WRITING EFFECTIVE USE CASES

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

Alistair Cockburn
Humans and Technology
copyright A.Cockburn, 1999-2000

Addison-Wesley

date: 2000.02.21

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 Software Development as a Cooperative Game. 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.

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

Acknowledgements

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

drafts and their educated commentary on various papers and ideas. Mike Jones at Beans & Brews thought up the bolt icon for subsystem use cases.

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

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

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

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

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

Preface . . . 1
Audience . . . 1
Organization . . . 1
Heritage of the ideas in this book . . . 2
The place of use cases in the Crystal book collection . . . 3
The samples used . . . 4
Acknowledgements . . . 4

Chapter 1 Introduction to Use Cases . . . 15
1.1 What is a Use Case (more or less)? . . . 15
Use Case 1: Buy stocks over the web . . . 17
Use Case 2: Get paid for car accident . . . 18
Use Case 3: Register arrival of a box . . . 19
1.2 Your use case is not my use case . . . 20
Use Case 4: Buy something (Casual version) . . . 22
Use Case 5: Buy Something (Fully dressed version) . . . 22
Steve Adolph: "Discovering" Requirements in new Territory . . . 25
1.3 Requirements and Use Cases . . . 26
A Plausible Requirements File Outline . . . 26
Use cases as a project linking structure . . . 28
(Figure 1.: "Hub-and-spoke" model of requirements) . . . 28
1.4 When Use Cases Add Value . . . 28
1.5 Manage Your Energy . . . 29
1.6 Warm up with a Usage Narrative . . . 30
Usage Narrative: Getting "Fast Cash" . . . 31

PART 1
The Use Case Body Parts

Chapter 2 The Use Case as a Contract for Behavior. 34

2.1 INTERACTIONS BETWEEN ACTORS WITH GOALS 34

Actors have goals. 34

(Figure 2.: An actor with a goal calls upon the responsibilities of another) 35

Goals can fail 36

Interactions are compound. 36

A use case collects scenarios 38

(Figure 3.: Striped trousers: scenarios succeed or fail) 38

(Figure 4.: The striped trousers showing subgoals.) 39

2.2 CONTRACT BETWEEN STAKEHOLDERS WITH INTERESTS 40

(Figure 5.: The SuD serves the primary actor, protecting off-stage stakeholders) . . . 40

2.3 THE GRAPHICAL MODEL 41

(Figure 6.: A stakeholder has interests) 42

(Figure 7.: Goal-oriented behavior made of responsibilities, goals and actions). . . . 43

(Figure 8.: The use case as responsibility invocation) 43

(Figure 9.: Interactions are composite) 43

Chapter 3 Scope 44

A Sample In/Out List 44

3.1 FUNCTIONAL SCOPE 45

The Actor-Goal List 45

A Sample Actor-Goal List: 45

The Use Case Briefs 46

A sample of use case briefs. 47

3.2 DESIGN SCOPE 47

(Figure 10.: Design scope can be any size) 48

Using graphical icons to highlight the design scope. 49

Examples of design scope 50

Use Case 6: Add New Service (Enterprise). 51

Use Case 7: Add new Service (Acura) 51

(Figure 11.: System scope diagram for Acura - BSSO.). 52

Use Case 8: Enter and Update Requests (Joint System) 52

Use Case 9: Add new Service (into Acura) 53

Use Case 10: Note new Service request (in BSSO) 53

Use Case 11: Update Service request (in BSSO) 53

Use Case 12: Note updated Request (in Acura) 53

(Figure 12.: Use case diagrams for Acura - BSSO) 54
(Figure 13.: A combined use case diagram for Acura-BSSO.). 54
Use Case 13: Serialize access to a resource 55
Use Case 14: Apply a Lock Conversion Policy 56
Use Case 15: Apply Access Compatibility Policy 56
Use Case 16: Apply Access Selection Policy 57
Use Case 17: Make Service Client Wait for Resource Access 57
3.3 THE OUTERMOST USE CASES 58
3.4 USING THE SCOPE-DEFINING WORK PRODUCTS. 60

Chapter 4 Stakeholders & Actors 61
4.1 STAKEHOLDERS 61
4.2 THE PRIMARY ACTOR OF A USE CASE 62
Why primary actors are unimportant (and important) 63
Characterizing the primary actors 66
A sample actor profile map: 66
4.3 SUPPORTING ACTORS 66
4.4 THE SYSTEM UNDER DISCUSSION, ITSELF 67
4.5 INTERNAL ACTORS AND WHITE-BOX USE CASES 67

Chapter 5 Three Named Goal Levels 69
(Figure 14.: The levels of use cases). 69
5.1 USER-GOALS (BLUE, SEA-LEVEL) 70
Two levels of blue. 71
5.2 SUMMARY LEVEL (WHITE, CLOUD / KITE). 72
Use Case 18: Operate an Insurance Policy 72
The outermost use cases revisited. 73
5.3 SUBFUNCTIONS (INDIGO/BLACK, UNDERWATER/CLAM) 73
Summarizing goal levels 74
5.4 USING GRAPHICAL ICONS TO HIGHLIGHT GOAL LEVELS 75
5.5 FINDING THE RIGHT GOAL LEVEL 75
Find the user's goal 75
Merge steps, keep asking "why" 76
(Figure 15.: Ask "why" to shift levels) 76
5.6 A LONGER WRITING SAMPLE: "HANDLE A CLAIM" AT SEVERAL LEVELS 77
Use Case 19: Handle Claim (business) 78
Use Case 20: Evaluate Work Comp Claim 79

Use Case 21: Handle a Claim (systems) 80
Use Case 22: Register Loss 83
Use Case 23: Find a Whatever (problem statement) 86

Chapter 6 Preconditions, Triggers, Guarantees **87**

6.1 PRECONDITIONS 87

6.2 MINIMAL GUARANTEES 89

6.3 SUCCESS GUARANTEE 90

6.4 TRIGGERS 91

Chapter 7 Scenarios and Steps **92**

7.1 THE MAIN SUCCESS SCENARIO, SCENARIOS 92
Main success scenario as the simple case 92
Common surrounding structure 92
The scenario body 93

7.2 ACTION STEPS 94
Guidelines for an action step 94
Guideline 1: It uses simple grammar 94
Guideline 2: It shows clearly, "Who has the ball" 95
Guideline 3: It is written from a bird's eye point of view 95
Guideline 4: It shows the process moving distinctly forward 95
Guideline 5: It shows the actor’s intent, not movements. 96
Guideline 6: It contain a ’reasonable’ set of actions 98
(Figure 16.: A transaction has four parts) 98
Guideline 7: It doesn’t "check whether", it "validates" 99
Guideline 8: It optionally mentions the timing 100
Guideline 9: Idiom: "User has System A kick System B" 100
Guideline 10: Idiom: "Do steps x-y until condition" 100
To number or not to number 101

Chapter 8 Extensions **103**

8.1 THE EXTENSION CONDITIONS 104
Brainstorm all conceivable failures and alternative courses 105
Guideline 11: The condition says what was detected 106
Rationalize the extensions list. 108
Roll up failures 108

