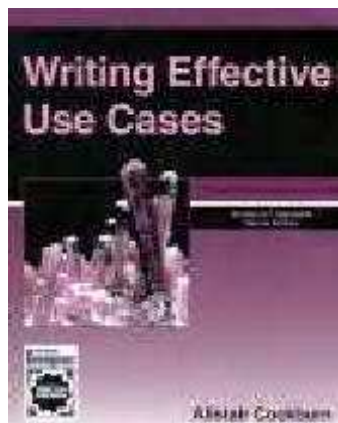# WRITING

EFFECTIVE USE CASES

Alistair Cockburn
Humans and Technology

pre-publication draft #3, edit date: 2000.02.21

published by Addison-Wesley, c. 2001.

Reminders

Write something readable.
Casual,readable use cases are still useful, whereas unreadable use cases won't get read.

Work breadth-first, from lower precision to higher precision.
Precision Level 1: Primary actor's name and goal
Precision Level 2: The use case brief, or the main success scenario
Precision Level 3: The extension conditions
Precision Level 4: The extension handling steps

For each step:
  Show a goal succeeding.
  Highlight the actor's intention, not the user interface details.
  Have an actor pass information, validate a condition, or update state.
  Write between-step commentary to indicate step sequencing (or lack of).
  Ask 'why' to find a next-higher level goal.

For data descriptions:
  Only put precision level 1 into the use case text.
  Precision Level 1: Data nickname
  Precision Level 2: Data fields associated with the nickname
  Precision Level 3: Field types, lengths and validations

Icons

Design Scope	Goal Level
Organization (black-box)	Very high summary
Organization (white-box)	Summary
System (black box)	User-goal
System (white box)	Subfunction
Component	too low

For Goal Level, alternatively, append one of these characters to the use case name:
  Append "+" to summary use case names.
  Append "!" or nothing to user-goal use case names.
  Append "-" to subfunction use case names.

The Writing Process

1. Name the system scope and boundaries.

   Track changes to this initial context diagram with the in/out list.
2. Brainstorm and list the primary actors.

   Find every human and non-human primary actor, over the life of the system.
3. Brainstorm and exhaustively list user goals for the system.

   The initial Actor-Goal List is now available.
4. Capture the outermost summary use cases to see who really cares.

   Check for an outermost use case for each primary actor.
5. Reconsider and revise the summary use cases. Add, subtract, or merge goals.

   Double-check for time-based triggers and other events at the system boundary.
6. Select one use case to expand.

   Consider writing a narrative to learn the material.
7. Capture stakeholders and interests, preconditions and guarantees.

   The system will ensure the preconditions and guarantee the interests.
8. Write the main success scenario (MSS).

   Use 3 to 9 steps to meet all interests and guarantees.
9. Brainstorm and exhaustively list the extension conditions.

   Include all that the system can detect and must handle.
10. Write the extension-handling steps.

    Each will ends back in the MSS, at a separate success exit, or in failure.
11. Extract complex flows to sub use cases; merge trivial sub use cases.

    Extracting a sub use case is easy, but it adds cost to the project.
12. Readjust the set: add, subtract, merge, as needed.

    Check for readability, completeness, and meeting stakeholders' interests.

PREFACE

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

Faced with writing, one suddenly comes face to face with the question, "Exactly what am I supposed to write - how much, how little, what details?" That turns out to be a difficult question to answer. The problem is that writing use cases is fundamentally an exercise in writing prose essays, with all the difficulties in articulating *good* that comes with prose writing in general. It is hard enough to say what a good use case looks like, but we really want to know something harder: how to write them so they will come out being good.

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

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

Audience

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

Writing coaches should find suitable explanations and samples to show their teams.

Course designers should be able to build courses around the book, issuing reading assignments as needed. However, as I include answers to many exercises, they will have to construct their own exam material :-).

Organization

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

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

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

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

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

There are four appendices: Appendix A discusses "Use Cases in UML", Appendix B contains "Answers to (Some) Exercises", Appendix C has a "Glossary" and Appendix D points to other "Readings" that may be of interest.

Heritage of the ideas in this book

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

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

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

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

The samples used

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

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

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

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

Crystal works from two basic principles:

• Software development is a cooperative game of group invention and communication. Software development improves as we improve people's personal skills and improve the team's collaboration effectiveness.

• Different projects have different needs. Systems have different characteristics, and are built by teams of differing sizes, containing people having differing values and priorities. It cannot be possible to describe the one, best way of producing software.

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

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

Acknowledgements

Thanks to lots of people. Thanks to the people who reviewed this book in draft form and asked for clarification on topics that were causing their clients, colleagues and students confusion. Special thanks to Russell Walters for his encouragement and very specific feedback, as a practiced person with a sharp eye for the direct and practical needs of the team. Thanks to Firepond and Fireman's Fund Insurance Company for the live use case samples. Pete McBreen was the first to try out the Stakeholders & Interests model, and added his usual common sense, practiced eye, and suggestions for improvement. Thanks to the Silicon Valley Patterns Group for their careful reading on early

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

PART 1
The Use Case Body Parts

PART 4
End Notes

1. INTRODUCTION TO USE CASES

What do use cases look like?

Why would different project teams need different writing styles?

Where do they fit into the requirements gathering work?

How do we warm up for writing use cases?

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

1.1 What is a Use Case (more or less)?

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

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

The use case, as a form of writing, can be put into service to stimulate discussion within a team about an upcoming system. They might later use that the use case form to document the actual requirements. Another team might later document the final design with the same use case form. They might do this for a system as large as an entire company, or as small as a piece of a software application program. What is interesting is that the same basic rules of writing apply to all these different situations, even though the people will write with different amounts of rigor, at different levels of technical detail.

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

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

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

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

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

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

Here are the first two use cases.

USE CASE 1: BUY STOCKS OVER THE WEB

Primary Actor: Purchaser
Scope: Personal Advisors / Finance package ("PAF")
Level: User goal
Stakeholders and Interests:
Purchaser - wants to buy stocks, get them added to the PAF portfolio automatically.
Stock agency - wants full purchase information.
Precondition: User already has PAF open.
Minimal guarantee: sufficient logging information that PAF can detect that something went wrong and can ask the user to provide details.
Success guarantee: remote web site has acknowledged the purchase, the logs and the user's portfolio are updated.
Main success scenario:
1. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwabb, etc.) from user.
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site, and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.
Extensions:
2a. User wants a web site PAF does not support:
    2a1. System gets new suggestion from user, with option to cancel use case.
3a. Web failure of any sort during setup:
    3a1. System reports failure to user with advice, backs up to previous step.
    3a2. User either backs out of this use case, or tries again.
4a. Computer crashes or gets switched off during purchase transaction:
    4a1. (what do we do here?)
4b. Web site does not acknowledge purchase, but puts it on delay:
    4b1. PAF logs the delay, sets a timer to ask the user about the outcome.
    4b2. (see use case *Update questioned purchase*)
5a. Web site does not return the needed information from the purchase:
    5a1. PAF logs the lack of information, has the user *Update questioned purchase*.

USE CASE 2: 🏠 GET PAID FOR CAR ACCIDENT ✎

Primary Actor: The Claimant
Scope: The insurance company ("MyInsCo")
Level: Summary
Stakeholders and Interests:
  the claimant - to get paid the most possible
  MyInsCo - to pay the smallest appropriate amount
  the dept. of insurance - to see that all guidelines are followed.
Precondition: none
Minimal guarantees: MyInsCo logs the claim and all activities.
Success guarantees: Claimant and MyInsCo agree on amount to be paid, claimant gets paid that.
Trigger: Claimant submits a claim
Main success scenario:
1. Claimant submits claim with substantiating data.
2. Insurance company verifies claimant owns a valid policy
3. Insurance company assigns agent to examine case
4. Insurance company verifies all details are within policy guidelines
5. Insurance company pays claimant and closes file.
Main success scenario:
1a. Submitted data is incomplete:
      1a1. Insurance company requests missing information
      1a2. Claimant supplies missing information
2a. Claimant does not own a valid policy:
      2a1. Insurance company declines claim, notifies claimant, records all this, terminates proceedings.
3a. No agents are available at this time
      3a1. (What does the insurance company do here?)
4a. Accident violates basic policy guidelines:
      4a1. Insurance company declines claim, notifies claimant, records all this, terminates proceedings.
4b. Accident violates some minor policy guidelines:
      4b1. Insurance company begins negotiation with claimant as to degree of payment to be made.

———————————————————————————

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

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

USE CASE 3: ▢ REGISTER ARRIVAL OF A BOX

    RA means "Receiving Agent".
    RO means "Registration Operator"
    <u>Primary Actor</u>: RA
    <u>Scope</u>: Nightime Receiving Registry Software
    <u>Level</u>: user goal
    <u>Main success scenario</u>:
    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.
    <u>Extensions:</u>
    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.
    <u>Variations</u>:
    4'. with and without Person id
    4''. with and without estimated value
    5'. RA leaves bags in box.

1.2 Your use case is not my use case

Use cases are a form of writing that can be put to use in different situations, to describe

* a business' work process,

* to focus discussion *about* upcoming software system requirements, but not be the requirements description,

* to be the functional requirements for a system, or

* to document the design of the system.

* They might be written in a small, close-knit group, or in a formal setting, or in a large or distributed group.

Each situation calls for a slightly different writing style. Here are the major subforms of use cases, driven by their *purpose*. They are further explained in "Use Case Body Parts," but you should become familiar with these notions right away.

- A close-knit group gathering requirements, or a larger group discussing upcoming requirements will write **casual** as opposed to the **fully dressed** use cases written by larger, geographically distributed or formally inclined teams. The casual form "short circuits" the use case template, making the use cases faster to write (see more on this below). All of the use cases shown above are fully dressed, using the full use case template and step numbering scheme. A example of casual form is shown below in Use Case 4:.

- Business process people will write **business** use cases to describe the operations of their business, while a hardware or software development team will write **external, system** use cases for their requirements. The design team may write **internal**, **system** use cases to document their design or to break down the requirements for small subsystems.

- Depending on the level of view needed at the time, the writer will choose to describe a multi-sitting or **summary** goal, a single-sitting or **user goal**, or a part of a user goal, or **subfunction.** Communicating which of these is being described is so important that my students have come up with two different gradients to describe them: by height relative to sea level (above sea level, at sea level, underwater), and by color (white, blue, indigo).

- Anyone writing requirements for a new system to be designed, whether business process or computer system, will write **black-box** use cases - use cases that do not discuss the insides of the system. Business process designers will write **white-box** use cases, showing how the company or organization runs its internal processes. The technical development team might do the same to document the operational context for the system they are about to design, and they might write white-box use cases to document the workings of the system they just designed.

