2-column style and the Rational Unified Process template (see both 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. And so I continue to use and recommend it.

Here is the basic template:

Use Case: <number> <the name should be the goal as a short active verb phrase>

Characteristic Information
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: Strategic, 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>
Success End Condition: <the state of the world upon successful completion>
Failed End Protection: <the state of the world if goal abandoned>
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>

I/O Variations
<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 (Optional)
<whatever your project needs for additional information>

One-column table style

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 create a lot of visual noise, and obscure the actual writing. However, that is merely personal preference, and so I show the template for the style:

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>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Protection	<the proper state of the world if goal abandoned>	
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>
VARIATIONS		**Branching Action**
	1	<list of variation s>

Wirfs-Brock's 2-column table style

Rebecca Wirfs-Brock invented a variant she calls a "conversation". This is further described in her various tutorial notes and upcoming books. Basically, it consists of putting all the primary actor's actions in the left-hand column, and all the system's actions in the right-hand column.

I find that while the 2-column form is clear, it makes the writing very long - a use case written in two columns often crosses two page boundaries, which is long indeed. By the time we put the goals into 3-9 steps at the appropriate goal level, the writing is so simple and clear that the people on the team no longer find the need for the two columns.

The more substantive problem with the 2-column format is that it does not support discussion of secondary actors. Perhaps one could add a third column for the secondary actors, but I have never heard Rebecca suggest, nor seen it done.

To be a little more clear about the intended use of this form, Rebecca reserves it for a little later in the requirements analysis process, when the team wants to make very sure they are clear what the actual, detailed responsibilities of the system are. Often, there is more user interface detail available at this stage, and so that UI detail gets mentioned. Larry Constantine has picked up the 2-column form as useful to enhance the design of the user interface (see [Constantine99]). Notice that both of these people are using this 2-column form for a later task in the development process, *after* the functional requirements have been written.

All of the above taken into account, many people do find the 2-column form attractive - either while they are just 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 2-column style:

Customer	System
1. Customer enters order number.	
	2. The system detects that the order number 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.
6. Customer exits the system.	

RUP style

Rational Software Corp. has published a use case template that uses simple paragraphs of text with no step numbering. To compensate for absence of numbering, the alternate courses are given headings and sections of their own.

Everything I have to say in this book works nicely with this template, which is attractive and easy to read, although a bit long-winded for my taste. See *Writing Sample: Example of a CRUD*

use case. Here is an excerpt from that writing sample, just to show the style. The interested reader should contact Rational Software for more details.

Title: Manage Reports

Brief Description

This Use Case describes and directs operations for Creating, Saving, Deleting, Printing, exiting and Displaying Reports. ...

Actors

- User (Primary)
- File System: typical PC file system or network file system with access by user. (Secondary)

Triggers

User Selects operations explicitly.

1. Flow of Events

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. etc.

Alternative Flows

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.**

Delete Report

a. etc.

2. Special Requirements

Platform

...

3. Pre-Conditions

- A data element exists on the machine and has been selected as the current element.

4. Post-Conditions

Success Post-Condition(s)

System waiting for user interaction. Report may be loaded and displayed, or ....

Failure Post-Condition(s)

- System waiting for user. The following lists some state possibilities: ...

5. Extension Points

None

If-statement style

Programmers inevitably get around to wanting to write "if" statements in the text. After all, it certainly is easier to say, "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 to the main use case.

If there were only one "if" statement in the use case, I would agree. Indeed, there is nothing in this model of use cases that precludes the use of "if ... then ... else". However, once there are even two "if" statements, then the required behavior of the system 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 report back later 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: Rewrite the following use case, getting rid of the "if" statements, and using goal phrases at the appropriate levels and alternate scenarios or extensions (for more on extensions, see section *Failure repair and roll-up*).

"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 simply 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 "Register Loss" in *Writing Samples: High-level to Low-level* your first test case. It shows a use case full of parallel, asynchronous, exceptional, co-processing activities, and how natural language deals with that in a way still quite easy to understand.

Whenever I consider the difficulty of creating a formal and yet easy-to-use language for use cases, I am impressed by the power of natural language and simply written prose. These things, which are difficult or complicated in programming languages, are easy to write and easy to understand in prose.

Interaction diagram style

A use case shows the interactions and internal actions of actors, interacting to achieve a goal. A number of diagram notations, particularly sequence charts, collaboration diagrams, activity diagrams, and Petri nets, can express approximately the same. Therefore, 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.

However, the diagrammatic notations suffer from two usability problems. The first is that you would like the end users and business executives to read your functional requirements document or business process description. Theses readers typically are not trained on visual notations, and have little patience to learn. By using the diagrammatic forms, you are fundamentally cutting off your core 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 impossible 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.

The UML use case diagram

The UML use case diagram, consisting of ellipses, arrows and stick figures, is *not* an interaction diagram, and 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 therefore 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" or "uses". 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, there is no way to show the sequencing of the action steps in the use case diagram, and no way to show which actor is doing the action. So the diagram is missing information at a very fundamental level. Also, it makes little sense to create ellipses for the internal, validation and state-changing actions of the system.

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.

In the ongoing evaluations of different forms that I hold, the diagram that is most often mentioned as useful is the topmost level of diagram. This is the one that shows the external actors and the user-goal use cases. It corresponds to the old "context diagram", and to the Actor-Goal list described in the section *Actor-Goal List*. The value of decomposing the ellipses drops rapidly from there. I discuss this more in the section *UML and the use case relations*.

Conclusion about different formats and styles

All of the above different forms and formats express the same basic information (with the exception of the use case diagram, of course). The recommendations and guidelines of this book apply or can be applied to all of them. Therefore, do not fuss too much about which format you are obliged to use on your project.

My preference is for single-column, numbered, plain text, full sentence form. I stay with that preference because when I visit a project and write the same use case in two or three different formats (one column, two column, numbered, unnumbered, table, plain text), the readers almost universally select the single-column, numbered, plain text, full sentence form. If your preference, or the preference of your readers (or team leader!) runs differently, then simply choose another form and get on with the writing.

Exercises

Exercise: Find the 4 most significant writing mistakes in the "Log In" use case, page 92, in Applying Use Cases.

11. Extension conditions

Extension conditions are where the really interesting parts of the requirements and design live. Capture them in the requirements, consider them during design, and test for them during testing.

Work in two stages:

1. First, brainstorm and include every possibility you and your colleagues can conjure up.
2. Second, evaluate, eliminate, merge, the ideas.

Capture all the alternative conditions in the section of the template called "Extensions" or "Alternate Courses".

Recall that there often is more than one way to operate a use case successfully. The form, however, builds from one, designated success sequence, patching the other successful paths, just as it does failure paths, onto that "main" success scenario. So include both failure conditions and the alternative success conditions.

Generally, I have found people do best when they list and evaluate the Extension conditions to the best of their abilities first, before starting to consider how the system should react to those conditions. The two reasons seem to be energy, and taking breaks. Considering how to handle the failures takes a lot of energy. By the time the team has written a few failure handling scenarios, they have lost the energy and flow needed to make a good failure conditions list. If, on the other hand, they complete the brainstorming, they can take a break, and the structure of the back end of the use case is in place for later work.

Brainstorm all conceivable failures and alternative courses

First, brainstorm all the possible ways in which the scenario could fail. Be sure to consider:

- The primary actor sending in bad data or requests.
- Failure to pass validation checks.
- Non-response or failure response from secondary actors.
- Inaction by the primary actor (timeouts).
- Bad data discovered on disk (even though this seems like an internal failure).
- Critical performance failures of the SuD that you must detect.

Include in your brainstorming all the alternative *success* paths you can see. The reason for this is that the use case form is a linearization of a basically cross-connected behavior pattern. We shall always start from a single, mainline success scenario. All other success paths will be built as extensions on this, and will be labeled by when and how the alternative success path varies from the main success scenario. It is for this reason that use case templates name that section heading "Extensions" or "Alternate Courses".

The most impressive performance I have seen on brainstorming failures was during a course, when I asked the students to brainstorm the failures on the ATM example. After 10 minutes, I called a halt, and two students said, "But we're only on step 3!" They had filled up an A4 page with all the failures they could think of in the first three steps, failures that had never even crossed my mind. Even after we crossed off the inappropriate conditions, they still had by far the most extensive list of failures I have ever seen.

A small, true story.

The opposite, negative effect of inadequate brainstorming happened on a real and large project.

Like most developers, we didn't want to consider what would happen in case the program encountered bad data in the database. So, like most developers, 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 of the company decided to see how his favorite customer, a large retail chain, was using his new sales devices. So he started our brand-new system and inquired about this large customer. The system replied, "No Data Found". A word to describe his state might be, "excited".

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, which had caused the error. Secondly, the error message was inaccurate. It should have said, "Some data missing." But more importantly, we had missed that *how the system reacts upon detecting bad internal data* is really part of the external requirements.

So 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 moral of the story is, include some internal errors, such as discovering missing data, in your failure list.

Two notes about this possible failures list.

- Your list should be longer than you will eventually use. The point during this stage is to try to capture all the aberrant situations that the system will encounter in its life. You can (and will) reduce the list later.
- You list will not be complete, even though you try. It regularly happens that you will discover a new failure condition while you are writing the failure-handling scenario, or when you add a new validation step somewhere inside the use case. Don't worry about this, just do the best you can during this brainstorming stage, and expect to come back and add to it, later.

Rationalize and reduce the extensions list

Some of the failure conditions you listed are likely to be out of scope, undetectable internal errors, or similar to other ones in the list.

Go through the list carefully, weeding out the ones you decide the system need not handle, and merging those that have the same net effect. When you are done, the list should ideally show all the alternative situations the system must handle, and only the ones that it must handle.

Here is a set of criteria to use in working through the list:

- Include only failures you can detect.
- Exclude most failures that are merely incorrect performance of the SuD. In most cases, the system will not be to detect them, so you can't handle them anyway. "ATM issues transaction receipt" is an example of an action that is not considered worth detecting failures on, and so the system will not detect that perhaps the dispenser slot is jammed.
- In some rare cases, where the internal failure is considered critical, be sure to include an explicit validation for the success of the action. In the case of "ATM issues cash", the

ATM has a specific check to make sure the cash actually came out of the slot. If it fails, the ATM will retract the cash, and not debit the customer's account.

- Include internal failures that have an effect that reaches the outside world. Bad data in the database is a standard example. Power failure is another.
- Include bad behavior of the external actors, since the system cannot control their actions.
- Consider timeouts, in particular. What happens if the primary actor walks away, or the secondary actor never replies?
- Include any condition that requires the system to do something to protect an interest of a stakeholder. Exclude any condition that does not require the system to react.
- Ask yourself, "Will you require the system to test for the failure of this?"
- When it doubt, include it. The damage from an extra condition is small compared to the damage of a forgotten extension (as in the bad database story, above).

At the end of this work, you will probably have a short and simple main success scenario, and a long list of failures and alternative courses to consider.

Exercise: Brainstorm the things that could go wrong during operation of an ATM.

Exercise: Brainstorm the things that could go wrong with the first user-goal use case for the PAF system.

Writing style

Write the condition as a phrase saying what is wrong or different here, compared to the main success scenario. Occasionally a sentence (expressed in the past tense, as though it already happened) is appropriate.

Examples:

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

Style with numbered steps. When the steps in the main success scenario are numbered, the condition names the step or steps where it might occur:

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

Small conventions about numbering alternative conditions.

Put a letter (a, b, c, ...) after the step number, to allow for multiple failures or alternatives at the same step of the main success scenario. Put a colon ("**:**") after the condition to make it clear to the reader that this is a condition and where the condition ends. You will be amazed how much easier the extensions are to read with this little convention.

If the condition can occur on several steps and you want to indicate that, simply list the steps where it can occur:

Example: 2-5a. User quit suddenly:

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

Example: *a. Network goes down:
*b. User walked away without notice (timeout):
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 a validation for the condition.

For more examples of ways to construct extensions, see *Writing Samples*.

Style with unnumbered steps. When the steps in the main success scenario are not numbered, you cannot refer to the specific step where the condition occurs. Therefore, make the condition description sufficiently long and accurate that the reader will know when the condition might occur. Put a blank line or space before each, and put the condition in *italics* so that it stands out. See the *CRUD use case* example in *Writing Samples*.

12. Failure repair and roll-up

Failures and success alternatives

For each extension condition, write how to deal with the alternative success path, or how to patch up the failure, either succeeding with the goal, or abandoning it.

The scenario fragment you are writing picks up at the step you numbered or named in the condition. E.g., "2a. ..." means that you will start the discussion at the moment of step 2 that the alternative condition is discovered. The condition has been discovered, now what happens?

Continue the story from this moment, in exactly the same way as you were writing in the Main Success Scenario. Use all of the guidelines about goal level, verb style, full sentences, that were discussed in *Overall Writing Style*. What you are writing is a scenario - or rather, a fragment of a scenario.

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

- The step that failed has been patched up, so that at the end of the extension writing, the story is exactly as though the step had not failed (example: the customer finally entered the right password, so it is as though no password error had occurred).
- The failure condition you indicated was preventing the step from happening, so, after the extension writing, the story starts up again at the beginning of that step (example: the user was unable to locate the web site, or the clerk was unable to find the customer's record, but now the web site/record has been found).
- The use case must abort due to total failure (example: power goes down, or the user quits).

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 third case, it is generally not necessary to say more than, "the use case ends", or "fail!" because the steps show the system setting the stakeholders' interests into place.

As a result, most extensions or alternate courses do not say where the story goes back to after the extension. Usually, it is obvious, and writing, "go to step xxx", after every extension makes the overall text less rather than more clear. 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 xxx".

Examples of all of these situations can be found in *Writing Samples*.

Small writing conventions.

Indent the numbers for the scenario fragment following a condition.

Example:

"2a. Insufficient funds:

2a1. System notifies customer, asks for a new amount.

2a2. Customer enters new amount."

Failures within failures

Inside the failure-fixing scenario fragment, you may find yourself with a possible other failure on your hands. In particular, your step may invoke another use case, or call for a validation step. Either could fail.

Simply continue the indentation, condition naming, scenario-fragment writing as has been described so far. At some point, your indentation and numbering will become so complex that you will decide to break the entire handling of the extension out into another use case entirely.

When you break out a sub-use case, simply give it a proper use case name, open up the template for a new use case, and fill in the details that you pulled out of the calling use case.

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

Here is an example, taken from *Writing Samples*:

6a. Clerk decides to exit without completing minimum information:

6a1. System warns Clerk it cannot finalize the form without date,

name or policy number, and adjuster.

6a1a. Clerk chooses to continue entering loss or to save as "intermediate".

6a1b. Clerk insists on existing without entering minimum information:

System discards any intermediate saves and exits.

In general, the overall cost of starting up a new use case is large enough that most experienced people delay breaking a scenario fragment out into its own use case, for as long as possible. The consensus I have heard is that if there were one more failure after 6a1b1, then they would create a new use case . However, if they can finish off the description at 6a1b1, they will. Notice in the

above example, that the writer did not even number the last step, in order to avoid the clutter of the last "6a1b1".

When to create a new use case, and when to keep adding to the current use case, is a matter of taste and practice. There are two things to recognize:

- There is a real cost associated with creating a new use case. It is cognitively more complex, it has a header and footer that needs filling out, it must be labeled, tracked, scheduled and maintained. Therefore, keeping the scenario fragment inside the use case has a better economic cost, up to the point that the use case just gets too hard to read. I put the limit of readability at about 3 pages of use case text, and four levels of indentation.
- The decision to move some piece of the story out into a use case, or from a small use case back into its calling use case, is a "small" decision. The use of goal phrases makes it a minor matter, since every step is written as a goal, and every use case is titled with a goal. There is no "glamour" for a step to become a use case in itself, and no "insult" for folding a use case into a couple of steps.

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

Exercise: Pull out 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: Write the full use case using the PAF system, filling out the failure repair steps.

Exercise: If you haven't already done so, fix the spark plugs use case given a few pages ago.

Failure roll-up

One of the nice things about use cases is that low-level failures roll up into high-level failures. It is one of the ways in which we avoid an explosion of scenarios while describing fairly complex behavior.

The user of the calling use case usually does not care why the sub-use case failed. Perhaps the user and the system in the sub-use case went through 3 or even 10 of the extensions, trying to patch up some failures in that sub-use case. At the end of the story, the end result was that the goal failed. In the calling use case, the user and system must try to patch up that failure.

As an example, suppose that you are working on our example Personal Advisor / Finance package, and are writing the user-goal use case, Update Investments. One of the last steps in Update Investments says, "User has PAF save the work", which calls the use case Save Work. Somewhere in Save Work, you will write extensions for "file already exists (user doesn't want to overwrite)", "directory not found", "out of disk space", "file write-protected" and so on. After the user has exhausted all the possible ways of saving the work, they end up back in Update Investments, faced with the fact that they can't save their work. What are they to do now? that is what you have to write in the Update Investments use case.

Usually, the calling use case often contains only one failure condition for many lower-level failures. This is "rolling up" the failures. The situation is portrayed diagrammatically in the following figure, which shows the "striped trousers" image of nested use cases.

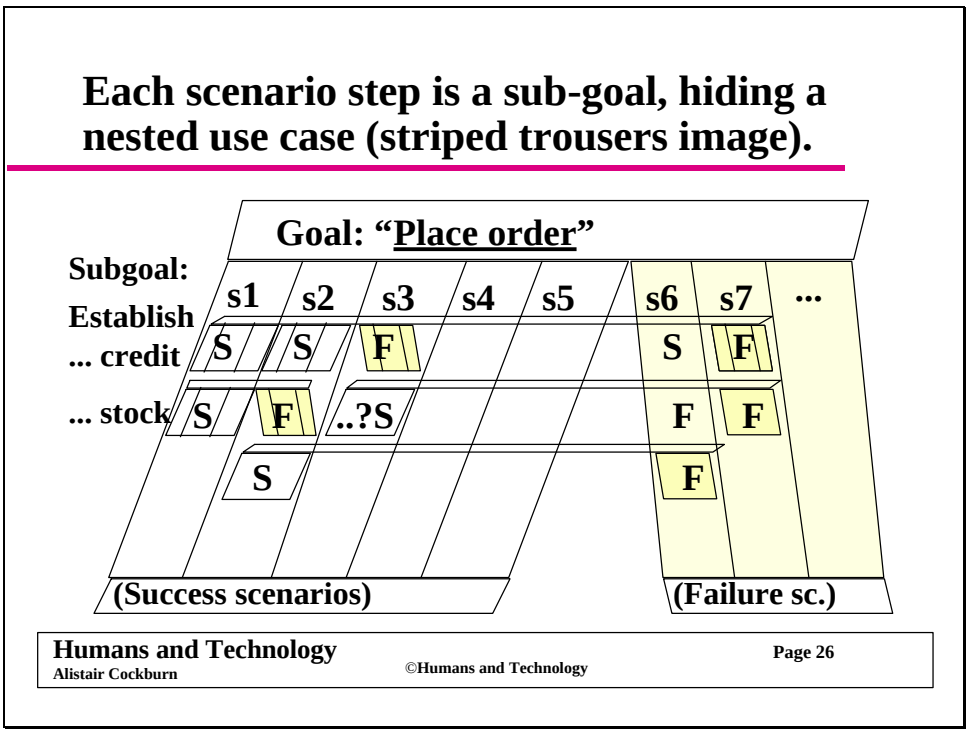


Figure xxx. "Striped trousers" image.

The belt of the trousers corresponds to the goal (Place Order in the figure, Update Investments, in the text example). The left leg corresponds to all ways of achieving success with the goal. The right leg corresponds to all ways of failing to achieve the goal. Each stripe corresponds to one scenario. Some scenarios end with success, some end with failure.

In naming a step, the outer use case is calling up a lower-level use case, i.e., a smaller pair of striped trousers. The step names the goal (the belt of the smaller trousers). The actor works along one or more scenarios (stripes) in the sub-use case. But in all cases, the sub-use cases ends by coming out the left leg (success) or the right leg (failure). The outer trouser (use case) is not interested in which stripes were used of the inner trouser, only whether the exit was out the left or the right leg. That determines which stripe of the outer trouser the actor is working along.

Practitioners of data-flow diagramming and functional decomposition report that this failure roll-up is one thing that makes use cases attractive to them. Even at the highest level of use case, failures are considered and reported, and rolled up into the appropriate vocabulary for the level.

13. Technology or Deferred Variations

Extensions serve to capture, "what the system does *now* is different." Most of the time, that is what there is to capture. There are a few times when extensions are not the right thing to build. Occasionally, what you really want to do is to put in a marker that "at this level of discussion, I

can't elaborate on a few, key differences in how step works, but we really need to keep track of these items." As Gerard Meszaros said, "The *what we are doing* is fundamentally the same, but the *how we might do it* is different."

I have come across fairly few occasions when this is needed. The few times I have encountered it, were for technology differences in the input and output of the system. Other people have encountered the need as special flags set by the data. Examples:

- You need to pay a customer for returned goods. You can pay by check, electronic funds transfer, or credit against the next purchase. At the level of writing the use case or its extensions, you write the appropriate thing: "Repay customer for returned goods." What you want now is a place to mark the three different ways of paying.
- You are specifying a new ATM. Technology has advanced to the point that customers can be identified by bank card, eye scan or fingerprints. Your main success scenario has the correct phrase: "User identifies him/herself, bank, and account #." What you want now is a place to mark the three technology choices.

Each of these technology choices would, at some lower level of use case, unfold into its own extension. Each has a noticeable impact on the cost and work plan, so you would like to preserve them, and "roll them up" to the highest level use case possible, as is possible with failure conditions.

The technology or deferred variation section of the use case template is rarely used. If you find yourself putting conditions and action steps there, then you are using the section incorrectly. The section is to mark lower-level technology variations that will need to be handled.

In UML, we would create an empty, "generic" use case called "Identify customer", and place three specializations under it, "Identify customer using card", "Identify customer using eye scan", and "Identify customer using fingerprint". We would then be stuck with those ellipses on the diagram - there is no way to roll them up.

In the textual format, also, we could use the generic use case style of writing:

1. Customer identifies self, bank and account #. Identification of him/herself is done using one of
   - bank card and PIN
   - eye scan
   - fingerprint
2. ...

However, this is cluttering up the writing of the main success scenario with information that really does not belong there. The better way is to use the Technology Variations section:

1. Customer identifies self, bank and account #.
2. ...

...

Technology variations

1a. Customer identification is done with one of
- bank card and PIN
- eye scan
- fingerprint

This puts in a marker that at some point, we might write a use case "Identify customer with bank card". On the other hand, this one phrase in Technology or Deferred Variations may be sufficient specification of behavior. Quite possibly, there is nothing more to say about the behavior of the variation. The difference might only be in the data format and technology of the interface.

Several examples of real uses of Technology or Deferred Variations are shown in *Writing Samples*.

14. Linking to other use cases

In the purely textual style of use case described in this book, every goal phrase is a potential reference to another use case. In UML terms, every step potentially "includes" another use case. As just described in the previous section, any step can be made into a use case, any use case can be included directly within another. This is one of the attractions of the goal-based use case model.