8.2 EXTENSION HANDLING 109
Guideline 12: Condition handling is indented 111
Failures within failures 111

Creating a new use case from an extension . . . 112

Chapter 9 Technology & Data Variations . . . 114

(Figure 17.: Technology variations using specialization) . . . 115

Chapter 10 Linking Use Cases . . . 116

10.1 SUB USE CASES . . . 116

10.2 EXTENSION USE CASES . . . 116

(Figure 18.: UML diagram of extension use cases) . . . 117

When to use extension use cases . . . 118

Chapter 11 Use Case Formats . . . 120

11.1 FORMATS TO CHOOSE FROM . . . 120

Fully dressed form . . . 120

Use Case 24: Fully Dressed Use Case Template <name> . . . *120*

Casual form . . . 121

Use Case 25: Actually Login (casual version) . . . *121*

One-column table . . . 122

Two-column table . . . 123

RUP style . . . 124

Use Case 26: Register for Courses . . . *125*

If-statement style . . . 127

OCCAM style . . . 128

Diagram style . . . 129

The UML use case diagram . . . 129

11.2 FORCES AFFECTING USE CASE WRITING STYLES . . . 130

11.3 STANDARDS FOR FIVE PROJECT TYPES . . . 134

For requirements elicitation . . . 135

Use Case 27: Elicitation Template - Oble a new biscum . . . *135*

For business process modeling . . . 136

Use Case 28: Business Process Template - Symp a carstromming . . . *136*

For sizing the requirements . . . 137

Use Case 29: Sizing Template: Burble the tramling . . . *137*

For a short, high-pressure project . . . 138

Use Case 30: High-pressure template: Kree a ranfath . . . *138*

For detailed functional requirements . . . 139

Use Case 31: Use Case Name: Nathorize a permion . . . *139*

11.4 CONCLUSION ABOUT FORMATS . . . 139

PART 2
Frequently Asked Questions

Chapter 12 **When are we done? 142**

Chapter 13 **Scaling up to Many Use Cases 144**

Chapter 14 **Two Special Use Cases 146**

14.1 CRUD USE CASES 146

Use Case 32: Manage Reports 146

Use Case 33: Save Report 148

14.2 PARAMETERIZED USE CASES 150

Chapter 15 **Business Process Modeling 153**

Modeling versus designing. 153

(Figure 19.: Core business black box). 154

(Figure 20.: New business design in white box) 154

(Figure 21.: New business design in white box (again)). 155

(Figure 22.: New business process in black-box system use cases) 156

Linking business- and system use cases 157

Rusty Walters: Business Modeling and System Requirements 158

Chapter 16 **The Missing Requirements 160**

Precision in data requirements. 161

Cross-linking from use cases to other requirements 163

(Figure 23.: Recap of Figure 1.“"Hub-and-spoke" model of requirements”) 163

Chapter 17 **Use Cases in the Overall Process 164**

17.1 USE CASES IN PROJECT ORGANIZATION 164

Organize by use case titles 164

(Figure 24.: Sample planning framework.) 164

Use cases cross releases 166

Deliver complete scenarios 167

17.2 USE CASES TO TASK OR FEATURE LISTS 167

Use Case 34: Capture Trade-in 169

Feature list for Capture Trade-in 170

17.3 USE CASES TO DESIGN 171

A special note to Object-Oriented Designers. 172

17.4 USE CASES TO UI DESIGN 174

17.5 USE CASES TO TEST CASES 174

Use Case 35: Order goods, generate invoice (testing example) 175
Acceptance test cases 175
17.6 THE ACTUAL WRITING 176
A branch-and-join process 176
Time required per use case.. 180
Collecting use cases from large groups 180
Andy Kraus: Collecting use cases from a large, diverse lay group 180

Chapter 18 Use Cases Briefs and eXtremeProgramming. 184

Chapter 19 Mistakes Fixed. 185
19.1 NO SYSTEM. 185
19.2 NO PRIMARY ACTOR 186
19.3 TOO MANY USER INTERFACE DETAILS 187
19.4 VERY LOW GOAL LEVELS 188
19.5 PURPOSE AND CONTENT NOT ALIGNED 189
19.6 ADVANCED EXAMPLE OF TOO MUCH UI. 189
Use Case 36: Research a solution - Before 190
Use Case 37: Research possible solutions - After 195

PART 3
Reminders for the Busy

Chapter 20 Each Use Case 200

Reminder 1. A use case is a prose essay 200
Reminder 2. Make the use case easy to read. 200
Reminder 3. Just one sentence form 201
Reminder 4. Include sub use cases 201
Reminder 5. Who has the ball? 202
Reminder 6. Get the goal level right 202
Reminder 7. Keep the GUI out 203
Reminder 8. Two endings 204
Reminder 9. Stakeholders need guarantees 204
Reminder 10. Preconditions 205
Reminder 11. Pass/Fail tests for one use case 206

Chapter 21 The Use Case Set 208

Reminder 12. An ever-unfolding story 208
Reminder 13. Corporate scope and system scope 208
Reminder 14. Core values & variations 209
Reminder 15. Quality questions across the use case set 212

Chapter 22 Working on the Use Cases. 213

Reminder 16. It's just chapter 3 (where's chapter 4?) 213
Reminder 17. Work breadth first 213
(Figure 25.: Work expands with precision). 214
Reminder 18. The 12-step recipe. 215
Reminder 19. Know the cost of mistakes 215
Reminder 20. Blue jeans preferred 216
Reminder 21. Handle failures. 216
Reminder 22. Job titles sooner and later 217
Reminder 23. Actors play roles 217
Reminder 24. The Great Drawing Hoax 218
(Figure 26.: "Mommy, I want to go home") 219
(Figure 27.: Context diagram in ellipse figure form.). 219
(Figure 28.: Context diagram in actor-goal format.). 220
Reminder 25. The great tool debate. 220
Reminder 26. Project planning using titles and briefs 222

PART 4
End Notes

Appendix A: Use Cases in UML 224

23.1 Ellipses and Stick Figures 224

23.2 UML's Includes Relation 225

Guideline 13: Draw higher goals higher. 225

(Figure 29.: Drawing Includes.) 225

23.3 UML's Extends Relation 226

(Figure 30.: Drawing Extends). 226

Guideline 14: Draw extending use cases lower 227

Guideline 15: Use different arrow shapes 227

Correct use of extends 227

(Figure 31.: Three interrupting use cases extending a base use case). 228

Extension points 228

23.4 UML's Generalizes Relations 229

Correct use of generalizes 229

Guideline 16: Draw generalized goals higher 230

(Figure 32.: Drawing Generalizes.) 230

Hazards of generalizes. 230

(Figure 33.: Hazardous generalization, closing a big deal). 231

(Figure 34.: Correctly closing a big deal) 231

23.5 Subordinate vs. sub use cases. 232

23.6 Drawing Use Case Diagrams 232

Guideline 17: User goals in a context diagram. 233

Guideline 18: Supporting actors on the right 233

23.7 Write text-based use cases instead 233

Appendix B: Answers to (some) Exercises 234

Exercise 6 on page 58 234

Exercise 7 on page 58 234

(Figure 35.: Design scopes for the ATM) 234

Exercise 13 on page 68 235

Exercise 14 on page 68 235

Exercise 16 on page 77 236

Exercise 17 on page 77 236

Exercise 20 on page 89 237

Exercise 23 on page 90 237

Exercise 26 on page 102 237
Exercise 27 on page 102 238
Exercise 29“Fix faulty ’Login’” 238
Use Case 38: Use the order processing system 239
Exercise 30 on page 109 239
Exercise 34 on page 113 240
Use Case 39: Buy stocks over the web 240
Exercise 37 on page 128: 241
Use Case 40: Perform clean spark plugs service 241

Chapter 25 Appendix C: Glossary. 242
Main terms 242
Types of use cases 243
Diagrams 245

Chapter 26 Appendix D: Reading 246
Books referenced in the text. 246
Articles referenced in the text. 246
Online resources useful to your quest.. 246

1. Introduction to Use Cases

What do use cases look like?

Why would different project teams need different writing styles?

Where do they fit into the requirements gathering work?

How do we warm up for writing use cases?

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

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

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

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

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

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

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

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

I show several examples of use cases below. The parts of a use case are described in the next chapter ("Use Case Body Parts"). For now, just note that the *primary actor* is the one with the goal that the use case addresses. *Scope* identifies the system that we are discussing, the *preconditions* and *guarantees* say what must be true before and after the use case runs. The *main success scenario* is a case in which nothing goes wrong. The *extensions* section describes what can happen differently during that scenario. The numbers in the extensions refer to the step numbers in the main success scenario at which each different situation gets detected (for instance, steps *4a* and *4b* indicate two different conditions that could show up at step 4). When a use case references another use case, the second use case is written in *italics* or <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"
Primary Actor: RA
Scope: Nightime Receiving Registry Software
Level: user goal
Main success scenario:
1. RA receives and opens box (box id, bags with bag ids) from TransportCompany TC
2. RA validates box id with TC registered ids.
3. RA maybe signs paper form for delivery person
4. RA registers arrival into system, which stores:
   RA id
   date, time
   box id
   TransportCompany
   <Person name?>
   # bags (?with bag ids)
   <estimated value?>
5. RA removes bags from box, puts onto cart, takes to RO.

Extensions:
2a. box id does not match transport company
4a. fire alarm goes off and interrupts registration
4b. computer goes down
leave the money on the desk and wait for computer to come back up.

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

1.2 Your use case is not my use case

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

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

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

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

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

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

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

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

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

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

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

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

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

USE CASE 4: BUY SOMETHING (CASUAL VERSION)

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

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

USE CASE 5: BUY SOMETHING (FULLY DRESSED VERSION)

Primary Actor: Requestor

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

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

Level: Summary

Stakeholders and Interests:

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

Company: wants to control spending but allow needed purchases.