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

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

You may want to test yourself on the use cases in this chapter. Use cases 1, 3, 5 were written for system requirements purposes, so they are fully dressed, black-box, system use cases, at the user-goal level. Use case 4 is the same, but casual instead of fully dressed. Use case 2 was written as the context-setting use case for business process documentation. It is fully dressed in form, it is black-box, and it is a summary-level business use case.

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

- A team is working on software for a large, mission critical project. They decide that extra ceremony is worth the extra cost, that a) the use case template needs to be longer and more detailed, b) the writing team should write very much in the same style, to reduce ambiguity and cost of misunderstanding, c) the reviews should be tighter, to scrutinize the use cases closer for omissions and ambiguities. Having little tolerance for mistakes, they decide to reduce tolerances (variation between people) in the use cases writing also.

- A team of three to five people is building a system whose worst damage is the loss of comfort, easily remedied with a phone call. They consider all the above ceremony a waste of time, energy and money. The team chooses a) a simpler template, b) to tolerate more variation in writing style, c) fewer and more forgiving reviews. The errors and omissions in the writing are to be caught by other project mechanisms, probably conversations among teammates and with the users. They can tolerate more errors in their written communication, and so more casual writing and more variation between people.

Neither is wrong. Those choices must be made on a project-by-project basis. This is the most important lesson that I, as a methodologist, have learned in the last 5 years. Of course we've been saying, "One size doesn't fit all" for years, but just how to translate that into concrete advice has remained a mystery for methodologists.

The mistake is getting too caught up in precision and rigor, when it is not needed. That mistake will cost your project a lot in expended time and energy. As Jim Sawyer wrote in an email discussion,

> "as long as the templates don't feel so formal that you get lost in a recursive descent that worm-holes its way into design space. If that starts to occur, I say strip the little buggers naked and start telling stories and scrawling on napkins."

I have come to the conclusion that it is incorrect to publish just one use case template. There must be at least two, a casual one for low-ceremony projects, and a fully dressed one for higher-ceremony projects. Any one project will adapt one of the two forms for their situation. The next two use cases show the same use case written in the two styles.

USE CASE 4: ⌂ BUY SOMETHING (CASUAL VERSION) ✐

The Requestor initiates a request and sends it to her or his Approver. The Approver checks that there is money in the budget, check the price of the goods, completes the request for submission, and sends it to the Buyer. The Buyer checks the contents of storage, finding best vendor for goods. Authorizer: validate Approver's signature. Buyer: complete request for ordering, initiate PO with Vendor. Vendor: deliver goods to Receiving, get receipt for delivery (out of scope of system under design). Receiver: register delivery, send goods to Requestor. Requestor: mark request delivered.

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

_____

USE CASE 5: ⌂ BUY SOMETHING (FULLY DRESSED VERSION) ✐

Primary Actor: Requestor

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

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

Level: Summary

Stakeholders and Interests:

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

Company: wants to control spending but allow needed purchases.

Vendor: wants to get paid for any goods delivered.

Precondition: none

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

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

Trigger: Requestor decides to buy something.

Main success scenario:

1. Requestor: *initiate a request*

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

3. Buyer: check contents of storage, find best vendor for goods

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

Extensions:
1a. Requestor does not know vendor or price: leave those parts blank and continue.
1b. At any time prior to receiving goods, Requestor can change or cancel the request.
 Canceling it removes it from any active processing. (delete from system?)
 Reducing price leaves it intact in process.
 Raising price sends it back to Approver.
2a. Approver does not know vendor or price: leave blank and let Buyer fill in or call back.
2b. Approver is not Requestor's manager: still ok, as long as approver signs
2c. Approver declines: send back to Requestor for change or deletion
3a. Buyer finds goods in storage: send those up, reduce request by that amount and carry on.
3b. Buyer fills in Vendor and price, which were missing: gets resent to Approver.
4a. Authorizer declines Approver: send back to Requestor and remove from active processing. (what does this mean exactly?)
5a. Request involves multiple Vendors: Buyer generates multiple POs.
5b. Buyer merges multiple requests: same process, but mark PO with the requests being merged.
6a. Vendor does not deliver on time: System does *alert of non-delivery*
7a. Partial delivery: Receiver marks partial delivery on PO and continues
7b. Partial delivery of multiple-request PO: Receiver assigns quantities to requests and continues.
8a. Goods are incorrect or improper quality: Requestor does *refuse delivered goods*. (what does this mean?)
8b. Requestor has quit the company: Buyer checks with Requestor's manager, either *reassign Requestor*, or return goods and *cancel request*.

Technology and Data Variations List: (none)
Priority- various
Releases - several
Response time - various
Freq of use - 3/day
Channel to primary actor: Internet browser, mail system, or equivalent
Secondary Actors: Vendor
Channels to Secondary Actors: fax, phone, car
Open issues:
 When is a canceled request deleted from the system?
 What authorization is needed to cancel a request?
 Who can alter a request's contents?

> What change history must be maintained on requests?
> What happens when Requestor refuses delivered goods?
> How exactly does a Requisition work, differently from an order?
> How does ordering reference and make use of the internal storage?

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

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

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

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

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

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

STEVE ADOLPH: "DISCOVERING" REQUIREMENTS IN NEW TERRITORY

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

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

How would you, as the analyst responsible for workflow management for that brand-new direct-to-plate system, gather requirements for something so totally new?

You could first find the use cases of the existing system, identify the actors and services of the existing system. But that only gives you the existing system. No one has done the new work yet, so all the domain experts are learning the system along with you. You are designing a new process and new software at the same time. Lucky you. How do you find the tracks on this fresh snow? Take the existing model and ask the question, "What changes?" The answer could well be, "Everything."

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

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

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

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

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

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

1.3 Requirements and Use Cases

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

- They really are requirements. You shouldn't have to convert them into some other form of behavioral requirements. Properly written, they accurately detail what the system must do.

- They are not all of the requirements. They don't detail external interfaces, data formats, business rules and complex formulae. They constitute only a fraction (perhaps a third) of all the requirements you need to collect - a very important fraction, but still only a fraction.

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

The following requirements outline is one that I find useful. I adapted it from the template that Suzanne Robertson and the Atlantic Systems Guild published on their web site and in the book, Managing Requirements (Robertson and Robertson, Addison-Wesley, 1999). Their template is fantastically complete (intimidating in its completeness), so I cut it down to the following form, which I use as a guideline. This is still too large for most projects I encounter, and so we tend to cut it down further, as needed. However, it asks many interesting questions that otherwise would not get asked, such as, "what is the human backup to system failure," and "what political consider-ations drive any of the requirements."

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

A Plausible Requirements File Outline

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

Chapter 2. The terms used / Glossary

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

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

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

Chapter 6. Human backup, legal, political, organizational issues
        Q1. What is the human backup to system operation?
        Q2. What legal, what political requirements are there?
        Q3. What are the human consequences of completing this system?
        Q4. What are the training requirements?
        Q5. What assumptions, dependencies are there on the human environment?

_____

   The thing to note is that use cases only occupy chapter 3 of the requirements. They are not all of the requirements. They are *only* but *all of* the behavioral requirements. Business rules, glossary, performance targets, process requirements, and many other things simply do not fall in the category of behavior. They need their own chapters (see Figure 1.).

Use cases as a project linking structure
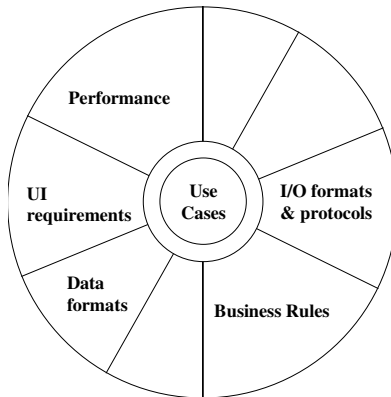
Figure 1. "Hub-and-spoke" model of requirements.

Use cases connect many other requirements details.

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

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

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

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

Requirements File Exercises

Exercise 1   Which sections of the requirements file outline are sensitive to use cases, and which are not? Discuss this with another person and think about why you come up with different responses.

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

1.4 When Use Cases Add Value

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

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

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

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

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

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

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

1.5 Manage Your Energy

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

- Give your stakeholders a chance to offer correction and insight about priorities early, and

- Permit the work to be split across multiple groups, increasing parallelism and productivity.

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

Precision is how much you care to say. When you say, "A 'Customer' will want to rent a video", you are not saying very many words, but you actually communicate a great deal to your readers.

When you show a list of all the goals that your proposed system will support, you have given your stakeholders an enormous amount of information from a small set of words.

Precision is not the same as accuracy. If someone tells you, "? is 4.141592," they are using a lot of precision. They are, however, quite far off, or inaccurate. If they say, "? is about 3," they are not using much precision (there aren't very many digits) but they are accurate for as much as they said. The same ideas hold for use cases.

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

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

1 *Actors & Goals*. List what actors and which of their goals the system will support. Review this list, for accuracy and completeness. Prioritize and assign to teams and releases. You now have the functional requirements to the first level of precision.

2 *Use case brief or main success scenario*. For the use cases you have selected to pursue, write the trigger and sketch the main success scenario. Review these in draft form to make sure that the system really is delivering the interests of the stakeholders you care about. This is the second level of precision on the functional requirements. It is fairly easy material to draft, unlike the next two levels.

3 *Failure conditions*. Complete the main success scenario and brainstorm all the failures that could occur. Draft this list completely before working out how the system must handle them all. Filling in the failure handling takes much more energy than listing the failures. People who start writing the failure handling immediately often run out of energy before listing all the failure conditions.

4 *Failure handling*. Finally, write how the system is supposed to respond to each failure. This is often tricky, tiring and surprising work. It is surprising because, quite often, a question about an obscure business rule will surface during this writing. Or the failure handling will suddenly reveal a new actor or a new goal that needs to be supported.

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

1.6 Warm up with a Usage Narrative

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

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

In this narrative, invent a fictional but specific actor, and capture, briefly, the mental state of that person, why they want what they want or what conditions drive them to act as they do. As with all of use case writing, we need not write much. It is astonishing how much information can be conveyed with just a few words. Capture how the world works, in this particular case, from the start of the situation to the end.

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

Here is an example of a usage narrative.

USAGE NARRATIVE: GETTING "FAST CASH"

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

_____

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

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

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

Usage Narrative Exercises

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

Exercise 4   Write a usage narrative for a person going into a brand new video rental store, interested in renting the original version of "The Parent Trap".

Exercise 5   Write a usage narrative for your current project. Get another person to write a usage narrative for the same situation. Compare notes and discuss. Why are they different, what do you care to do about those differences - is that tolerance in action, or is the difference significant?