Also, as described in the preceding section, any use case might fail. Therefore, when a sub-use case is called out, the writer must take care of handling the failure of the sub-goal. This writing is described in *Failure repair and roll-up*.

Exercise: Find a condition in a user-goal use case for the PAF system that requires a sub-use case to be broken out. Write that sub-use case, and sew it, both success and failure, back into your user-goal use case.

Extension use cases

On the rarest of occasions, you need to track the extension scenario as its own use case, in its entirety, triggering condition and all (see next section for when this might occur)

In this situation, proceed exactly as with a scenario extension or failure condition, except starting with a new use case. Here is how it works, illustrated with the situation of a person using the ATM of a competing bank.

- Open up a new, "extending" use case ("Use ATM of competing bank", in this example).
- In the Trigger section of the extending use case, write what condition in the other, or base use case causes you to be here - exactly as you would have written it in the base use case, but with both the step number and the base use case's name (e.g., "Detected that the customer's "home" bank is a competing bank, in step 2 of use case *Use ATM*.").
- Fill in the actors, stakeholders, etc., as usual.
- Pick up the story, step 1 of the main success scenario, at the moment after the condition has been detected. Describe the normal handling of the situation.

- Consider alternative courses, alternative success paths, failure conditions, exactly as usual, and write them exactly as always.
- Refer to the places in the base use case, where the behavior is different, as you need to. Complete as usual.
- Say how the success and end states of extending use case fit back in with the base use case.

On the one hand, this new use case is simply an extension of the base use case that got broken out into its own space. On the other hand, this new use case is odd because it specifically names internal parts of the base use case. For this reason, keep the use of "extending use cases" to a minimum. They should not be the normal writing style, but rather, the most exceptional..

There are two primary differences between this form of linkage and the standard linkage of simply naming the goal of the sub-use case:

- In the recommended linkage, the condition phrase sits inside the base use case. The base use case, in its failure handling, names the sub-use case simply by naming the goal in the step (see any example in this book in which an extension step contains a phrase in italics). When an "extension use case" is used, however, the condition is *not even mentioned* in the base use case! The location of the failing step, the failure condition, and the failure handling are all put into the extending use case.
- In the recommended linkage, the base use case actually names the sub-use case. You can read the base use case and know what other use cases can be triggered. In the other form of linkage, the base use case *contains no reference* to the extending use case! You cannot know which other use cases are busy extending the base use case, without using a tool of some sort to scan the entire use case database. This is intentional, and part of the use case form.

When to use extension use cases

I have seen only three justified uses of this sort of extension in six years of reviewing use cases. Two of the three occasions were when the extension was very large, very significant, and needed to be tracked specially. They were really exceptional cases.

The most natural occasion on which to use an extension use case when there are many asynchronous services the user might request, services which do not disturb the main flow of the main success scenario. This sort of situation is common with shrink-wrapped software such as a word processor.

In a word processor, the user's main activity is typing. However, he or she might suddenly decide to change the zoom factor ... or the font size ... or run the spell checker ... or any of literally dozens of things. In most of these cases, we want the operation of the typing activity to remain ignorant of what else is happening to the document. More importantly, in the requirements document, we want software development teams B, C and D to be able to come up with new ideas, new services, and not have to force team A to update their use case, "Write text", for every new service that gets thought up. We want to be able to extend the requirements document without trauma.

In such a situation, it is quite proper and appropriate that the base use case remain ignorant of the many extension use cases being built, and it is then possible to add any number of extension use cases without damaging either the requirements for the basic software, or the team doing the development.

Exercise: 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 sew this new use case into your previous use case(s) about withdrawing money in two ways: as a sub-use case that your base use case simply refers to in the ordinary way, and as an extension use case. Discuss with a colleague which you prefer, and why.

Related Concepts

1. All the use cases together

All the use cases together make up an ever-unfolding story about the system. Usually there are two or three key primary actors, with a few very high level goals. You will write one strategic, probably business use case for each of these primary actors, at the outermost design scope possible. Each of these strategic use cases will probably name the user goals the primary actor will care about at different moments.

The user goal use cases are the unfolding of the top levels of the story. The user goal use cases unfold into subfunction use cases, which may unfold again into more subfunction use cases.

An executive can read the top levels of the story, and stop reading at any point. The system implementers will probably only start reading at the user goal level, and read down into the subfunctions.

How can you tell when you are done writing all the use cases? The answer is, when you have identified all the primary actors, and all their "blue" goals, and written all the blue use cases, along with the strategic and subfunction use cases needed to support them.

2. Business process modeling with use cases

You might observed by now that this "actor - goal - stakeholder" business seems applicable to describing the business process of a company. That is entirely correct, and there are advantages to using use cases for business process modeling.

Most of this book appears to be about designing software systems. That is because most of the people writing use cases at the moment are using them for software systems. However, everything about writing use cases in this book applies equally for business use cases. One person wrote in and said,

> "I hope you plan on addressing business use cases in your book, and how they can relate to system use cases. I've recently been doing research on business process engineering, and outside of the work by Ivar Jacobson, nobody seems to like using business use cases, but rather opt for workflow/activity diagrams to model business processes."

In fact, quite a number of people are using use cases for business modeling, and they do quite well for describing the operation of a business in an easy-to-read form - the ever-unfolding story that starts with successful achievement of a goal, and then adds complications as extensions to it. Those who use a different modeling technique to document the business process are missing out on the synergy between business process use cases and system use cases.

Business process modeling with use cases

This book contains several examples of business use cases: *Use case example 1: Get paid for Car Accident* in the opening chapter, and in *Writing Samples*, *High-level to low-level*, and *Buy Goods*. In *High-level to Low-level*, you can see two different writing styles for business use cases,

and also the differences business and system use cases for the same goal "Handle a Claim". A fragment of a purely business use case is shown below.

1. A reporting party who is aware of the event reports a claim 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.

Stakeholder & interest:
Insurance company Divisions who sell Insurance company policies
Insurance company Customers who have purchased policies
Depart of Insurance who sets market conduct
Claimants who have loss as a result of act of an insured
Insurance company Claims Division
Future Customers

Business use case fragment from *Writing Sample: Handle a Claim*.

The business process starts with an external actor who views your organization as a system that offers a service. That actor, probably a person or an organization, has a goal, and initiates an interaction with the organization. At this moment, we are in exactly the situation described in the Actors & Goals model. In addition, the organization must satisfy the interests of certain stakeholders, so the Stakeholders & Interests model is also in play.

The business use case, then describes how the business or organization responds to a triggering event, to satisfy to the goal of an external actor and the interests of the stakeholders. It is written exactly as any use case.

Relating business and system use cases

The good news is that a "business" use case has exactly the same appearance as a "system" use cases, and so all the training in writing, reading and reviewing use cases can be applied to both business use cases and system use cases. The *ever-unfolding story* can start in the business use cases and continue to unfold, naturally, in the system use cases. That is the synergy that business and system use cases offer. The system use case is an expansion of the system use case, a specific design of the business using technology.

The bad news that since a business use case has exactly the same appearance as a system use case, both writers and readers will accidentally cross levels, writing system behavior into the business use cases and business operations into the system use cases. This is not necessarily a bad idea, if done deliberately, but often, the writers and readers don't realize they doing so.

It is fairly common that a reader will start reading a business use case, and criticize it for being "too high-level". They don't notice that that it is a business use case and not intended to provide system behavior detail. Alternatively, a business use case writer will accidentally include a great

detail of system behavior. The readers of the use case, business executives, lose interest in the writing because it contains inappropriately low levels of detail (read also, *User goals and level confusion*, and *Mistakes fixed: Raising the goal level*).

How can you help reduce the mistaken confusion of business and system use cases? Firstly, and always, mark both the scope and level in the use case template, read them both and train your readers to read them both at the start of every reading.

Marking the two fields is still insufficient, though. People need a much stronger reminder. You might use one of two alternatives:

- Use a differently shaped template or type face. I actually don't know how to make this work effectively, given that the same tool and template will probably be used for both. I include it because perhaps some reader will find an effective solution along these lines.
- Attach a graphic logo to either the use case title or the "scope" field. See *Using icons to indicate scope and level*.

I have only experimented with these to date, and the tools have not been quite right for realizing either. But the situation seems to require a strong visual difference, to help people orient themselves for both the writing and the reading.

Designing business to system use cases

You may have noticed that I have been carefully using the phrase, "business process modeling" instead of "business process reengineering" or "business process design". A use case *documents* a process, whether that process is the business process or the process for using a computer. The process it documents can be the design of an outer system, or the specification for one of its component subsystems. Specification alternates with design at each level of design scope, and the use case documents the specification or the design.

I want to be clear that the design of the business does *not* directly imply the specification of the computer system. There is a core business process, and then there is an act of invention regarding how technology might be used to link different parts of the business process for efficiency.

Doing "business system modeling" and then "deriving" the system specification directly will leave you with the same business process you had before. That is all well and good if that is what you are after, but most organizations want their business processes to be *better* after the introduction of technology.

The improved business process will be *invented* from the core business process and ideas about how technology might help. The effectiveness of the new business process depends on the particular inventions named.

Let's look at the unfolding of design scope, how the design of the business leads into the specification of the computer system, and how use cases play into the picture.

Core business requirements

You are about to do major reconstructive surgery on you business' overall processes, perhaps following the lead of Hammer's Reinventing the Organization. What you need to find is the ultimate *core* of your business. You probably will start with

- *Stakeholders* in the behavior of the company and any outcome,

and work out what their interests are. Then comes some introspection and invention as to what business you truly are in, after which point you may finally consider the entire company or organization as a single, black box reacting to outside stimuli. At this point you get:

- *External, primary actors* with operative goals you propose the organization satisfy.
- *Triggering events* that the organization must / should / might respond to.

That is the starting point for the design of the business process. There are no use cases at this moment, although the defining framework for the use cases, discussed early on, is present.

Again comes raw invention, or design. Hammer gives many examples of how different acts of invention lead to different business designs, and the relative effectiveness differences, in Reinventing the Organization. So far:

- We documented the stakeholders.
- We invented the proposed overall responsibility and services of our system / organization.
- We documented and also invented the actors and events we wish to support with that overall responsibility and service list.

Next,

- we invent how the organization will handle those events, actors and stakeholders.
- we document that as the business process, perhaps using use cases.

Business operations model in use cases

Once you have a core actor-event-stakeholder model, and have invented the way you think your organization might respond to the events and satisfy the stakeholders, then you can write down the new business process. If you are not reinventing the organization, then you can go straightaway to describing the way the business works using use cases.

Either way, document the business process as use cases, using the same form and rules as any other use case. Treat the outermost containing system (the company or overall organization) as a single, black box, initially. Write the use cases showing the interaction of the organization with the external players, even showing the interaction between external actors, as in *Buy Goods* (see *Writing Samples*).

After you have the use cases for the organization as a black box, open up the black box, and write "white-box" business use cases (described in the next chapter). These show your business process design, the interactions of the players in the organization. Very often, technology is not mentioned in these use cases, but rather, the essential goals of the people and sub-organizations.

Most often, the technology, including computer systems, is not an active player in the overall process, but is a repository, a conduit between players, just as a telephone is. Deciding how that

conduit can be improved to improve the overall business process, is the next act of invention and design.

- We invent how the technology might support the desired process, or we co-invent both the business process and the computer system requirements at the same time.
- We document the system's functional requirements (mostly free of user interface design), perhaps using use cases.

Invent and document business process with technology

Typically, the computer system serves as a holder of information put in by one person and picked up by another. In principle, the computer can be replaced (in the descriptions) by baskets of paper shipped from person to person. Your task, or your team's, is to invent how active conduits such as a computer, or perhaps a fleet of palm computers, or walkie-talkies, can simplify, improve, and streamline the process. That act is an act of design, and the use cases do not dictate the answer.

Naming the use of the technology in business use cases is similar to naming the user interface design in system use cases, and similarly brittle over the (longer) scale of time that a business process operates. In the case of business use cases, you may decide that such documentation is a good use of time and resources.

Document business operation with technology

If you name the computer system's place in the process, you are inventing the *responsibilities* of the computer system, as discussed in the Actor & Goals model of use cases.

That is all right to do, as long as you are aware that a use case containing system operations is actually a specification of the system to be designed, or a documentation of the system already designed, and hence is redundant with the system specification use case.

How to handle the overlap between business process use cases showing technology, and system requirements use cases, is a matter of personal preference. In *Use case example 2. Handle a Nightime Deposit* (also included in the *Writing Samples*), the writer chose to include the business process steps as a *context* for understanding the actual trigger and use of the system. This person intended the use case as a software system functional requirements specification, and included the surrounding business process to show that the new system function

In *Writing Samples: Handle a Claim (business)*, however, the writer chose to leave out the technology, creating a purely business use case. The documentation of just the business goals left open the invention of how the computer might assist. In *Handling a Claim (system)*, the writer shows how the technology supports the business process.

My personal preference is to write the business process without technology, leaving the invention open, and then to write the contextual business process steps into the system requirements use case, showing how the system operation fits into the ongoing operations.

After writing the functional specification for the computer system, finally

- We invent a user interface and internal system design to meet the functional requirements, sometimes we co-invent both the functional requirements and the user interface or internal design (the latter two constraining what is possible).

- In rare cases, we document the user interface or internal design with use cases (see *Writing Sample: Documenting a Design Framework*).

Further comments on business modeling and use cases

Use cases are a concise and readable account of a system's behavior with respect to actors, events and stakeholders. As such, they can serve well in business process modeling. When they are used, there is a completely natural fit to showing how the computer system's operation fits in with and supports the overall operation of the organization. The system's operation is part of the *ever-unfolding story* that the top-level business use case starts, with documenting alternating with designing/inventing.

In *Other Voices*, FirePond's Russell Walters comments on his experiences using business process use cases with system requirements use cases. Bruce Anderson of the IBM Consulting Group comments on different forms of business process modeling feeding the system requirements use cases. Independent consultant Steve Adolph comments on using use cases to discover requirements vs. using them to document known requirements.

3. White-box use cases

Most of the use cases people write are written to *specify* a system prior to implementing. But that is not the only use for use cases. There are two occasions when people want to expose the workings of the system and describe how the parts of the system collaborate to deliver the system's services. "White-box" is a term meaning showing the insides of the system under discussion, and so these use cases are called "white-box" use cases.

The two times most common times when white-box use cases are written are:

- Writing use cases describing business processes, prior to reengineering the business or as a contextual part of writing system requirements.
- Using the use case form to document the design of a system.

See *Writing Samples* for an example of white-box use cases.

4. The Missing Requirements

Use cases are only "chapter two" of the requirements document, the functional requirements. That means they do not contain information such as performance requirements, business rules, user interface design, data descriptions, finite state machine behavior, priority, and probably some other information. There are several ways to deal with this missing information.

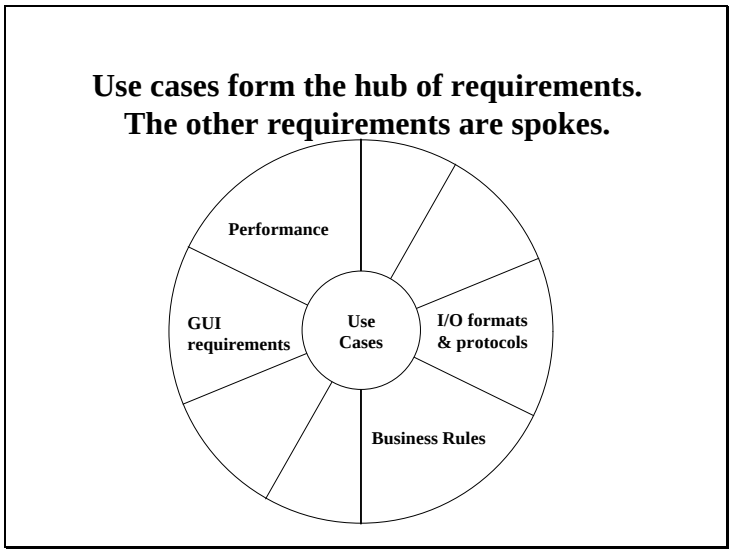
Some of the information can profitably be attached to each use case. Use case priority, expected frequency of occurrence, performance demand, delivery date, list of secondary actors, and open issues, can be placed at the end of the use case. The template I use has a section at the end for all this sort of information. Different projects adjust this collection of information to contain whatever they feel is important to them.

In many cases, a simple spreadsheet or table captures the information well. Many people report using 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, you can use the other columns for:

- primary actor

- trigger
- delivery priority
- estimated complexity
- probable release
- performance requirement
- state of completion
- ...and whatever else you need.

You can then resort the table by actor, or priority, or completeness, or complexity, and get alternate views of the situation.



Hub & spokes image of use cases and the requirements file.

The third way in which use cases help with the other information, is that they act as a sort of "hub" that links together different sorts of information. I/O 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 description (see more on *Data Descriptions*, below). Complex business rules do not fit neatly into the use case narrative, but again, we can link to a page containing the business rule. 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.

The thing to be careful about is not using use cases to describe requirements that don't fit well into the use casel form. Use cases are for describing interactions. Occasionally, I hear someone complain that it is hard to detail the requirements for a two-tape merge operation in use cases. Indeed, I wholeheartedly agree. A two-tape merge operation is best described using algebraic or tabular forms, not use cases!

Perhaps 25% of the entire requirements file is suited to for describing in use case form, the other 75% will be written using other forms. It just happens that that 25% is a central part of the

requirements file, and connects many other requirements, as the "hub" on the hub-and-spoke figure indicates.

Data descriptions

The constant recommendation in the book has been to write the "intent" of the actor, to write the data that gets passed in the most summary form. E.g., the sentences should read similar to, "Customer supplies name and address."

However, it is clear to every programmer that this is not sufficient to design to. The program designer, and the user interface designer, need to know what exactly is meant by address, which fields it contains, the lengths of the fields, the correction rules for addresses, zip codes, phone numbers, and the like. All of this information belongs in the requirements file somewhere (and not in the use case!).

Recall the discussing of "precision" in the section, *Precision, Tolerance, Hardness*. What we are after is a way to manage our energy in getting the precision needed to carry out the design. I separate the information into three levels:

- Information nicknames - the first level of precision
- Field lists - the second level of precision
- Field details & checks - the third level of precision

Information nicknames. We write "customer's name, address, and phone number," to indicate that three parcels of information are needed. We expect to unfold the description of each of *name*, *address*, and *phone number*. For writing and reading a use case, these nicknames are sufficient and appropriate. 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, it is better to break the details out of the use case, and simply link to those descriptions from all of the use cases that reference it.

Field lists. It may happen that by "customer's name", we intend to collect first name, middle name and last name (or, perhaps, just the middle initial). What is it exactly is needed for "address" - addresses around the world have so many different formats? The second level of precision is to list what separate field items are needed for each of the nicknames.

There are many strategies for dealing with the second level of precision, and it is not for this book to standardize on any one. Here are two. You may know of another (see also Constantine and Lockwood's book, Software for Use and Luke Hohmann's book, GUIs without Glue).

The first strategy is to have one page of description for each nicknamed item. Thus, under "customer name", you will identify that you need three fields: the customer's first name, middle name, last name. That's all. Over time, you will add to this page, until it contains all the details about those fields (see *Field details & checks*, below).

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. It is quite likely that these three parcels of information will tend to 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 a database page for "name, address, and phone number". In that page 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. The difference between the two strategies is that in the second strategy, (a) you put clusters of nicknamed information onto one page, and (b) when you expand to more precision, you will not expand on the same page, but will put the extra detail on a page per 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 these data items. You might also use different people to define the second and third levels of precision for the data. It is a good thing that the second level of precision is kept separate from the use case.

Field details & checks. What the programmers and database designers really need to know is detail on the order of: "How many characters long can the customer's name be?", and "What restrictions are there on the 'Date of Loss' field?" These are field types, field lengths, and field checks. Different project teams put this information in different places. Some put it into a chapter of the requirements file called "Data requirements", or "External data formats". Some put it into another Lotus Notes page set or database called "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 & 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.

5. Parameterized use cases

We can take advantage of the fact that naming a goal in a step is like a subroutine call (see also the "includes" relation in the UML section), and use cases are a human-to-human communications medium, to shorten our writing.

Sometimes we are faced with writing a series of use cases that are all about the same. The most common example is Find a Customer, Find a Product, Find a Promotion, and the like. If we are using casual use cases, there is not a problem with just writing four or five of these use cases. However, if we are using the fully dressed template, and we see 5, 10 or more of these Find a Whatever use cases coming, and if we are going to hand these use cases to different developers, then we would like something better.

It seems that finding a thing, whatever thing it might be, must use basically the same logic:

- user specifies the thing
- system searches, brings up a list
- user selects, perhaps resorts the list, perhaps changes search
- in the end system finds the thing (or doesn't)

If that is true, then it is likely that one programming team will create the generic searching mechanism, and various other teams will make use of that in some specialized way.

What is different each time, is

- the searchable qualities of the thing
- the display details of the thing
- the sorting details of the things.

So we write euphemisms, or nicknames, for the parts we want to swap out or substitute on a case-by-case basis. We call these, "parameterized use cases". The *Writing Sample: Find a Whatever* shows one style of writing a parameterized use case.

The writer of the use case writes

- "The user identifies the searchable qualities of the thing."
- "The system finds all matching things and displays their display values in a list."
- "The user can resort them according to the sort criteria."

The underlines phrases indicate the writer about to write the step, "User finds a customer", needs to supply three things to make it clear what is going on:

- what is the thing (customer),
- what field of the thing to display (this is the second level of precision of data)
- what sorting fields to use (also in the second level of precision of the data).

One project separated their data descriptions into a set of Lotus Notes pages, so they could write, "Find a customer using Customer search details." The reader of the use case would know that the intent of the actor was to find a customer, and that the display and sort details would be provided on the linked page called "customer search details".

Using this neat trick, the use case stayed clean and uncluttered by details of searching, sorting, resorting, and the like, the common searching logic was localized and written only once, consistency was guaranteed, and the people who really needed to know the details for programming purposes could find their details.

6. UML & the relations Includes, Extends, Generalizes

UML defines graphics and three relations between use cases that people are determined to use. It is a moving target, and does not address writing style, which is what this book is mostly about, so I limit the discussion of UML (version 1.3) to just this section.

Write text-based use cases instead

Put your energy into writing easy-to-read use cases in simple prose, and all the relations between use cases will become straightforward and second nature, and you won't understand why other people are getting tied up in knots about them. Simply stated, if you spend very much time studying and worrying about the graphics and the relations, then you are spending your energy in the wrong place.

I am indebted to Bruce Anderson of IBM's European Object Technology Practice for the comment he made during a panel on use cases at OOPSLA '98. One member of the audience asked him why it was that everyone else on the panel was concerned about scenario explosion and how to use "extends", and he wasn't. Bruce's answer was, "I just do whatever Alistair said to do."