Vendor: wants to get paid for any goods delivered.

Precondition: none

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

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

Trigger: Requestor decides to buy something.

Main success scenario:

1. Requestor: *initiate a request*

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

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

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

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

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

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

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

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

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

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

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

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

STEVE ADOLPH: "DISCOVERING" REQUIREMENTS IN NEW TERRITORY

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

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

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

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

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

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

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

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

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

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

1.3 Requirements and Use Cases

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

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

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

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

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

A Plausible Requirements File Outline

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

Chapter 2. The terms used / Glossary

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

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

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

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

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

Use cases as a project linking structure

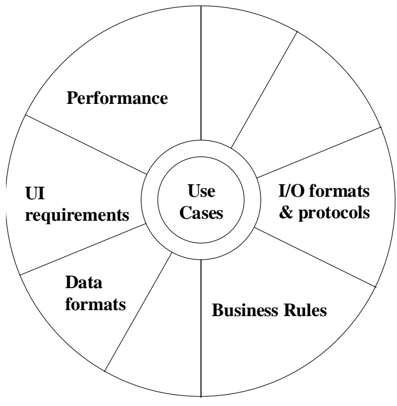


Figure 1. "Hub-and-spoke" model of requirements.

Use cases connect many other requirements details.

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

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

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

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

Requirements File Exercises

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

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

1.4 When Use Cases Add Value

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

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

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

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

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

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

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

1.5 Manage Your Energy

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

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

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

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

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

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

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

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

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

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

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

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

1.6 Warm up with a Usage Narrative

A usage *narrative* is a situated example of the use case in operation - a single, highly specific example of an actor using the system. It is not a use case, and in most projects it does not survive

into the official requirements document. However, it is a very useful device, worth my describing, and worth your writing.

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

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

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

Here is an example of a usage narrative.

USAGE NARRATIVE: GETTING "FAST CASH"

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

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

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

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

Usage Narrative Exercises

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

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

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

PART 3
REMINDERS FOR THE BUSY

20. Each Use Case

Reminder 1. A use case is a prose essay

Recall from the preface, *"writing use cases is fundamentally an exercise in prose essays, with all the difficulties in articulating good that comes with prose writing in general."*

Russell Walters of Firepond Corporation wrote,

> I think the above statement clearly nails the problem right on. This is the most misunderstood problem, and probably the biggest enlightenment for the practicing use case writer. However, I'm not sure the practitioner can come to this enlightenment on their own, well, at least until this book is published. :-) I did not understand this as the fundamental problem, and I had been working with the concept of use cases for 4 years, until I had the opportunity to work alongside you. And even then, it wasn't until I had a chance to analyze and review the *before* and *after* versions of the use case you assisted with rewriting [Use Case 36: on page 190] when the light bulb came on. Four-plus years is long time to wait for this enlightenment! So, if there is only one thing the readers of this book walk away understanding, I hope it is the realization of the fundamental problem with writing effective use cases.

Use this reminder from Rusty to help keep your eyes on the text, not the diagrams, and be aware of the writing styles you will encounter.

Reminder 2. Make the use case easy to read.

You want your requirements document short, clear, easy to read.

I sometimes feel like an 8th grade English teacher walking around, saying,

> "Use an active verb in the present tense. Don't use the passive voice, use the active voice. Where's the subject of the sentence? Say what is really a requirement, don't mention it if it is not a requirement."

Those are the things that make your requirements document short, clear, and easy to read. Here are a few habits to build to make your use cases short, clear, and easy to read:

1 Keep matters short and too the point. Long use cases make for long requirements, which few people enjoy reading.

2 Start from the top and create a coherent story line. The top will be a strategic use case. The user goal, and eventually, subfunction-level use cases branch off from here.

3 Name the use cases with short verb phrases that announce the goal to be achieved.

4 Start from the trigger, continue until the goal is delivered or abandoned, and the system has done

any bookkeeping it needs to, with respect to the transaction.

5 Write full sentences with active verb phrases that describe the subgoals getting completed.

6 Make sure the actor is visible in each step.

7 Make the failure conditions stand out, and their recovery actions readable. Let it be clear what happens next, preferably without having to name step numbers.

8 Put alternative behaviors in the extensions, rather than in *if* statements in the main body.

9 Create extension use cases only under very selected circumstances.

Reminder 3. Just one sentence form

There is only one form of sentence used in writing action steps in the use case:

* a sentence in the present tense,
* with an active verb in the active voice, describing
* an actor successfully achieving a goal that moves the process forward.

Examples are,

"Customer enters card and PIN."
"System validates that customer is authorized and has sufficient funds."
"PAF intercepts responses from the web site, and updates the user's portfolio."
"Clerk *finds a loss* using search details for "loss"."

Use this sentence form in business use cases, system use cases, summary, user, subfunction use cases, with the fully dressed or the casual template. It is the same in the main success scenario and in the extension scenario fragments. Master this sentence style.

It is useful to have a different grammatical form for condition part of an extension, so it doesn't get confused with the action steps. Use a sentence fragment (possibly a full sentence, preferably (but not always) in the past tense). End the condition with a colon ('**:**') instead of period.

"Time-out waiting for PIN entry:"
"Bad password:"
"File not found:"
"User exited in the middle:"
"Data submitted is not complete:"

Reminder 4. Include sub use cases

What you would do quite naturally if no one told you to do otherwise, is to write a step that calls out the name of a lower-level goal or use case, as in:

"Clerk *finds a loss* using search details for "loss"."

In the terms of the Unified Modeling Language, the calling use case just *included* the sub use case. It is so much the most obvious thing to do, that it would not even deserve mention if there weren't writers and teachers encouraging people to use the UML *extends* and *specializes* relations (for my views, see “Appendix A: Use Cases in UML”).

As a first rule of thumb, always use the *includes* relation between use cases. People who follow this rule report that they and their readers simply have less confusion with their writing than people who mix *includes* with *extends* and *specializes*. For the other occasions, see“When to use extension use cases” on page 118.

Reminder 5. Who has the ball?

Sometimes people write in the passive voice or from the point of view of the system itself, looking out at the world. This produces sentences like: "Credit limit gets entered." This sentence doesn’t mention who it is that enters the credit limit.

Write from the point of view of a bird up above, watching and recording the scene. Or write in the form of a play, announcing which actor is about to act. Or pretend for a moment that you are describing a soccer game, in which actor1 has the ball, dribbles it, then kicks it to actor2. Actor2 passes it to actor3, and so on.

Let the first or second word in the sentence be the name of the actor who owns the action. Whatever happens, make sure it is always clear who has the ball.

Reminder 6. Get the goal level right

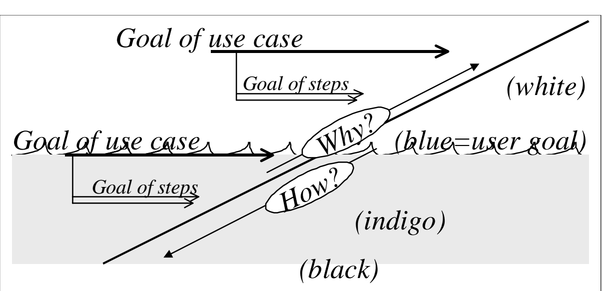
- Review 5.5“Finding the right goal level” on page 75 for the full discussion.
- Make sure the use case is correctly labeled with its goal level: summary, user, or subfunction.
- Periodically review to make sure you know where "sea level" is for your goals, and how far below (or above) sea level the steps are. Recall the tests for sea level goals:
  * It is done by one person, in one place, at one time (2-20 minutes).
  * The actor can go away happily as soon as this goal is completed.
  * The actor (if an employee) can ask for a raise after doing many of these.
- Recall that most use cases have 3-9 steps in the main success scenario, and that the goal level of a step is typically just below the goal level of the use case. If you have more than 9 steps, look for steps to merge in
  * places where the same actor has the ball for several steps in a row, and
  * places where the user’s movements are described. They are typically user interface movements, violating Guideline 5:“It shows the actor’s intent, not movements.” on

page 96

* places where there is a lot of simple back-and-forth between two actors. Ask if they aren't really trying to accomplish something one level higher with all that back-and-forth.