PART 1
THE USE CASE
BODY PARTS

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

Learning to write good use cases is harder. The writer has to master three concepts that apply to every sentence in the use case and to the use case as a whole. Odd though it may seem at first glance, keeping those three concepts straight is difficult. The difficulty shows up as soon as you start to write your first use case. These three concepts are:

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

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

While reading, hang onto these summary definitions:

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

2. THE USE CASE AS A CONTRACT FOR BEHAVIOR

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

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

2.1 Interactions between Actors with Goals

Actors have goals

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

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

Figure 2. An actor with a goal calls upon the responsibilities of another.
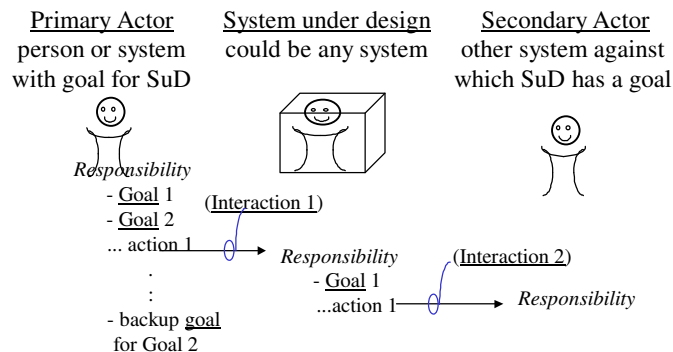
To carry out its job responsibility, the system formulates subgoals. It can carry out some subgoals internally. It needs the help of another, *supporting*, actor to carry out others. This supporting actor may be a printing subsystem or it may be another organization, such as a partner company or government agency.

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

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

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

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

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

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

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

Goals can fail

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

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

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

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

Interactions are compound

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- All the interactions relate to the same goal of the same primary actor.

- The use case starts at the triggering event, and continues until the goal is delivered or abandoned, *and* the system's completes its responsibilities with respect to the interaction

A use case collects scenarios

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

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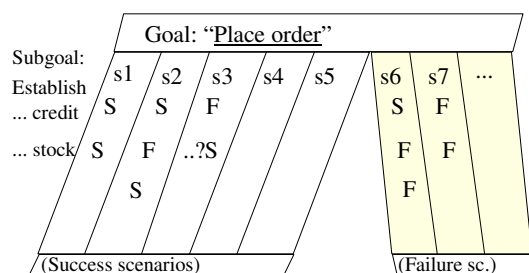
A useful metaphor for this is illustrated in the *striped trousers* image (Figure 3.). The belt on the trousers names the goal that holds all the scenarios together. There are two legs, one for the scenarios that end in success, one for the scenarios that end in failures. Each stripe corresponds to a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

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

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

Figure 4. The striped trousers showing subgoals.

In Figure 4., I add to the striped trousers image to show a sub use case fitting into the use case that names it. A customer who wants to *Place an Order*. One of the customer's subgoals is to *Establish*

Credit. That subgoal is complex, and might succeed or fail: it is a use case we have rolled up into a single step. The step "Customer establishes credit" is the belt on another set of trousers. In the stripe or scenario containing that step, the subgoal either succeeds or not. In scenarios 1 and 2 on the drawing, the subgoal works. In scenarios 3 and 7, the subgoal fails. In scenario 3, however, the next subgoal for establishing credit succeeds, and the scenario ends with success. In scenario 7, the second attempt also fails, and the entire use case ends with failure to place an order.

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

The principles we see from the trousers image are that:

- Some scenarios end with success, some end with goal failure.

- A use case collects together all the scenarios, success and failure.

- Each scenario is a straight description for one set of circumstances with one outcome.

- Use cases contain scenarios (stripes on the trousers), and a scenario contains sub use cases as its steps.
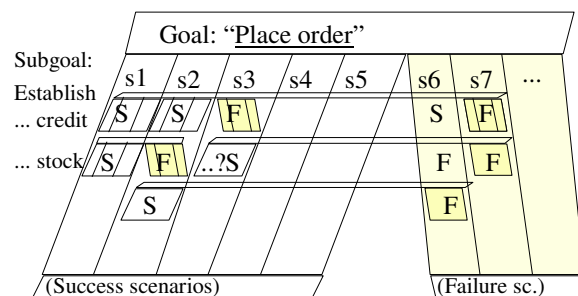
- A step in a scenario does not care which stripe in the sub use case was used, only whether it ended with success or failure.

We shall make use of these principles throughout our writing.

2.2 Contract between Stakeholders with Interests

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

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

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

The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfy the stakeholders' interests.
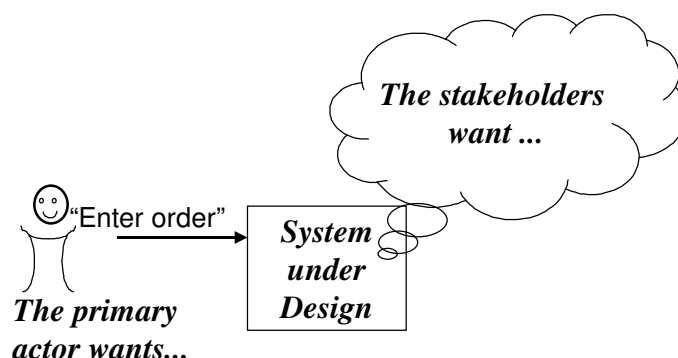
Figure 5. The SuD serves the primary actor, protecting off-stage stakeholders**.**

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

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

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

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

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

2.3 The Graphical Model

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

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

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

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

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

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

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

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

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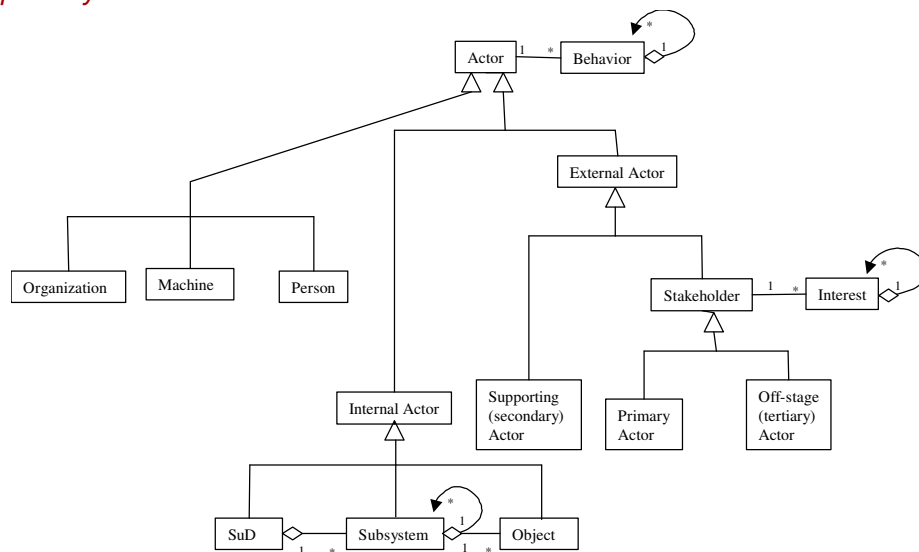
Figure 6. A stakeholder has interests. An actor has behaviors. A primary actor is also a stakeholder.

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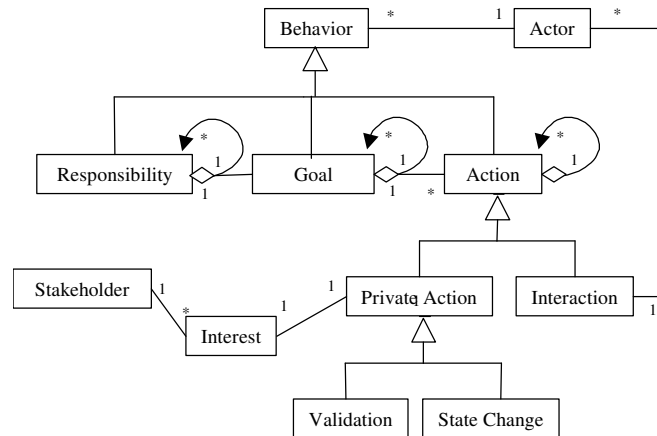
Figure 7. Goal-oriented behavior made of responsibilities, goals and actions.

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

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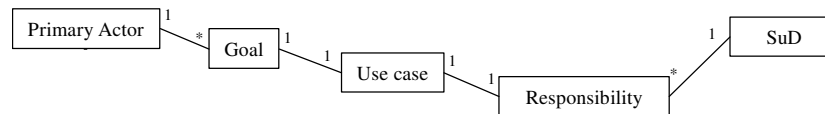
Figure 8. The use case as responsibility invocation. The use case is the primary actor's goal, calling upon the responsibility of the system under design.

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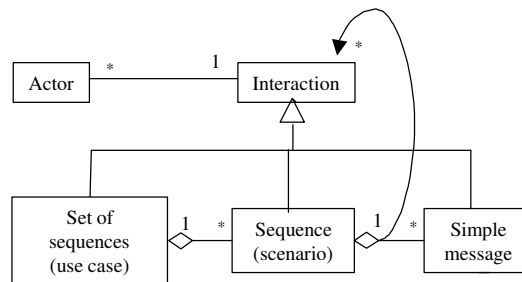
Figure 9. Interactions are composite**.** 'N' actors participate in an interaction. Interactions are composite, decomposing into use cases, scenarios, and simple messages. Once again, the word *sequence* is used as a convenience.

3. SCOPE

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

Keeping track of the scope of the project, or even just the scope of a discussion can be difficult. Rob Thomsett introduced me to a wonderful little tool for tracking and managing scope discussions, the *In/Out List*. It is absurdly simple and remarkably effective. It can be used to control scope discussions for ordinary meetings as well as project requirements.

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

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

A SAMPLE IN/OUT LIST

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

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

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

3.1 Functional scope

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

The Actor-Goal List

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

A SAMPLE ACTOR-GOAL LIST:

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

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

number in which the system will support that goal. You will update this list continually over the course of the project so that it always reflects the status of the system's functional boundary.

Some people add additional columns: *Trigger*, to identify those that will get triggered by time instead of a person; or the three items *Business Priority, Development Complexity, Development Priority*, so they can separate the business needs from the development costs to derive the development priority.

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

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

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

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

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

The Use Case Briefs

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

The *brief* of a use case is a 2-6 sentence description of the use case behavior, mentioning only the most significant activity and failures. It reminds people of what is going on in the use case. It is useful for estimating work complexity. Teams constructing from commercial, off-the-shelf components (COTS) use this description in preparing to select the components. Some projects, those having extremely good internal communications and continual discussion with their users, never write more than these use case briefs for their requirements. They keep the rest of the requirements in the continual discussions, prototypes, and frequently delivered increments.

You can put the use case brief either in a table, as an extension to actor-goal list, or directly into the use case body as its first draft. Here is a sample table of briefs, thanks to Paul Ford and Paul Bouzide of Navigation Technologies.