That seems like a terribly self-serving sentence to put into this book. However, people who write text-based use cases, as I am describing them, simply do not run into the problems that

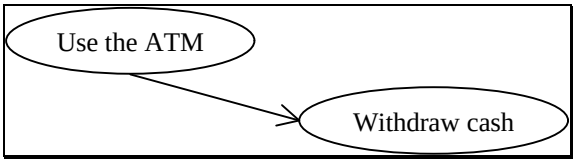
people do who fiddle with the stick figures, ellipses, and the UML relations. The fancy relations come naturally when simply write an unfolding story. They become an issue only if you focus on them. A growing number of use case teachers and coaches have discovered this, and an increasing number are downplaying the use of ellipses and arrows in requirements gathering.

"Includes"

A "base" use case "includes" another if it names it in an action step. This is the normal and obvious relationship between a higher-level use case and a lower-level use case, as used throughout this book. The included use case describes a lower-level goal than the base use case.

Every step in a use case is written as sentence about a goal that succeeds. The verb phrase part of that sentence is the name of a potential sub-use case. If you don't 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, or it "includes the behavior" of the "included" use case, in UML 1.3 terminology. Prior to UML 1.3, it was said to "use" the lower level use case. UML 1.3 refers to the higher level use case as the "base" use case, and the lower level use case as the "included" use case.

In UML notation, an arrow goes from the base use case to the included use case, signifying that the base use case "knows about" the included one. The following figure illustrates that "Use the ATM" includes "Withdraw cash".



"Includes" or "uses" as a subroutine call

It should be fairly obvious to most programmers that the "includes" relation is the old and familiar subroutine call from programming languages dating back to the 1950s. This is not a problem or a disgrace, rather, it is a natural use of a natural mechanism that we use both in our daily lives and in programming. On occasion, it is appropriate to parameterize use cases, pass them function arguments, and even have them return values. See, for example, the use case *Find a Whatever*, in the *Writing Samples*. Keep in mind that the purpose of a use case is to communicate effectively with another person (not a CASE tool or a compiler).

"Extends"

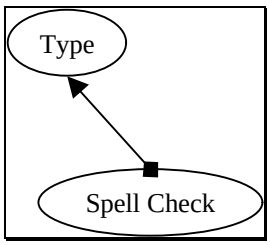
In UML terms, an *extending* use case *extends* a *base* use case if the base use case does not name the extending use case, but rather, the opposite. The extending use case names some internal point in the course of the base use case, names a condition, picks up the course of behavior, and when done, deposits the actor back at the point in the base use case that got interrupted. Rebecca Wirfs-Brock refers to the extending use case as a "patch" on the base use case (many programmers should be able to relate to this analogy!).

That is exactly what the scenario extensions have been doing *inside* the purely textual use cases we have discussed so far, and what the extension use cases were doing in the last section,

Extension Use Cases. So an extension use case can be thought of as just a scenario extension that outgrew the use case in which it was embedded. In UML the usage is the same.

In UML notation, an arrow goes from the *extending use case back to the base use case*, signifying that the extending use case "knows about" the base use case (the reverse of the "includes" relation.

The following figure illustrates that "Spell check" extends "Type". I have deliberately used a different sort of arrow to differentiate the "extends" arrow from the "includes" and the "generalizes". In the basic UML, there is no difference between the arrow, one merely writes the word "extends" or "includes" or "generalizes" somewhere along the arrow. More on this in *Drawing habits*.



A critique of UML's "extends" relation

The circumstance that caused "extends" to be invented in the first place was the business practice of *never touching* the requirements file of a previous system release. Since Ericsson was building telephony systems, and in telephony the business is to add asynchronous services (see *When to use extension use cases*), the extends relation was both practical and applicable. The new team could, *without touching a line of the original system requirements*, build on the old, safely locked requirements document, adding the requirements for a new, asynchronous service at whatever point in the base use case was appropriate.

I am sad to sat that the UML 1.3 standard violates the original purpose of extends and violates basic principles of document management. In UML 1.3, you are supposed to name, in the base use case, those places at which extensions are permitted! These are called "extension points".

- The first violation of document management is given by the extends relation in its original form - that you have to name the place in the base use case where the extension occurs. Once the base use case is edited, that location may move. The saving grace, in Ericsson's case, was that the requirements document was locked, and so the sentences were guaranteed not to move around. This is, by the way, a valid criticism of the numbered-step use case form. It is one more reason to discourage the use of extension use cases. I suspect that extension points were introduced to deal with this violation.
- The more serious violation is that now, you must go back and modify the base use case whenever you write an extension use case that needs a new extension point! The original purpose of this exercise was to avoid having to modify the base use case when new, asynchronous services get added, but now we have to type into both the old and the new

use case! With that much intrusion, you would be better off using the simpler "includes" relation in the first place.

- There is an additional violation. You are permitted (in some textbooks actually "encouraged") to expose the extension points in the use case diagram. This causes two problems on its own. First, you are actually publishing the internal parts of the use case into the global view offered by the use case diagram. Secondly, the extensions take up most of the space in the ellipse, obscuring the much more important goal name.
- Finally, and this is not specific to the "extends" relation, UML 1.3 does not show different

In the end, I have come to view "extension points" as a mistake in UML 1.3.

So, first of all, don't use extension use cases, unless you have a really good reason. If you do use them, absolutely don't expose the extension points in the use case diagram.

Follow the advice of the section, *Reminder: Use Includes.*

"Generalizes"

A lower level use case may "specialize" a higher-level use case. In principle, the lower level use case is a "similar species" of use case. Specifically, "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" (from UML 1.3 specification).

Correct use of "generalizes"

Good test phrases for any "generalized" concept is "generic", or "some kind of". If you find yourself describing a generic interaction, or a generic actor, or some kind of this general sort of interaction, or some kind of this general sort of actor, then you have a candidate for "generalizes". One actor may be a generalization or a specialization of another, and one use case may be a generalization or specialization of another.

We have already seen (section *Actors*) that the Clerk is the more general actor, and the Manager is the more specialized. The Manager can do everything the Clerk can (and more). The important test is that

- the specialized actor (the Manager) can do every use case the general actor (the Clerk) can do.

This is the pass/fail test on the generalization relation between actors

To understand "generalizes" for use cases, let us return to the ATM.

Goal: Use the ATM

Primary actor: Customer

Scope: ATM Level: Strategic

Main Success Scenario

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

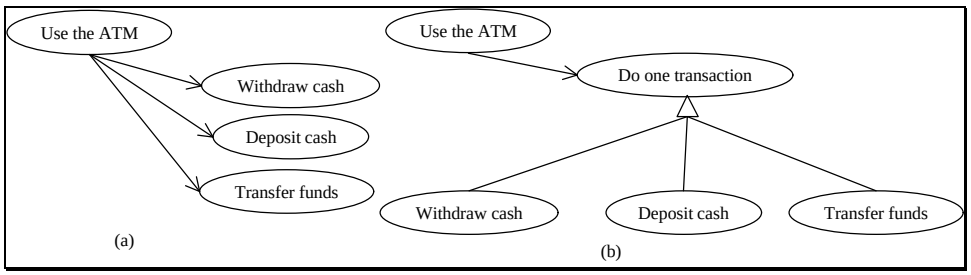
4. ATM returns card.

In step 3, what does the customer do? Generically speaking, "a transaction". The term "generic" tips us off that there is a generalized action happening.

In a plain prose use case, 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.

In UML, we have a choice. We can ignore the whole generalizes business, and just "include" the specific operations (see figure (a), below). Or, we can create a "general" use case, called "Do an ATM transaction", and show the specific operations as specializations of it (see figure (b), below).

In UML notation, an arrow goes from the specialized use case back to the general use case. Notice that once again, I have chosen a different sort of arrow for "generalizes" than "extends" (see *Drawing habits*).



Use whichever you prefer. Personally I shy away from generalizes relations, and the reason this time has to do with UML graphics conventions.

In UML, all arrows look alike. You are supposed to annotate the arrow with the type of relation, but that is a very small visual signal, and lost in the complexity of most drawings. If I had drawn the above figures with similar arrow types, you would not be able to tell whether

- "Use the ATM" extends, includes or generalizes "Do one transaction:. You can make a good guess, from reading the use case names.
- "Withdraw cash" and the others extend or generalize "Do one transaction". This time, reading the use case name doesn't help. You have to read the little words along the arrow.

In UML, you are permitted to change the visual appearance of stereotypes. That is what I have done in the drawings, so that the different relations have different appearances.

If your drawing tool has clearly different styles of arrows for the relations, then, it makes sense to draw the empty but general "Do one transaction" use case. The general use case provides a

visual placeholder for the reader to see the different kinds of specific transactions. It also differentiates between these operations and the other use cases that are "included" via arrows.

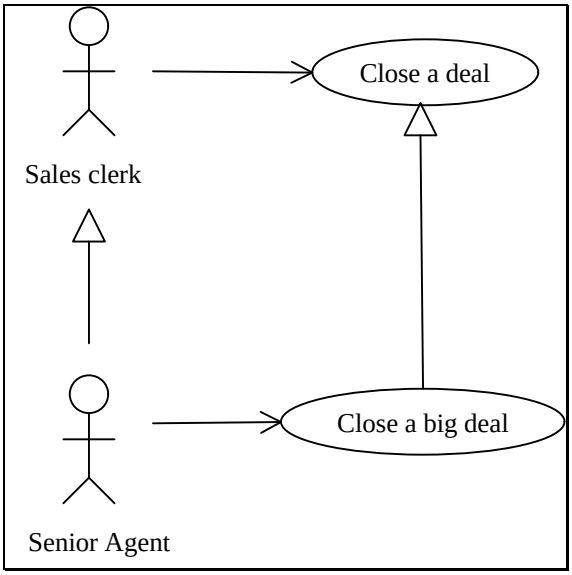
Notice that none of these issues arise in purely textual use cases. You simply write that the use case do one of the following operations, and list the operations.

Hazards of "generalizes"

I think that the authors of UML 1.3 simply have not thought through the consequences of their definitions. The generalizes relation abounds with hazards, the greatest of which is that you cannot safely combine specialization of actors with specialization of use cases.

There is one specific idiom to stay away from in use case drawings. It is when a specialized actor uses a specialized use case of a general use case of the general actor (see figure, adapted from Applying Use Cases, p.89).

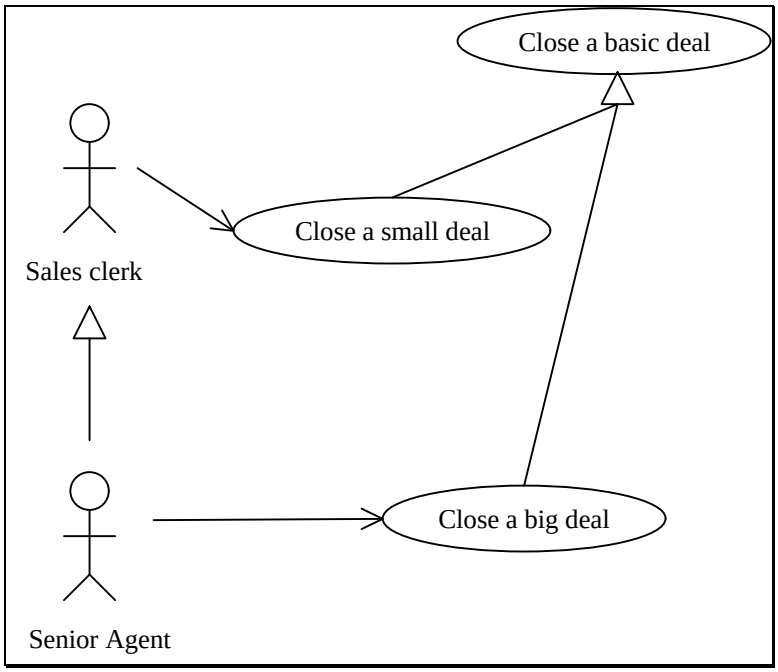
The fundamental rule is that a specialized use case can be substituted wherever a general use case is mentioned. The figure below announces that an ordinary Sales clerk can close a big deal, when what the writer would like to express is that only a Senior Agent can close a big deal, which is much like an ordinary deal.



The corrected drawing is shown below. The question now is, does closing a small deal really "specialize" closing a basic deal, or does it "extend" it? Since working with purely textual use cases will not put you in this sort of puzzling, trivial, and economically wasteful situation, I leave the that question as an exercise to the interested reader. Note that the above mistake was published in Applying Use Cases, a book written by use case experts. If even they can make such a fundamental mistake, your team is likely to, also.

Even Martin Fowler has been unable to sort out the difference between "extends" and "generalizes". In the 2nd edition of UML Distilled, he writes, "Use generalization when you are

describing a variation on normal behavior and you and you wish it to casually." I have no idea what a "casual" variation on normal behavior is. I suspect that Martin is advising you not to pay much attention to the defined and convoluted meanings of the relations, but just to draw something you like, hopefully something that you and your colleagues can agree on the meaning of.



There is one last critique I have of the generalizes relation. In the object-oriented programming world, there is as yet no consensus of what "subtyping" of behavior exactly is, what properties and options that term brings along with it. The industry having not settled what subtyping and substitutability of behavior is, means that there can be no standard interpretation of what subtyping or specialization of use cases means.

If you insist on using the generalizes relation, my suggestion is to make the generalized use case empty (as in "Do a transaction", above).

UML "Subordinate" use cases vs. Cockburn "sub-use cases"

In the subtext of the UML specification 1.3, the UML authors introduce a new 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 open up the SuD, and show how the use cases of the SuD *components* work together to deliver the use case of the SuD. But, in an odd turn, the components themselves are not explicitly shown. It is though you were to draw an "anonymous

collaboration diagram", a special sort of functional decomposition, that you are then supposed to explain with an proper collaboration diagram.

The specification says, "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."

The purpose of introducing these peculiar relations in the explanatory text of the use case specification is unclear. 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 level. Since I use the "includes" or "subroutine call" style of relationship between use cases, whenever possible, the higher level use case will "call" the "sub-" use case. I have been careful in this book to refer to the higher level use case as the "calling" use case, and the lower level use case as the "sub-" use case.

In my earlier, I used the words "subordinate" and "superordinate" for higher and lower level use cases. Since UML 1.3 has taken those words, I shifted vocabulary. My experience is that, unless told, people do not even notice anything to notice about the terms "calling use case" and "sub-use case". They are obvious to even the beginning writer and novice reader of use cases.

Drawing habits

In the previous sections about UML, I commented about the confusion caused by UML using the same style of arrow for all three relations. The text that is written alongside each arrow is simply not adequate to make clear the intended relations, and the result is bound to cause miscommunication and error.

These are my recommendations for drawing:

- *Always* draw higher level goals higher up on the page or screen than lower level goals. When you do this, the "includes" arrows will always point down, and the other two will always point up.
- *Always* draw generalization arrows pointing up into the bottom, preferably the center, of the general use case, never into the sides.
- Try to get different arrows for the different relations. Preferably use the same triangle for generalization as is used in the rest of UML. If you can do this, then your three arrows will be instantly recognizable:
- The '"includes" arrow is an open arrow pointing down.
- The "extends" arrow is an open or other arrow pointing up.
- The "generalizes" arrow has a triangle head, and points up.
- On the main or "context" drawing, do not show any use cases lower than user-goal level.

Exercises

Exercise: Construct legitimate examples of "extends" and "generalizes", for some use cases and actors of your choice.

Exercise: Fix Exhibit 7-3, page 89, in Applying Use Cases. Draw it in a different way, to avoid the hazards described in this section.

7. Mapping to UI, design tasks, design, and test

Use cases to UI

Better than whatever I can write in this book, Larry Constantine and Lucy Lockwood have written in Software for Use, and Luke Hohmann has written in GUIs without Glue.

The short of the story is this: you have on staff someone with the assignment, and hopefully the skill, to invent pleasant-to-use user interfaces. They 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.

It is therefore, not the place of the use case to describe the UI design. It is, however, very often beneficial to paste a snapshot of the designed GUI into the use case, at the bottom, so that readers can see what design is being nominated. Also beneficial, is to create what looks like a finite state machine, showing the screen flow, how the different screens link to each other.

Often, the use case writers will find it helpful to pretend they are typing into a screeen, 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. It is sometimes useful for the UI designer to see these forms and sketches. Pay attention that these forms are not interpreted as requirements, but rather as indications of how the usage experts view their task.

Use cases to detailed requirements (engineering tasks)

Subj: update
From: R.
To: arc@acm.org ('Cockburn, Alistair')

Alistair,

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.

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. Seeing how the development team was latching on to this document, and knowing it lacked the detail we wanted because we were hoping to focus around the use cases, we knew we needed to do something. 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. I had a copy

of Rational's RequisitePro that has the ability to highlight text in a Word document, and request to make a requirement out of it. Then you can create a view that shows essentially the same hierarchy we have in the scope document.

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.

Have you seen this predicament before, as well as how would you recommend dealing with it?

Best regards,

R.

I have encountered situations like this, too. In some cases, the "detailed requirements document" was written first, and then someone decided to write use cases from them (this seems a little backwards to me, but that's what happened, several times). In other cases, the organization (the designers, in particular) was set up to work from "numbered requirements paragraphs", much like the detailed requirements R. describes.

The tension is this, that when the programmer/designers are implementing, say, the Undo mechanism, the Transaction Logging mechanism, or the Search Screen Framework, they are not working with the operational flow of the use case. They are working on a single line item buried in one use cases - or maybe, the same line repeated across multiple use cases. These designers cannot describe their work in use case language, except to say, "I'm working on the Search Screen part of use case 5."

At the same time, the buyers, users and testers of the software don't really care about the individual line items. They want to see the application in full flow, delivering value.

Personally, on projects up to 50 people in size, I have never had to break apart the use cases into line items. Perhaps that is because I stress personal communication and cooperation on my projects, and have been fortunate with the cultures I have worked in. On these projects, we were 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.

If you can get away with doing this, it is obviously far less expensive than the double labor R. describes.

If you can't get away with doing this, then you are faced with the exact exercise R. describes.

The goal of the exercise is to break the requirements description into pieces that can be allocated to single developers, or single development teams. Each piece becomes a program "feature" or "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.

When I get my preferences, I prefer to write use cases, and generate the task list informally, rather than creating and maintaining two official documents, one usage-centered and one feature-centered.

Use cases to design

There are four things to say about transitioning from use cases to design, some good news and some bad news.

Design 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, as introduced by Ward Cunningham and Kent Beck [Beck89] and described by Rebecca Wirfs-Brock and company [WB91] (check the online material at [CunninghamCRC] and [CockburnCRC]), because that design technique is based on walking through scenarios, exactly as provided by use cases. Even with other design techniques, they serve well to remind the designers of when the design is complete and handles all exceptions.

Out-of-order appearance

However, the design task does not map itself tidily to use case units. Usually, a class, component or structure is used by several use cases. It is likely that some "interesting" information about the structure is contained in a use case scheduled for a future release, i.e., the designers will not see that information until later.

There are two approaches to this. The first is to have the designers scan all the use cases to collect all the information that might apply to their design task. This can clearly only be done on smaller projects. If you can manage to do this, fine. For most projects, this is not a realistic option.

The alternative is to recognize that the software will change over its lifetime, and to relax about that. Design each release as well as practical, recognizing that sometime in the next year or two, new information will surface that will cause a change. This is not new news, or particularly bad news, it is simply the shape of the world. Even if your designers read every use case, it is still quite likely that a new requirement will surface and cause a change.

Use case per release

It would be soothing to say that a use case maps to a release, or that a tidy set of use cases map to a release. Some projects manage to do this, by constructing the use cases exactly so that a full use case is designed in one increment of the project and release in completess. However, organizing your use cases this way means writing separate use cases for each extension of the funtionality of the system. This can be done, using the extension use case mechanism. However, most organizations find that their use case set becomes difficult to read.

Most organizations write the use cases to be as readable as possible, and live with the fact that they will release portions of a use case at a time. To track this, they highlight in yellow, or italics, the parts that will be released, and refer to "the highlighted portions of use case 5".

Note that this discussion presumes that your organization is using incremental staging and delivery of the system, with increments of about four months or shorter. This incremental development has become a standard recommendation on modern software projects (for further information on incremental development, see Surviving Object Oriented Projects [Cockburn98] or check online at [CockburnVW].

Design of other aspects

Design means more than design of the operational flow. Designers draw upon all of the chapters of the requirements document. Many are particularly interested in the detailed descriptions of interface formats, which, as we have seen, are not contained in the use cases.

This is not a drawback to use cases, or a problem in any respect. Simply recognize that the use cases are there to show the system supplying value to its users, and handling the behavioral complexity of the interface. That is their purpose. It is the purpose of the detailed data descriptions, finite state machines, business rules, and other sections of the requirements document, to show the other requirements.

Use cases to test cases

The use cases provide a ready-made functional test suite for the system. Most test groups are delighted to see the use cases. It is often the first time they have ever been given something so easy to work with by the design team. Even better, they are given this test suite right at requirements time!

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 kindly provided by Pete McBreen (http://www.cadvision.com/roshi/papers.html). Notice his use of stakeholders and interests to help identify the test cases, and how his test cases contain specific values.

Use case: Customer: Buy goods
Customer places order for goods, an invoice is generated and sent out with the ordered items.
Success End Condition
Goods have been allocated to the Customer
Invoice has been created (Customer Invoicing Rule applies)
Picking list has been sent to distribution
Failure End Condition
Goods are not allocated to the Customer
Customer account information unchanged
Transaction attempt has been logged

Main Success Scenario
1.Customer: Select items and quantities
2.System: Allocate required quantities to customer
3.System: Obtain authenticated invoicing authorization
4.Customer: Specify shipping destination
5.System: Send picking instructions to distribution

Extensions
2a Insufficient stock to meet required quantity for item
2a1 Customer: confirm reduced quantity
2b out of stock on item
2b1 System: determine shipping date
2b2 Customer: Accept delayed delivery
3a Customer is bad credit risk (link is 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 will need more test cases. The Main Success Scenario test case should come first since it is nice to show how the system works in the high volume case. Often this test case can be generated before all of the extension conditions and recovery paths are known.)

Main Success Scenario Tests
- Initial system state/inputs
Customer Fred (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

Bad Credit risk
- Initial system state/inputs
Customer Joe (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

8. CRUD use cases

There is a debate raging at the moment as to how to organize all those little use cases that say, "Create a new X", "Update an X", "Delete an X". These simple operations on entities in the system are known as the CRUD operations (for the old Create, Replace, Update, Delete operations on the old database systems). The question is, are the Create / Update / Delete use cases all part of one bigger use case, "Manage X", or are they three 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. Susan Lilly advocates this view (see *Other Voices*).

However, the CRUD use cases can clutter up the use case list, tripling the number of individual use cases to track. Therefore, some people prefer to write just one use case, "Manage X", with three paths through it, the main path for updating X, with an extension for creating and an extension for deleting X. An example of this form of writing is shown in *Writing Samples*.