- Ask, "why is the user/actor doing this action?" The answer you get is the next higher goal level. You may be able to use this goal to merge several steps. Review the diagram below to see how goal of steps fit within goals of use cases at different levels.

(Recap of Figure 15."Ask "why" to shift levels" on page 76.)



Reminder 7. Keep the GUI out

Verify that the step you just wrote captures the real intent of the actor, not the just movements in manipulating the user interface. This advice applies when you are writing functional requirements, since, clearly, you can write use cases to document the user interface itself.

In a requirements document, describing the movements of the user in manipulating the interface has three drawbacks:

* the document is needlessly longer;
* the requirements become brittle, meaning that small changes in the user interface design cause a change in the "requirements" (which weren't requirements after all);
* it steals the work of the UI designer, whom you should trust will do a good job.

Most of the use cases in this book should serve as good examples for you. I select this extract of Use Case 20:"Evaluate Work Comp Claim" on page 79 as a representative example:

1. Adjuster reviews and evaluates the reports, ...
2. Adjuster rates the disability using ...
3. Adjuster sums the disability owed, ...
4. Adjuster determines the final settlement range.

I select this as an example of what not to do:

2. The system displays the Login screen with fields for username and password.
3. The user enters username and password and clicks 'OK'.
4. The system verifies name and password.

5. The system displays the Main screen, containing function choices.
6. The user selects a function and clicks ’OK’.

It is very easy to slip into describing the user interface movements, so be on your guard.

Reminder 8. Two endings

Every use case has two possible endings, success and failure.

When an action step calls a sub use case, bear in mind that the called use case could succeed or fail. If the call is in the main success scenario, then the failure is handled in an extension. If it is called from an extension, describe both success and failure handling in the same extension (see, for example, Use Case 22:“Register Loss” on page 83).

You actually have two responsibilities with respect to goal success and failure. The first is to make sure that you deal with failure of every called use case. The second is to make sure that your use case satisfies the interests of every stakeholder, particularly in case the goal fails.

Reminder 9. Stakeholders need guarantees

A use case does *not* only record the publicly visible interactions between the primary actor and the system. If it did only that, it would not make acceptable behavioral requirements. It would only document the user interface.

The system enforces a contractual agreement between stakeholders, one of whom is the primary actor, the others of whom are not there to protect themselves. The use case describes how the system protects all their interests under different circumstances, with the user driving the scenario. It describes the guarantees the system makes to them.

Take the time to name the stakeholders and their interests in each use case. You should find 2-5 stakeholders: the primary actor, the owner of the company, perhaps a regulatory agency, and perhaps someone else. Perhaps it is the testing or maintenance staff who has an interest in the operation of the use case.

Usually, the stakeholders are the same for most use cases, and usually their interests are very much the same across use cases. It soon takes little effort to list their names and interests. Typically, these are the sorts of interests:

* The primary actor's interest is the use case name, it usually is to get something.
* The company's interest is usually to ensure that they don't away with something for free, or that they pay for what they get.
* The regulatory agency's interest usually is to make sure that the company can demonstrate that they followed guidelines, usually that some sort of a log is kept.
* One of the stakeholders typically has an interest in being able to recover from failure in the middle of the transaction, i.e., more logging sorts of interests.

See that the main success scenario and extensions address the stakeholders' interests. This takes very little effort. Read the text of the use case, starting with the main success scenario, and see whether those interests are present. You will be surprised by how often one is missing. Very often, the writer has not thought about the fact that a failure can occur in the middle of the main success scenario, leaving no log or recovery information. Check that all failure handling protects *all* the interests of all the stakeholders.

Often, a new extension condition reveals a missing validation in the main success scenario. On occasion, there are so many validations being performed that the writer moves the set of checks out into a separate writing area, perhaps creating a *business rules* section.

Pete McBreen wrote me on the first time his group listed the stakeholders' interests. They chose a system they had already delivered. They discovered, in that list, all the change requests for the first year of operation of their software. They had successfully built and delivered the system without satisfying certain needs of certain stakeholders. The stakeholders figured this out, of course, and so the change requests came in. What excited this team was that, had they written down the stakeholders and interests early on, they could have avoided (some number of) those change requests. As a result, Pete is a strong advocate of capturing the stakeholders' interests in the use cases. Performing this check takes very little time, and is very revealing for the time spent.

The guarantees section of the template documents how the use case satisfied these interests. You might skimp on writing the guarantees on a less critical, low ceremony project on which the team has good personal communications. You will pay more attention to documenting the guarantees on more critical projects, where potential for damage or cost of misunderstanding is higher. However, in both cases, your team should at least go through the mental exercise of checking both exits of the use case, the success and failure exits.

It is a good strategy to write the guarantees *before* writing the main success scenario, because then you will think of the necessary validations on the first pass, instead of discovering them later and having to go back and change the text.

Read 2.2“Contract between Stakeholders with Interests” on page 40 and 6.2“Minimal Guarantees” on page 89 for more details on these topics.

Reminder 10. Preconditions

The preconditions section of the use case declares its valid operating conditions. The precondition must be something the system can ensure will be true. You document the preconditions because you will not check those conditions again in the use case.

The most common preconditions is that the user is logged on and validated. The other time a precondition is needed is when a second use case picks up the thread of activity part-way through a first use case. The first use case sets up a particular condition that the second relies upon. An

example is when the user selects a product or other partial choice in the first use case, and the second one uses knowledge of that product or choice in its processing.

Whenever I see a precondition, I know there is a higher-level use case in which the precondition gets established.

Reminder 11. Pass/Fail tests for one use case

It is nice when we can find simple pass/fail tests to let us know when we have filled in a part of the use case correctly. Here are the few I have found. All of them should produce a "yes" answer.

Table 20-1: Pass/fail tests for one use case

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

Table 20-1: Pass/fail tests for one use case

Field	Question
Main success scenario.	**15** Does it run from trigger to delivery of the success end condition?
	16 Is the sequence of steps right (does it permit the right variation in sequence)?
	17 Does it have 3 - 9 steps?
Each step in any scenario.	**18** Is it phrased as an goal that succeeds?
	19 Does the process move distinctly forward after successful completion of the step?
	20 Is it clear which actor is operating the goal (who is "kicking the ball?)
	21 Is the intent of the actor clear?
	22 Is the goal level of the step lower than the goal level of the overall use case? Is it, preferably, just a bit below the use case goal level?
	23 Are you sure the step does not describe the user interface design of the system?
	24 Is it clear what information is being passed?
	25 Does the step "validate", as opposed to "checking" a condition?
Extension condition.	**26** Can and must the system detect it?
	27 Must the system handle it?
Technology or Data Variation List.	**28** Are you sure this is not an ordinary behavioral extension to the main success scenario?
Overall use case content.	**29** To the sponsors and users: "Is this what you want?"
	30 To the sponsors and users: "Will you be able to tell, upon delivery, whether you got this?"
	31 To the developers: "Can you implement this?"

21. THE USE CASE SET

Reminder 12. An ever-unfolding story

For a development project, there is one use case at the top of the stack, called something like "Use the ZZZ system". This use case is little more than a table of contents that names the different outermost actors and their highest-level goals. It serves as a designated starting point for anyone looking at the system for the first time. It is optional, since it doesn't have much of a storyline, but most people like to see just one starting place for their reading.

It calls out the *outermost use cases*, which show the summary goals of the outermost primary actors of the system. For a corporate information system, there is typically an external customer, the marketing department, and the IT or security department. These use cases show the interrelationships of the sea-level use cases that define the system. For most readers, the "story" starts with one of these use cases.

The outermost use cases unfold into user-goal or sea level use cases. In the user-goal use cases, the design scope is the system being designed. The steps show the actors and system interacting to deliver the user's immediate goal.

A step in a sea-level use case unfolds into an underwater (indigo, or subfunction) use case if the sub use case is complicated or is used in several places. Subfunction use cases are expensive to maintain, so only use them when you have to. Usually, you will have to create subfunction-level use cases for *Find a Customer*, *Find a Product* and so on.

On occasion, a step in an indigo use case unfolds to another, deeper indigo use case.

The value of viewing the use case set as an ever-unfolding story is that it becomes a "minor" operation to move a complicated section of writing into its own use case, or to fold a simple sub use case back into its calling use case. Each action step can, in principle, be unfolded to become a use case in its own right.

See 10.1"Sub use cases" on page 116.

