Patterns for Effective Use Cases

The Agile Software Development Series

Cockburn • Highsmith
Series Editors

Foreword by **Craig Larman**

Steve Adolph
Paul Bramble
With Contributions by
Alistair Cockburn
and Andy Pols

Preface

Use cases are a popular requirements modeling technique, yet people often struggle when writing them. They understand the basic concepts of use cases, but find that actually writing useful ones turns out to be harder than one would expect. One factor contributing to this difficulty is that we lack objective criteria to help judge their quality. Many people find it difficult to articulate the qualities of an effective use case.

This book examines the problems people encounter while writing use cases. It describes simple, elegant and proven solutions to the specific problems of writing use cases on real projects. We have identified approximately three-dozen patterns that people can use to evaluate their use cases. We have based these patterns on the observable signs of quality that successful projects tend to exhibit. Our goals are to provide a vocabulary for discussing and sharing these properties with other people, provide advice for writing and organizing use cases effectively, and provide some "diagnostics" for evaluating use cases. We want these patterns to become second nature. Our hope is to find people saying, "Do we have a **SharedClearVision(95)**?" or "Does this use case have **CompleteSingleGoal(132)**?" when discussing their use cases.

Audience

This book is intended for anyone who deals with use cases and wishes to learn more about them. It assumes that you have a working knowledge of use cases, and have some experience writing them. It is the follow-up to Alistair Cockburn's Writing Effective Use Cases. If you are unfamiliar or inexperienced with use cases, then we recommend that you read Alistair's book first.

Use Cases are helpful for designing business processes, software based or not. We do not intend for this book to be "yet another software development book", written in "geek-speak" that is indiscernible to all but the most technically gifted, and readable only by the most determined. Yet, we understand that use cases are predominately a software development tool, and we, being software developers, cannot help but focus on this area. We hope that this book helps all the participants of a project understand the software development community, and we have tried to illustrate this topic in a usable manner for all who work with use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Organization

This book is organized as a catalog of approximately three dozen patterns offering criteria for evaluating the quality of use cases. Each pattern describes a specific guideline or "sign of quality" that you can use to judge the caliber of a use case in a particular area. Because each organization has its own culture and its own way of doing things, use cases that work well for one organization may completely miss the mark in another. Patterns document generalized solutions and as such provide a simple and yet effective mechanism for describing the characteristics of quality use cases. As such, patterns can transcend organizational differences, allowing people to tailor the solution to their specific needs.

We document our patterns using the style of Christopher Alexander, in which we define the problem, the context in which it occurs and a solution. We find this format to be very readable, and is more appropriate for our topic and diverse audience. Pattern names appear in bold type. For example, **VisibleActorIntent(161)** and **EverUnfoldingStory(117)** are both patterns described in this book. We include the page number in parenthesis after the pattern name to help you locate its description.

The patterns are organized into categories. For example, chapter two describes organizational patterns, chapter three describes process patterns, and chapter four describes the use case set patterns.

Each pattern has one or more examples demonstrating either the benefit of implementing the solution recommended by the pattern or the consequences of what happens when you don't. We based as many of these examples on live projects as we could; however, we sanitized them, to protect the companies and people involved (quite rightly), as well as streamlined them for readability. Moreover, we simplified many of them because real use cases are often long and can be quite complicated, especially the ones demonstrating bad practices. We hope that you find these examples useful. You may apply some of our other patterns to these samples, and find ways to improve them.

We illustrate our patterns with a story that runs throughout the book, following a group of developers from a national travel agency as they write some use cases for their new product, the "*Wings Over the World*" travel reservation system. This story portrays an environment in which many of the problems described in the book can occur and therefore provides a background for discussing the patterns. The story also helps us provide examples that succinctly demonstrate a pattern that may be difficult or impossible to illustrate with real examples. While the *Wings Over the World* examples may be contrived, they are based on our experiences, and many of the people, conversations, examples, and even the Red Eye flight are composites of real events, real conversations, and real use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

How to Use this Book

Reading a pattern catalog cover to cover is usually not most people's idea of fun. Your first reading of this book can be either to familiarize yourself with the patterns or serve as a tutorial for what makes a quality use case. Either way, read chapter one, which gives you the background for use cases, patterns, and the pattern categories.