A SAMPLE OF USE CASE BRIEFS

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

3.2 Design scope

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

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

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

A small, true story

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

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

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

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

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

_____

Figure 10. Design scope can be any size.
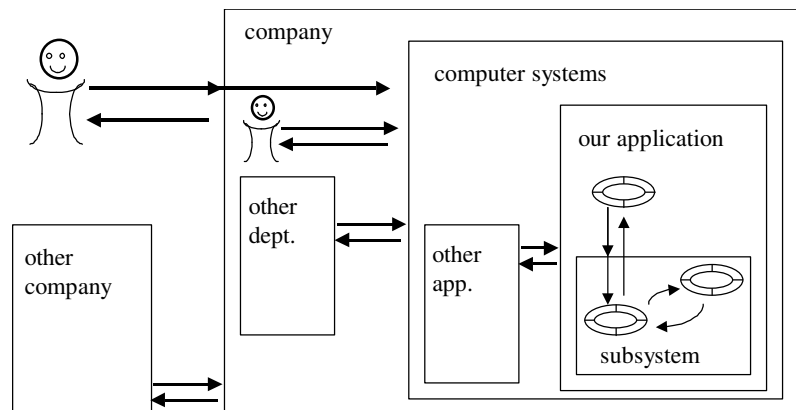
Typically, the writer considers it obvious what the design scope of the system is. It is so obvious that they don't mention it. However, once there are multiple writers and multiple readers, then the design scope of a use case is not at all obvious. One writer is thinking of the entire corporation as the design scope (see Figure 10.), one is thinking of all of the company's software systems, one is

thinking of the new, client-server system, and one is thinking of only the client or only the server. The readers, having no clue as to what is meant, get lost or misunderstand the document.

What can we do to clear up these misunderstandings?

The only answer I have found is to *label each and every use case with its design scope*, using specific names for the most significant design scopes. To be concrete, let us suppose that MyTelCo is designing NewApp system, which includes a Searcher subsystem. The design scope names are:

- "**Enterprise**" scope (put the real name here, e.g. **MyTelCo**) 🏠 signifies that you are discussing the behavior of the entire organization or enterprise in delivering the goal of the primary actor. Label the *scope* field of the use case with the name of the organization, e.g., *MyInsCo*, rather than just writing "the company". If discussing a department, use the department name. Business use cases are written at enterprise scope.

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

- "**Subsystem**" scope (put the subsystem name in here, e.g. **Searcher**) ⬚ means you have opened up the main system and are about to talk about how a piece of it works.

Using graphical icons to highlight the design scope

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

Recall, in the following, that a *black-box* use case does not discuss the internal structure of the system under discussion, while a *white-box* use case does.

- A *business* use case has the enterprise as its design scope. Its graphic is a building. Color it grey 🏠 if you treat the whole enterprise as a black box. Color it white 🏠 if you talk about the departments and staff inside the organization.

- A *system* use case has a computer system as its design scope. Its graphic is a box. Color it grey ▱ if you treat it as a black box, white ▱ if you reveal how its componentry works.

- A *component* use case is about a subsystem or component of the system under design. Its

graphic is a bolt (as in nuts and bolts): . See the use case set *Documenting a Design Framework* for an example of a component use case.

Examples of design scope

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

. . .

3.3 The Outermost Use Cases

In "Enterprise to system scope" on page 50, I recommend writing two use cases, one for the system under design, and one at an outer scope. Now we can get more specific about that.

> For each use case, find the outermost design scope at which it still applies, and write a summary level use case at that scope.

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

Typically, there are only 2-5 outermost use cases for the entire system, so it is not the case that every use case gets written twice. There are so few of them because each outermost use case merges the primary actors having similar goals on the same design scope, and pulls together all the lower level use cases for those actors.

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

Let's visit the outermost use cases for MyTelCo and its Acura system, described a little earlier.

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

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

The customer is clearly outside MyTelCo, and so there is one outermost use case with the customer as primary actor and MyTelCo as scope. This use case will be a summary level use case, showing MyTelCo as a black box, responding to the customer's request, delivering the service, and so on. In fact, the use case is outlined in Use Case 6:"Add New Service (Enterprise)." on page 51.

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

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

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

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

3.4 Using the Scope-Defining Work Products

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

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

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

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

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

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

4. STAKEHOLDERS & ACTORS

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

Look for actors in:

- the *stakeholders* of the system.

- the *primary* actor of a use case;

- the system under design, itself (*SuD*);

- the *supporting* actors of a use case;

- *internal actors,* components within the SuD;

4.1 Stakeholders

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

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

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

A short, true story

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

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

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

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

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

4.2 The primary actor of a use case

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

. . .

4.3 Supporting actors

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

. . .

4.4 The system under discussion, itself

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

. . .

4.5 Internal actors and white-box use cases

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

. . .

5. THREE NAMED GOAL LEVELS

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

> "I want this sales contract. To do that I have to take this manager out to lunch. To do that I have to get some cash. To do that I have to withdraw money from this ATM. To do that I have to get it to accept my identity. To do that I have to get it to read my ATM card. To do that I have to find the card slot."

> "I want to find the tab key so I can get the cursor into the address field, so I can put in my address, so I can get my personal information into this quote software, so I can get a quote, so I can buy a car insurance policy, so I can get my car licensed, so I can drive."

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

Giving names to goal levels helps. The following sections describe the goal level names and icons I have found useful, and finding the goal level you need at the moment. Figure 14.illustrates the names and visual metaphors I use.
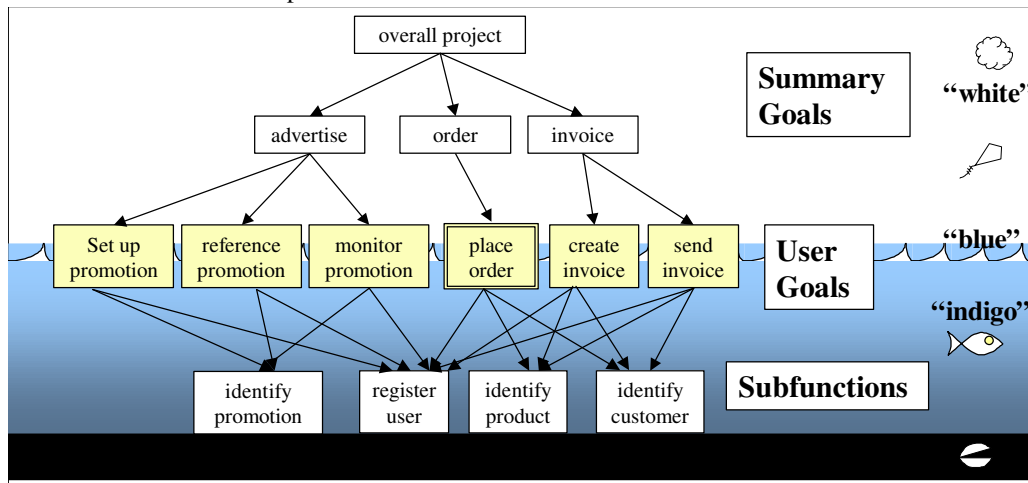
Figure 11. The levels of use cases. The use case set reveals a hierarchy of goals, the *ever-unfolding story*.

5.1 User-goals (blue, sea-level ⌣⌣)

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

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

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

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

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

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

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

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

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

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

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

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

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

A small, true story

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

Two levels of blue

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

. . .

5.2 Summary level (white, cloud◌ / kite✐)

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

- They show the context in which the user goals operate,

- They show life-cycle sequencing of related goals,

- They provide a table of contents for the lower-level use cases, both lower white use cases and blue use cases.

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

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

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

USE CASE 6: 🏠 OPERATE AN INSURANCE POLICY ⬭

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

Other examples of white use case are

* Use Case 19:"Handle Claim (business)" on page 78,

* Use Case 20:"Evaluate Work Comp Claim" on page 79, and

* Use Case 21:"Handle a Claim (systems)" on page 80.

. . .

5.3 Subfunctions (indigo/black, underwater ⤢/clam ⬡)

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

Subfunction use cases are underwater, indigo. Some are *really* underwater, so far underwater that they sit in the mud on the bottom. Those we color black, to mean "this is so low level, please

don't even expand it into a use case" ("It doesn't even swim... it's a clam!"). It is handy to have a special name for these ultra-low-level use cases, so that when someone writes one, you can indicate it shouldn't be written, that its contents ought to rolled into another use case.

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

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

. . .

5.4 Using graphical icons to highlight goal levels

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

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

- Very summary (very white) use cases get a cloud, ⬡. Use this on that rarest of occasions, when you see that the steps in the use case are themselves white goals.

- Summary (white) use cases get a kite, ◹. This is for most summary use cases, whose steps are blue goals.

- User-goal (blue, sea-level) use cases get waves, ∿.

- Subfunction (indigo) use cases get a fish, ⊳◎. Use this for most indigo use cases.

- Some subfunctions (black) should never be written. Use a clam, ⊖, to mark a use case that needs to be merged with its calling use case.

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

5.5 Finding the right goal level

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

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

. . .

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

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

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

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

. . .

6. PRECONDITIONS, TRIGGERS, GUARANTEES

. . .

7. SCENARIOS AND STEPS

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

. . .

8. EXTENSIONS

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

. . .

9. TECHNOLOGY & DATA VARIATIONS

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

   . . .

10. LINKING USE CASES

10.1 Sub use cases

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

> User saves the report

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

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

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

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

10.2 Extension use cases

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

> You are designing a new word processing program, call it *Wapp*. The user's main activity is typing. However, the user might suddenly decide to change the zoom factor or the font size, run the spell checker, or do any of a dozen different of things, not directly connected to typing. In fact, you want the typing activity to remain ignorant of what else might happen to the document.

> Even more importantly, you want different software development teams to come up with new ideas for services, and not force them all to update the base use case for each new service. You want to be able to extend the requirements document without trauma.

The characteristics of such a situation are

* There is a main activity, which can be interrupted.

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

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

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

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

Here are some extracts for the word processor *Wapp*, to illustrate.

. . .

11. USE CASE FORMATS

. . .