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

Design scope can cause confusion. People have different ideas of where, exactly, the boundaries of the system are. In particular, be very clear whether you are writing a *business use case* or a *system use case*.

A business use case is one in which the design scope is business operations. The use case is about an actor outside the organization achieving a goal with respect to the organization. The

business use case often contains no mention of technology, since it is concerned with how the business operates.

A system use case is one in which the design scope is the computer system about to be designed. The use case is about an actor achieving a goal against the computer system. It is a use case about technology.

The business use case is often written in white-box form, describing the interactions between the people and departments in the company, while the system use case is almost always written in black-box form. This is usually appropriate because the purpose of most business use cases is to describe how the present or future design of the company works, while the purpose of the system use case is to create requirements for new design. The business use case described the inside of the business, the system use case describes the outside of the computer system.

If you are designing a computer system, you should have both business and system use cases in your collection. The business use case show the context for the system's function, the place of the system in the business.

To reduce confusion, always label the scope of the use case. Consider creating a graphic icon to pictorially illustrate whether it is a business or system use case (see "Using graphical icons to highlight the design scope" on page 49). Consider placing a picture of the system inside its containing system, within the use case itself, to illustrate the scope pictorially (see Use Case 8:"Enter and Update Requests (Joint System) ." on page 52).

Reminder 14. Core values & variations

People keep inventing new use case formats. Experienced writers seem to be coming to a consensus on core values for them. Two papers in 1999 conference [Firesmith99], [Lilly99] described a top dozen or so mistakes made in writing use cases. The mistakes and fixes described in those articles echo the core values.

Core values

Goal-based. Use cases are centered around the goals of the primary actors, and the subgoals of the various actors, including the SuD, in achieving that goal. Each sentence describes a subgoal getting achieved.

Bird's eye view. The use case is written describing the actions as seen by a bird above the scene, or as a play, naming the actors. It is not written from the "inside looking out".

Readable. The ultimate purpose of a use case, or any specification, is to be read by people. If they cannot easily understand it, it does not serve its core purpose. You can increase readability by sacrificing some amount of precision and even accuracy, and make up for the lack with increased conversation. But once you sacrifice readability, your constituents won't read them.

Usable for several purposes. Use cases are a form of behavioral description that can be used for various purposes at various times in a project. They have been used:

* to provide black-box functional requirements.
* to provide requirements to an organization's business process redesign.
* to document an organization's business process (white box).
* to help elicit requirements assertions from users or stakeholders (being discarded afterward, as the team members write the final requirements in some other form).
* to specify the test cases that are to be constructed and run.
* to document the internals of a system (white box).
* to document the behavior of a design or design framework.

Black box requirements. When used as a functional specification technique, the SuD is always treated as a black box. Project teams who have tried writing white-box requirements (guessing what the insides of the system will look like) report that those use cases were hard to read, not very well received, and were brittle, changing as design proceeded.

Alternative paths after main success scenario. Jacobson's original idea of putting alternative courses after the main success scenario keeps showing up as the easiest to read. Putting the branching cases inside the main body of text seems to make the story too hard to read.

Not too much energy spent. Continued fiddling with the use cases does not keep increasing their value. The first draft of the use case brings perhaps half of the value of the use case. Adding to the extensions keeps adding value, but changing the wording of the sentences ceases to improve the communication after a short while. At that point, your energy should go to other things, such as checking the external interfaces, the business rules and so on, all part of the rest of the requirements. This comment about diminishing returns on writing varies, of course, with the criticality of the project.

Suitable variants

Even keeping to core values, a number of acceptable variants have been discovered.

Numbered steps vs. simple paragraphs. Some people number the steps, in order to be able to refer to them in the extensions section. Other people write in simple paragraphs and put the alternatives in similar paragraph form. Both seem to work quite well.

Casual vs. fully dressed. There are times when it is appropriate to allocate a lot of energy to detail the functional requirements, and other moments, even on the same project, when that is not a good use of energy. See 1.2“Your use case is not my use case” on page 20 and 1.5“Manage Your Energy” on page 29. This is true to such an extent that I don't even recommend just one template any more, but always show both the casual and fully dressed templates. Different writers

sometimes prefer one over the other. Each works in its own way. Compare Use Case 25:“Actually Login (casual version)” on page 121 with Use Case 5:“Buy Something (Fully dressed version)” on page 22.

Prior business modeling with vs. without use cases. Some teams like to document or revise the business process before writing the functional requirements for a system. Of those, some choose use cases to describe the business process, and some choose another business process modeling form. From the perspective of the system functional requirements, it does not seem to make much difference which business process modeling notation is chosen.

Use case diagrams vs. actor-goal lists. Some people like actor-goal lists to show the set of use cases being developed, while others prefer use case diagrams. The use case diagram, showing the primary actors and their user-goal use cases, can serve the same purpose as the actor-goal list.

White box use cases vs. collaboration diagrams. There is a near equivalence between white-box use cases and UML's collaboration diagrams. You can think of use cases as textual collaboration diagrams. The difference is that a collaboration diagram does not describe the components' internal actions, which the use case might.

Unsuitable variants

"If" statements inside the main success scenario. If there were only one branching of behavior in a use case, then it would be simpler to put that branching within the main text. However, use cases have many branches, and people lose the thread of the story. Individuals who have used *if* statements report that they soon change to the form with main success scenario followed by extensions.

Sequence diagrams as replacement for use case text. Some software development tools claim to support use cases, because they supply sequence diagrams. While sequence diagrams also show interactions between actors,

- sequence diagrams do not capture internal actions (needed to show how the system protects the interests of the stakeholders);
- sequence diagrams are much harder to read (they are a specialized notation, and they take up a lot more space);
- it is nearly impossible to fit the needed amount of text on the arrows between actors;
- most tools force the writer to hide the text behind a pop-up dialog box, making the story line very hard to follow;
- most tools force the writer to write each alternate path independently, starting over each time from the beginning of the use case. This duplication of effort is tiring, error prone, and is also hard on the reader, who has to detect what difference of behavior is presented in each variation.

Sequence diagrams are not a good form for expressing use cases. People who insist on sequence diagrams, do so in order to get the tool benefit of automated services: cross-referencing, back-and-forth hyperlinks, ability to change names globally. While these services are nice (and lacking in the textual tools currently available), most writers and readers agree they are not sufficient reward for the sacrificed ease of writing and reading.

GUIs in the functional specs. There is a small art to writing the requirements so that the user interface is not specified along with the needed function. This art is not hard to learn, and is worth learning. There is strong consensus not to describe the user interface in the use cases. See 19.6“Advanced example of too much UI” on page 189, and the book by Constantine and Lockwood, Designing Software for Use.

Reminder 15. Quality questions across the use case set

I have only three quality questions that cross the use case set:

- Do the use cases form a story that unfolds from highest level goal to lowest?
- Is there a context-setting, highest level use case at the outermost design scope possible for each primary actor?
- To the sponsors and users: "Is this everything that needs to be developed?"

22. Working on the Use Cases

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

Use cases are only a small part of the total requirements gathering effort, perhaps "chapter 3" of the requirements. They are a central part of that effort, they act as a core or hub that links together data definitions, business rules, user interface design, the business domain model, and so on. But they are not all of the requirements. They are the behavioral requirements.

This has to be mentioned over and over, because such an aura has grown around use cases that some teams try to fit every piece of requirements into a use case, somehow.

Reminder 17. Work breadth first

Work breadth first, not depth first, from lower precision to higher precision. This will help you manage your energy. See Section 1.5"Manage Your Energy" on page 29. Work in this order:

1 **Primary actors**. Collect all the primary actors as the first step in getting your arms around the entire system for a brief while. Most systems are so large that you will soon lose track of everything in it, so it is nice to have the whole system in one place for even a short time. Brainstorm these actors to help you get the most goals on the first round.

2 **Goals**. Listing all the goals of all the primary actors is perhaps the last chance you will have to capture the entire system in one view. Spend quite some time and energy getting this list as complete and correct as you can. The next steps will involve more people, and much more work. Review the list with the users, sponsors, and developers, so they all agree on the priority and understand the system being deployed.

3 **Main success scenario**. The main success scenario is typically short and fairly obvious. This tells the story of what the system delivers. Make sure the writers show how the system works once, before investigating all the ways it can fail.