Neither way is "wrong", and I do not have enough evidence to form a rule one way or the other. My personal tendency is to write "Manage X", to get the advantage of less clutter, unless it turns out not to work well for some reason, usually related to the complexity of the writing, and only then break it out into "Create X", "Update X", "Delete X". I like to think of splitting up and merging use cases as a minor operation, and that reducing the clutter in the requirements folder is important.

9. 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 of AGCS 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!

In this book, I have pulled out the dominant issue: "How does one write consistenly readable use cases?" I find that the answer is constant, given the actor-goal / full, active sentence style. Nonetheless, you may find yourself in a situation with some combination of the issues listed below, and that may cause you to be patient, to be tolerant, to strategize, or to work differently.

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..." (it was good enough for Dad, so it's good enough for me).
- Multiple cutures:
  - 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

- Management Cycle Phase – results in Clandestine Analysis - Paul what does this mean?
- Reality Cycle Phase – leads to Creeping Analysis - Paul ditto?
- Watch out, people tend to stress the things they know
- Scheduling related to depth of ?? (Paul?) knowledge related to 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.
- Complete Model versus Incomplete Model (See cooperation between teams)
  - When in the lifecycle is this occurring (see Alistair's completeness issue??)
  - 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.

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 (Paul ??)
- 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 dipersed

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
  - Achieve Completeness
  - People want to fill out 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 to resolve
- System Complexity
  - Analysis paralasis – complexity of system overwhelmes analyst.
  - Number of actor profiles
  - Number of function points
  - Kind of system
    - Simple user system
    - Real Time System
    - Embedded System (Must be error resistent)

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 Requirments/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?
- 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 preicision to get before getting some depth.

10. Tools.

None of the tools on the market are perfect, in fact none is better than barely usable.

Lotus Notes. Lotus Notes has been the best I have seen as late as mid-1999. The notes are easy to read and write, and most importantly you can create specialized views of the list of databases, so you can scan across use cases, or open one up for reading. You can hyperlink related use cases, and append notes to a use case. When you update the template, all use cases in the database get updated. All of that is good.

However, renumbering the steps and extensions as you edit the use case soon becomes a nuisance, and the links eventually become out of date. Manually inserted backlinks become out of date very quickly. 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.

All in all, however, the feature that makes Lotus Notes most attractive is the ability to create a view of the list of use cases. The first useful view highlights use cases by priority, release, state of completion, and title. The other useful view highlights use cases by primary actor or subject area, level, and title. The other critical feature is the ability to update the template and have all use cases automatically incorporate the new template.

Word processors with hyperlinks. Modern word processors allow hyperlinks, which makes the far superior to the word processors without hyperlinks. Word processing software is standard, and the users are trained on them. You can often create a template for people to use. Being a word processor, the tool encourages people to write as though they were writing a story, which is good.

With the advent of hyperlinking, it is practical to put each use case into its own file. It is then practical and effective to hyperlink from a higher-level use case to the sub-use case. Of course, there is no automated reverse traversal of the link.

There is, however, no way to scan across all the use cases as there is with Lotus Notes. Changes to the template do not propagate to old use cases, so you are likely to end up with multiple use cases forms in the requirements file.

Local relational databases. A number of people have created, or started to create, a template for Microsoft Access or equivalent database product. Although it would seem natural that the database can support the use case metamodel, in practice, the usability of the result is so poor, that I have never encountered a project that has managed to have all of the use cases in the database.

CASE tools. CASE tools are built to manipulate graphics, not text. The few that I have seen hide the textual description of each step behind a dialog box, making it invisible on the main view of the use case. Some use interaction diagrams (sequence charts) as the diagrammatic capture of the use case.

The attraction of the CASE tool is that the cross-referencing facility is complete, providing for automated back links. However, the usability is so bad that I have seen team members mutiny, and revert to word processing, rather than use the CASE tool.

Requirements capture tools. Specialized requirements capture tools, such as DOORS, are becoming more commonplace. I have not seen anyone try to capture use cases in these tools on a real project. They promise to provide automated forward and backward hyperlinks, and are text based, rather than diagram based, so it is possible one of them will work well for you. However, none of them that I know of incorporates a metamodel of use cases close to the one I describe here, so I would expect there to be some mismatches, in practice. Still, given that use cases are linked text, you should be able to make them work

11. Scale: rolling up use cases

Scale is a matter of hiding things in order to cover more material. To show a map of the city, we might omit everything less than 100 feet long. The six-foot bookshelf in my library does not show up at all. When you choose a scale at which to operate, it dictates the amount of precision that you can use. Scale is a matter of getting more of the system into view at one time. There are four ways to getting more of the system into view.

Use lower precision

Just as a 1:30 or 1:100 scale drawing of your office will not show the buttons on your keyboard or the bookshelf supports, so the 1:30 or 1:100 scale drawing of your software design will not show all the attributes of each class. When you need to view all the use cases, try just viewing their names.

Select items for relevance

A cartographer omits bus lines and houses on a county map, because they are not relevant to the task at hand. With use cases, select the use cases that serve just the primary actor(s) that you care about for the discussion, or just those use cases being developed by the development team you are discussing, or just those use cases for the release

Bundle multiple items together.

The city map shows just a rectangle for a block; the state map shows just one shape for each city. If you want to see more, you have to move to a different scale. With use cases, create *use case clusters* or *subject areas*.

A use case cluster is a set of use cases that are managed together. It may be that a "white", or strategic, use case collects together all those use cases, and so the white use case is an excellent clustering mechanism (see Layering, below). On the other hand, you may find that there are a disparate set of use cases that you nonetheless like to think of at the same time. On one project, the system maintained a mock checkbook for corporate customers of the company. 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.

Larger projects deal with different *subject areas* entirely. One company had Invoicing, Promotions, Staffing, and other subject areas. Another had separate subject areas for the different stages in the life of its routing life cycle: Booking, Tracking, Delivery; there were also Routing, Billing, and other subject areas. Typically, different development teams will work on the different subject areas.

Layering

Finally, there are layers. Layers correspond to goal levels, and are discussed extensively in this book. Higher-level use cases roll up lower-level use cases, showing the life cycles involved and variations on the life cycle. Thus, to save space in a table or chart, you might work with, "Handle a claim (white)", thereby collecting together all the use cases related to handling a claim.

12. Estimating, Planning, Tracking.

The use case names give you a structure for planning and tracking the project. I have good news and bad news for you here. I'll give the bad news first.

Ideally, you would like to deliver an entire use case on a software release. However, if you are using an incremental release strategy, you will find that release 2 might only deliver the most frequently used parts of the use case, that you are saving the infrequently used parts for a later release (some people use yellow highlighting on the text to indicate which parts are being delivered first).

This complicates the way you talk about your releases. You cannot say, "Use case 15 will be delivered in release 2." You have to say, "Release 2 delivers the yellow highlighted parts of use case 15." To be honest, that is the way I have seen most projects work, and it works passably well. But it is not as tidy as we might like.

The alternative is not better. You might split the use case into two or three parts, one for each release. Now you can say, "Use case 15 will be delivered in release 2." However, you have doubled or tripled the number of use cases in your database, and increased the complexity of linkage between them. Instead of 50 or 80 use cases, you have 120 or 200 use cases to manage. Most project teams prefer the first choice to the second. On small projects, the second strategy might work better. I have encountered both strategies in use. Typically, it will be clear to you which is better for your situation.

There is a third strategy. During requirements gathering, write the use cases as though each was going to be delivered at one time, even if you know they will be split. At the beginning of each incremental development period, have the team write a separate use case for the functionality they are planning to deliver in this period. Then the team can show exactly what they will deliver

out of the full use case. On the next increment, the team might write their use case with two highlightings, 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 third strategy, the number of use cases still multiplies, but in a limited and controlled way. The team has the sensation of managing (e.g.) 80 base use cases, plus however many they are delivering in this incremental period.

All of that was the bad news. Once you decide how to manage that issue, you can proceed with the good news.

The use cases give the management team a handle on "usable function" being delivered to the users. With each release, it is clear what value is being delivered.

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. In a nice twist on this theme, Kent Beck, in his "Planning Game" has the developers estimate the development cost first, and the business sponsors can decide on the priority of each use case in the context of that development cost. You may choose to fill these two columns in two passes.

Finally, in the next 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.

The net benefits of use cases are two:

- The list of use case names gives the project planners a structure for manipulating the development priority and timeframe for the use cases.
- The use case list clearly shows the value to the business of each release.

A small, true story.

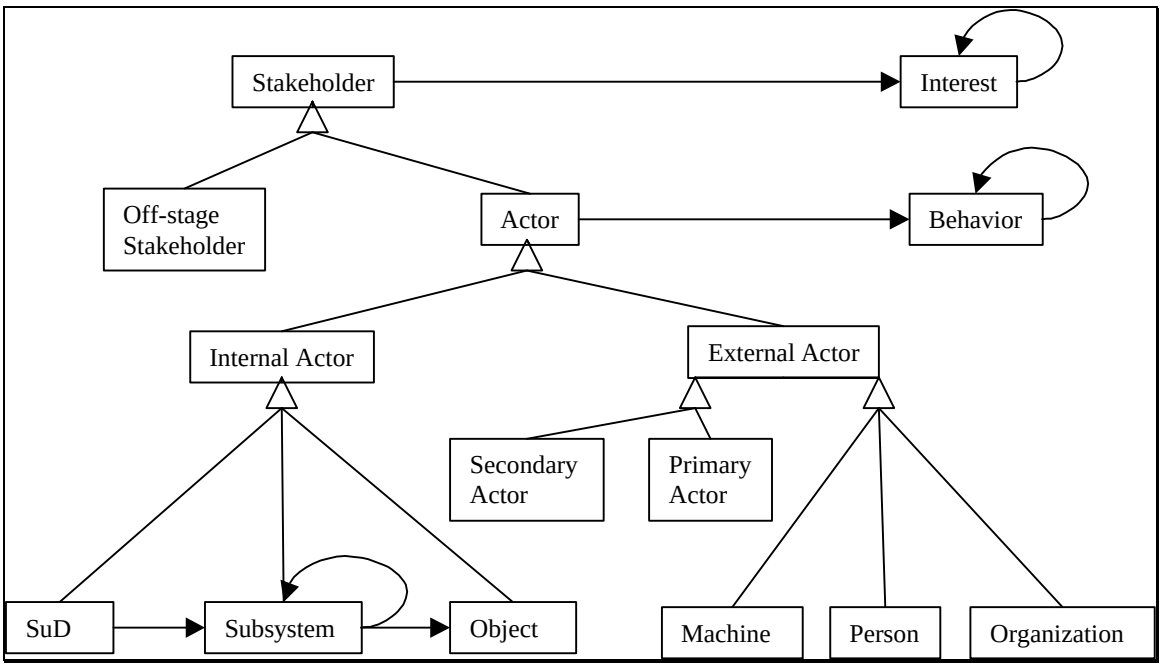
A person was given the assignment to decide what business processes to support in the next releases of the software. She found she had a *7- page* actor-goal list! She created the table with estimated complexity and value, trigger, and so on. She sat down with the executive sponsor for the project, and they trimmed the potential list to half a page. She then wrote the main success scenario for those business processes, sat down with the executive sponsor again, and trimmed the list of steps to consider to about half a page of system-level goals. She then went on a tour of the branch offices, and a few weeks later came back with a clear picture of which of those business steps would most affect and benefit the branch office workers. From this, short list, they identified the four use cases to develop over the next six months.

13. A Metamodel for Use Cases

For those interested models and metamodels, the following is a UML drawing corresponding to the Stakeholders & Interests model of use cases. As basic "truth-in-advertising", I suspect that this model, as all untested models, contains mistakes. I don't know how to debug it without several years of playing with it. Therefore, please consider it as a first approximation, and write to me with observations on it.

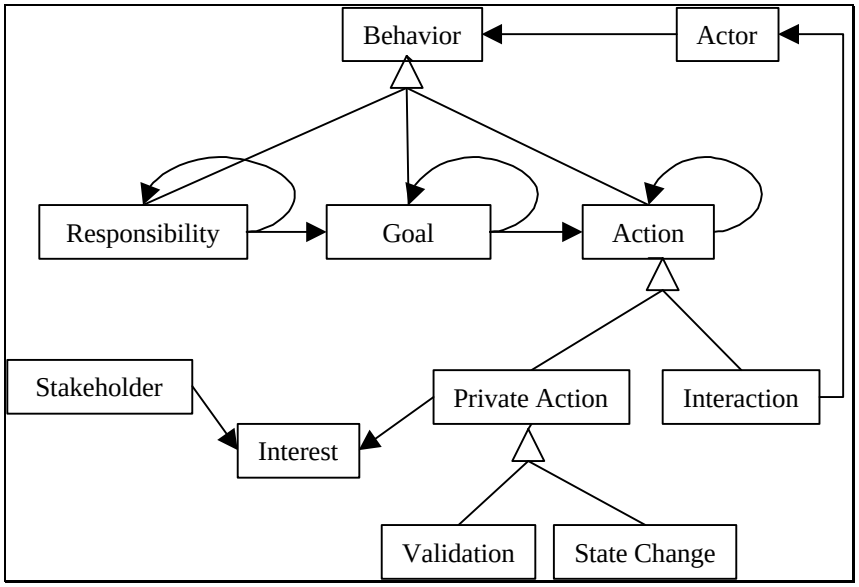
An actor is a stakeholder

This model fragment expresses that an actor is a stakeholder. A stakeholder has interests, and an actor has behavior.



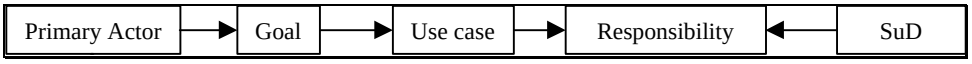
Behavior

This model fragment expresses that actions are part of goal-oriented behavior, that interactions connect the actions of one actor with another, and that the private actions we care about are those that capture the interests of stakeholders.



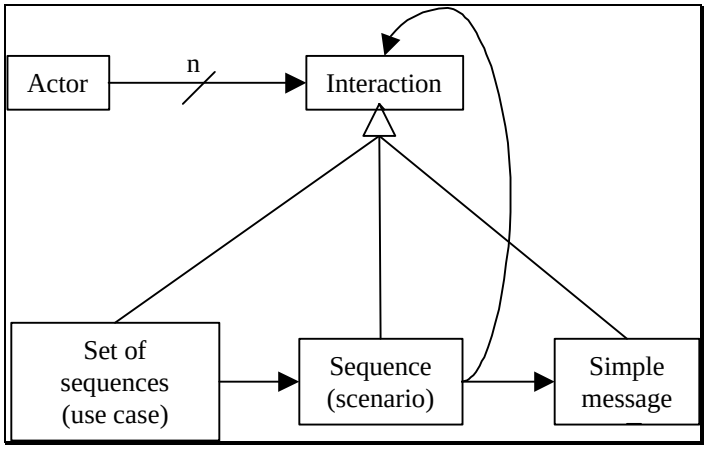
Use case as contract

This model fragment expresses that the use case is the primary actor's goal, calling upon the responsibility of the system under design (SuD).



Interactions

This model fragment expresses that (n) actors participate in an interaction, and shows the recursive decomposition of interaction by use case, scenario, and simple message. Once again, the word "sequence" is used as a convenience.


Actor
n
Interaction
Set of sequences (use case)
Sequence (scenario)
Simple message

Reminders

1. Recipe for writing use cases

At this point, we can summarize the basic process of writing use cases. Here is the "simple", 12-step description of what has been said so far.

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 strategic level use cases covering all the above.
Step 5. Reconsider & revise the strategic use case. Add, subtract, merge goals.
Step 6. Pick a goal / use case to expand. Optionally, write a system-in-use story to become acquainted with the story of the use case.
Step 7. Fill in the stakeholders and interests, the preconditions, the success end conditions, the failure protection. Double the goals and interests against those conditions.
Step 8. Write the main success scenario for the use case. Double check it against the stakeholders' interests.
Step 9. Brainstorm and list the possible failure conditions.
Step 10. Write how the actors and system should recover from each failure.
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.

2. Quality Matters

Recall from the prologue:

> "The problem is that writing use cases is fundamentally an exercise in writing natural language essays, with all the difficulties in articulating "good" that comes with natural language prose writing in general."

Russell Walters, of Firepond corporation wrote in,

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 along-side yourself. 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 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.

Kind regards,
Rusty Walters

Quality within one use case

Pass/Fail Tests

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 a the few I have found. All of them should produce a "yes" answer.

1. *Use case title*. Is the name an active-verb goal phrase, the goal of the primary actor?
2. *Use case title.* Can the system deliver that goal?
3. *Scope and Level*: Are the scope and level fields filled in?
4. *Scope*. 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. *Scope*. If the Scope is the actual system being designed, do the designers have to design everything in whatever is mentioned in the Scope, and nothing outside it?
6. *Level.* Does the writing of the steps match the goal level stated in Level?
7. *Level.* Is the goal really at the level mentioned?
8. *Actor.* Does it have behavior?
9. *Primary actor*. Does it have a goal against the SuD that is a service promise of the system?
10. *Preconditions*. Writing varies by hardness and tolerance, per project, but the questions is: Are they mandatory, and can they be ensured by the SuD?
11. *Preconditions*. Is it true that they are never checked in the use case?
12. *Stakeholders and interests.* (Usage varies by hardness and tolerance) Are they mentioned?
13. *Success end condition.* Are all stakeholders interests satisfied?
14. *Failure protection.* (Usage varies by hardness and tolerance) If present, are all the stakeholders' interests protected?
15. *Main success scenario*. Does it run from trigger to delivery of the success end condition?
16. *Main success scenario*. Is the sequence of steps right (does it permit the right variation in sequence)?
17. *Main success scenario*. Does it have 2 - 11 steps? (You may run across a use case that really has more, but I'll bet that if you merge steps and raise goal levels, your shorter version will be easier to read).
18. *Each step in any scenario*. Is it phrased as an goal that succeeds?
19. *Each step in any scenario*. Does the process move distinctly forward after successful completion of the step?

20. *Each step in any scenario*. Is it clear which actor is operating the goal (who is "kicking the ball?)
21. *Each step in any scenario*. Is the intent of the actor clear?
22. *Each step in any scenario*. 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. *Each step in any scenario*. Are you sure the step does not describe the user interface design of the system? (It might, if that is the purpose of the use case).
24. *Each step in any scenario*. Is it clear what information is being passed?
25. *Each step in any scenario*. Does the step "validate" a condition, as opposed to "checking" a condition?
26. *Extension condition.* Can and must the system detect it?
27. *Extension condition.* Must the system handle it?
28. *Technology or Deferred Variation.* Are you sure this is not an ordinary behavioral extension to the main success scenario?

Making the use case easier to read.

You want your requirements document

- short,
- clear,
- easy to read.

I 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." However, these 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 files, which few people enjoy reading.
2. Start from the top and create a story line. The top will be a strategic use case. The task-level, and eventually, subfunction-level use cases branch off from here.
3. Give the use cases names that are short verb phrases, naming 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 sub-goals 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.
9. Use "includes" relationships between use cases whenever possible.

Quality across the use case set

1. Does the list of user goals capture all the services all the primary actors want from the system?
2. Do the use cases form a story that unfolds from highest level goal to lowest?
3. Is there a context-setting, highest level use case at the outermost design scope possible for each primary actor?

Know the cost of mistakes

Failures in quality in the use cases have a cost that depends on your system type and project character. The better the internal communications are between your usage experts and developers, the lower the cost of quality failures: the people will simply talk to each other and straighten matters out. That is simply good news, since it means your project can run at a lower cost.

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 goes up.

Recognize where your project sits along this spectrum, and pay attention not to get too worked up over relatively small mistakes on a small, casual projects, but do get rigorous if the consequences of misunderstanding are great. In other words, know the cost of mistakes.

3. A bad use case is still a good use case.

The best part about use cases is that just writing them at all probably will improve the level of communication between your users and executives and the developers.

You can make two mistakes with a use case - writing too little, or writing too much. Please write too little.

As a first pass, or a fallback plan, simply have the requirements writer write a little story about the system being used. That story will communicate the intent of the user, the service the system provides, the core functionality of the system and the some of the information that gets passed back and forth.

This serves as a basis for discussion between the parties involved in developing the system. On small - even on medium-sized projects - this can be sufficient. The users or requirements writers sit close enough to the developers that the developers can simply walk down the hall, or phone them, and ask what is meant by a phrase, or please check the field checks being performed, or whatever.

- I have seen a 50-person, 15M$ project succeed with just the main success scenario and a few failures marked, in a simple paragraph text form. This project had excellent in-team communications, which permitted them to work that way.
- The Chrysler Comprehensive Compensation project team, building software to pay all of Chrysler's payroll using the "Extreme Programming" methodology, did not go further than just a sketch of functionality for each use case. They put so little into each use case that they called them "stories" rather than use cases, and wrote them on single cards. Each story card was really a promise for a conversation between a requirements expert and a developer. Significantly, the entire team of 14 people sat in only two, large, adjacent rooms, and had excellent in-team communications.

When in doubt, write less text, at a higher level of goal and 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 then will 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 have closed communications on the team, instead of opening them. It is a common fault that programmers write at too low of a goal level. Have someone else draft the first and high-level round of use cases, and let the programmers fill in the details, if your organization works by having the programmers write the requirements.

The point is that diminishing returns set in very quickly with use cases. Naming the goals and drafting the storyline is tremendously valuable. After that, the energy required to polish the use cases goes up very quickly, and the value flattens out very soon. I have belabored every point in this book so that you can build good writing habits into your reflexes, and tell where you or your team can still improve.

If I were on a project with a severely limited time- and energy-budget (and good internal communications), here is how I would organize the effort:

- Collect the users and goals, as before, and as always.
- Draft the main storyline in paragraph form (see *Casual template*).
- Brainstorm failures, as before, and as always.
- Draft the failure repair, and jot down the repair steps.
- Skip fancy efforts at numbering and use case cross-linkage tools.

If I were on a project with the usual, limited time- and energy-budget, and poor internal communications, I would work on the communications. See, for example, the risk reduction pattern, *Holistic Diversity*, and related tips in Surviving Object-Oriented Projects.

The use cases in *Writing Samples, A Bank Deposit System*, show one person's way of saving energy, and still capturing the information needed to develop the system. The writer was one of the two programmers on the project. In terms of the methodology grid given in Chapter 2, this was a "2-person, loss of discretionary moneys" project, and so could operate with very low ceremony, high tolerance and low hardness. They correctly judged the weight of methodology they needed for their project, and worked accordingly.

4. Just chapter two.

Use cases are only a small part of the total requirements gathering effort, perhaps "chapter 2" of the requirements document or file. 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.

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. Refer to the sections, *The Missing Information* and *Project Planning and Tracking*.

5. An ever-unfolding story.

To guide your writing, think of the set of use cases as an ever-unfolding story or play.

There is one, top level use case 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.