If your objective is just to become familiar with the patterns, simply read the introduction of each subsequent chapter, and then read the problem, context, and solution of each pattern. If you find a pattern interesting, or particularly germane to your specific situation, then read it more closely. Don't feel that you need to fully understand one pattern before you examine another. Once you have this knowledge, you can use the book as a reference. Store it on your shelf within easy reach, so that you can look up a pattern later when you are have a problem writing a use case, or you are reading one, and something just looks wrong.

This book can also be used as a tutorial to understand the signs of quality for a well-written use case. After reading chapter one, skim through each chapter and read the *Wings Over the World* example to get a feel for the environment that gives rise to many of the problems addressed by these patterns. Read the chapter introduction to understand the pattern category and get a brief overview of each of the patterns in that chapter. Skim the patterns themselves, read the problem statement, and the solution. Take a look at the examples and make sure you understand how the pattern improves the quality of the use case or the process to create the use case.

A great approach to learning these patterns is to hold "brown bag" or "lunch and learn" sessions, which many people use to learn patterns. To do this, have someone present one or two patterns at a lunchtime seminar every week. Then discuss the pattern, its intention and its tradeoffs, then look at the examples, and discuss how they compare with your organization's use cases.

What About UML?

The Unified Modeling Language is the software industry's latest attempt to sign a modeling language non-proliferation treaty with itself. When most people think of use cases and UML they simply think of diagramming tools. Others think about UML defining semantics for include and *extends* in UML parlance. Some are UML enthusiasts, and others are not. One thing on which we all agree is that rigidly following UML semantics does not guarantee quality use cases. Many people who have never even heard of UML write great use cases. However, UML is an important influence on use cases and therefore we have included guidelines for the effective use of UML with the relevant patterns.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Why Don't We Use a Single Use Case Template for the Examples?

With the exception of the *Wings Over the World* use cases, most of the use cases presented throughout this book are sanitized versions of real production use cases. Many of these use cases appear different because they are the works of different writers. We could have created a uniform template to present all the examples, but we decided not to because we did not want to advocate a specific use case template. There are enough around now, and just as we did not want to imply there is only one true way to write a use case, we also did not want to imply there is only one true use case template.

We chose to use different styles for our use case diagrams for the same reason, using a variety of different drawing and CASE tools to generate them.

The Agile Software Development Series.

This book is one in a collection of books, the Agile Software Development Series that highlights light, human-powered software development techniques. Some books discuss a single technique, some a single role on the project, and some discuss team collaboration issues.

The book series is based on two common principles:

> 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;
>
> Focusing on skills, communication and community allows the project to be more effective and more agile than focusing on processes and plans.

Accordingly, the book series runs along three main tracks:

> How one person can improve their effectiveness on projects through particular techniques;
>
> How a group of people can improve their combined effectiveness through various group techniques; and
>
> Examples of particular, successful methodologies that you can use as models for your own adaptations.

Agile Software Development elaborates the ideas of software development as a cooperative game, of methodology as coordination of culture, and of methodology families. It separates the different aspects of methodologies, techniques from activities, work products and standards.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Agile Software Development Ecologies discusses problems in software development, principles that are found in common across the diverse experts in the Agile Alliance, and common agile practices.

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.

Guidelines for Writing Effective Use Cases is a handbook for writing high-quality use cases, providing a set of patterns for gauging the quality of use cases, and offering suggestions for improvement when your use cases fail to meet them. You can apply the concepts from this follow-up to Writing Effective Use Cases to all types of use cases, regardless of your organization's style.

Among the other books in the series, Improving Software Organizations and Surviving Object-Oriented Projects [SOOP] are in the line of group techniques, and Crystal Clear [CC] is in the line of sample methodologies.

Pattern Heritage

Paul Bramble began researching the concept of Use Case Patterns in the summer of 1997, after numerous suggestions from his colleague Linda Rising (author of The Pattern Handbook and The Pattern Almanac). He had been defining a generic use case development process for their company, when he realized that use cases are highly dependent on the personalities of the group using them. Recognizing that this research required the input of multiple people and experiences, he co-organized a workshop on use case patterns with Alistair Cockburn at OOPSLA 98 in Vancouver, BC. Eleven participants attended the workshop, and four of them (Paul, Alistair, Andy Pols and Steve Adolph) agreed to continue the research. It took them well over two years to define their use case pattern language, requiring several meetings in the Western United States and Canada, as well as two ChiliPLoP conferences to workshop several of their patterns.