4 **Failure / Extension conditions**. Capture all the extension conditions before worrying about how to handle them. This is a brainstorming activity, which is quite different than researching and writing the extension handling steps. Also, the list serves as a worksheet for the writers. They can write in spurts with breaks, without worrying about losing their place. People who try to fix each condition as they name them typically never complete the failure list. They run out of energy after writing a few failure scenarios.

Figure 25. Work expands with precision.

<table>
<tr><td rowspan="24">Actor</td><td rowspan="12">Goal</td><td rowspan="6">Success Action</td><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="6">Success Action</td><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="12">Goal</td><td rowspan="6">Success Action</td><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="6">Success Action</td><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td rowspan="3">Failure Condition</td><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
<tr><td>Recovery Action</td></tr>
</table>

5 **Recovery steps**. These are built last of all the use case steps, but surprisingly, new user goals, new actors, and new failure conditions are often discovered during the writing of the recovery steps. Writing the recovery steps is the hardest part of writing use cases, because it forces the writer to confront business policy matters that often stay unmentioned. It is when I discover an obscure business policy, a new actor or use case while writing recovery steps, that I feel a vindication or payoff for the effort of writing them.

6 **Data-fields**. While formally outside the use case writing effort, often the same people have the assignment of expanding data names (such as "customer information") into lists of data fields (see 18.“Use Cases Briefs and eXtremeProgramming” on page 184).

7 **Data-field details & checks**. In some cases, different people write these details and checks at the same time the use case writers are reviewing the use cases. Often, it will be IT technical people who write the field details, while IT business analysts or even users write the use cases. This represents the data formats to the final level of precision. Again, they are outside the use cases proper, but have to be written eventually.

Reminder 18. The 12-step recipe

Step 1: Find the boundaries of the system (Context diagram, In/out list).

Step 2: Brainstorm and list the primary actors. (Actor List)

Step 3: Brainstorm and list the primary actors' goals against the system. (Actor-Goal List)

Step 4: Write the outermost summary level use cases covering all the above.

Step 5: Reconsider & revise the strategic use cases. Add, subtract, merge goals.

Step 6: Pick a use case to expand or write a narrative to get acquainted with the material.

Step 7: Fill in the stakeholders, interests, preconditions and guarantees. Double check them.

Step 8: Write the main success scenario. Check it against the interests and the guarantees.

Step 9: Brainstorm and list possible failure conditions and alternate success conditions.

Step 10: Write how the actors and system should behave in each extension.

Step 11: Break out any sub use case that needs its own space.

Step 12: Start from the top and readjust the use cases. Add, subtract, merge. Double check for completeness, readability, failure conditions.

Reminder 19. Know the cost of mistakes

The cost of lowered quality in the use case depends on your system and project. Some projects need next to no quality in the writing of the requirements document, because they have such good communications between users and developers:

> The Chrysler Comprehensive Compensation project team, building software to pay all of Chrysler's payroll using the "eXtreme Programming" methodology [Beck99], never went further than use case briefs. They wrote so little that they called them "stories" rather than use cases, and wrote each on an index card. Each was really a promise for a conversation between a requirements expert and a developer. Significantly, the team of 14 people sat in two (large) adjacent rooms, and had excellent in-team communications.

The better the internal communications are between your usage experts and developers, the lower the cost of omitting parts of the use case template. People will simply talk to each other and straighten matters out.

If you are working with a distributed development group, a multi-contractor development group, very large development group, or on life-critical systems, then the cost of quality failure is higher. If it is critical to get the system’s functionality correctly written down, then you need to pay close attention to the stakeholders and interests, the preconditions and the minimal guarantees.

Recognize where your project sits along this spectrum. Don't get too worked up over relatively small mistakes on a small, casual projects, but do get rigorous if the consequences of misunderstanding are great.

Reminder 20. Blue jeans preferred

Odd though it may sound, you will typically do less damage if you write too little, compared with too much. When in doubt, write less text, using higher level goals, with less precision, and write in plain story form. Then you will have a short, readable document - which means that people will bother to read it, and will then ask questions. From those questions, you can discover what information is missing.

The opposite strategy fails: if you write a hundred or so, low-level use cases, in great detail, then few or no people will bother to read the document, and you will shut communications on the team, instead of opening them. It is a common fault mistake programmers write at too low of a goal level, so this mistake happens quite often.

A small, true story.

> I helped on a successful 50-person, 15M$ project. We wrote only the main success scenario and a few failures, in a simple paragraph text form. This worked because we had excellent communications. Each requirements writer was teamed with 2-3 designer-programmers. They say next to each other or visited many times each day.

Actually, enhancing the quality of in-team communications helps every project. The teaming strategy described in the just-mentioned project, is the *Holistic Diversity* pattern from Surviving Object-Oriented Projects.

Reminder 21. Handle failures

One of the great values of use cases is naming all the extension conditions. It happens on many projects that there is a moment when the programmer has just written,

```
If <condition>
    then <do this>
    else ..?.
```

She or he stops. "else..?." she (he) muses. "I wonder what the system is supposed to do here? The requirements don't say anything about this condition. I don't have anyone to ask about this odd situation. Oh, well, ...", and then types something quick into the program,

```
    else <do that>
```

The "else" handling was something that should have been in the requirements document. Very often, it involves significant business rules. I often see usage experts huddling together or calling associates to straighten out, "just what should the system do under these circumstances?"

Finding the failure conditions and writing the failure handling often turns up new actors, new goals, and new business rules. Often, these are subtle and require some research, or change the complexity of the system.

If you are only used to writing the success scenario, try capturing the failures and failure handling on your next use cases. You are likely to be surprised and heartened by what you uncover.

Reminder 22. Job titles sooner and later

Job titles are important at the beginning and at the end of the project, but not in the middle. Pay attention to them early and pay attention to them later on, and don't worry about them in the midst of use case writing.

At the beginning of the project, you need to collect all the goals the system must satisfy, and put them into a readable structure. Focusing on the job titles or societal roles that will be affected by the system allows you to brainstorm effectively and make the best first pass at naming the goals. With a long list of goals in hand, the job titles also provide a clustering scheme to make it easier to review and prioritize the nominated goals.

Having the job titles in hand also allows you to characterize the skills and work styles of the different job titles. This information informs your design of the user interface.

Once people start developing the use cases, discussions will surface about how roles overlap. The role names used for primary actor become more generic (e.g., "Orderer") or more distant from the people who will actually use the system, until they are only place-holders to remind the writers that there really is an actor having a goal.

Once the system starts being deployed, the job titles become important again. The development team must

* assign permission levels to specific users, permission to update or perhaps just read each kind of information,
* prepare training materials on the new system, based on the skill sets of the people with those job titles, and which use cases each group will be using,
* package the system for deployment, packaging clusters of use case implementations together.

The job titles, important at the beginning and relatively insignificant in the middle, become important again at the end.

Reminder 23. Actors play roles

Actor means either the job title of the person using the system, or the role that person is playing at the moment of using the system (assuming for the moment it is a *person*). It is not really significant which way we use the term, so don't spend too much energy on the distinction.

The important part is the goal. That says what the system is going to do. Just exactly who calls upon that goal will be negotiated and rearranged quite a lot over the life of the system. When you discover that the Store Manager also can act in the capacity of a Sales Clerk, you can:

* Write in the use case, the primary actor is "Either Sales Clerk or Store Manager" (UML fans, draw the arrow from both actors to the ellipse).
* Write "The Store Manager might be acting in the role of Sales Clerk while executing this use case" (UML fans, draw a generalization arrow from Store Manager to point to Sales Clerk).
* Create an "Orderer" to be the primary actor. Write that a Sales Clerk or Store Manager is acting in the role of Orderer when running this use case (UML fans, draw "generalization" arrows from Sales Clerk and Store Manager to point to Orderer).

None of these is wrong, so you can choose whichever you find communicates to your audience.

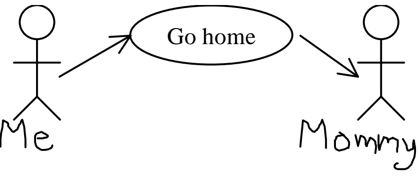
- Recognize that a person fills many roles, a person in a job is just a person filling a role, and that a person with a job title acts in many roles even when acting in the role of that job title.
- Recognize that the important part of the use case is not the primary actor's name, but the goal.
- Recognize that it is useful to settle a convention for your team to use, so that they can be consistent in their use.

Review “Why primary actors are unimportant (and important)” on page 63 and Reminder 22.“Job titles sooner and later” to see how actor names shift to role names and get mapped back to actor names again.