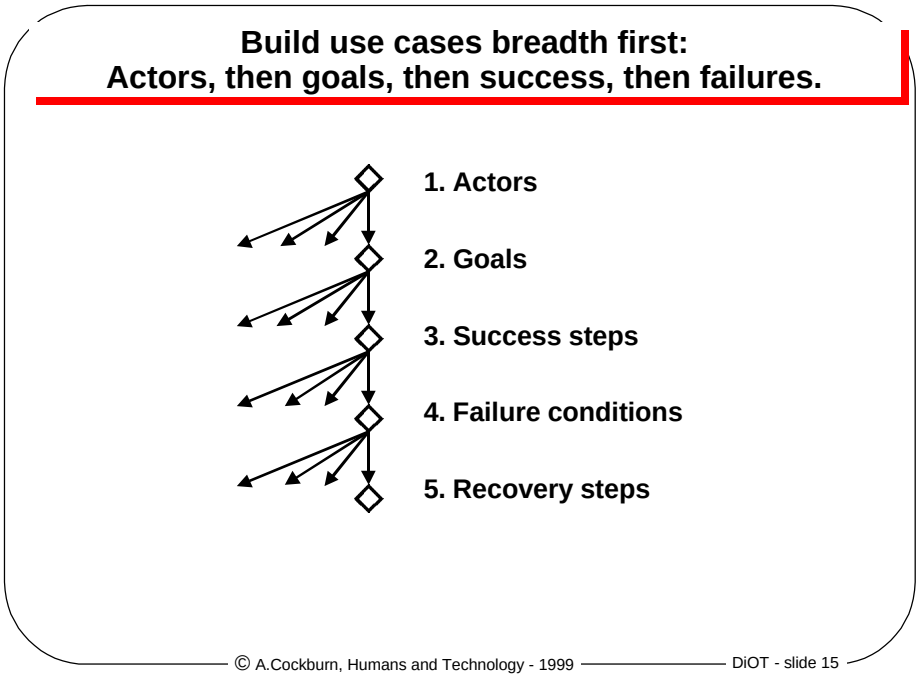
It names the highest-level (white / strategic) goals of the outermost actors, working against the outermost possible system scope. Typically, there will be an external customer, the marketing department, and the IT or security department. These use cases show the life-cycle relationship of the individual user goals, and most often contain "blue" or user-goal level steps (e.g. get a quote, buy a policy, close a policy). There are often only 3 - 5 of these use cases. For most readers, the "story" starts with one of these use cases.

The steps in the outermost use cases name and unfold into use cases having users acting on specific goals against the system under design, i.e., blue use cases. In those blue use cases, the design scope is the system being designed. The steps in the blue use cases show how the actors and system interact to deliver the goal, each step moving the process distinctly forward.

A step in a blue use case only unfolds into its own (indigo, or subfunction) use case if it is quite complicated, or has several steps to it *and* is used in several places. Subfunction-level 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 (but see the *Writing Sample* for "Find a whatever", and the section *Parameterized use cases*). See also the section *Two levels of blue* about the cases in which a blue (user-goal) use case calls another blue (user-goal) use case.

6. Breadth first, low precision

To manage your energy, work breadth first, not depth first, working from lower precision to higher precision. At each stage, you will produce the appropriate information for rationalization and review, making sure of your priorities before going on with the next level of work. Each level of work involves considerably more effort than the previous, so trimming work on one level saves a great deal of energy on the lower levels (see figure).



1. Primary actors. Collecting all the primary actors is 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. Brainstorming these actors will 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 (for a long and tricky example, see *Writing Samples*). 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. Putting all the failure or extension conditions into one place before elaborating on them brings two advantages. First, the activity is a brainstorming activity, which, as an activity, is different than deciding and sequences steps for fixing the failure. Second, the list serves as a work list for the writers. The writers can take a break from writing and know, when they come back, where they are and what needs to be done. Writers who try to fix each condition as they name them, typically run out of energy after writing a few failure scenarios, and never complete the failure list.
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. Field lists. While formally out of the use case writing effort, often the same people are charged with expanding the data nicknames into the lists of fields associated with each nickname (see *Data descriptions*, in *The Missing Information*).
7. Field details & checks. See *Data descriptions*, in *The Missing Information*, for more on field details and checks. In some cases, different people can be writing 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.

7. Just one sentence style.

There is only one style of sentence used in writing action steps in a 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"."

This sentence style is used in business use cases, system use cases, strategic-goal use cases, user-goal use cases, subfunction use cases, using the fully dressed template or the casual template. Master this one sentence style and you will write use cases that are short, clear and easy to read.

A second grammatical form is used, to indicate extension conditions. It is useful to use a different grammatical form for these, so they don't get confused with action steps. Use simple descriptive phrases or sentences preferably (but not always) in the past tense:

"Timeout"
"Bad password"
"File not found"
"User exited in the middle."
"Data submitted is incomplete." (note: not the past tense! Can you do better?)

Inside the extension, the action steps again look exactly as everywhere else: "Actor successfully achieves goal".

8. Clear intentions, goal in sight.

Reread the sentence you just wrote, and make sure the goal really is in sight, and it really is the goal of the actor.

Did you write, "System checks credit limit"? Is that really the goal of the system? It probably is not. If the goal of the system were really to "check" the credit limit, then the program would end, saying, "Yep, we checked that credit limit. It has been *checked*!"

In most cases, checking the credit limit is only the preAcura to what we care about - taking the order. To take the order, we have to make sure the customer has an adequate credit limit. "Make sure" was the operative phrase there. The goal is to "make sure the customer has an adequate credit limit." That is the phrase that goes into the scenario (whether main or extension).

But what happens if the customer doesn't have sufficient credit? That is indeed important, and gets noted in an extension: "Insufficient credit:", followed by more goal-achieving sentences.

Note how, in this use case fragment taken from the *Writing Samples*,, the intent of the actor is always clear:

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.
5. Adjuster checks reserves ...
6. Adjuster seeks authorization for settlement ...
7. Adjuster documents the file.
8. Adjuster sends any correspondence ...
9. Adjuster continues to document file regarding all settlement activity.

This is the writing for a business use case. For an example involving a system use case, read *Answers to Exercises: Buy stocks over the web* "

9. Who's got the ball?

Sometimes people write in the "passive" voice: "Credit limit gets entered." This sentence is missing the real subject - "who" it is who enters the credit limit.

Remember to write from the point of view of a bird up above, watching and recording the scene. Or, in the form of a play, announcing which actor is about to act.

I occasionally like to think of a use case as describing a soccer game: actor1 has the ball, dribbles it, kicks it to actor2. Actor2 passes it to actor3. And so on.

Check each sentence, to make sure that it is clear who owns the action in the step. Almost always, the first or second word in the sentence is the name of the actor who owns the action in that step. Sometimes the actor passes control to another actor, sometimes not. Whatever happens, make sure it is always clear, who's got the ball.

10. Satisfy the stakeholders

It is incorrect to think that the use case only shows the interactions with the primary actor. The use cases are your *functional requirements* for the system being described or designed. That means the use case shows *all* the actors in action, and shows how *all* the stakeholders' interests are satisfied by this system.

It takes surprisingly little effort to check that you have addressed the stakeholders' interests. Usually, there are 3-5 stakeholders: the primary actor, the owners of the company, perhaps a regulatory agency, and maybe someone else. Perhaps the testing staff, or maintenance staff has an interest in the operation of the use case. Usually, these interests repeat for all use cases, and usually their interests are very much the same across use cases.

- The primary actor's goal is their interest, and 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, i.e., usually that some sort of a log is available.
- 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.

Typically, these are the sorts of interests. The way to check that they are satisfied is just to read the text of the use case, starting with the main success scenario, and see whether those interests are present.

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, i.e., the interests of the regulatory agency or whoever wants to be able to replay the interrupted transaction are not represented.

Write the Success End Condition. This should be fairly obvious text, but writing it specifically with regard to all the stakeholders sometimes turns up a missing element.

Now write the Failure protection. I have found that this often turns up a missing element in the main success scenario. If the failed end protection is that there is a log in case of catastrophic failure, then the main success scenario probably needs a step about writing out an intermediate state.

Check the failure scenarios. Often, a new validation shows up in a failure scenario - by the fact that the validation failed. Often, the writer did not write that validation into the main success scenario, where it belongs (so that its failure can be noted).

Checking the stakeholders' interests is surprisingly revealing. The first time someone tested out these tests, they found that all of the change requests they received for the first year of operation of their software, turned up in the stakeholders' interests list. In other words, they have successfully build and delivered the system *without* satisfying the needs of various stakeholders. The stakeholders figured this out, of course, and so the change requests came in.

The good news is that performing this check takes very little time, and is quite revealing for the time spent.

11. User goals and level confusion.

The single hardest thing about use cases is getting the goal levels right, for each use case and each step.

Your two anchor points for the discussion are:

> User's goal ("blue") level.
>
> 3-10 steps per use case.

Find the blue goal

The use cases for a business can be linked in a complex graph all the way from "Make us all rich", down to "Hit the Tab key." In all of the levels of goal, only one level stands out:

> 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. Business process modeling people call it an "elementary business process". I call it the "user's goal". I have given it the designated color, blue, because you may have associations with some of the other words, and you probably have no associations with a "blue" use case. So I define just what is a "blue" level of goal, for a computer system being described:

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

This is a blue, or user-goal, or elementary business process level. Everything relates to this level, either above or below. I even sometimes refer to this level as "sea level" (see figure), because there is only one of it.

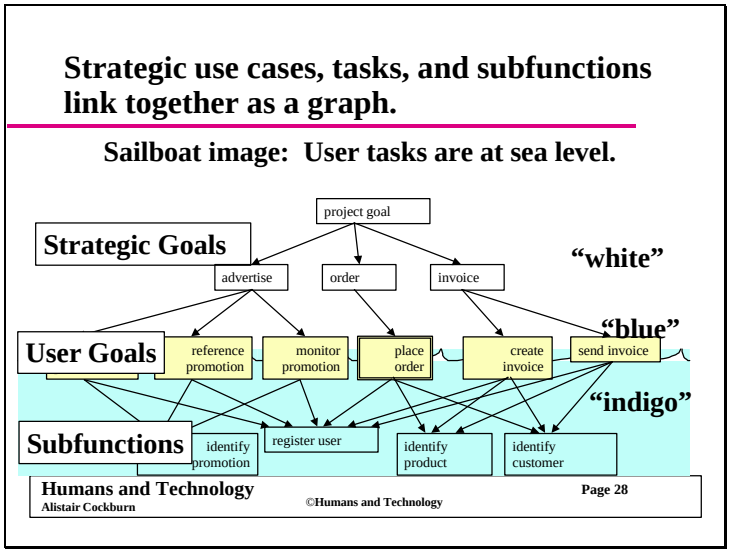


Figure. Sailboat image of use case graph.

Therefore, the very first thing is to ask, if the use case is labeled as "user goal", is: "Is this what the customer / user / primary actor really wants from the system, now?"

In most cases, when people drafting use cases, the answer is "no" - the use case goal is below the level of what the primary actor actually is interested in. Most people draft use cases at the indigo, or subfunction level. The correction is to ask one of the two questions,

"What does the primary actor really want?"

or

"Why is this actor doing this step / goal?"

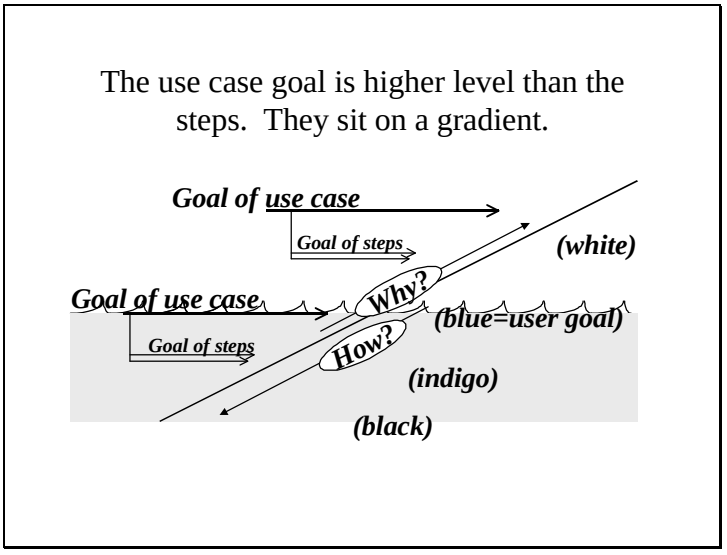


Figure. Ask "why" to shift levels.

The answer to the "why" question is probably the actors real goal. Check it again, and repeat as needed.

Merge steps, keep asking "why"

The second anchor point is the length of the use case. Most well-written use cases I have seen had between 3 and 8 steps. I have never seen one longer than 11 steps that couldn't be shortened and improved in the shortening.

Why is this? I have no idea. I don't think there is anything magical about those numbers. If I were to guess, I'd say that people do not construct or tolerate processes that take more than that many intermediate goals. I keep waiting for a legitimate counter-example, just to prove that the numbers have no deep significance.

Whatever the reason for the numbers, you can use this observation to improve your use cases. Many beginning use case writers write use cases 10 to 20 steps long. They make one of two mistakes:

1. They include user interface details that they shouldn't.
2. They write action steps at too low a level.

Check for both of these. They are described separately in *Mistakes Fixed*, in *Getting the GUI Out*, and *Raising the Goal Level*.

Label each use case

The first and final thing to do is to label the goal level of the use case. First, because that is what you should do. Last, because you probably didn't do that, and so you need to go back and do it before you save the use case.

The templates included with this book have two fields to help the reader understand the use case contains: the *design scope*, and the *goal level*. I never proceed without checking these two fields. The first announces what is *not* in the system under discussion, and hence needs to be mentioned. The second announces at what level this use case is operating.

Once the use case is labeled, then you can check

- whether the title or goal of the use case is appropriate for that level.
- whether the steps in the use case are appropriate steps for that level. They should be just one level below.

12. Corporate scope, system scope

If goal levels are the dominant source of confusion is, design scope is right behind. The writer and the reader often have different ideas of "just exactly what are the boundaries of the system we are describing?" *Business use cases*, and *system use cases* are a pair of terms that many people have invented, spontaneously and independently. Because of the evident clarity of the terms, they are fairly safe to use.

A "business" use case is one in which the topic of discussion, or scope, is business operations. The use case is about an actor outside the organization achieving a goal against the organization. The goal level is usually user-goal or strategic.

A "system" use case is one in which the design scope is the (typically) 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, even if it successfully finesses the details of user interface design. The level of greatest interest is the user-goal, although we write the other levels as called for.

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 an as-yet undetermined design. The system use case describes the outside of the computer system, the business use case described the inside of the business.

The business use case often contains no mention of technology. These use cases concerns are with how the business operates, what the goals are of the employees or departments, and not the use of the technology.

Exceptions to these generalizations can be found, of course:

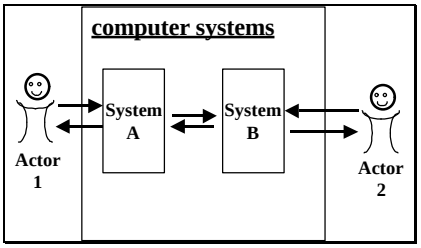
- Writing a black-box business use case as a preAcura to the white-box ones, to illustrate the operational environment in which the white-box business use cases are situated.
- Writing a white-box system use case to show how the subsystems work together to deliver the overall systems' functions. The white-box use case might be used for documentation, or to provide context for the use cases that will be written as requirements for the subsystems.

Making scope easier to discern

Always mark the scope portion of the use case template. If there is no scope descriptor in the use case template, collect together all the use cases of one design scope.

Consider using a graphic icon at the top left of the use case to pictorially illustrate whether it is a business or system use case (see " Using graphic icons to indicate scope and level" in section *Goal Levels*).

Consider drawing a picture of the actual system scope inside its containing system, within the use case itself, to illustrate it specifically.



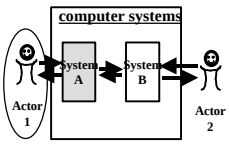
Here is an example. System, A is being built to replace an old system, B. For a year or two, during development, function will be transferred from B to A piecewise, which means the two systems will co-exist and synchronize with each other. A new entry will be put into system A, and then modified using system B. Use cases have to be written for both systems, for system A because it is entirely new, and for system B, which will be modified to accept input from the new, system A. Confusion is ready to set in.

We have to write as many as four use cases. Note that they are all "blue" level use cases, system scope.

- Actor 1, acting as primary actor on System A, enters a new X into system A, which, as part of its responsibility, notifies system B of the new entry.
- System A, acting as primary actor on System B, notifies system B of the new entry.
- Later, Actor 2, acting as primary actor on System B, modifies X.
- System B, acting as primary actor on system A, updates system A with the new state of X.

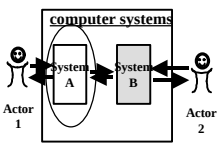
This situation is confusing at best, even with properly labeled templates and trained staff. Let us see how a bit of embedded graphics can help. In each of the following, the system under design is colored gray, and the primary actor of the use cases has an ellipse.

Use case 1. Enter new X into A
Scope: System A
Level: Blue
Primary actor: Actor 1



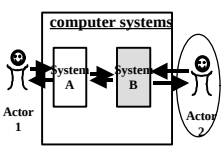
...use case body follows...

Use case 2. Note new X in B
Scope: System B
Level: Blue
Primary actor: System A



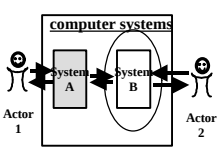
...use case body follows...

Use case 3. Update X into B
Scope: System B
Level: Blue
Primary actor: Actor 2



...use case body follows...

Use case 4. Note updated X into A
Scope: System A
Level: Blue
Primary actor: System B



...use case body follows...

13. Actors, roles and goals

When we say "actor", we could mean either the job title of the person using the system, or the role that person is playing at the moment of using the system (pretending 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. That is what this reminder is about.

The important part is the goal. This says what the system is going to do. Who, exactly, will call upon that goal, will be rearranged quite a bit over the life of the system.

When pressed for the rigorously correct answer for what we mean by "actor", the answer would be "role". However, we use both job title and role during the use case gathering activity (see *Job titles first and last*). Thus, we might start by noting that people with job titles Sales Clerk and Store Manager are going to use the new computer system. We might identify that Sales Clerk is the primary actor for "Order a product". Later, we notice that the Store Manager can also "Order a product". We can:

- Write in the use case that the primary actor is "Either Sales Clerk or Store Manager" (in UML, draw the arrow from both actors to the ellipse).
- Say that, "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 a new role, "Orderer", so that Orderer can be the primary actor. We would then say that Sales Clerk or Store Manager is acting in the role of Orderer when running this use case (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 well for your audience. The first point is to recognize that a person fills many roles, that 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. The second point is to recognize that the important part of the use case is not the primary actor's name, but the goal. The third is to recognize that it is useful to settle a convention for your team to use, so that they can be consistent in their use.

14. Job titles first and last.

Job titles are important at the beginning and at the end of the project.

At the beginning of the project, you need to collect all the goals the system must satisfy, and put them into a readable structure. Focussing 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 will inform the user interface design.

Once people start developing the use cases, however, the issues about overlapping roles and multiple primary actors described in the last reminder start showing up. The role names used for primary actor become more generic ("Orderer") or more distant from the people who will actually use the system. And so the job titles of the primary actors become less and less important, until they are only place-holders, reminding 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, relatively insignificant in the middle, become important again at the end.

15. Two exits.

Every use case has two possible exits, success and failure. Pay attention to both. Write down the names of the stakeholders and their interests. Write, and later doublecheck, how all the interests of the stakeholders are satisfied at successful completion of the use case, when the primary actor's goal is achieved. Also write, and later doublecheck, how the interests of the stakeholders are protected in the face of various kinds of failure. Recognize that the use case will have two exits (on rare occasion, I have seen a use case that can exit with partial success).

The success exit of a sub-use case returns to the statement in the calling use case that named the sub-use case. That statement could be in the main success scenario of the calling use case, or in one of the extensions - all steps are phrased as goals that succeed.

The failure exit of a use case returns to the condition in the calling use case that names the failure, so handling of that failure can be described. In that failure handling, either the failure will be repaired, or the calling use case's goal will itself fail.

16. Preconditions, postconditions.

The Precondition

The precondition names something that you want the SuD to ensure has happened before the use case starts. You want the system designers to go to extra lengths to have the system guarantee that this use case absolutely won't start unless these conditions are met, because these are the conditions that will never be checked during the steps in the use case.

If you find that an extension names a failure of a condition named in the precondition, then either the precondition or the extension contains a writing error.

If you discover that the use case actually can start without the precondition satisfied - if it is only the most degenerate or unusual situation, but it is technically permitted - then the precondition contains a writing error.

An example of a common precondition is "Use logged in and validated." An example of an overstated precondition might be that the use case "Evaluate order" cannot run unless it contains at least one order item. Usually, that is neither an actual requirement (an order evaluates to zero cost if it contains no items), nor can it be enforced (the user is permitted to evaluate any order at any time, and so there is no prior use case in which the contents of the order are checked).

How bad are these writing errors, how costly to your project? That depends on the characteristics of the project, as described at the beginning of the book. On a less critical project, or one with good internal communications, the cost is negligible. You might have overstated the

preconditions, but when a team member questions it, you provide the correction in a verbal response. The teammate then goes and corrects the code and tests. You have set up a project environment which is strong on personal communication, and can use a "soft" use case form, and a "low ceremony" process. In this situation, that is all right.

On a larger project, with longer, thinner communication lines between people, the cost is greater. It is more likely that the programmer, not able to ask the question of the use case writer, spends quite a bit of time trying to understand how to ensure the precondition, or whether to. In this case, a more "hardened" use case form is appropriate, with more attention to correctness.

This topic is discussed more in "Your use case is not my use case", in *Precision, Tolerance, Hardness*.

The Postconditions

You can skimp on the writing of the postconditions on less critical projects, where low ceremony process and softer use cases are applicable. You will pay more attention to the postconditions 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.

The obvious success end condition is that the primary actor has achieved the goal. Less obvious but fully as important is that every stakeholder must have their interests satisfied. Most often neglected in the use case text are logging and bookkeeping duties.

Review the stakeholders and their interests as you read the main success scenario, at least mentally. This does not take a lot of energy, and it usually is revealing. When writing the use case, write down the stakeholders and their interests *first*, then write down what must be the success end condition. Then you are in good shape to write a main success scenario that will meet those conditions.

Now think about failure of the goal. There are many ways to fail, more ways than you can tidily fit into the failure protection field. In all failures, though, the interests of the stakeholders must be protected. Reread the main success scenario, thinking about whether it protects the stakeholders against sudden and catastrophic failures.

Write the failure protection to illustrate the ways in which the system must protect the interests of the stakeholders. Sometimes these will take the form of "if" or "only if" statements (usually phrased in the past tense). Here are two examples, one with and one without the "if" condition:

> "Sufficient transaction status was collected to roll back or later continue with the transaction."
>
> "The account was only debited if the cash was successfully dispensed."

Once again, it is a good strategy to write the failure protection *before* writing the main success scenario, because then you will think of those protections while writing the first time, instead of discovering them later and having to go back and change the text.