Future Work

We know that there are many more potential patterns out there that people use every day – whether they are conscious of it or not – to create high quality use cases. We sincerely hope this book helps to inspire debate and motivate others to capture these patterns.

Acknowledgements

Copyright © 2001 Adolph, Bramble All Rights Reserved

This book has been a long time in the making. It has seen many rewrites and has been influenced by many people. We would like to thank our families who had to put up with our zany brainstorming and arm-wrestling writing sessions, especially our wives Fariba and Christine, who proofread and edited this text countless times. Also, Elizabeth, Michael, and Rebecca for being patient with their daddy as he wrote on “their” computer.

Alistair Cockburn, Andy Pols, Steve, and Paul jointly put the structure of the pattern language in place in a series of work sessions we loosely termed Bear Plop, Horse Plop, and Bird Plop. Through those sessions, we identified the overall pattern language structure and the initial names, forces, and resolutions for the patterns. Alistair and Andy provided important feedback about the pattern language, as the two of us continued to develop the material for this book.

We are indebted to Linda Rising who not only reviewed the manuscript, but also offered invaluable help on structuring our patterns and was a great source of encouragement. Linda planted the original seeds for this book, and served as unofficial pattern consultant.

The genesis of this work was the Use Case Pattern Workshop at OOPSLA 98. Our thanks to co-leader Greg Gibson, and those who contributed to the founding of this research: John Artim, Greg Gibson, Ted Lefkovitz, Dr. Ben Lieberman, Susan Lilly, Rick Ratliff, Dan Rawsthorne, and Dr. Liping Zhao. The group did an excellent job identifying the various forces involved in writing and using use cases. We were particularly influenced by Rick’s Sans GUI and Breadth First patterns, even though we didn’t use these patterns directly in this book.

Early versions of this pattern language were workshopped at ChiliPLoP, and we thank Neil Harrison and David Delano for shepherding us during this part of its development. Neil contributed his Liberating Form pattern, which while not used in this language certainly helped us to clearly understand the benefit of use cases. Kim Lee also made contributions during the ChiliPLoP workshop, and we are indebted to her for her elegant Borodin Quartet pattern. While the pattern is not part of our language, it helped influence our process patterns.

A special thanks goes to Dan Rawsthorne, who was part of the original OOPSLA sessions and who joined us at the ChiliPLoP sessions. He contributed the sections explaining the application of the patterns to UML, helping us wrap up a lot of loose ends.

Reviewers play an important role in any book, and we had good ones: Linda Rising, Dan Rawsthorne, Pete McBreen, Rusty Walters, and Don Olson. Thanks for the good reviews and the constructive advice!

We are also indebted to the following people:

- Craig Larman for introducing Steve Adolph and Paul Bramble to each other.

Copyright © 2001 Adolph, Bramble All Rights Reserved

- Joshua Kerievsky for turning a two hour flight delay into a wonderful opportunity to understand the Alexandrian pattern form.
- David Roberts for suggesting a Graduate Alternatives pattern which became our PromoteAlternatives pattern.
- Colleagues and friends Gary Wong, and Arsham (Ash) Hamidi who took time to review early drafts.
- The team at Addison-Wesley for their help and patience.
- Lastly, the people who have influenced our careers, including: Dr. Donald Miller and Terry Packer, who helped us believe in ourselves, provided a fertile learning environment and the freedom to follow our passion.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1 What is a Quality Use Case?

The hardest part of building a software system is deciding precisely what to build. - Fredrick Brooks [BROO1987]

1.1 Why Use Cases at All?

"I understand the requirements, but what does it actually do?" is a question often asked by systems analysts, business analysts, product managers and programmers when confronted by two hundred pages of traditional IEEE standard "The system shall…" style functional requirements. After reading these convoluted documents, many of us have often gone back to the clients and users and pleaded, "What do you want this system to do? Tell me a story about how you are going to use this system."

People like stories, and from one point of view, use cases are simply stories about how people (or other things) use a system to perform some task. In this sense, use cases are nothing new; we have always had ways of telling stories about systems. We have used everything from flowcharts, to message traces, to storyboards, to just plain prose. So what are the advantages of a use case?