Reminder 24. The Great Drawing Hoax

For reasons that remain a mystery to me, many people have focused on the stick figures and ellipses in use case writing since Jacobson's first book came out, and neglected to notice that use cases are fundamentally a text form. The strong CASE tool industry, which already had graphical but not text modeling tools, seized upon this focus and presented tools that maximized the amount of drawing in use cases. This was not corrected in the OMG's Unified Modeling Language standard, which was written by people experienced in textual use cases. I suspect the strong CASE tool lobby affected OMG efforts. "UML is merely a tool interchange standard", is how it was explained to me on several occasions. Hence, the text that sits behind each ellipse somehow is not part of the standard, and is a local matter for each writer to decide.

Figure 26. "Mommy, I want to go home".



Whatever the causes, we now have a situation in which many people think that the ellipses *are* the use cases, even though the ellipses convey very little information. Experienced developers can be quite sarcastic about this. I thank Andy Hunt and Dave Thomas for this lighthearted spoof, mocking the cartoonish "requirements made easy" view of use cases. From The Pragmatic Programmer, 1999.

It is important to recognize that the ellipses cannot possibly replace the text. The use case diagram is (intentionally) lacking sequencing, data, and receiving actor. It is to be used

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

These are all fine, as described in Reminder 14.“Core values & variations” on page 209. Just remember that use cases are fundamentally a text form, so use the ellipses to augment, not replace the text. The following two figures below show two ways of presenting the context diagram.

Figure 27. Context diagram in ellipse figure form. (Adapted from Booch, Martin, Newkirk, Object-Oriented Design with Applications, Addison-Wesley, 2000.)

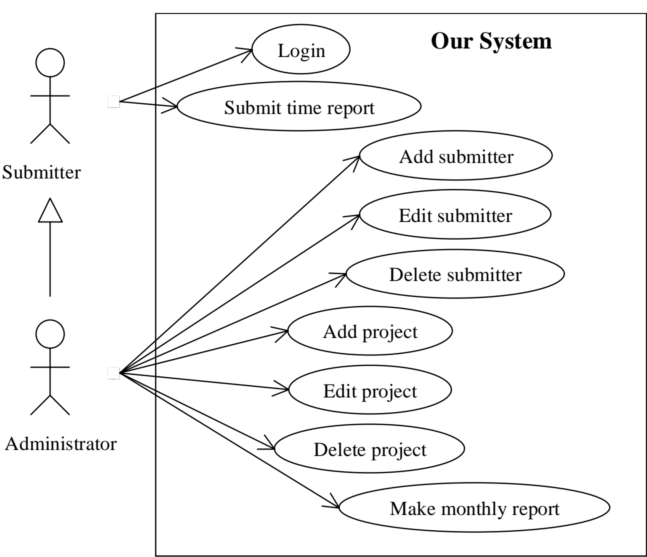


Figure 28. Context diagram in actor-goal format. Same actors and goals, common use cases repeated for clarity.

Table 22-1: Actor-Goal list for context diagram

Actor	Goal
Submitter	Log in
	Submit time report
Administrator	Log in
	Submit time report
	Add submitter
	Edit submitter
	Delete submitter
	Add project
	Edit project
	Delete project
	Make monthly report

Reminder 25. The great tool debate.

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

Lotus Notes. Still my favorite, Lotus Notes has no metamodel of use cases, but supports cooperative work, hyperlinking, common template, document history, quick view-and-edit across the use case set, and easily constructed varieties of sortable views, all significant advantages. It allows the expanding data descriptions to be kept in the same database, but in different views. When you update the template, all use cases in the database get updated. It is fairly easy to set up, and extremely easy to use. I have used Lotus Notes to review over 200 use cases on a fixed-cost project bid, with the sponsoring customers.

The drawback of Lotus Notes, as of any of the plain text tools, is that renumbering steps and extensions while editing a use case soon becomes a nuisance. The hyperlinks eventually become out of date. Manually inserted backlinks become out of date very soon. There are no automated backlinks on the hyperlinks, so you can't tell which higher-level use cases invoke the use case you are looking at.

What makes Lotus Notes most attractive to me is the ease of use combined with the way the annotated actor-goal list is a dynamically generated view of the use case set. Just write a new use case, and the view immediately shows its presence. The view is simultaneously a hyperlinked table of contents, an actor-goal list, and a progress tracking table. I like to view the use cases either by priority, release, state of completion, and title, or by primary actor or subject area, level, and title.

Word processors with hyperlinks. With hyperlinking, word processors finally became viable for use cases. Put the use case template into a template file. Put each use case into its own text file using that template, and it becomes easy to create links across use cases. Just don't change the file's name! Writers are familiar with word processors, and are comfortable using them to write stories.

They have all the drawbacks of Lotus Notes. More significantly, there is no way to list all the use cases, sorted by release or status, and click them open. This means that a separate, overview list has to be constructed and maintained, which means it will soon be out of date. There is no global update mechanism for the template, and so multiple versions of the template tend to accrete over time.

Relational databases. I have seen and heard of several attempts to put the model of actors, goals, and steps, into a relational database such as Microsoft Access. While this is a natural idea, the resulting tools have been awkward enough to use that the use case writers went back to using their word processors.

Requirements management tools. Specialized requirements management tools, such as DOORS or Requisite Pro, are becoming more common. Such tools provide automated forward and

backward hyperlinks, and are intended for text-based requirements descriptions. On the minus side, none that I know of supports the model of main success scenario and extensions that is the heart of use cases. The few use cases I have seen from these sorts of tools are very lengthy, with a great deal of indenting, numbering, and lines, making the use case text hard to read (remember Reminder 2.“Make the use case easy to read.” on page 200, and Reminder 20.“Blue jeans preferred” on page 216). If you are using such a tool, find a way to make the story shine through.

CASE tools. On the plus side, CASE tools support global change to any entity in its metamodel, and automated back links for whatever it links. However, as described earlier, CASE tools tend to be built around boxes and arrows, doing poorly with text. Sequence diagrams are not an acceptable substitute for textual use cases, and most CASE tools offer little more than a dialog box for text entry. I have seen writing teams members mutiny, and revert to word processing, rather than use their CASE tool.

That leaves you with a less than pleasant choice to make. Good luck.

Reminder 26. Project planning using titles and briefs

Review 17.1“Use Cases In Project Organization” on page 164 for the pluses and minuses of using use cases to track the project’s progress, and an example of using the actor goal list as a project planning framework. Here are the reminders.

The use case planning table. Put the actors and goals in the left-most two columns of a table, and in the next columns record any of the following as you need to: business value, complexity, release, team, completeness, performance requirement, external interfaces used, and so on.

Using this table, your team can negotiate over the actual development priority of each use case. They will discuss business need versus technical difficulty, business dependencies and technical dependencies, and come up with a sequencing of development.

Delivering partial use cases As described in “Use cases cross releases” on page 166, you will quite often to decide to deliver only part of a use case in a particular release. Most teams simply use a yellow highlighter or bold text to indicate which portion of a use case is being delivered. You will want to write in the planning table the *first* release in which the use case shows up, and the *final* release, in which will deliver the use case in its entirety.

PART 4
END NOTES

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

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

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

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

APPENDIX A: USE CASES IN UML

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

23.1 Ellipses and Stick Figures

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

The trouble starts when you or your readers believe that the diagrams define the functional requirements for the system. Some people get infatuated with the diagrams, thinking they will make a hard job simple (as in Figure 26.“"Mommy, I want to go home"” on page 219). They try to capture as much as possible in the diagram, hoping, perhaps, that text will never have to be written. Here are two typical events, symptoms of the situation.

> A person in my course recently unrolled a taped-together diagram several feet on a side, with ellipses and arrows going in all directions, *includes* and *extends* and *generalizes* all mixed around (distinguished, of course, only by the little text label on each arrow). He wanted coaching on whether their project was using all the relations correctly, and was unaware it was virtually impossible to understand what his system was supposed to *do*.
>
> Another showed with pride how he had "repaired" the evident defect of diagrams not showing the order in which sub use cases are called. He added yet more arrows to show which sub use case preceded which other, using the UML *preceeds* relation. The result, of course, was an immensely complicated drawing, that took up more space than the equivalent text, and was harder to read. To paraphrase the old saying, he could have put 1,000 readable words in the space of his one unreadable drawing.