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

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

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

USE CASE 7: FULLY DRESSED USE CASE TEMPLATE <NAME>

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

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

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

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

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

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

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

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

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

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

Main Success Scenario:

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

<step #> <action description>

Extensions

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

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

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

Technology and Data Variations List

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

<step or variation # > <list of variations>

<step or variation # > <list of variations>

Related Information

<whatever your project needs for additional information>

. . .
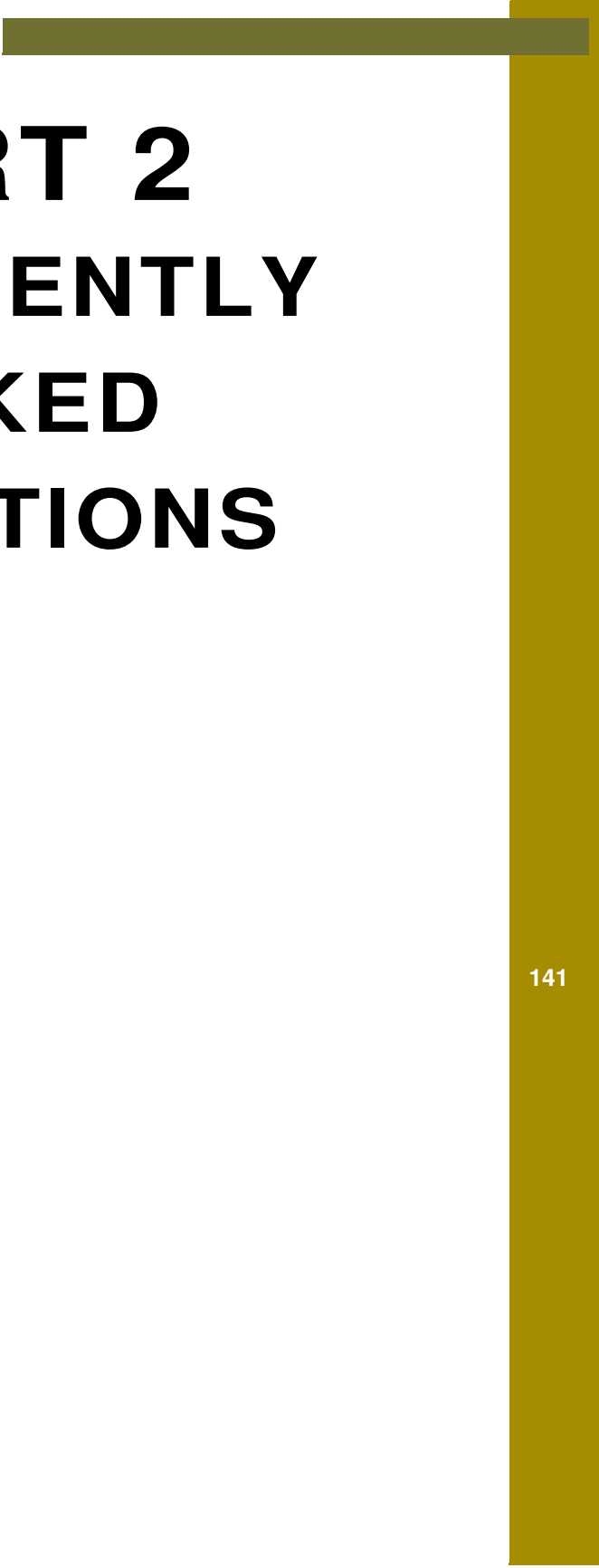
11.1 Conclusion about formats

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

PART 2
FREQUENTLY ASKED QUESTIONS

12. WHEN ARE WE DONE?

You are "done" when

- You have named all the primary actors and all the user goals with respect to the system.

- You can captured every trigger condition to the system either as a use case trigger or an extension condition.

- You have written all the user-goal use cases, along with the summary and subfunction use cases needed to support them.

- Each use case is clearly enough written that

    – the sponsors agree they will be able to tell whether or not it is actually delivered.

    – the users agree that is what they want or can accept as the system's behavior.

    – the developers agree they can actually develop that functionality.

- The sponsors agree that the use case set covers all they want (for now).

. . .

13. SCALING UP TO MANY USE CASES

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

. . .

14. TWO SPECIAL USE CASES

14.1 CRUD use cases

There is not yet a consensus on how to organize all those little use cases of the sort, *Create a Frizzle, Retrieve a Frizzle, Update a Frizzle, Delete a Frizzle*. These are known as CRUD use cases, from the Create, Retrieve, Update, Delete operations on databases. The question is, are they all part of one bigger use case, *Manage Frizzles*, or are they separate use cases?

. . .

14.2 Parameterized use cases

We are occasionally faced with writing a series of use cases that are almost the same. The most common examples are Find a Customer, Find a Product, Find a Promotion, etc. It is probable that just one development team will create the generic searching mechanism, and other teams will make use of that mechanism.

Writing half-a-dozen similar use cases is not much of a problem with casual use cases. However, writing six similar fully dressed use cases is a chore, and it won't take the writers long to want a short cut. I'll describe that short cut using the *Find a Whatever* example first mentioned in Use Case 23:"Find a Whatever (problem statement)" on page 86.

. . .

15. BUSINESS PROCESS MODELING

Everything in this book applies to business processes as well as to software systems design. Any system, including a business, that offers a set of services to outside actors while protecting the interests of the other stakeholders can be described with use cases. In the case of businesses, the readability of use cases is quite helpful.

There are examples of business use cases in other parts of this book, specifically:

* Use Case 2:"Get paid for car accident" on page 18
* Use Case 5:"Buy Something (Fully dressed version)" on page 22
* Use Case 6:"Operate an Insurance Policy" on page 59
* Use Case 19:"Handle Claim (business)" on page 78
* Use Case 20:"Evaluate Work Comp Claim" on page 79

. . .

16. THE MISSING REQUIREMENTS

It is all very well to give the advice, "Write the *intent* of the actor, just use a *nickname* for the data that gets passed," as in "Customer supplies name and address." However, it is clear to every programmer that this is not sufficient to design to. The programmer and the user interface designer need to know what exactly is meant by address, which fields it contains, the lengths of the fields, the validation rules for addresses, zip codes, phone numbers, and the like. All of this information belongs in the requirements somewhere - and not in the use case!

Use cases are only "chapter three" of the requirements document, the behavioral requirements. They do not contain performance requirements, business rules, user interface design, data descriptions, finite state machine behavior, priority, and probably some other information.

"Well, where are those requirements?!" the system developers cry! It is all very well to say the use cases shouldn't contain those, but they have to get documented sometime.

. . .

17. USE CASES IN THE OVERALL PROCESS

. . .

18. USE CASES BRIEFS AND EXTREMEPROGRAMMING

The ultralight methodology, Extreme Programming, or XP, uses an even lighter form of behavioral requirements than the ones I show in this book (see <u>Extreme Programming Explained</u>, by Kent Beck, Addison-Wesley, 1999). In XP, the usage and business experts sit right with the developers. Since the experts are right there, the team does not write down detailed requirements for the software, but writes *user stories* as a sort of promissory note to have a further discussion about the requirements around a small piece of functionality.

An XP user story, in its brevity, may look either like the *use case briefs* described in "A sample of use case briefs" on page 47, or a system *feature* as described in "Feature list for Capture Trade-in" on page 170.

Each XP user story needs to be just detailed enough that both the business and technical people understand what it means and can estimate how long it will take. It must be a small enough piece of work that the developers can design, code, test and deploy it in three weeks or less. Once meeting those criteria, it can be as brief and casual as the team can get manage. It often gets written just on an index card.

When the time comes to start working on a user story, the designer simply takes the card over to the business expert and asks for more explanation. Since the business expert is always available, the conversation continues, as needed, until the functionality is shipped.

On rare occasion, a small, well-knit development team with full-time users on board will take usage narratives or the use case briefs as their requirements. This only happens when the people who own the requirements sit very close to the people designing the system. The designers collaborate directly with the requirements owners during design of the system. Just as with XP's user stories, this can work *if* the conditions for being able to fulfill on the promissory note are met. In most projects they are not met, and so it is best to keep the usage narrative acts as a warm-up exercise at the start of a use case writing session, and the use case brief as part of the project overview.

19. MISTAKES FIXED

The most common mistakes in writing are leaving out the subjects of sentences, making assumptions about the user interface design, and using goal levels that are too low. Here are some examples. The purpose of this section is not to quiz you, but to sharpen your visual reflexes.

The first examples are short; the last one is a long example from a real project. Practice on the small ones, then tackle the last one.

. . .

Chapter 19. Mistakes Fixed
Parameterized use cases - Page 150

PART 3
REMINDERS FOR THE BUSY

20. EACH USE CASE

Reminder 1. A use case is a prose essay

. . .

Reminder 2. Make the use case easy to read.

. . .

Reminder 3. Just one sentence form

. . .

Reminder 4. Include sub use cases

. . .

Reminder 5. Who has the ball?

. . .

Reminder 6. Get the goal level right

. . .

Reminder 7. Keep the GUI out

. . .

Reminder 8. Two endings

. . .

Reminder 9. Stakeholders need guarantees

. . .

Reminder 10. Preconditions

. . .

Reminder 11. Pass/Fail tests for one use case

. . .

21. THE USE CASE SET

Reminder 12. An ever-unfolding story

. . .

Reminder 13. Corporate scope <u>and</u> system scope

. . .

Reminder 14. Core values & variations

. . .

Reminder 15. Quality questions across the use case set

. . .

22. WORKING ON THE USE CASES

Reminder 16. It's just chapter 3 (where's chapter 4?)

. . .

Reminder 17. Work breadth first

. . .

Reminder 18. The 12-step recipe

. . .

Reminder 19. Know the cost of mistakes

. . .

Reminder 20. Blue jeans preferred

. . .

Reminder 21. Handle failures

. . .

Reminder 22. Job titles sooner <u>and</u> later

. . .

Reminder 23. Actors play roles

. . .

Reminder 24. The Great Drawing Hoax

For reasons that remain a mystery to me, many people have focused on the stick figures and ellipses in use case writing since Jacobson's first book came out, and neglected to notice that use cases are fundamentally a text form. The strong CASE tool industry, which already had graphical but not text modeling tools, seized upon this focus and presented tools that maximized the amount of drawing in use cases. This was not corrected in the OMG's Unified Modeling Language

standard, which was written by people experienced in textual use cases. I suspect the strong CASE tool lobby affected OMG efforts. "UML is merely a tool interchange standard", is how it was explained to me on several occasions. Hence, the text that sits behind each ellipse somehow is not part of the standard, and is a local matter for each writer to decide.

Figure 25. "Mommy, I want to go home".

Whatever the causes, we now have a situation in which many people think that the ellipses *are* the use cases, even though the ellipses convey very little information. Experienced developers can be quite sarcastic about this. I thank Andy Hunt and Dave Thomas for this lighthearted spoof, mocking the cartoonish "requirements made easy" view of use cases. From The Pragmatic Programmer, 1999.
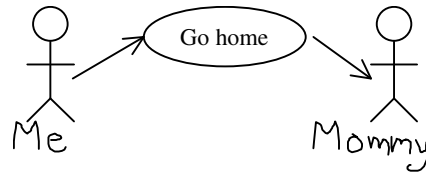
It is important to recognize that the ellipses cannot possibly replace the text. The use case diagram is (intentionally) lacking sequencing, data, and receiving actor. It is to be used

* as a table of contents to the use cases,
* as a context diagram for the system, showing the actors pursuing their various and overlapping goals, and perhaps the system reaching out to the secondary actors,
* as a "big picture", showing how higher-level use cases relate to lower-level use cases.

These are all fine, as described in Reminder 14."Core values & variations" on page 201. Just remember that use cases are fundamentally a text form, so use the ellipses to augment, not replace the text. The following two figures below show two ways of presenting the context diagram.