17. Use includes.

I *always* use the "includes" relation between use cases. People who follow this rule report that they simply do not have the confusion around use cases that other people talk about. This entire

book coaches you how to do that. An increasing number of writers and teachers are coming to the same conclusion.

"Includes" means that the higher-level use case contains the name of the lower-level use case. "Includes" simply means that in the calling use case, you name the sub-use case. In the way of writing shown in this book, it is what you do on every step. It is the most obvious thing to do, and would not even deserve mention, if there weren't writers and teachers going around trying to teach people to use "extends" and "generalizes".

Like all good rules, it gets broken occasionally. In five years, I have seen three places when it was appropriate to write use the "extends"

Read the section *UML & the relations Includes, Extends, Generalizes* for the detailed discussion of how and when to use these other two relations.

18. Core use case values and variants

A number of teachers, consultants, and project teams have come up with variants on the style of use cases discussed in this book. Consensus is growing on core values. See, for example Susan Lilly and Don Firesmith's papers in TOOLS 99 avoiding the top 10 or 20 mistakes in writing use cases.

Core values

Readable. The ultimate purpose of use cases, or any specification, is to be read by people. If it can't be understood, it doesn't serve its core purpose. You can sacrifice some precision and even accuracy for readability, but you can only sacrifice so much readability before the use cases stop meeting their purpose. Using the writing styles in this book, the use cases can be as precise as you need, and still readable.

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 2/3 of the value of the use case. Correcting and adding to the extension keeps adding value, but changing the wording of the sentences ceases to improve the communication after a short while. After that, 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.

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".

Usable for several purposes. Use cases are a form of behavioral description that can be used at various times in a project, and for various purposes. Although most often named as a black-box functional specification deliverable, they have also been used:

- to provide requirements to an organization's business process redesign.
- to document an organization's business process (white box).

- to encourage and elicit requirements from users or stakeholders, being discarded afterward, as the team members write the requirements in a previously standardized form (e.g., numbered paragraphs and "system shall" statements).
- 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.

Although in each case, the use cases are written by different categories of people, and for different purposes, in different parts of the project or corporate life, the rest of the core values, and the writing style, are in common.

Black box requirements. When used as a functional specification technique, the SuD is *always* treated as a black box. The project teams my colleagues and I have interviewed, who have tried writing use case requirements guessing what the insides of the system will look like, report that those use cases were hard to read, not so well received, and "brittle", changing as the understanding of the system internals changed.

Alternative paths after main success scenario. Jacobson's original idea of putting the alternative courses after the main success scenario keeps being voted as the preferred, and easiest to read form. Putting the branching cases inside the main body of text makes the story too hard to read and evaluate (see "no if statements", below).

Suitable variants

Even keeping to most of the advice in the book, and the core values above, a number of acceptable variants have been discovered.

Either numbered steps or simple paragraphs. While some number of people number the steps, in order to be able to refer to them in the extensions section, other people choose to write in simple paragraphs and put the alternatives in similar paragraph form. One such is shown in *Writing Samples: Manage Reports*.

Casual or fully dressed template. There are times and projects for which the high energy is appropriate. There are other projects, and other moments, even on the same project, when it is not a good use of energy to label each step and extension, the pre- and postconditions, stakeholders, performance requirements, etc. (this is described in *Precision, Tolerance, Hardness*), and the casual template is appropriate. This is true to such an extent that I don't even publish or recommend just one template any more, but always show both. The short example at the end of this section shows the "log in" use case written in low energy style in simple paragraphs.

Full sentences or theatre-style. Some people write full sentences, with noun, verb, and so on. Others choose to separate the actor's name at the start of step, as is done in plays. One such is shown in "Example of white-box business use case using fully dressed template" in *Precision, Hardness, Tolerance*.

Prior business modeling with or without use cases. Some organizations go straight to writing the functional requirements for a system (in use cases, of course ;-). Some prefer to revise or at least document the business process around the system-to-be. Of those, some choose use cases as the way to describe the business process, and some choose other business process modeling notations. From the perspective of the system functional requirements spec, it does not seem to make much difference which business process modeling notation is chosen.

Use case diagrams or 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 actors linked to their user-goal use cases can serve the same purpose as the actor-goal list. That diagram, however, can also show the secondary actors, which the actor-goal list does not do.

White box documentation or collaboration diagrams. There is a close equivalence of white-box use cases and UML's collaboration diagrams. One might think of, or use, use cases as a textual collaboration diagram. The difference, besides being textual, is that a use case does describe what the SuD must accomplish internally, which the collaboration diagram does not.

Use case: Login. 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.

Casual user case. (from Booch, Martin, Newkirk, OO Design with Applications, 3rd edition.)

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, just writing, "If ... then ...". However, as soon as there are two of these, people tend to lose the thread of the story. Use cases have many branches, and often branches within the branches. People who have tried putting "if" statements inside the main story consistently report that they changed to the form using main success scenario and extensions or alternate courses. Notice that even in the simple and casual "login" use case above, the authors separated out the extension conditions, presenting them after the main story line.

Sequence diagrams as replacement for use case text. Some software development tools claim to support "use cases" because they supply sequence diagrams, which also show interactions between actors. However,

- Sequence diagrams do not show the internal responsibilities of the SuD, which the use cases should cover (if being used for functional specifications).
- Sequence diagrams are much harder for the evaluating sponsor or user to read, because they are a specialized notation, and because they take up more space.
- It is difficult to impossible to fit the needed amount of text on the arrows between actors.
- Most tools force the writer to hide the actual text behind a pop-up dialog box, making the story line impossible to follow.
- Most tools force the writer to write each alternate path from the beginning of the use case, causing much duplication of effort, and making it hard for the reader to detect what difference of behavior is presented in each variation.

In short, sequence diagrams are not a good form for expressing use cases. Those people who do insist on sequence diagrams, do so in order to get the tool benefit of automated services: cross-referencing, back-and-forth hyperlinks, global name changing ability. 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 (see Core Values, above).

GUIs in the functional specs. There is a small art in writing the specification so that the user interface is not specified along with the function. This art is not hard to learn and is worth it. There is strong consensus not to describe the user interface in the use cases. See *Mistakes Fixed: Getting the GUI out*, and Designing Software for Use, by Larry Constantine and Lucy Lockwood.

19. Planning around goals.

In *Planning and Tracking*, I discussed the pluses and minuses of use cases in setting up tracking project progress. The actor-goal list is a most useful project planning resource, even when the system is not delivered in full use case increments. This is the reminder to construct a use case planning table and to be aware of partial use case delivery.

The use case planning table

Put the actors and goals in the left-most two columns of a table, and in the next columns you can record any of: business value, implementation complexity, scheduled release, owning implementation team, degree of completeness, and other project-planning information (you can put other sorts of information in other, similar, tables: performance requirement, external interfaces used, and so on). See the figure below for an illustration taken from a real project (yes, use case #1 is not there - it was a strategic, not task-level goal).

Actor	**Task-level Goal**	**Business Need**	**Technical Difficulty**	**Priority**	**Use-case Nr.**
Any	*Check on requests*	Top	Large job in general case	1	2
Authorizor	*Change authorizations*	High	Simple	2	3
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

Example of a use case planning table.

Using this table, your team can negotiate over the actual development priority of each use case. They will discuss business need versus technical diffiiculty, business dependencies and technical dependencies, and come up with a sequencing of development.

Delivering partial use cases

As described in *Planning and Tracking*, 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. In this case, you might want to write in the table the *first* release in which the use case shows up, and the *final* release, which will deliver the use case in its entirety (and you don't have to worry about it any more).

Partially complete classes / components

One of the things we have learned, working this way, is that the domain classes are only ever complete with respect to the delivered functionality - they are never "done" in some complete sense. A new path through a partially delivered use case, or a new use case, could trigger a change or addition to a class or component.

It seems that this is a cultural shock for people coming from the earlier, database development paradigm.

There was a time when the database designers, drawing the entity-relationship, would insist on discovering every attribute of every entity. Adding a new field to an existing table and optimizing all the database indices was considered an enormously expensive activity. The economics of development therefore suggested at least identifying and marking all the attributes that would *ever* be referenced. The released software could then be called 20% complete, 40% complete, up to 100% complete, with respect to the identified attributes.

The economics - and culture - are different in object-oriented development, and in organizations that do not need heavily optimized database indices. In these projects, only that part of the class or entity is defined that is needed immediately. Adding an attribute to a class or table is considered a "minor" operation.

Therefore, the class or entity can only ever be called, "complete with respect to the delivered use cases." The team does not try to guess into the future, and so it is not possible to say what the ultimate 100% completion point is. It is not possible to say that the class is "30% complete". It is possible to say that class is "30% complete with respect to these use cases." The very next use case, or the one to be developed a year from now, might change the contents of the class.

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 former, database design culture, but working on a project in the latter, object-oriented 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."

20. The Great Drawing Hoax

For reasons that remain a mystery to me, many people focussed 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, already having graphical metamodeling tools, and no 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 in 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.

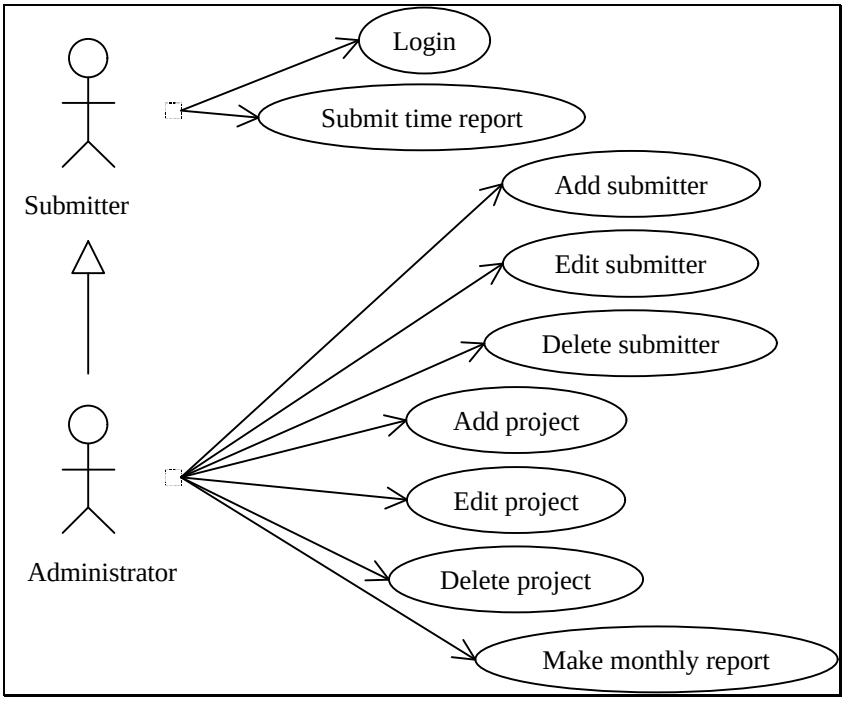
Whatever the causes, we now have a situation in which many people think that the ellipses *are* the use cases. One student in my lecture asked, "When do we write the text? Is it when we get to the leaf level decomposition of ellipses?" This despite the fact that the experts are actively downplaying the ellipses, teaching people to write text.

It is important to recognize that the ellipses cannot possibly replace the text. The use case diagram is (intentionally) lacking. It does not sequencing, data, or receiving actor. The use case diagram is to be used

- as a sort of table of contents to the use cases,
- as a sort of "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 form of "big picture", showing how higher-level use cases relate to lower-level use cases.

Everything that can be done with use case diagrams can be done with text, and done in a way that is easier for the end users to read and respond to. The text supports the ever-unfolding story from the highest goals of the strategic business use case down to the most detailed interaction.

Although some people, like myself, do not use the diagrams at all in their use case work, some people like seeing the context diagram (as in the example, below), and some people like working with the ellipses as a table of contents, down to one or two levels of expansion. This is fine, as described in *Core values and variants*.



Example of a use case "context" diagram.
(Adapted from Booch, Martin & co, OO Design with Applications.)

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

The same use cases in actor-goal format.
(Common use cases repeated for clarity)

There is one value to drawing the exploded use case diagram: showing project progress. Consider posting the sailboat image of the use case graph (shown below, and described in *User goals and level confusion*). Color each use case ellipse *yellow* as it is written and accepted into the

project, *green* as design work on it starts, and *red* as it is completed and delivered. Over the course of the project, the exploded use case graph will show the status of the project, growing to completion. It will always be clear what is being worked on at the moment, and what is remaining. (A little truth-in-advertising here: I have suggested this and heard this suggested, but have yet to see it actually done!).

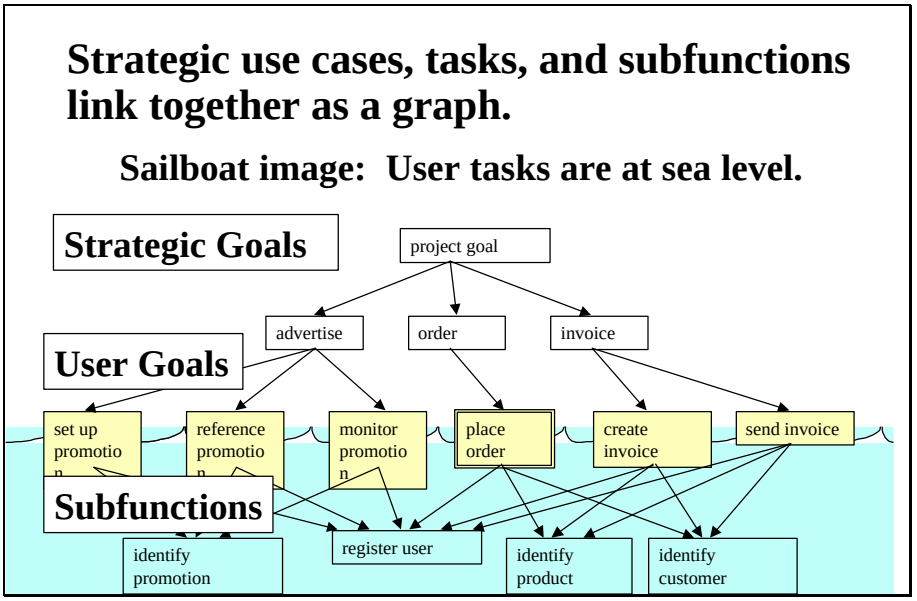


Figure. Sailboat image of use case graph.

In short,

- Remember that use cases are fundamentally a text form.
- Use the ellipses to augment, not replace the text.

21. The Great Tool Debate.

Sadly, use cases are not supported very well by any tools on the market (late 1999, to my knowledge). Given the size of the market, many tool vendors claim to support them, either in text or in graphics. However, none of the tools contain a metamodel even close to that described in *A metamodel of use cases* in an easy-to-use form. 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 also allows the expanding data descriptions to be kept in the same database, but in different views. 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.

Still, there are unfortunate aspects to using Lotus Notes. Renumbering the steps soon becomes a time-consuming annoyance. When we insert a step between steps 3 and 4, we have to renumber not only every step after 4, but also every line in the extensions section, starting from 4a. There are no automated back links, so that while naming and linking to a sub-use case is easy, there is no good way to get from the sub-use case to the list of calling use cases. Links, being manually maintained, eventually get out of date. This is small at the beginning, but becomes a growing nuisance over long periods of time.

Even with those drawbacks, it still comes out the best, in my view, due to ease of use and the view-and-edit across the use case set.

Simple word documents. With the inclusion of hyperlinking, word processing software finally became a viable tool for use cases. Put the use case form into a template file. Put each use case into its own file, and it becomes easy to create a link to the use case. Just don't change the file's name! Word processors are available and known tools.

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.

Requirements management tools. I have seen requirements management tools, such as Requisite Pro, and DOORS (I am sure there are others), nominated to capture use cases, but have not seen them tried on a project. On the plus side, these are tools intended to track requirements bidirectionaly and have an internal metamodel and hence bi-directional hyperlinks where they support links. 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, which will limit their effectiveness.

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.

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 based around box-and-arrow metamodels, and do 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.

Mistakes Fixed

1. Getting the GUI out

I have talked about avoiding naming the use interface definition in the use case. That is difficult for some people to imagine. It is perhaps best illustrated through a few examples.

Russell Walters and Dave Scott of 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 challenge was to make it look good according to the principles of writing effective use cases. The "after" version is 1/4 as long, containing the same basic information, but without constraining the user interface.

Before:

Use Case: Research a solution (Ed. note: alias, "Select a product")
Scope: Our web system
Level: User-goal
Primary actor: Shopper - a consumer or agent wanting to research products they may purchase.
Main success scenario: (note: this is in 2-colum, "conversation" form)

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:	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.
23. Shopper will answer questions	
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:	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.
35. Shopper will answer questions	
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.

Alternatives

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.

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.

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.

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.

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*

At any point during the research process, the Shopper may request to be contacted: Initialize Request For Contact Use Case.

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.

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.

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*

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.*

Notes on the Before:

The writers of the use case had selected the 2-column form, to make the separation of action clearer, even though it would make the use case longer. Also, they could or did not visualize any other user interface design than the question-and-answer model, so they described that question and answer sequence in the use case.

My approach was, first, to put it into simple story form in 1-column format, and second, to try to revise and remove any UI design I saw. To do so, I focussed on the intention of the user at every point. Eventually, I imagined that there would be a completely different design design that would somehow accomplish the same effect. I conjured up in my mind a design in which there would be no typing at all, but that the user would only click on a picture or several . I was then able to capture the intent and remove UI dependency. As you will see, this also shortened the writing enormously.

I also rechecked the goal level in each statement, to see if perhaps the goal of the step was too low level, and could be raised. Here is the result.

After:

Use Case EC5 Research possible solutions

Context of use: The Shopper gets guided through a tailored product selection process according to their personal characteristics as revealed by their selections along that process (perhaps they answer a series of questions, or click on some pictures or text - what they click on instructs the program as to what to ask or present next). At the end of the questioning process the shopper will be presented with product recommendations that meet their personal characteristics. At any point within the questioning process the shopper can choose to bypass the remaining questions and the system will move the shopper to product recommendations and present products based on limited shopper's personal characteristics. In the case that the shopper bypasses all questions the shopper will be presented will all products to directly select. Having the product identified, by whichever path, the shopper can add it to the shopping cart, and start over with another product. The system captures market data (product selection, Shopper navigation, and characteristics) along the way.

Scope: web software system

Level: Task

Preconditions: none

Success End Condition: 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.

Failed End Condition: <the state of the world if goal abandoned>

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.
3. The **system** logs the navigation information along the way.
4. Shopper selects a final product configuration, considers it.
5. Shopper adds it to the shopping cart.
6. The **system** adds to the shopping cart the selected product, and the navigation information.
7. 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

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: The system does the best it can, logs all the incoming information, and continues as it can.

1b. Shopper wants to continue a previous, saved, partial solution:

1b.1. Shopper identifies themselves (Establish Identity) and their saved solution.

1b.2. The **system** recalls the solution and returns the shopper to where they 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 a 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:

2d.1. The system issues a sales lead (Generate Sales Lead).

2e. System has been setup to require the Shopper to identify themselves: (Establish 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: (View Other web-site).

2h. System is setup to interact with known Customer Information System:

2h.1. System interacts with Customer Information System (Retrieve Customer Information).

2h.2. System uses results to start from some advanced point in the question/answer series.

2i. Shopper indicates they want to know their finance credit rating because it will influence their product selection process: (Obtain 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 will (Retrieve Billing History).

2j.2. The System will utilize the shopper's billing history to influence the product selection process.

4a. 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.

5a. Shopper decides not to add the product to the shopping cart: System retains the navigation information for later.

7a. 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.

7b. The shopper asks to save the contents of the shopping cart:

7b.1. The system prompts the shopper for name and id to save it under, and saves it.

7b.1a. The Shopper's identity has not been determined: (Establish Identity)

7c. The shopper has a trade-in to offer: (Capture Trade-In).

7d. Shopper decides to finance items in shopping cart: (Obtain Finance Plan)

7e. Shopper leaves the E-Commerce System when shopping cart has not been saved:

7e.1. The system prompts the shopper for name and id to save it under, and saves it.

7e.1.a The system prompts the shopper for name and id to save it under, and the shopper decides

to exist without saving: system logs navigational information and ends the shopper's session.

7f. Shopper selects an item in shopping cart and wishes to see availability of matching product (new or used) from inventory:

7f.1. Shopper requests to (Locate Matching Product from Inventory).

7f.2. Shopper wishes to fulfill (exchange) item in shopping cart with matching product from inventory.

7f.3. System ensures there is one item in shopping cart configured to inventory item.

7f.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.

8a. At any time, the shopper can ask to be contacted (Request Contact).

RELATED INFORMATION (optional)

Superordinate Use Case: Shop over the web, Sell Related Product

OPEN ISSUES (optional)

Extension 2c. What questions are legal? What other systems are hooked in? What are the requirements for the interfaces/

Notes on the After:

One of the open questions at the start of this work was whether the 1-column format would still show the system's responsibility clearly to the designers. Rusty's evaluation was that all the real requirements were still there, and were clearly marked. In fact, he was happier because:

- the result was shorter and easier to read, while keeping all the actual requirements,
- the design was less constrained.

2. Raise the goal level

Before:

Fragment of a use case:

...
3. User provides name.
4. User provides address.
5. User provides telephone number.
6. User selects product and quantity.
7. System validates that user is a known user.
8. System opens a connection to warehouse storage system.
9. System requests current stock levels from warehouse storage system.
10. Warehouse storage system returns current stock levels.
11. System validates that requested quantity is in stock.
...

Notes on the Before:

It is clear that this is going to be a long use case. We can't criticize the above steps on the grounds that they describe the user interface too closely. What can we do to shorten the writing and make it clearer what is going on?

After:

...
3. User provides personal information (name, address, telephone number) and selects product and quantity.
4. System validates that user is a known user and that the product is in stock.
5. System validates with the warehouse storage system that sufficient quantity of the requested product is in stock.
...

Notes on the After:

I did three things to shorten the text.

1. I merged the data items into a single steip. The original text used separate steps for the individual data items. I asked, "What is the user trying to provide, in general?", and answered, "personal information". "Personal information" is a handle, a nickname that people will use, to refer to all of the pieces of information that will be collected about the person. It is the data description at the first level of precision (see *Data Descriptions*). By writing the handle of the data collected, we not only shorten the use case, but we allow for the details of the collected data to be changed over time, without affecting the use case in any major way.

2. I merged several, different, pieces of information going in the same direction, into one step. This is not always the best thing to do. Sometimes "providing personal information" is so different from "selecting product and quantity", that the writers prefer to have them on separate lines.

That is a matter of taste. I always try writing into one step all the information going in one direction. If it looks too cluttered, or it turns out that the extensions beg to have them separated, then I separate them again.

3. I looked for a slightly higher-level goal. I asked, "Why is the system doing all these things in steps 8-11?", and answered, "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 usually captures the requirements as clearly as before - even more clearly, since it is so much shorter.

3. Put the actor back in

Before:

Fragment of use case for withdrawing money from an ATM:

...

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

...