First, use cases give us a semi formal framework for structuring the stories. Ivar Jacobson gave us the concepts of actors and use cases and rules for how actors and use cases communicate. It wasn't just enough to tell a story, the story had to have a purpose, or in Ivar Jacobson's words, "*yield a result of measurable value to an individual actor of the system*" [JAC1995].

Just as excessive structure and formality can make requirements unusable, so can the complete lack of structure. If everyone is free to tell stories about the system in any manner they feel free to choose, how can you determine if the requirements are correct? How do you find redundant stories? How do you find holes in the stories? Some structure is necessary; otherwise people's creativity will work at cross-purposes.

It is this semi-formal structuring that liberates the creativity of people. Rigid formal requirement models can be stifling, and are unusable to most people because they have not been expertly trained in the appropriate modeling technique. This semi formal structuring makes it relatively easy for the end user of a system to read the document with very little training. This means that the end users are more likely to actually read the requirements document, and are better able to substantiate the system proposal while it is still in the writing stage.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Second, use cases describe the system requirements for the error situations, in every use case and at every level of description. Since much or most of the system complexity lies in handling error situations, this means that the associated difficulties are detected and discussed early, rather than late in the development cycle.

Third, although use cases are essentially a functional decomposition technique, they have become a popular element of object-oriented software development. Several people, including Jacobson [Jac1992] and Larman [Lar2001] describe methodologies for realizing the objects necessary to implement the behavior described by the use case. One can write a set of use cases describing the system's functional behavior and then use these techniques to design the objects necessary to implement that behavior.

Finally, use cases provide good scaffolding on which to hang other project information. The project manager can build estimates and release schedules around them. Data and business rule specifiers can associate their requirements to the use cases in which they are needed. User interface designers can design and link their designs to the relevant use cases. Testers can construct test scenarios from the success and failure conditions described in the use cases. Many modern software development processes are built around use cases.

1.2 What's So Hard About Telling Good Stories?

Writing use cases was supposed to be easy. One reason for their popularity is that a well-written use case is relatively easy to read. People may suppose that easy to read also means easy to write, but that is a mistake. It can be terribly hard to write easy to read stories. Use cases are stories, prose essays, and so bring along all the associated difficulties of story writing in general. As Rusty Walters wrote in [reference], "I did not understand this as the fundamental problem for [the first] four years".

The following example illustrates some very common mistakes encountered by teachers of use case writing. This use case fragment describes the actions a student performs when registering for her courses. It is not a horrible use case; we have all written some like this, but it is a long way from being a good use case.

Use case 1b - Register for courses (main scenario, poorly written version)

1. Display a blank schedule.
2. Display a list of all classes in the following way: The left window lists all the courses in the system in alphabetical order. The lower window displays the times the highlighted course is available. The third window shows all the courses currently in the schedule.
3. Do

Copyright © 2001 Adolph, Bramble All Rights Reserved

4. Student clicks on a course.
5. Update the lower window to show the times the course is available.
6. Student clicks on a course time and then clicks on the "Add Course" button.
7. Check if the Student has the necessary prerequisites and that the course offering is open.
8. If the course is open and the Student has the necessary prerequisites, add the Student to the course. Display the updated schedule showing the new course.

   If no, put up a message, “You are missing the prerequisites. Choose another course”.
9. Mark the course offering as "enrolled" in the schedule.
10. End do when the Student clicks on "Save Schedule"
11. Save the schedule and return to the main selection screen.

Use Case 1-1 Use Case Horror: Example of a Poorly Written Use Case

The problems with this fragment include:

Too much user interface detail. In many poorly written use cases, we often see references to mouse clicks, list boxes and window design. In the normal course of events, the use case is written as a set of *requirements* that the system must satisfy. The user interface design details are not usually requirements, they are usually a design choices. Those design choices are made later, after the use cases have been written and reviewed. The design choices are often changed later, still satisfying the overall requirements. Use case experts universally consider including the user interface design inside the use cases as a *very bad* mistake. It is costly because not only does it adds writing and reviewing time; but it makes the requirements document longer and hence more likely not to be read carefully. Furthermore, it makes the requirements set "brittle," in the sense that small design decisions will invalidate or force an expensive revision of the requirements document. This is the single most critical mistake to avoid. The **Adornments(147)** and **TechnologyNeutral(177)** patterns describe how to steer clear of excessive detail.