. . .

Reminder 25. The great tool debate.

Sadly, use cases are not supported very well by any tools on the market (now, early 2000). Many companies claim to support them, either in text or in graphics. However, none of the tools contain a metamodel close to that described in *2.3"The Graphical Model" on page 41*, and most are quite hard to use. As a result, the use case writer is faced with an awkward choice.

. . .

Reminder 26. Project planning using titles and briefs

. . .

Chapter 22. Working on the Use Cases

PART 4
END NOTES

How could I not discuss the Unified Modeling Language and its impact on use cases? UML actually impacts use case writing very little. Most of what I have to say about writing effective use cases fits inside one ellipse. Appendix A covers ellipses, stick figures, *includes, extends, generalizes*, the attendant hazards, and drawing guidelines.

Appendix B provides answers to selected exercises. I hope you do those exercises, and read the discussions provided with the answers.

Appendix C is a glossary of the key terms used in the book.

Appendix D is a list of the articles, books, and web pages I referred to along the way.

APPENDIX A: USE CASES IN UML

The Unified Modeling Language defines graphical icons that people are determined to use. It does not address use case content or writing style, but it does provide lots of complexity for people to discuss. Spend your energy learning to write clear text instead. If you like diagrams, learn the basics of the relations, and then set a few, simple standards to keep the drawings clear.

23.1 Ellipses and Stick Figures

When you walk to the whiteboard and start drawing pictures of people using the system, it is very natural to draw a stick figure for the people, and ellipses or boxes for the use cases they are calling upon. Label the stick figure with the title of the actor and the ellipse with the title of the use case. The information is the same as the actor-goal list, but the presentation is different. The diagrams can be used as a table of contents.So far, all is all fine and normal.

The trouble starts when you or your readers believe that the diagrams define the functional requirements for the system. Some people get infatuated with the diagrams, thinking they will make a hard job simple (as in Figure 25.""Mommy, I want to go home"" on page 203). They try to capture as much as possible in the diagram, hoping, perhaps, that text will never have to be written. Here are two typical events, symptoms of the situation.

 A person in my course recently unrolled a taped-together diagram several feet on a side, with ellipses and arrows going in all directions, *includes* and *extends* and *generalizes* all mixed around (distinguished, of course, only by the little text label on each arrow). He wanted coaching on whether their project was using all the relations correctly, and was unaware it was virtually impossible to understand what his system was supposed to *do*.

Another showed with pride how he had "repaired" the evident defect of diagrams not showing the order in which sub use cases are called. He added yet more arrows to show which sub use case preceded which other, using the UML *preceeds* relation. The result, of course, was an immensely complicated drawing, that took up more space than the equivalent text, and was harder to read. To paraphrase the old saying, he could have put 1,000 readable words in the space of his one unreadable drawing.

Drawings are a two-dimensional mnemonic device that serve a cognitive purpose: to highlight relationships. Use the drawings for this purpose, not to replace the text.

With that purpose in hand, let us look at the individual relations in UML, their drawing and use.

23.2 UML's *Includes* Relation

A *base* use case *includes* an *included* use case if an action step in the base use cases calls out the included one's name. This is the normal and obvious relationship between a higher-level and a lower-level use case. The included use case describes a lower-level goal than the base use case.

The verb phrase in an action step is potentially the name of a sub use case. If you never break that goal out into its own use case, then it is simply a step. If you do break that goal out into its own use case, then the step *calls* the sub use case (in my vocabulary), or it *includes the behavior of* the included use case, in UML 1.3 vocabulary. Prior to UML 1.3, it was said to *use* the lower level use case, but that phrase is now out of date.

A dashed arrow goes from the (higher-level) base use case to the included use case, signifying that the base use case "knows about" the included one, as illustrated in Figure 29..

Guideline 13: Draw higher goals higher

Always draw higher level goals higher up on the diagram than lower level goals. This helps reduce the goal-level confusion, and is intuitive to readers. When you do this, the arrow from a base use case to an *included* use case will *always* point down.

Figure 29. Drawing *Includes*.

UML permits you to change the pictorial representation of each of its elements. I find that most people drawing by hand simply draw a *solid* arrow from base to included use case (drawing dashed ones by hand is tedious). This is fine, and now you can justify it :-). When drawing with a graphics program, you will probably use the shape that comes with the program.
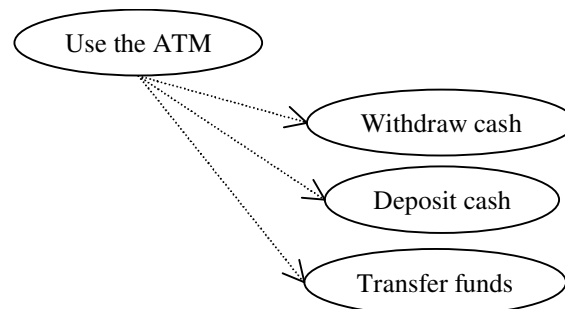
It should be evident to most programmers that the *includes* relation is the old subroutine call from programming languages. This is not a problem or a disgrace, rather, it is a natural use of a natural mechanism, which we use in our daily lives and also in programming. On occasion, it is appropriate to parameterize use cases, pass them function arguments, and even have them return values (see 14."Two Special Use Cases" on page 144). Keep in mind, though, that the purpose of a use case is to communicate with another person, not a CASE tool or a compiler.

23.3 UML's *Extends* Relation

An *extending* or *extension* use case *extends* a *base* use case if the extending one names the base one and under what circumstances it interrupts the base use case. The base use case does not name the extending one. This is useful if you want to have any number of use cases interrupt the base one, and don't want the maintenance nightmare of updating the higher level use case each time a new, interrupting use case is added. See Section 10.2"Extension use cases" on page 66.

Behaviorally, the extending use case specifies some internal condition in the course of the base use case, with a triggering condition. Behavior runs through the base use case until the condition occurs, at which point behavior continues in the extending use case. When the extending use case finishes, the behavior picks up in the base use case where it left off.

Rebecca Wirfs-Brock colorfully refers to the extending use case as a *patch* on the base use case (programmers should relate to the analogy of program patches!). Other programmers see it as a text version of the mock programming instruction, the *come-from* statement.

We use the extension form quite naturally when writing extension conditions within a use case. An *extension use case* is just the extension condition and handling pulled out and turned into a use case on its own (see*10.2"Extension use cases" on page 66*). Think of an extension use case as a scenario extension that outgrew its use case and was given its own space.

The default UML drawing for *extends* is a dashed arrow (the same as for *includes*) from extending to base use case, with the phrase <<extends>> set alongside it. I draw it with a hook from the extending back to the base use case, as shown in *Figure 30.,* to highlight the difference between *includes* and *extends* relations.

Figure 30.(a) shows the default UML way of drawing *extends* (example from <u>UML Distilled</u>. Figure 30. (b) shows the hook connector.

Figure 30. Drawing *Extends*.
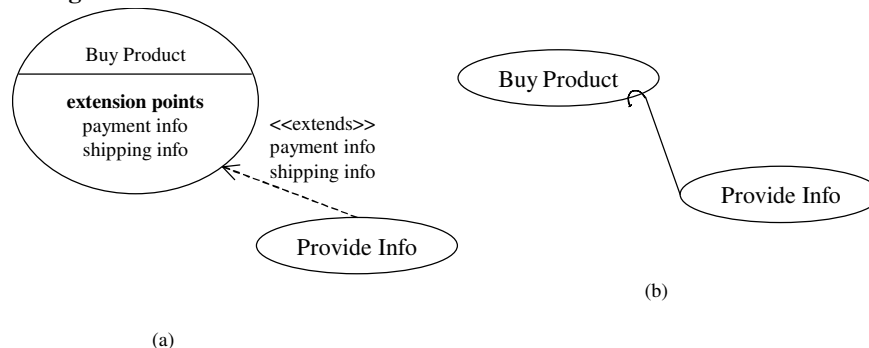
Guideline 14: Draw extending use cases lower

An extension use case is generally at lower level than the use case it extends, and so it should similarly be placed lower on the diagram. In the *extends* relation, however, it is the lower use case that knows about the higher use case. Therefore, draw the arrow or hook *up* from the extending to the base use case symbol.

Guideline 15: Use different arrow shapes

UML deliberately leaves unresolved the shape of the arrows connecting use case symbols. Any relation can be drawn with an open-headed arrow and some small text that says what the relation is. The idea is that different tool vendors or project teams might want to customize the shapes of the arrows, and the UML standard should not prevent them.

The unfortunate consequence is that people simply use the undifferentiated arrows for all relations. This makes drawings hard to read. The reader must study the small text to detect which relations are intended. Later on, there are no simple visual clues to help remember the relations. This combines with the absence of other drawing conventions to make many use case diagrams truly incomprehensible.

Therefore, take the trouble to set up different arrow styles for the three relations.

- The standard *generalizes* arrow in UML is the triangle-headed arrow. Use that.

- The default, open-headed arrow should be the frequently used one. Use it for *includes.*

- Create a different arrow for *extends*. I have started using a hook from extending to base use case. Readers like that it is immediately recognizable, doesn't conflict with any of the other UML symbols, and brings along its own metaphor, that an extending use case has its hooks in the base use case. Whatever you use, work to make the *extends* connector stand out from the other ones on the page.

Correct use of *extends*

"When to use extension use cases" on page 118 discusses the main occasions on which to create extension use cases. I repeat those comments here.

The most common is when there are many asynchronous services the user might activate, which should not disturb the base use case. Often, they are developed by different teams. These situations show up with shrink-wrapped software packages as illustrated in Figure 31..

Figure 31. Three interrupting use cases extending a base use case.
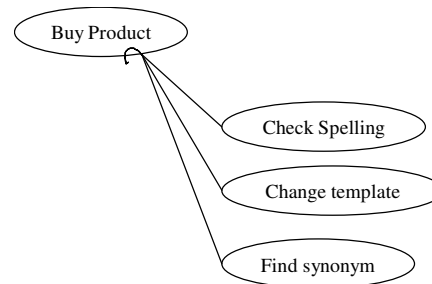
The second situation is when you are writing additions to a locked requirements document. In an incrementally staged system, you might lock the requirements after each delivery. You would then *extend* a locked use case with one that adds function.

Extension points

The circumstance that caused *extends* to be invented in the first place was the practice of never touching the requirements file of a previous system. In the original telephony systems where these were developed, the business often added asynchronous services, and so the *extends* relation was practical, as just described. The new team could build on the safely locked requirements document, adding the requirements for a new, asynchronous service at whatever point in the base use case was appropriate, without touching a line of the original system requirements.

But referencing behavior in another use case is problematic. If no line numbers are used, how should we refer to the point at which the extension behavior picks up? And if line numbers are used, what happens if the base use case gets edited and the line numbers change?