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

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

23.2 UML's *Includes* Relation

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

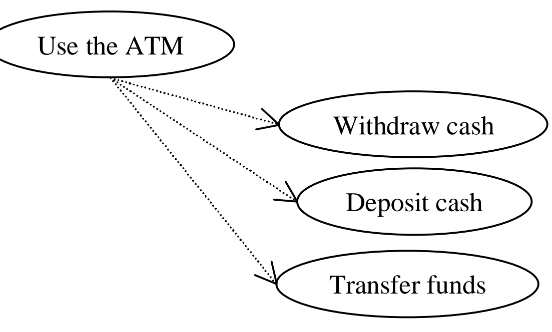
The verb phrase in an action step is potentially the name of a sub use case. If you never break that goal out into its own use case, then it is simply a step. If you do break that goal out into its own use case, then the step *calls* the sub use case (in my vocabulary), or it *includes the behavior of*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

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

Guideline 13: Draw higher goals higher

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

Figure 29. Drawing *Includes*.



UML permits you to change the pictorial representation of each of its elements. I find that most people drawing by hand simply draw a *solid* arrow from base to included use case (drawing dashed ones by hand is tedious). This is fine, and now you can justify it :-). When drawing with a graphics program, you will probably use the shape that comes with the program.

It should be evident to most programmers that the *includes* relation is the old subroutine call from programming languages. This is not a problem or a disgrace, rather, it is a natural use of a natural mechanism, which we use in our daily lives and also in programming. On occasion, it is appropriate to parameterize use cases, pass them function arguments, and even have them return values (see 14."Two Special Use Cases" on page 146). Keep in mind, though, that the purpose of a use case is to communicate with another person, not a CASE tool or a compiler.

23.3 UML's *Extends* Relation

An *extending* or *extension* use case *extends* a *base* use case if the extending one names the base one and under what circumstances it interrupts the base use case. The base use case does not name the extending one. This is useful if you want to have any number of use cases interrupt the base one, and don't want the maintenance nightmare of updating the higher level use case each time a new, interrupting use case is added. See Section 10.2"Extension use cases" on page 116.

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

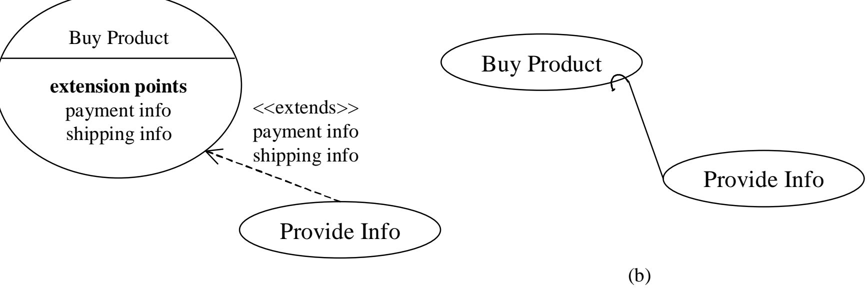
Rebecca Wirfs-Brock colorfully refers to the extending use case as a *patch* on the base use case (programmers should relate to the analogy of program patches!). Other programmers see it as a text version of the mock programming instruction, the *come-from* statement.

We use the extension form quite naturally when writing extension conditions within a use case. An *extension use case* is just the extension condition and handling pulled out and turned into a use case on its own (see*10.2"Extension use cases" on page 116*). Think of an extension use case as a scenario extension that outgrew its use case and was given its own space.

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

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

Figure 30. Drawing *Extends*.



Guideline 14: Draw extending use cases lower

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

Guideline 15: Use different arrow shapes

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

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

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

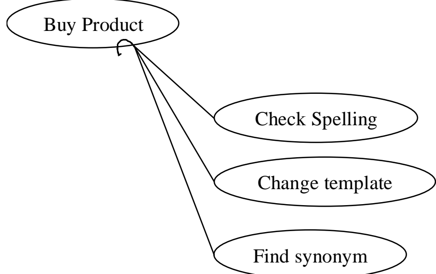
- The standard *generalizes* arrow in UML is the triangle-headed arrow. Use that.
- The default, open-headed arrow should be the frequently used one. Use it for *includes*.
- Create a different arrow for *extends*. I have started using a hook from extending to base use case. Readers like that it is immediately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

Correct use of *extends*

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

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

Figure 31. Three interrupting use cases extending a base use case.



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

Extension points

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

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

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

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

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

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

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

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

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

Of course, you could just say that.

23.4 UML's *Generalizes* Relations

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

Correct use of *generalizes*

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

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

____

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

- Transfer money
- Check balance

Customer does transactions until selecting to quit
4. ATM returns card.
____

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

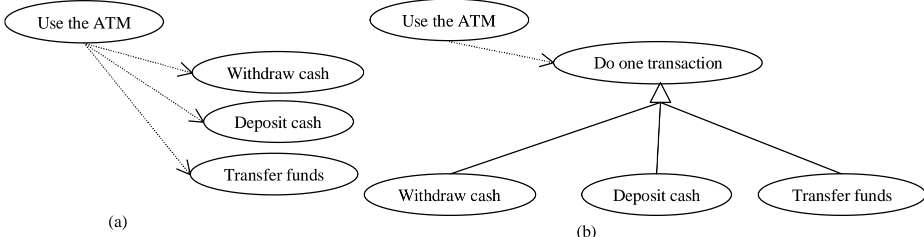
Actually, we have two choices. We can ignore the whole generalizes business, and just *include* the specific operations, as shown in Figure 32.(a). Or, we can create a *general* use case for "Do one ATM transaction", and show the specific operations as specializations of it, as in Figure 32.(b).

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

Guideline 16: Draw generalized goals higher

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

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



Hazards of *generalizes*

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

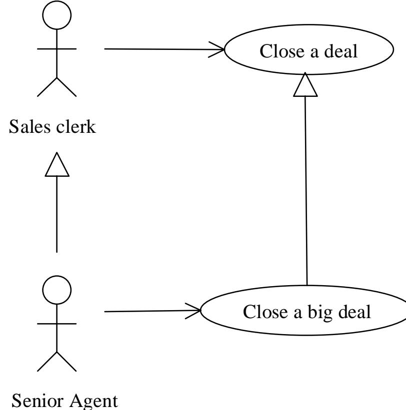


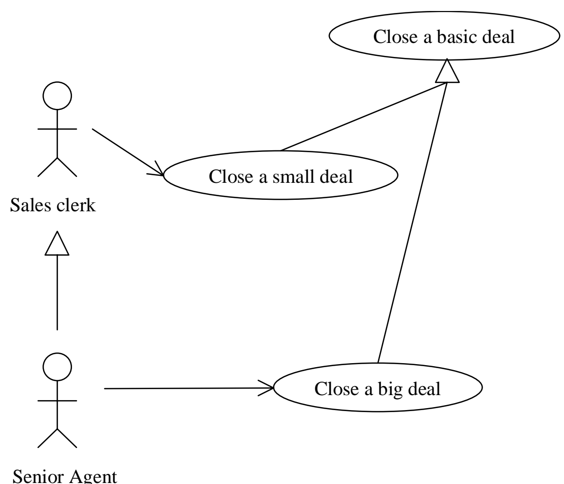
Figure 33. Hazardous generalization, closing a big deal.

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

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

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

Figure 34. Correctly closing a big deal.



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

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

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

23.5 *Subordinate* vs. sub use cases

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

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

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

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

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

23.6 Drawing Use Case Diagrams

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

Guideline 17: User goals in a context diagram

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

Guideline 18: Supporting actors on the right

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

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

23.7 Write text-based use cases instead

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

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

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

24. 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.

25. Appendix D: Reading

Books referenced in the text.

Beck, K., Extreme Programming Explained, Addison-Wesley, 1999.

Cockburn, A., Surviving Object-Oriented Projects, Addison-Wesley, 1998.

Cockburn, A., Software Development as a Cooperative Game, Addison-Wesley, due 2001.

Constantine, L., and Lockwood, L., Software for Use, Addison-Wesley, 1999.

Hohmann, L., GUIs with Glue, in preparation as of 2000.

Roberson, S. and Robertson, R., Managing Requirements, Addison-Wesley, 1999.

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

Articles referenced in the text.

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

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

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

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

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

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

Online resources useful to your quest.

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

http://www.usecases.org

http://members.aol.com/acockburn

http://www.foruse.com

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

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?"