Too many use cases at low goal levels. Computer programmers, who often are "stuck" with the job of writing the requirements document, have a tendency to produce numerous low-level use cases on the level of "Authorize user". These writers are very interested in describing individual system functions and features, largely because those are the functions they will have to implement. However, requirements documents written at such a level are very long, and difficult for end users to read. These documents do not show

Copyright © 2001 Adolph, Bramble All Rights Reserved

well what the system will contribute to the lives of the end consumers of the system. The **CompleteSingleGoal(132)** pattern describes how to properly structure use cases to avoid this problem.

Using a use case for non-behavioral information. It happens that a writer is told, "Use cases are great. Write everything in use cases." But a use case is not good for everything, it is really only good for describing behavior. Everything that the system must *do* should really go into a use case, but everything else should really go into some other format. Some use case writers will write immensely detailed use cases describing a user interface form being filled out each field in the form getting one or two lines in the use case. A much better approach is to create an **Adorment(147)** by simply attaching the form to the back of the use case and writing in the appropriate step: "User provides the information on form XYZ (see attachment)." This shortens both the writing and the reading, without sacrificing detail. Performance requirements, complex business rules, data structures, and product line descriptions, are all valuable, but better captured with other requirements tools such as tables, formulae, state machines, or even simply placed in another section of the requirements document.

Too long. The above three common errors produce use cases that are long and hard to read. The main success scenario of a well-written use case is short, usually only three to nine steps long. (Oddly, many people feel embarrassed with such a short start to their use case -- they should not fear, however, as there are usually more than enough extension conditions to make the use case worth writing and reading.) In general, the rule is that shorter is better. The pattern **LeveledSteps(173)** describes how to write balanced, reasonably sized use cases.

Not a complete goal accomplishment. While shorter is better, some use case writers do not capture the complete behavior for goal accomplishment, but only describe a fragment of the necessary behavior. This causes trouble during implementation, since the use cases do not connect to each other, and the programmers have to make guesses as to how to sew them together. A related mistake is not considering all the possible failure conditions or alternative behaviors. Once again, the programmers will discover these in their programming, and will either have to guess at what to program or bring the project to a halt while someone investigates what the system should do. The patterns **CompleteSingleGoal(132)** and **ExhaustiveAlternatives(140)** provide advice on associating goals with use cases and including all necessary failure conditions.

Sentence fragments. A relatively minor, but still noticeable mistake is writing in sentence fragments, as done in the poorly written use case shown in use case 1-1. One could argue that minor writing errors like this don't matter, but on any but the smallest projects

Copyright © 2001 Adolph, Bramble All Rights Reserved

there are many use case writers and readers. Omitting the actors' names in the action steps easily causes confusion over the course of the project, a damage far greater than the cost of simply remembering to write full sentences at the beginning. The pattern **VisibleActorIntent(161)** describes how to write scenarios with clear, unambiguous steps.

1.3 Why a Use Case Pattern Language?

There are no absolute criteria we can use to differentiate between good quality use case and a poor quality use case. Although books have been published to explain use cases, teachers have always had a difficult time saying *why* the good ones were good and what was wrong with the bad ones. To see the difference between good and bad, and the difficulty in identifying what makes the difference, try your hand at comparing this fragment against the poorly written use case in use case 1-1.

Use Case 1a *Register For Course*

1. Student requests a new schedule.

2. The system prepares a blank schedule form and pulls in a list of open and available courses from the Course Catalog System.

3. Student selects primary and alternate courses from the available offerings.

4. For each course, the system verifies that the Student has the necessary prerequisites and adds the Student to the course, marking the Student as "enrolled" in that course in the schedule.

5. When the Student indicates the schedule is complete, the system saves the schedule.

Use Case 1-2 Main Scenario for a well written Use Case

Notice that the well-written use case is much shorter, contains fewer details, and is easier to read than the first one. Yet we cannot simply say, “Write concise, less detailed use cases that are easy to read”. Some problems are long and incredibly complex, involving many details, and as a result yield long, detailed, and somewhat difficult to read use cases, no matter how well written.