Recall, if you will, that the line numbers are really line labels. They don't have to be numeric, and they don't have to be sequential. They are just there for ease of reading and so the extension conditions have a place to refer to. Usually, however, they are numbers, and they are sequential. Which means that they will change over time.

Extension points were introduced to fix these issues. An *extension point* is a publicly visible label in the base use case that identifies a moment in the use case's behavior by nickname (technically, it can refer to set of places, but let us leave that aside for the moment).

Publicly visible extension points introduce a new problem. The writers of a base use cases are charged with knowing where it can get extended. They must go back and modify it whenever someone thinks up a new place to extend it. Recall that the original purpose of *extends* was to avoid having to modify the base use case.

You will have to deal with one of these problems. Personally, I find publicly declared extension points more trouble than they are worth. I prefer just describing, textually, where in the base use case the extending use case picks up, ignoring nicknames, as in the example below.

If you do use extension points, don't show them on the diagram. The extension points take up most of the space in the ellipse, dominating the reader's view and obscuring the much more important goal name (see Figure 30.). The behavior they refer to does not show up on the diagram. They cause yet more clutter.

There is one more fine point about extension points. An extension point name is permitted to call out not just *one* place in the base use case, but as many as you wish, places where the extending use cases needs to add behavior. You would want this in the case of the ATM, when adding the extension use case *Use ATM of Another Bank*. The extending use case needs to say,

> "Before accepting to perform the transaction, the system gets permission from the customer to charge the additional service fee.
>
> ...
>
> After completing the requested transaction, the system charges the customer's account the additional service fee."

Of course, you could just say that.

23.4 UML's *Generalizes* Relations

A use case may *specialize* a more general one (and vice versa, the general one *generalizes* the specific one). The (specializing) child should be of a "similar species" to the (general) parent. More exactly, UML 1.3 says, "a generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior and extension points defined in the parent use case, and participates in all the relationships of the parent use case".

Correct use of *generalizes*

A good test phrase is *generic*, using the phrase "some kind of". Be alert for when find yourself saying, "the user does some kind of this action", or saying, "the user can do one of several kinds of things here". Then you have a candidate for *generalizes*.

Here is a fragment of the *Use the ATM* use case.

---

1. Customer enters card and PIN.
2. ATM validates customer's account and PIN.
3. Customer does a transaction, one of:
   - Withdraw cash
   - Deposit cash

>   - Transfer money
>   - Check balance
> Customer does transactions until selecting to quit
> 4. ATM returns card.

What is it the customer does in step 3? Generically speaking, "a transaction". There are four *kinds of* transactions the customer can do. *Generic* and *kinds of* tip us off to the presence of the generic or generalized goal, "Do a transaction". In the plain text version, we don't notice that we are using the *generalizes* relation between use cases, we simply list the kinds of operations or transactions the user can do and keep going. For UML mavens, though, this is the signal to drag out the *generalization* arrow.

Actually, we have two choices. We can ignore the whole generalizes business, and just *include* the specific operations, as shown in Figure 32.(a). Or, we can create a *general* use case for "Do one ATM transaction", and show the specific operations as specializations of it, as in Figure 32.(b).

Use whichever you prefer. Working in prose, I don't create generalized use cases. There is rarely any text to put into the generic goal, so there is no need to create a new use case page for it. Graphically, however, there is no way to express "does one of the following transactions", so you have to find and name the generalizing goal.

Guideline 16: Draw generalized goals higher

Always draw the generalized goal higher on the diagram. Draw the arrowhead pointing up into the bottom of the generalizing use case, not into the sides. See Figure 32. and Figure 34. for examples.

Figure 32. Drawing *Generalizes*. Converting a set of *included* use cases into specializations of a generic action.
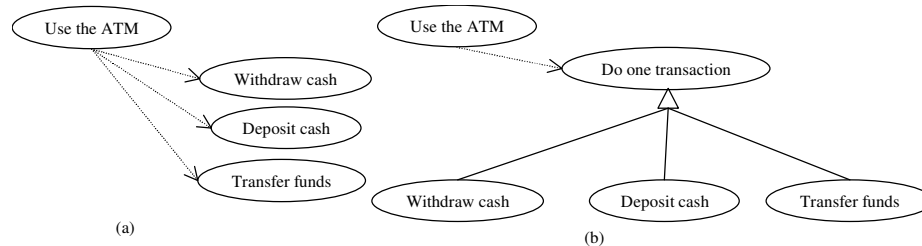
Hazards of *generalizes*

Watch out when combining specialization of actors with specialization of use cases. The key idiom to avoid is that of a *specialized actor using a specialized use case*, as illustrated in Figure 33. "Hazardous generalization, closing a big deal".
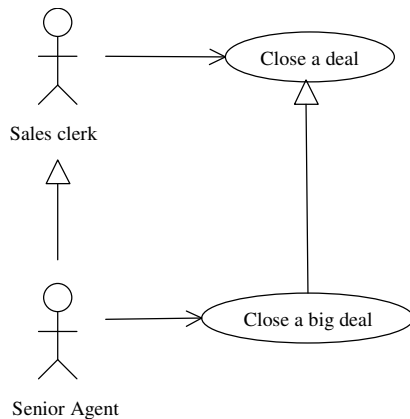
Figure 33. Hazardous generalization, closing a big deal.

Figure 33. is trying to express the fairly normal idea that a Sales Clerk can close any deal, but it takes a special kind of sales clerk, a Senior Agent, to close a deal above a certain limit. Let's watch how the drawing actually expresses the opposite of what is intended.

From Section 4.2"The primary actor of a use case" , we recall that the specialized actor can do every use case the general actor can do. So the Sales Clerk is a generalized Senior Agent. To many people, this seems counterintuitive, but it is official and correct.

The other specialization seems quite natural: Closing a Big Deal is a special case of closing an ordinary deal. However, the UML rule is, "A *specialized use case can be substituted wherever a general use case is mentioned*". Therefore, the drawing says that an ordinary Sales Clerk can close a Big Deal!

Figure 34. Correctly closing a big deal.

The corrected drawing is shown in Figure 34."Correctly closing a big deal" . You might look at this drawing and ask, does closing a small deal really *specialize* closing a basic deal, or does it *extend* it? Since working with text use cases will not put you in this sort of puzzling and economically wasteful quandary, I leave that question as an exercise to the interested reader.

In general, the critique I have of the generalizes relation is that the professional community has not yet reached an understanding of what it means to subtype and specialize behavior, what properties and options are implied. Since use cases are descriptions of behavior, there can be no standard understanding of what it means to specialize use cases.

If you do use the generalizes relation, my suggestion is to make the generalized use case empty, as in *Do a transaction*, above. Then the specializing use case will supply *all* the behavior, and you only have to worry about the one trap described above.
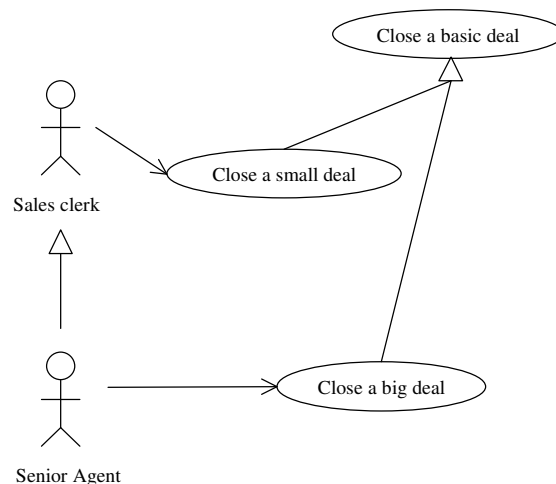
23.5 *Subordinate* vs. sub use cases

In the extended text section of UML specification 1.3, the UML authors describe a little-known pair of relations between use cases, one that has no drawing counterpart, is not specified in the object constraint language, but is simply written into the explanatory text. The relations are *subordinate use case*, and its inverse, *superordinate use case*.

The intent of these relations is to let you show how the use cases of *components* work together to deliver the use case of a larger system. In an odd turn, the components themselves are not shown. The use cases of the components just sit in empty space, on their own. It is as though you were to draw an anonymous collaboration diagram, a special sort of functional decomposition, that you are later supposed to explain with a proper collaboration diagram.

> "A use case specifying one model element is then refined into a set of smaller use case, each specifying a service of a model element contained in the first one. ... Note though, that the structure of the container element is not revealed by the use cases, since they only specify the functionality offered by the elements. The subordinate use cases of a specific superordinate use case cooperate to perform the superordinate one. Their cooperation is specified by collaborations and may be presented in collaboration diagrams." (UML 1.3 specification)

The purpose of introducing these peculiar relations in the explanatory text of the use case specification is unclear. I don't propose to explain them. The reason that I bring up the matter is because I use the term "sub use case" in this book, and someone will get around to asking, "What is the relation between Cockburn's sub use case and the UML subordinate use case?"

I intend sub use case to refer to a goal at a lower goal level. In general, the higher level use case will call (*include*) the sub use case. Formerly, I said "subordinate" and "superordinate" for higher and lower level use cases. Since UML 1.3 has taken those words, I have shifted vocabulary. My experience is that people do not find anything odd to notice about the terms "calling use case" and "sub use case". These notions are clear to even the novice writer and reader.

23.6 Drawing Use Case Diagrams

When you choose to draw use case diagrams with stick figures and ellipses, or just with rectangles and arrows, you will find that the ability of the diagram to communicate easily to your readers is enhanced if you set up and follow a few simple diagramming conventions. Please don't hand your readers a rat's next of arrows, and then expect them to trace out your meaning. The guidelines mentioned above, for the different use case relations, will help. There are two more drawing guidelines that can help.

Guideline 17: User goals in a context diagram

On the main, context diagram, do not show any use cases lower than user-goal level. The purpose of the diagram is, after all to provide context, to give a table of contents for the system being designed. If you decompose use cases in diagram form, put the decompositions on separate pages.

Guideline 18: Supporting actors on the right

I find it helpful to place *all* the primary actors on the left of the system box, leaving room on the right for the supporting (secondary) actors. This reduces confusion about primary versus secondary actors.

Some people never draw supporting actors on their diagrams. This frees up the right side of the box so that primary actors can be placed on both sides.

23.7 Write text-based use cases instead

If you spend very much time studying and worrying about the graphics and the relations, then you are expending energy in the wrong place. Put your energy into writing easy-to-read prose. In prose, the relations between use cases are straightforward, and you won't understand why other people are getting tied up in knots about them.

This is a view shared by many use case experts. It is somewhat self-serving to relate the following event, but I wish to emphasize the seriousness of the suggestion. My thanks to Bruce Anderson of IBM's European Object Technology Practice for the comment he made during a panel on use cases at OOPSLA '98. A series of questions revolved around the difference between *includes* and *extends* and the trouble with the exploding number of scenarios and ellipses. Bruce responded that his groups don't run into scenario explosion and don't get confused. The next questioner asked why everyone else was concerned about "scenario explosion and how to use *extends"*, but he wasn't. Bruce's answer was, "I just do what Alistair said to do." His teams spend time writing clear text, staying away from *extends*, and not worrying about diagrams.