Notes on the Before:

This use case shows everything the ATM does, but does not show the actor's behavior. In general, this sort of use case is harder to understand than the ones with full sentences and actor behavior, and harder to verify and correct. In some cases, critical information is omitted about the actor's behavior, often having to do with sequencing.

After:

This book already contains enough examples of the ATM use case written in full actor-action style. I won't repeat them here.

4. Kent Beck's Three Bears Pattern

Kent Beck presented a learning pattern he called the *Three Bear's Pattern*. This pattern is based on perceptual psychology, that a person learns a perceptual gradient by naming different points along the gradient. Kent's explanation is simpler (the following is my paraphrase):

> Goldilocks ate some porridge that was too hot, some that was too cold, and some that was just right. She lay on a bed that was too hard, one that was too soft, and one that was comfortable.
>
> You, too, can learn what is a comfortable zone when faced with a gradient or spectrum of choice. Deliberately overdo to one end of the spectrum, then deliberately overdo to the other end of the spectrum. At this point, you will be able, much more easily, to be able to place the third attempt at a comfortable place along the gradient.

In terms of the "burnt pancakes" way of talking about quality, discussed by Luke Hohmann in Journey of a Software Professional, this might correspond to making use cases with batter: too thick, too watery, and finally, about the right consistency

Use cases are full of gradients, gradients of design scope, gradients of goal level and step size. Try Kent's Three Bears pattern to your use case writing once or twice to learn those gradient.

Exercise: ***The three bears visit an ATM***. Write the papa bear version, with extremely, overly detailed steps about how it goes. Write the mama bear version, with too little detail and too few steps. Write the baby bear version, with a good level of detail and step size.

Exercise: We are trying to learn what is "user-goal" level, and working on the Personal Advisor / Finance software. Name a goal level that is way too high to be useful for the spec, name a goal level that is way too low (you might use it as a subfunction use case), and name one that is "about right".

Answers to Exercises

Exercise 3A. System-in-use story for an ATM:

FAST CASH OPTION.

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.

Exercise 7A:

The "enter" button.
The key panel.
The ATM itself.
The bank's computer system, ATM plus mainframe.
The bank as an entity.
The electronically connected industries.
The community in which you are operating, or the subset of the world for this problem.

Exercise 8A:

	Not an actor	Primary actor	Secondary actor	System under design
The ATM				yes
The customer		yes		
The bank teller		yes		
The ATM card	no, it is a data item unless it has behavior of its own			
The bank	no, it is the containing system of the containing system			
The front panel	no, it is a subsystem (an actor in a lower-level set of systems)			
The bank robber		yes		
The bank owner	no, a stakeholder			
The serviceman		yes		
The printer	no, a subsystem			
The mainframe			yes	

The net link	no, an interface (probably)			

Exercise 9B:

The containing system is the bank's electronic and computing parts (not the people).

	Not an actor	Primary actor	Secondary actor	System under design
The ATM	no, it is now a subsystem			
The customer		yes		
The bank teller		yes		
The ATM card	no, it is a data item unless it has behavior of its own			
The bank	no, it is the containing system			
The front panel	no, it is a subsystem			
The bank robber		yes		
The bank owner	no, a stakeholder			
The serviceman		yes		
The printer	no, a subsystem			
The mainframe	no, it is now a subsystem			
The net link	no, it is now a subsystem			

Exercise 9A:

Strategic (white): Take someone out for dinner.
Strategic (white): Use the ATM
Task (blue): Get money from the ATM.
Subfunction (indigo): Enter PIN
Subfunction (black): Find the Enter button.

Exercise 9B:

Actor	Goal	Level
Serviceman	Put ATM into working order	strategic
	Run ATM self-test	task
Bank Clerk	Restock money	task
	Refill supplies	task
Customer	Use the ATM	strategic
	Withdraw cash	task
	Deposit money	task
	Transfer money	task

	Check balance	task

Exercise 10A:

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 10B:

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.
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 12A:

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 13C:

"Perform clean spark plugs service"
Precondition: car taken to garage, engine runs
Success postcondition: engine runs smoothly
Failure postcondition: customer notified of larger problem, car not fixed.
Main Success Scenario:
1. cover fender with protective materials.
2. open hood and 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)

Exercise 13D:

"Buy stocks over the web"
Primary Actor: Purchaser / user

Scope: PAF
Level: Task
Precondition: User already has PAF open.
Success End Condition: remote web site has acknowledged the purchase, PAF logs and the user's portfolio are updated.
Failed End Protection: sufficient log information exists that PAF can detect that something went wrong and can ask the user to provide the details.
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, asks the user to *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 14A. Failure end conditions for withdrawing money.

The customer's account is not debited if the money was not issued; the ATM logged each step sufficiently to detect where a transaction got aborted, and what completed.

The three bears visit an ATM

Papa bear (too detailed)

Mama bear (not enough detail)

Baby bear (OK detail)

Three bears pattern and user-goal level

Too high: "Get rich"
Too low: "Save file"
About right: "Make a new investment"

Writing Samples

1. Moving high level to low level

I selected the following use cases because they are examples built using the Stakeholder-Interests model. There is almost no discernible difference in the writing - it is just that the writing was double-checked against the stakeholders and their interests along the way. The inclusion of stakeholders was an experiment, and considered a successful one at the end of the writing period. Including them did not cost much energy or time, while giving us a tighter framework for thinking about what we were producing, and helped us notice when we had accidentally omitted steps.

These use cases start from very high ("white" / "strategic"), move through "blue" or "user goal" level, down to "indigo' or "subfunction", illustrating how to shift goal levels. Watch also as the system scope shrinks from "company operations" to "all computer systems" to just the system under design.

UC #: 1010 Handle Claim (business)

Scope: Insurance company Operations
Level: Business strategic
Release: Future Status: Draft Revision: Current
Context of use: Claims Adjuster handles claim.
Primary Actor: Claimant
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 reports a claim 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 End Condition: Claim is resolved and closed
Failed End Protection:
Stakeholder & interest:
Insurance company Divisions who sell Insurance company policies

Insurance company Customers who have purchased policies
Depart 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 this use case: This is a "business" use case - the "system under design" is the entire 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 main 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.

UC #: 1014 Evaluate Work Comp Claim

Scope: Insurance company Operations
Level: -White (Business, 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 End Condition: Claim is evaluated and settlement range determined.
Failed End Protection: Additional investigation or medical evaluations are completed until claim is ready to be re-evaluated for settlement.
Stakeholder & 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 this use case: This is a again a "business" use case - the "system under design" is the entire operations of the company. However, it uses a lower-level goal than the preceding one: in this, the work of an adjuster is shown, work that may take days, weeks or months. For that reason, it is considered a "strategic goal" use case - it contains many single-sitting activities.

The reason I include this use case is because as business use case, it starts to show how the computer system might be used to simplify the life of the adjuster. Also, step 7 was added because of the interests of the Department of Insurance.

UC #: 2 Handle a Claim (system)

Scope: "System" means all computer systems combined
Level: White (Strategic)
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 Variations:
8. Settlement payment is
a. by check
b. by interbank transfer
c. by automatic prepayment of next installment
d. by creation and payment of another policy

Frequency of occurrence: to be documented
Success End Condition: Claim is closed
Failed End Protection: Claim must be reopened
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:

Writer:
Critical Reviewers:

Open Issues: What are the time periods for archiving claims?

Commentary on this use case: To many people on the project, this "system" use case seemed so vague as to be useless. However, it served three purposes that paid for its writing time.

First, it glues together a number of user-goal use cases, showing how they fit within the business guidelines. It also shows how and when the closing, purging and archiving happen, which otherwise was a mystery to a number of people writing use cases and implementing the system.

Second, it put into the official log certain sequencing and business rules which were unknown to some of the team. The team has spent 3 work hours the day before this was written trying to guess those rules. Once this use case was written, by the people who knew the ruled, the rest of the team was saved many more work hours of discussion about the topic.

Third, this use case serves as an introduction and table of contents to new readers of the set of use cases, from new hires to the company executives. The executives could see that the key processes were included. The newcomers could learn how the company worked, and could start to drill down into the larger number of user-goal use cases.

UC #: 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

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.

2f. 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)).

2f1. 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. Timeout: 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: ??
Success End Condition: loss information is captured and stored
Failed End Protection: Nothing happens.
Stakeholder & interest:

Business Rules:

0. When does saved loss goes to System A (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: Search details for policy, Policy index information, Preliminary loss information, loss information specific to claim line, 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: Thanks, Susan and Nancy!
Critical Reviewers: Alistair, Eric, ...

Open Issues:
How often does it autosave.
Agent acknowledgement cards, where and how do they print etc.?
Coverage verification box? necessary or is it implied if coverage was copied to field?

Commentary on this use case: This is one of the most interesting and complex use cases I have seen. It shows why we like use cases to be "prose written for a human audience".

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. And yet, there is a "normal" sequence they should try to work through. And still, the computer should to attempt to facilitate the work at every turn. We wrote at least four completely versions of this use case, trying to be clear, showing the normal work path, and showing the computer working asynchronously. Perhaps on the 7th or 8th revision, we would have found something better, but the writers felt they had passed the point of diminishing returns and stopped with this (quite adequate) version.

This use cases makes multiple invocations to "Find a whatever", each time mentioning a different thing to find, and different search criteria. Indeed, the rules the system must obey while supporting a search for a policy are different from those it must obey while supporting a search for a customer. The team came up with this ingenious solution to avoid rewriting the standard steps for searching for something: match lists, sorting criteria, resorting, researching, no items found, etc. It was both readable and precise.

The extension, '*a. Power failure', introduced the notion of intermediate saves. The intermediate save became important, because it 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 surprising for the team. It ended with the failure condition '6b', which dealt with timeout 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 how failures fit within failures. The writers could have turned this into its own use case. They decided that 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 taking them, before breaking the extension out into its own use case.

Extension '2-5a' shows again 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.

UC#: 24 Find a Whatever

Note: Among the little things that plague use case writers are sets of twisty little use cases, almost all the same... but not quite. One group adopted a clever tactic - they parameterized both the name and the arguments to the use case, setting up a writing convention.

Any sentence that read Find a Customer or Find a Product would mean "use the use case Find a Thing, where 'thing' is Customer (or Product, or whatever)". The particular data and search restrictions were logged on a separate sheet in Lotus Notes, so the correct invocation would be "Find a customer using Customer search details."

With this neat convention, the logistical details of Find a Thing could be written just once, and used in many similar but different contexts.

Exercise: write the use case for "Find a Whatever", whose trigger is that the user decides to locate or identify a whatever. The use case should allow the user to enter search criteria, sort, etc. 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.

2. Buy Goods: a business use case

Use Case: 5 Buy Goods

Use Case: 5 Buy Goods

Context of use: Buyer issues request directly to our company, expects goods shipped and to be billed.

Scope: Company *Level*: Summary

Preconditions: We know Buyer, their address, etc.

Success End Condition: Buyer has goods, we have money for the goods.

Failed End Condition: We have not sent the goods, Buyer has not spent the money.

Primary Actor: Buyer, any agent (or computer) acting for the customer

Trigger: purchase request comes in.

Main Success Scenario

1. Buyer calls in with a purchase request.
2. Company captures buyer's name, address, requested goods, etc.
3. Company gives buyer information on goods, prices, delivery dates, etc.
4. Buyer signs for order.
5. Company creates order.
6. Company ships ordered goods to buyer.
7. Company ships invoice to buyer.
8. Buyers pays invoice.

Extensions

3a. Company is out of one of the ordered items:

3a1. Renegotiate order.

4a. Buyer pays directly with credit card:
4a1. Take payment by credit card (use case 44)
8a. Buyer returns goods:
8a. Handle returned goods (use case 105)

Technology Variations

1. Buyer may use
   - phone in,
   - fax in,
   - use web order form,
   - electronic interchange

8. Buyer may pay by
   - cash or money order
   - check
   - credit card

Project Information

Priority	Release Due	Response time	Rate of use
Top	Rel. 1.0	5 minutes for order, 45 days until paid	200/day

Superordinate Use Case: Manage customer relationship (use case 2)
Subordinate Use Cases:
- Create order (use case 15)
- Take payment by credit card (use case 44)
- Handle returned goods (use case 105)

Channel to primary actor: may be phone, file or interactive
Secondary Actors: credit card company, bank, shipping service
Channels to Secondary Actors:
Open issues
What happens if we have part of the order?
What happens if credit card is stolen?

Commentary: The real use case, from which this was extracted, had a long set of extensions dealing with what happened when an authorized broker sold goods to the military, who might then return parts, etc. Even with that extension, the entire use case took less than two pages.

The sponsoring executive, when glancing over the text, immediately asked, "What about automatic warehouse triggering?" We were panicked, until we discovered that was out of the system's scope. Yet it revealed how appropriate our writing was, that the omission was so clear to him.

As with other very high-level use cases, this one serves as an introduction, an overview, and a lead-in to the user-goal level use cases.

3. Handling a nighttime bank deposit

My thanks to the Central Bank of Norway for allowing me to reproduce these draft use cases. These are presented essentially as written (with a few Norwegian phrases replaced by their English equivalent). The team of three programmers never needed to take them beyond this writing in order to do their job, an example of good use of the principle of diminishing returns: even an approximate use case may be sufficient for the job.

The system about to be redesigned was that used by the Central Bank of Norway to credit ordinary bank customers for the cash deposits they put into their local banks. Notice who pays for the service - that sure surprised me!

The writer of the use cases was a programmer, who used acronyms to shorten his typing.

PA means "Primary Actor".

NB is the Central Bank of Norway (Norges Bank).

BBS is the IT funneling agency that sits between the banks and NB.

RA means "Receiving Agent".

RO means "Registration Operator"

DS means "Department Supervisor"

UC1: Get credit for nighttime money deposit

Primary Actor: Customer

System: Overall banking system: Bank, NB, BBS

1. Customer drops money into Bank night-safe deposit
2. Bank gives money to NB, who counts it three times and gives receipts to Bank and Customer
3. NB issues credit to Customer's Bank's account through BBS, who sends the credit notice into NB.
4. NB invoices Bank for the work..
5. Credit shows up on Customer's next statement.

3'. NB credits Customer's Bank's account in NB, faxes that back to Bank, who then credits the Customer's account.

System: Nightime Receiving Registry

Potential actors: Receiving Operator: RO
Department Supervisor: DS

RA goals

1. Register arrival of a box (bags in box, who delivered it, date, <estimate of amount>)
2. Change registry for an old arrival

DS goals

1. Get reports

UC MO1. Register arrival of a box

RA means "Receiving Agent".
RO means "Registration Operator"
DS means "Department Supervisor"

Primary Actor: RA
System: Nightime Receiving Registry Software
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.

UC RO1. Register a bag from a box

PA. Registration's Operator RO
System: Nighttime Receiving Registry Software
1. RA hands cart full of bags to RO.
2. RO registers bag externals with system
    RA id
    date, time
    bag id
    bag contents as written on outside of bag (amount NOK in cash, checks)
    Customer's account number and filial number

where is the box id?

Commentary: These use cases are considerably less formal than the early ones - and took less energy to write, and yet capture the information the programming team needed. They needed to cover the logistics of the situations they would encounter, and so the extensions and variations were very important to them. And they needed to write down the data they were obliged to capture in the computer.

Beyond that, since there were only 3 people, sitting next to each other, they didn't need to get fancy about the templates they used. These use cases are a good example of what, in Software Development as a Cooperative Game, is referred to as a "marker, intended to remind". The use cases took considerable effort to write, the writer had to double check the exact procedures involved, especially for the extensions and variations sections and for the data required. Therefore, once they were written, they served the important function of "reminding" the team of what they once knew. Hence the use of acronyms for the actors, and the cut-down template, were quite appropriate.

4. Manage Reports (a CRUD use case)

Note: These use cases were written by people who had attended my course and then were writing on their own, using the Rational Unified Process suggested template for their work. These two use cases show one approach to the problem of CRUD use cases ("create, replace, update, delete'). CRUD use cases plague use case writers, because they are similar and simple, which implies they should take little time and space, but they are hard to fold together. If folded together, they are ungainly, but if split apart, they quickly clutter up the use case file.

The question always is: is there just one use case with all the basic operations, or are there 3 or 4 separate use cases to track? This example shows one group's choice.

Different people have different responses on how to deal with these "little" use cases. Susan Lilly prefers to keep them all separate, so she can associate "which actors have permission to change the item" with the use cases on the use case diagram. Rusty Walters of Firepond is comfortable with the choices made by the authors of these sample use cases.

Compare these writing samples with the various responses in *Email Discussions: How to deal with CRUD use cases?*.

Thanks to John Colaizzi for these use case samples.

Use case 1: Manage Reports

1. Title: Manage Reports

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.

Actors

There are two actors for this use case which are:

- User (Primary)
- File System: typical PC file system or network file system with access by user. (Secondary)

Triggers

User Selects operations explicitly using the Explorer interface.

1. Flow of Events

Basic Flow – Open, Edit, Print, Save, and Exit report

f. User selects Report by clicking report in Explorer and selects open (open also triggered by Double clicking on a report in the Explorer).
g. System displays report to screen.
h. User sets report layout etc. using use case: "Specify Report Specs".
System displays altered report
i. Steps **c** and **d** repeat until user is satisfied
j. User Exits report using use case: "Exit Report"
k. 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.

Alternative Flows

Create New Report

c. 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".
d. Use case flow continues with Basic flow at step **b.**

Delete Report

b. User selects Report by clicking report in Explorer and selects Delete.
c. System opens report (or makes it current if it is already open) and requests validation from user for deleting report.
d. Report is closed and resources cleaned up
e. System removes report entry from report list and report data is removed from storage medium

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.

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.

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

2. Special Requirements

Platform

The platform type must be known for control of the report display operations and other UI considerations.

3. Pre-Conditions

- A data element exists on the machine and has been selected as the current element.

4. Post-Conditions

Success Post-Condition(s)

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.

Failure Post-Condition(s)

System waiting for user. The following lists some state possibilities:

- Report may be loaded
- Report list still valid

5. Extension Points

None

Use Case 2: Save Report

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".

Actors

There are two actors for this use case which are:

- User (Primary)
- File System: typical PC file system or network file system with access by user. (Secondary)

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.

6. Flow of Events

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
c. 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.

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.

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.

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.
   1. 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.

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.

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.

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

7. Special Requirements

None

8. 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"

9. Post-Conditions

Success Post-Condition(s)

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".

Failure Post-Condition(s)

- 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)

10. Extension Points

None

5. Looping inside an extension

Note: In the ATM example, there is an awkward moment when the writers discover they have to deal with a loop in the extension. The ATM should ask for the PIN up to three times, and then keep the card. How does one write this? Here are three versions, all of which I was willing to accept. However, the third version comes out as the most attractive, and I have seen this technique used to good effect with complicated extensions.

We pick up the story at extension 1a. Invalid PIN detected...

--- Group 1's version ---
1a. Bad PIN:
1a1. ATM prompts customer to reenter PIN
1a2. Customer enters PIN, and ATM validates it.
1a2a. Good PIN: continue with main scenario
1a2b. Bad PIN:
1a2b1. ATM prompts customer to reenter correct PIN
1a2b2. Customer enters PIN, and ATM validates it.
1a2b2a. Good PIN: continue with main scenario
1a2b2b. Bad PIN:
1a2b2b1. ATM prompts customer to reenter correct PIN
1a2b2b2. Customer enters PIN, and ATM validates it.
1a2b2b2a. Good PIN: continue with main scenario
1a2b2b2b. Bad PIN: ATM keeps card, writes to security log, notifies customer.

--- Group 2's version ---
1a. Bad PIN:
  1a1. ATM prompts customer to reenter PIN.
  1a2. Customer enters PIN, and ATM validates it.
    1a2a. Good PIN: continue with main scenario
    1a2b. 3rd failed attempt at entering PIN:
        ATM keeps card, writes to security log, notifies customer.

--- Group 3's version ---
1a. 1st & 2nd incorrect PIN submission:
  1a1. ATM prompts customer to reenter PIN.
  1a2. Customer enters PIN, and ATM validates it, continues with main scenario.

1b. 3rd incorrect PIN submission:
  1b1. ATM keeps card, writes to security log, notifies customer.

6. Documenting a Framework

Note: The original description of this service was 18 pages long with many drawings. After a week's intense work, the designers decided they were happier with this, much shorter, description, which somehow makes the problem appear simple :-). Another example of the 100x difference between writing a readable use case, and reading a readable use case.

Unlike most of the other use cases here, I had nothing at all to do with the writing of these use cases, although the writers had been working from my "Structuring Use Cases with Goals" article at the time. I have left their text intact, so you can see someone else's writing style. Thanks to Dale Margel of Nova Gas in Calgary for the use cases.

Service Description: This service is used by objects to protect critical sections of code from unsafe access by multiple Processes. These critical code sections read or modify the value of a Resource. The permission to do this is controlled by a Resource Lock. The Resource Lock calls are normally placed inside the critical sections of code by the programmer, and thus are not normally visible from outside the object. The Resource Lock does not actually control the Resource, it can only control access to itself - and thereby protect a Resource when used properly.

If a Process intends to read or modify the value of a Resource, it asks the Resource Lock for permission to do so. The Resource Lock will grant access if it is safe to do so, or it will suspend the Process until it can be safely resumed. If the Process will be reading a Resource, it requests Shared access; if it is to modify a Resource, then it requests Exclusive access. The Resource Lock serves these requests on a first-in basis, and ensures that all requesting Processes may access the Resource safely.

Once a Process has completed accessing the Resource, it asks the Resource Lock to release its access to the Resource. The Resource Lock may resume other conflicting requests at this time. When a Process is waiting for access, it is suspended. A Process may specify the maximum amount of time which it is willing to wait for access. The system may also specify the maximum time that any Process may hold access to a Resource. In the event of a waiting timeout of a holding timeout, the Process is notified of the error.

A process may request access to a Resource it already has access to, but it must release access for each time that it has been granted. The Process will not loose access to the Resource until all granted access has been released. Any combination of Shared or Exclusive access can be held concurrently by a Process.

UC CC1: ***Serialize access to a resource***

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 conversion policy (see CC 4) to the request.

2b. Resource Lock finds that the resource is already in use:
    2b1. The Resource Lock applies a compatibility policy (see CC 2) 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 a selection policy (see CC 3) to grant access to any suspended Service Clients.

5b. Holding Timer is still running:
5b1. Resource Lock cancels Holding Timer.

Variations:
1. The specified requested access can be:
- For Exclusive access
- For Shared access

2c. The Lock holding timeout can be specified by:
- The Service Client
- A Resource Locking Policy
- A global default value.

UC CC 2 Apply access compatibility policy
Used By: CC 1 Resource Locking
Main Success Scenario
1) Resource Lock checks that request is for shared access.
2) Resource Lock checks that all current usage of resource is for shared access.

Extensions

2a. Resource Lock finds that the request is for exclusive access.
2b1. Resource Lock makes Service Client wait for resource access (see CC 5)
(the process is resumed later by the CC 3 Lock serving strategy

3c. Resource Lock finds that the resource is being exclusively used:
2c1. Resource Lock makes Service Client wait for resource access (see CC 5)
Variations:
1) The compatibility criterion may be changed.