People who write good text-based use cases simply do not run into the problems of people who fiddle with the stick figures, ellipses and arrows of UML. The relations come naturally when you write an unfolding story. They become an issue only if you dwell on them. As more consultants gain experience both ways, an increasing number reduce emphasis on ellipses and arrows, and recommend against using the *extends* relation.

APPENDIX B: ANSWERS TO (SOME) EXERCISES

...

25. APPENDIX C: GLOSSARY

Main terms

Use case. A use case expresses a contract between the stakeholders of a system about its behavior. It describes the system's behavior and interactions under various conditions as it responds to a request on behalf of the stakeholders, the *primary actor*, showing how the primary actor's goal gets delivered or fails. The use case collects together the scenarios related to the primary actor's goal.

Scenario. A scenario is a sequence of action and interactions that occurs under certain conditions, expressed without *if*s or branching.

A *concrete scenario* is a scenario in which all the specifics are named: the actor names and the values involved. It is equivalent to describing a story in the past tense, with all details named.

A *usage narrative*, or just *narrative*, is a concrete scenario that reveals motivations and intentions of various actors. It is used as a warm-up activity to reading or writing use cases.

In requirements writing, scenarios are written using placeholder terms like "customer" and "address" for actors and data values. When it is necessary to distinguish these from *concrete scenarios* they can be called *general scenarios*.

Path *through a use case* and *course* *of a use case* are synonyms for *general scenario*.

The *main success scenario* is the one written in full, from trigger to completion, including goal delivery and any bookkeeping that happens after. It is a typical and illustrative success scenario, even though it may not be the only success path.

An *alternate course* is any other scenario or scenario fragment written as an extension to the main success scenario.

An *action step* is the unit of writing in a scenario. Typically one sentence, usually describes behavior of only one actor.

Scenario extension. A scenario fragment that starts upon a particular condition in another scenario.

The *extension condition* names the circumstances under which the different behavior occurs.

An *extension use case* is use case that interrupts another use case, starting upon a particular

condition. The use case that gets interrupted is called the ***base use case.***

An ***extension point*** is a tag or nickname for a place in a base use case where an extension use case can interrupt it. An extension point may actually name a set of places in the base use case, so that the extension use case can collect together all the related extension behaviors that interrupt the base use case for one set of conditions.

A ***sub use case*** is a use case called out in a step of a scenario. In UML, the calling use case is said to ***include*** *the behavior of* the sub use case.

Interaction. A message, a sequence of interactions, or a set of interaction sequences.

Actor. Something with behavior (able to execute an *if* statement). It might be a mechanical system, computer system, a person, an organization or some combination.

An ***external actor*** is an actor outside the system under discussion.

A ***stakeholder*** is an external actor which is entitled to have its interests protected by the system, and satisfying whose interests requires the system to take specific actions. Different use cases can have different stakeholders.

A ***primary actor*** is a stakeholder who requests the system to deliver a goal. Typically but not always, the primary actor initiates the interaction with the system. The primary actor may have an intermediary initiate the interaction with the system, or may have the interaction triggered automatically on some event.

A ***supporting*** or ***secondary*** actor is a system against which the SuD has a goal.

An ***off-stage*** or ***tertiary*** actor is a stakeholder of a use case who is not the primary actor.

An ***internal actor*** is either the system under discussion (SuD) itself, a subsystem of the SuD, or an active component of the SuD.

Types of use cases

A use case ***brief*** is a one-paragraph synopsis of the use case.

A ***casual use case*** is one written in simple, paragraph, prose style. It is likely to be missing project information associated with the use case, and is likely to be less rigorous in its description than a fully dressed use case.

A ***fully dressed use case*** is written with one of the full templates, identifying actors, scope, level, trigger condition, precondition, and all the rest of the template header information, plus project annotation information.

A *black-box use case* does not mention any components inside the SuD. Typically used in the system requirements document.

A *white-box use case* mentions the behavior of the components of the SuD in the description. Typically used in business process modeling.

A *summary-level use case* is one that takes multiple user-goal sessions to complete, possibly weeks, months or years. Sub use cases can be any level of use case. Marked graphically with a cloud ⬡ or a kite ✎ . The cloud is used for use cases that contain steps at cloud or kite level. The kite is used for use cases that contain user-goal steps.

A *user-goal use case* satisfies a particular and immediate goal of value to the primary actor. Typically performed by one primary actor in one sitting of 2-20 minutes (less if the primary actor is a computer), after which they can leave and proceed with other things. Steps are user-goal or lower. Marked graphically with waves 〰 .

A *subfunction use case* is one satisfying a partial goal of a user-goal use case or of another subfunction. Steps are lower-level subfunctions. Marked graphically with a fish ⋈◎ or a clam ⊖ . Using the clam signifies that the use case is too low level and should not be written at all.

The phrase *business use case* is a short-cut phrase indicating that the use case puts the emphasis on the operation of the business rather than the operation of a computer system. It is possible to write a business use case at any goal level, but only at *enterprise* or *organization* scope.

The phrase *system use case* is a short-cut phrase indicating that the use case puts the emphasis on the computer or mechanical system rather than the operation of a business. It is possible to write a system use case at any goal level and at with any scope, including *enterprise* scope. A system use case written at enterprise scope highlights the effect of the SuD on the behavior of the enterprise.

Enterprise scope means the SuD is an organization or enterprise. Labeled on the use case with the name of the organization, business or enterprise. Marked graphically with a building in gray 🏠 or white ⌂ depending on whether the use case is black- or white-box.

System scope means the SuD is a mechanical/ hardware/ software system or application. Labeled on the use case with the name of the system. Marked graphically with a box in gray ▦ or white ▢ depending on whether the use case is black- or white-box.

Subsystem scope means the SuD in this use case is a portion of an application, perhaps a subsystem or framework. Labeled on the use case with the name of the subsystem, and marked graphically with a threaded bolt ⟨▥▥ .

Diagrams

Use case diagram. In UML, the diagram showing the external actors, the system boundarry, the use cases as ellipses, and arrows connecting actors to ellipses or ellipses to ellipses. Primarily useful as a context diagram and table of contents.

Sequence diagram. In UML, the diagram showing actors across the top, owning columns of space, and interactions as arrows between columns, with time flowing down the page. Useful for showing one scenario graphically.

Collaboration diagram. In UML, a diagram showing the same information as the sequence diagram but in a different form. The actors are placed around the diagram, and interactions are shown as numbered arrows between actors. Time is shown only by numbering the arrows.

26. APPENDIX D: READING

Books referenced in the text.

Beck, K., <u>Extreme Programming Explained</u>, Addison-Wesley, 1999.

Cockburn, A., <u>Surviving Object-Oriented Projects</u>, Addison-Wesley, 1998.

Cockburn, A., <u>Software Development as a Cooperative Game</u>, Addison-Wesley, due 2001.

Constantine, L., and Lockwood, L., <u>Software for Use</u>, Addison-Wesley, 1999.

Hohmann, L., <u>GUIs with Glue</u>, in preparation as of 2000.

Roberson, S. and Robertson, R., <u>Managing Requirements</u>, Addison-Wesley, 1999.

Wirfs-Brock, R., Wilkerson, B., Wiener, L., <u>Designing Object-Oriented Software</u>, Prentice-Hall, 1990.

Articles referenced in the text.

Beck, K., Cunningham, W., "A laboratory for object-oriented thinking", ACM SIGPLAN 24(10):1-7, 1989.

Cockburn, A., "VW-Staging", http://members.aol.com/acockburn/papers/vwstage.htm

Cockburn, A., "An Open Letter to Newcomers to OO", http://members.aol.com/humansandt/papers/oonewcomers.htm

Cockburn, A., "CRC Cards", http://members.aol.com/humansandt/papers/crc.htm

Cunningham, W., CrcCards", http://c2.com/cgi/wiki?CrcCards

McBreen, P., "Test cases from use cases", http://www.cadvision.com/roshi/papers.html

Online resources useful to your quest.

The web has huge amounts of information. Here are a few starting points.

http://www.usecases.org

http://members.aol.com/acockburn

http://www.foruse.com

http://www.pols.co.uk/usecasezone/

Chapter 26. Appendix D: Reading
Write text-based use cases instead - Page 240

F
Flexography 44, 46, 48, 50, 52, 54, 56, 58, 60

Pass/Fail Tests for Use Case Fields

All of them should produce a "yes" answer.

Field	Question
Use case title.	**1** Is the name an active-verb goal phrase, the goal of the primary actor?
	2 Can the system deliver that goal?
Scope and Level:	**3** Are the scope and level fields filled in?
Scope.	**4** Does the use case treat the system mentioned in the Scope as a black box? (The answer may be 'No' if the use case is a white-box business use case, 'Yes' if it is a system requirements document).
	5 If the Scope is the actual system being designed, do the designers have to design everything in the Scope, and nothing outside it?
Level.	**6** Does the use case content match the goal level stated in Level?
	7 Is the goal really at the level mentioned?
Primary actor.	**8** Does the named primary actor have behavior?
	9 Does it have a goal against the SuD that is a service promise of theSuD?
Preconditions.	**10** Are they mandatory, and can they be put in place by the SuD?
	11 Is it true that they are never checked in the use case?
Stakeholders and interests.	**12** Are they mentioned? (Usage varies by formality and tolerance) Must the system satisfy their interests as stated?
Minimal guarantees.	**13** If present, are all the stakeholders' interests protected?
Success guarantees.	**14** Are all stakeholders interests satisfied?
Main success scenario.	**15** Does it run from trigger to delivery of the success guarantee?
	16 Is the sequence of steps right (does it permit the right variation in sequence)?
	17 Does it have 3 - 9 steps?

Field	Question
Each step in any scenario.	**18** Is it phrased as an goal that succeeds?
	19 Does the process move distinctly forward after successful completion of the step?
	20 Is it clear which actor is operating the goal (who is "kicking the ball?)
	21 Is the intent of the actor clear?
	22 Is the goal level of the step lower than the goal level of the overall use case? Is it, preferably, just a bit below the use case goal level?
	23 Are you sure the step does not describe the user interface design of the system?
	24 Is it clear what information is being passed?
	25 Does the step "validate", as opposed to "checking" a condition?
Extension condition.	**26** Can and must the system detect it, and handle it?
	27 Is it phrased as what the system actually detects?
Technology or Data Variation List.	**28** Are you sure this is not an ordinary behavioral extension to the main success scenario?
Overall use case content.	**29** To the sponsors and users: "Is this what you want?"
	30 To the sponsors and users: "Will you be able to tell, upon delivery, whether you got this?"
	31 To the developers: "Can you implement this?"