*UC CC 3: **Apply access selection policy***
Used By: CC1 Resource Locking
When: Resource Lock must determine which (if any) waiting requests should be served
Note: This strategy is a point of variability.

Main Success Scenario
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.

UC CC 4: ***Apply a lock conversion policy***
Used By: CC 1 Resource Locking

Main Success Scenario
1) Resource Lock checks that request is for exclusive access.
2) Resource Lock checks that Service Client already has shared access.
3) Resource Lock checks 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.
3a. Signal Service Client that requested access could not be granted.
3b. Fail!

4a. Resource Lock finds that there are other Service Clients using the resource.

4a1. Make Service Client wait for resource access (see CC 5)

*UC CC 5: **Make Service Client wait for resource access.***

Used By: CC 2,4 Resource Locking

Main Success Scenario

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 timeout has been specified:
1a1. Resource Lock starts timer

2a Waiting Timer expires:
2a1. Signal Service Client that requested access could not be granted.
2a2. Fail!

Variations:

The Lock waiting timeout can be specified by

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

Process Matters

1. Work in Pairs

There are three basic ways to write use cases: alone, in pairs, or in large groups with a facilitator. I have found the following to be effective:

1. Start in a group. Brainstorm the primary actors, brainstorm their goals. Review the actor-goal list. Write one "system-in-use" user story together, so everyone gets the feel of the activity.
2. Break into pairs. Have two people at a time draft the use cases, sometimes sitting together, sometimes alone, as they choose. The two people can check the correctness of the main success scenario, brainstorm the failure and alternate conditions, and work out the failure handling steps. It is useful to have two people, since they will have many questions about the rules of the business, what is really the requirement versus what is merely a holdover characteristic from the old days. They can double check that the GUI has not crept in, and that the goals are at the right levels.
3. Circulate the papers. One large team, surprisingly, decided to print out and circulate the use cases on paper. I learned why: on paper, the use case writers gather everyone's comments at one time, and then make one editing pass through the use case, merging all the suggestions. When they had tried to deal with online suggestions, they found they were doing much more revising, sometime making a change based on one person's suggestion, then taking that change out to meet another person's suggestion. Whether you do it all online, or by paper, find a peer group to check the level of writing and the business rules inside the use cases.
4. Review with a designer and a usage expert. Include a technical person to be sure that the use case contains enough detail for them to implement from (excluding the data descriptions and UI design). Include a usage expert to make sure all the requirements are true requirements and that the business really works that way.
5. Do a group review. Include in this group all of: software designers, business experts, usage experts, and UI designers. After this review, the use case will probably be in shape for design to start.
6. Update the use case after the group review. The use case has now hit its baseline point, design can probably start, and changes should be made only when mistakes are found.

2. An experience of gathering use cases

The following is the text of an article by Andy Kraus, published in Object Magazine, May, 1996 (copyright Object Magazine, permission to reproduce as long as reference is given, still needs to be obtained).

"...we facilitated Use case sessions with an average of fifteen to twenty participants to elicit the "true" requirements.

Requirements had probably changed since the RFP had been written -- Strong as the RFP had been as a vehicle for communicating known requirements to us, some of us believed that it was imperative for us to make our first contract activity "validating" the REP requirements, amending

them where appropriate, with the client's assistance and concurrence. Work had started on collecting and recording their original requirements nearly two years prior to project start. It had been nearly eight months since those requirements had been frozen into their final RFP form. We believed that there was a strong possibility that the requirements had changed during that time period, perhaps significantly.

The RFP did not specify an object-oriented implementation --

There were no Use cases in the RFP -- and Use cases were and continue to be the major thread connecting all the phases of our object-oriented development process, from requirements through acceptance testing. We had been convinced that Use case modeling was a proven analysis technique for eliciting, understanding and defining functional system requirements. It was our belief that Use cases would tree the users to focus on the what, not the how of their requirements.

Use cases did free the users to focus on the what not the how of the requirements. According to Chris, a Crime Analyst with the department, use cases actually describe how the system is used, not how the physical screens look. Further, the use case sessions were designed to focus on high level descriptions of uses rather than creating a specific description for every single use (see, Jacobson, I, “Use cases in large-scale Systems”, ROAD, 1(6), 1995). Since these discussions were kept)separate from physical screen layout discussions, disagreements over trivial issues were kept to a minimum and allowed the participants to focus on the important issues within their expertise. It was surprising how the use case sessions facilitated the generation of many new ideas that will ultimately make the system much better.

Be prepared to get different requirements when using use cases. According to Nancy, the Administrator, "Use cases provided a way to document and capture the end users needs and gave the user the opportunity to truly have input as to how the system should look and feel. Using this process, however, may change the original scope of the project which requires the need to rethink some issues “

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. Our sessions were held in two conference rooms with roughly equivalent layouts. Session participants were seated around tables arranged in an elongated "u" shape open to the front of the room where the facilitators stood near a large white board, a projection screen and two flip chart stands. The session scribe was seated near the back of the room on one side of the "U" equipped with a laptop computer as well as a tape recorder. 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, Getting the right people into the sessions proved challenging. 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 to be used on the engagement as well as object-oriented techniques and facilitation to the sessions. He developed and conducted the orientation session and training in eliciting Use cases in addition to leading many of the sessions.

A "SME" (Subject Matter Expert) provides the domain expertise to jump start the process and keep it on track once it's moving. -- This person brought over twenty years experience as an Officer with over ten years of eliciting related I/T work from requirements gathering to development to project management to the sessions. His practical experience proved invaluable in time saved as well as the addition of a different perspective to the proceedings.

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 (County) Department of Information Services, participated in the sessions. They provided insights into the emerging requirements, particularly with respect to the external system 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. Scribes should not involve themselves in the actual session interaction, nor should they editorialize. Except where necessary to clarify information being captured, scribes should be silent partners during sessions. We had several scribes working the early sessions who proved invaluable in capturing the emerging Use cases as well as significant talking points from the sessions.

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.

If you're going to create a large number of Use cases you need automation beyond simple word processing. We used a word processor and form for our initial storage and manipulation of the Use cases. As with the flip charts, we realized about fifty Use cases into the process that we needed a better and smarter tool. Unfortunately our schedule and resource constraints prevented us from developing such a tool in time to use in the sessions. we couldn't find a commercial Use case tool that did what we wanted, so we built our own. We developed a Use case tool based in a relational database, front-ended by a 4GL app builder. This tool provided us powerful textual search capability as well as easy update and reporting. we plan on extending the tool to generate matrices to assist us in managing our project: 1) Use cases x Objects, 2) Use case x Ul Part and 3) Use cases x Requirement. A further extension to the tool will be the ability to use it in a "workgroup" mode.

A job title may not an "actor" make. lt 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 secondary actors, e.g., supervisor, in the initial formulation of actors. In our sessions, no "supervisor" actor was identified. 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 et torts 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 us. Case brainstorming. After hitting a serious case of writer's block on the second day of the sessions we had a group of frustrated users on our hands as well as some scared managers. We met long into the night to find a way to clear the day's hurdle. Finally it was determined that 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 stop 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 to get "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. In our experience, 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, Limits, System useponsibilities, Success and Failure Use cases, help facilitate the process (see below).

By the end of the third week we found ourselves continually referring to previously built use cases while constructing new ones. In response to the redundancy we began to define and evolve what we called "Standard Descriptors". The descriptors held prescribed arrangements of object attributes for the new system divided along class lines, e.g., People, Places, Locations. The descriptor sets provided a pathway to a consistent presentation of information for both query arguments and returns. 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. In the absence of standards based Use cases, we chose to build our Use case template to build our Use case template on one recommended to us by a consultant (Alistair Cockburn), who had validated his template at a Workshop on Object Oriented Design (WOOD). The following items were part of our template:

1. Number
2. Name
3. Actors
4. Goal
5. Inputs
6. Outputs
7. Primary Success Scenario
8. Alternate Scenarios
9. System Responsibilities
10. Event Trigger (for non-human actor Scenarios)
11. Maps to Requirement in RFP on Page#, Paragraph#"

We found this Scenarios template worked for us, and hope something similar will work for you. “

Five Project Standards

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 *Forces affecting use case writing styles*. Here, finally, I separate out five particular situations and nominated standards. They are:

- Doing business process modeling
- Eliciting requirements, even if use cases will not be the final form
- Drafting / sizing system requirements
- Writing functional requirements on a short, high-pressure project
- 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. Simply selecting between the 5 choices may clarify your group's position.

To complete this section, I need to stop being generic. Writing the scope as "corporate", "operations", and "system" does you a disservice, since that isn't what you write in the scope field. You write the name of the organizations. Let us say, that our organization is MyInsCo. When it becomes relevant to talk about software systems, I shall discuss the new system, Acura, being introduced to augment and eventually replace the old system, BSSO.

The business process modeling project

Your ultimate purpose may be to design the business itself. Or, it may be to describe the new business process that will be installed along with the new software system. In either case, the readers will be experienced line staff, department managers, and senior executives. The use cases must be easily read. The detail given in the descriptions of the data passed from actor to actor is less important than the sequencing, and the fact that certain steps are mentioned at all. Failure handing is important, since it reveals important business operations rules.

- The System under Discussion is the organization itself, how it handles external triggers and internal responsibilities.
- "MyInsCo" and "MyInsCo operations" are the only two design scopes likely to be used.
  -  "MyInsCo", i.e., the organization itself as a black box, is used for the top-level and for the outermost use cases. These use cases show the organization, as a black box, responding to external stimuli and interacting with external actors. They will be used either as a "spec" against which the business process must be designed, or as introductory use cases, setting the context for the later, white-box use cases.
  -  "MyInsCo Operations" is used to show the organization's design in action, to show the parts of MyInsCo - people or departments, or occasionally, computer systems - working together to deliver the organization's responses. These use cases have a System under Discussion which encompasses all of the operations, and are written as "white-box" use cases.
- The primary actor is almost always a person (or another organization).
- "Strategic" and "user-goal" are the only two goal levels likely to be used.
  -  A "strategic" level use case is one that involves multiple wait periods, and hence multiple user-goal use cases. It is possible bu unlikely that "subfunction" level goals will be used in business modeling.
  -  A "user-goal" level use case is one that happens while the primary actor waits.
-  The final use case wil be a fully dressed use case, either in full paragraphs or in numbered steps. I show the numbered form in this standard, but the paragraph form may be substituted. During early sketching activities or during elicitation, the "casual" use case form may be used, but that will not survive into the final deliverable.

The applicable icon sets are:

 = strategic, black box

 = strategic, white box (corporate operations)

 = user-goal, white box (corporate operations)

Here is the template presented using simple text. Your form may differ depending on the tool you are using. Where I write MyInsCo, you will write the organization's real name. I show strategic, black box icons for the example.

Use Case: Accomplish whatever

Scope: <MyInsCo / MyInsCo Operations>
Level: <Strategic / User-goal > (or white / blue, if you prefer)
Context:
Primary actor:
Stakeholders & Interests:
Success End Condition:
Failure protection:
Preconditions:
Triggers:
Main Success Scenario:
1. ...
2. ...
3. ...
Extensions:
1a. ...
1a1. ...
Frequency of occurrence:
Open Issues: ...

Drafting / sizing system requirements

Your purpose here is to sketch out the system requirements so that you can estimate the size and shape of the system, and save them as reminders for later. Later, you might elaborate them into fully dressed use case requirements, or you might choose to design directly from the draft, casual use cases.

The System under Discussion is whatever it is - organization, or department, or computer system, etc. Therefore, any of the design scopes might be appropriate, any kind of primary actor might be encountered

For speed, you will use the casual form of template, even though it leaves out some information, and your writers will leave out more.

Here is the casual template presented using simple text. Your form may differ depending on the tool you are using. I show system-scope, black box, user-goal icons for the example.

Use Case: Accomplish whatever

Scope: whichever **Level:** whichever

Context: put here preconditions or conditions of normal usage

Primary actor: whomever

Put here a paragraph of texts describing the actors achieving success in the main success scenario ...

Put here another paragraph of text mentioning some of the alternate paths and the handling ...

Frequency of occurrence:

Open Issues: ...

Functional requirements for a short, high-pressure project

Your purpose here is to get the requirements, but since the project is short and under heavy time pressure, you prefer to avoid the overhead of numbers. Unlike drafting the requirements, however, you want additional information recorded.

Here is the basic template.

Use Case: **Accomplish whatever**

Scope:  Acura **Level:**  User goal

Context:

Primary actor:

Stakeholders & Interests:

Success End Condition:

Failure protection:

Preconditions:

Triggers:

Main Success Scenario:

... A paragraph of texts 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: ...

The requirements elicitation task

Your ultimate purpose is to *discover* the requirements. You may or may not eventually write the requirements in use case form. The game, therefore, is to move quickly through the use cases, drafting them in a lively work session.

You want to retain the stakeholders and interests, to help remind the people about the hidden requirements, but you don't worry so much about other aspects of requirements, since they may get recast into another format.

Use Case: Accomplish whatever 

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

Detailed functional requirements, longer project, start of increment

Your purpose here is to seriously collect the requirements, making use of all of the features of use cases. The System under Design may be anything, and the actos and goals, similarly anything. You will use the fully dressed form. I show the system scope, user-goal icons in this example.

Use Case Name: Accomplish whatever

Scope:
Level:
Context:
Primary actor:
Stakeholders & Interests:
Success End Condition:
Failure protection:
Preconditions:
Triggers:
Main Success Scenario:
1. ...
2. ...
3. ...

Extensions:
1a. ...
1a1. ...

Frequency of occurrence:
Open Issues: ...

Other Voices

1. Susan Lilly, RSA, inc.

SusanLilly 2-page intro to ucs
separate ucs by security/entitlement cf CRUD (get exampe)
susan likes dgms, alistair doesn't
know and write to your audience

2. Russell Walters, Firepond

Subj:Use case samples from Firepond

Date: 7/30/99 2:39:51 PM Mountain Daylight Time
From: russellw@firepond.com (Walters, Russell)
To: ACockburn@aol.com (ACockburn@aol. com (E-mail))

Alistair,

I'm really glad to hear you have decided to write a book on use cases. There are so many issues, nuances, styles, and general confusion in the industry.

Just the other day our group had a long discussion about the work by Larry Constantine and Lucy Lockwood on essential use cases in their book "Software for Use". We came to the conclusion that Larry speaks of concrete uses cases that are written very action oriented and specific or implied about the user interface. There idea with the essential form is to avoid any mention of user interface and to get away from actions, but concentrate on intentions. However, it seems their essential form is maybe a little too abstract. I think the goal oriented and user intention form that you teach, void any user interface speak, is the proper level of abstraction.

I hope you plan on addressing business use cases in your book, and how they can relate to system level use cases. I've recently been doing research on business process engineering (BPR), and outside of the work by Ivar, nobody seems to like using business use cases, but rather opt for workflow/activity diagrams to model business processes. To pass along another lesson learned: Team members as well as our intended audience easily got confused about the different scopes of a use case (business vs system)when simply reading the narratives. In fact, some use case descriptions at the system level I found making reference to use cases at the business scope. It wasn't until I visually drew the hierarchical relationships in a diagram, as well as the UML use case diagram did the levels really become apparent. Even though you don't necessarily use the UML use case diagrams, it would be beneficial to show some sample use cases in Cockburn style, and how they would be properly modeled with the UML use case relationships (generalization, extends, includes). Then again, I'm not even sure the inventors know how to properly use these relationships in a real system. ;-) And, please, what every you do, don't show another ATM example!

Sorry about the ATM examples, Rusty ;-)

Email discussions

This section is still undergoing work... do I really need it, given the length of the book?
... currently in a separate file for easier trimming ...
emails.doc

oopsla uc questions

1. how many is enough?
   AA: How many rooms belong in a building?
   UC = user task; how many of those?
2. is it like functional decomp?
   AA: to me, yes, to others, no,
   but shows failure conditions at each level, unlike fn decom
3. estimate using ucs?
   AA: need also algorithmic complexity, technology infrastructure
   and capability of development staff to estimate
   but...20 programmer years did 160 task-level ucs
   so... 8 uc per programmer year, or 1 uc per 6 programmer weeks.
4. non-functional requirements?
   AA: need to deal with them also, separately
5. what if the system already exists?
   AA: can use ucs to describe system succinctly
6. how do you package or cluster if there are many many?
   cluster by major topic (invoicing, payroll, advertising)
7. are we contributing to global warming with uc discussions?
   AA: certainly.

Related work on use cases

This section will contain thumbnails of other people's work, and references to web sites.
Ivar Jacobson
Don Firesmith
Susan Lilly
Constantine and Lockwood
Schneider & co
Larman
Collins-Cope: http://www.ratio.co.uk/rsi.htm markcc@ratio.co.uk (Mark Collins-Cope)

Glossary

Main terms

Use case. A use case is the statement of the goal the primary actor has toward the system's declared responsibilities, and the collection of possible scenarios between the system under discussion and various actors, showing how the primary actor's goal might be delivered or might fail.

Scenario / Concrete scenario / Generified scenario / Path / Course.

A scenario is a sequence of interactions that happens under certain conditions, with the intent to achieve the primary actor's goal, and having a particular result with respect to that goal. Normally spoken, a scenario is phrased in generic terms, using placeholders for the identity of the primary actor and the actual values passed around.

A concrete scenario is a scenario in which every actor involved is given a specific name, and actual values are named. It is equivalent to describing a true story in the past tense, with all details name.

A generified scenario is the official term for a scenario which has placeholders or generic terms for the actors and values passed.

A path is a generifies scenario or scenario fragment within a use case.

A course is the same as a path.

System-in-use story or User Story. A system-in-use story or user story is a concrete scenario that also shows the motivations and intentions of the various actors, used as a light communications device or warm-up to the use case writing.

Actor. Something with behavior.

External actor. An actor outside the system under design.

Internal actor. The system under discussion, a subsystem of that, or an individual person or object inside it.

Primary actor. The external actor with the goal the system is supposed to satisfy as one of its services.

Secondary actor. An external actor against which the system under design has a goal.

SuD. The system under discussion, or system under design.

Interaction. A compound or recursive structure containing individual messages, sequences of interactions or sets of sequences of interactions between two actors.

Stakeholder and Interest. Any one or thing that has an accepted vested interest in the behavior of the system. The interest is what they care the system provides or protects.

Extension (to main success scenario). An extension to a scenario is a scenario fragment that starts upon a particular condition and rejoins the main scenario or terminates in failure. Just as an extension use case, a scenario extension names the point in the main scenario where it picks up.

Types of use cases

Briefs. A use case brief is a one-paragraph synopsis of the use case.

Casual. 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.

Fully Dressed. A fully dressed use case uses one of the accepted full templates, identifying actors, scope, level, trigger condition, precondition, and all the rest of the template header information, plus the project annotation information.

Business. A business use case puts the emphasis on the operation of the business rather than the operation of the computer system. It is likely to have the entire business as its scope (see Corporate Use Case). It may be written black-box or white-box style.

System. A system use case puts the emphasis on the (typically) computer system that is being designed. It is likely to have the computer system as its scope, but there is likely to be one use case that takes the entire corporation as its design scope, but highlights in it the aspects of the behavior that reflect on the computer system's behavior.

Corporate. A use case having the corporation as its design scope.

Computer system. A use case having the computer system as its design scope. Normally, the computer system under discussion is named explicitly.

Innards. The parts or components of the system under design.

White-box. A use case that shows the innards of the system under discussion, showing how the parts cooperate to deliver the system's services. Typically used in business process modeling.

Black-box. A use case that does not reveal the innard of the system under discussion. No internal parts may be mentioned in a black-box use case. Typically used in the system requirements document.

Strategic. A use case taking multiple user-goal sessions to complete, often on the order of weeks, months or years.

User-goal. A use case that satisfies a particular and immediate goal of the primary actor, a goal of value to them. Typically is performed by one actor, in one sitting, 2-20 minutes (could be less, if the primary actor is a computer), after which they can leave and proceed with other things.

Subfunction. A use case whose goal level is below user goal. Is a sub-task within a user-goal use case or subfunction.

Other terms

Course. A full sequence of behavior within a use case.

Main course. The main success scenario.

Alternate course. Any course other than the main success scenario.

Calling use case. The higher-level use case that includes or names this use case. In UML, called the "including" use case.

Sub-use case. The use case named by a calling use case. In UML, the "included" use case.

Superordinate use case - subordinate use case. Unofficial UML relations, named but not properly defined in the UML definition documents. The superordinate use case is a use case for a containing system. The corresponding subordinate use cases are the use cases for each of the (possibly unnamed) components of that containing system, such that their composite, summed behaviors produce the equivalent result as the superordinate use case on the containing system.

Sequence of steps. A scenario fragment.

Use case diagram. In UML the diagram showing the external actors, the system boundarry, the use cases as ellipses, and arrows connecting the actor to ellipses or ellipses to ellipses. Primarily useful as a context diagram and table of contents into the use cases.

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 space, and interactions are shown as arrows between actors. Time is shown only by numbering the arrows.

Goal. What the primary actor wants from the system under discussion.

Extension condition. The circumstances under which a different set of behavior will occur. Most extension conditions name things that go wrong during the main success scenario, but some name alternative successful choices of behavior.

Step. A unit of writing in a use case. Typically one sentence, usually describes behavior of only one actor.

Where do I find answers on ?

	Base Concepts	Related Concepts	Reminders	other
actor, primary actor, job titles	Actors & Stakeholders		Actors, Roles and Goals / Job Titles First and Last	
design scope	Design Scope		Corporate Scope, System Scope	
goal levels	Goal Levels		User Goals and Level Confusion	*Mistakes Fixed*: Raise the Goal Level
transfering into design		Mapping to UI etc.		
transfering into GUI design		Mapping to UI etc.		*Mistakes Fixed*: Getting the GUI out
business use cases	Design Scope	Business Process Modeling with Use Cases	Corporate Scope, System Scope	
estimation		Estimating, Planning, Tracking	Planning around Goals	
UML's use cases		UML & the relations etc.		
the writing style and template we should use	Overall Writing Style	Forces Affecting Use Case Writing Styles	A Bad Use Case etc. / Just One Sentence Style / Clear Intentions, etc. / Who's Got the Ball? / Use Includes / Core Use Case Values etc.	*Preliminaries*: What does a Use Case Look Like? / *Five Project Standards*
test cases from use cases		Mapping to UI, Design Tasks, Design and Test		
managing our energy and timing of writing	The Actor List		Breadth-First, Low Precision	*Preliminaries:* Precision, Hardness, Tolerance
the detailed requirements		The Missing Requirements	Just Chapter Two	
process			Recipe for Writing etc.	*One-Page Summary / Process Matters*
"extends"		UML & the Relations, etc.	Use Includes	
tools		Tools	The Great Tool Debate	
transfering into text from		UML & the Relations, etc.	The Great Drawing Hoax	

drawings				
"roles" vs. "job titles"			Actors, Roles and Goals / Job Titles First and Last	