To make matters worse, each development organization has its own culture, its own people, and its own way of doing things. What works for one organization may not work for another. This disparity makes it impossible to define a “one size fits all” process that everyone can use to create high quality use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

We want to capture guidelines that can help us write good use cases, and can help us evaluate existing ones. We must find some way to describe these terms so that they are meaningful in different organizations and development cultures.

To counter the common problems in writing use cases and push the result toward well-written use cases, we have constructed and cataloged in this handbook a small set of patterns that gives us a vocabulary for the describing the characteristics of a good quality use case. Put another way these are characteristics that we consider to be signs that quality is present in the writing. Some of these patterns apply to a single sentence in the use case, some apply to a single scenario, some to the set of extensions or to the use case itself. More are needed to discuss multiple use cases and more still to discuss the entire use case set, even the place of a use case in the requirements document. We find that simply describing the use case itself is insufficient, and discussions quickly move from the use cases themselves to the teams writing them and the processes they use for constructing and reviewing the use cases.

The patterns in this language describe the signs of quality about the use cases and the writing process. These signs of quality serve several purposes. They provide a vocabulary for writing use cases, giving people the words they need to express what they want to see, or change, in a set of use cases. While we do not expect these patterns to help the starting writer produce excellent use cases, they can be invaluable for the more experienced writer, offering time-tested advice for improving the quality of their use cases. These patterns are best considered as a diagnostic tool, and should be of great use for reviewing the use case drafts to improve their quality. The absence of any sign indicates that something important is missing because a good set of use cases exhibits all of these patterns.

1.4 What are Patterns

We based our pattern style on Christopher Alexander's work on *pattern languages* [ALE1977, ALE1979] His patterns capture just the information we need, and in a highly readable form. Each pattern indicates *what is present in a good example*, what sorts of thoughts or trade-offs push the writer toward and away from a good result, our recommendation for dealing with them, and examples of these ideas in action.

Alexander, a building architect, recognized common structures in cities, communities, and buildings that he considered to be “alive”. . His research resulted in the creation of what he called a language, one he believed would enable people to design almost any kind of building and community, a language based on the way that people resolved those forces that show up over and over again, in building situations all over the world. He called these reoccurring themes “Patterns”.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Alexander wrote “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you could use this solution a million times over without doing it the same way twice.” [ALE1979] Alexander intended that patterns answer questions such as “where should I place a terrace, how should I design the front entrance, or even how should I organize my community?"

Alexander wrote a 500-page book to describe what a pattern is. The one-sentence version might be, “a pattern is a three part rule which expresses a certain relationship between a certain context, a problem, and a solution.” Like a rule, a pattern has a problem, and then proposes a solution to the problem. The context helps us understand when the solution to the problem is appropriate.

Alexander’s patterns never really caught on in the architectural community, but they became the rage in the software development community around 1995, with the publication of the book Design Patterns: Elements of Reusable Object Oriented Software. This book used patterns to capture good solutions to common problems programmers experienced when designing software. Of course, like any worthwhile idea in technology, shortly thereafter, pattern-hype grew out of control, propelling patterns as the next another magic bullet to all software development.

Our feeling is that it is time to get patterns back to their appropriate place: as signs of quality, and as strategies. When used as signs of quality, a pattern expresses what is present in a well-formed example (of whatever is being discussed). An example in room lighting design is Alexander's *Light on Two Sides of Room* (light enters the room from two sides [ALE1977]. An example in this book, for use cases, is **IntentionRevealingName(144)** (the name of a use case is a verb phrase describing the intention of the primary actor). When used to capture a strategy, a pattern names a way to deal with conflicting pressures. An example in software design from the Design Patterns book is *Visitor* “*…when needing multiple asynchronous traversal over a complex data structure, create a different object for each traversal, which keeps track of its own state in the traversal”* [GAM1995]. An example from software project management is Cockburn's *Gold Rush* (when you don't have time to both capture requirements completely and then design the program carefully, do them in parallel, carefully monitoring communication and rework issues [Cockburn98]).

Patterns as signs of quality are aids for diagnosing, revising and improving a group's work. Patterns as strategies help people thread their way through complex situations. These two contributions of the pattern form should not be lost amid the hype over the term.

Although we like the form of Alexander's pattern writing, it is our experience that people can become confused by the word "pattern." That word often brings to the listener's mind repeating visual structures on ties, carpets and wallpaper. Even today, some people who are familiar with the concept of software patterns think of them as plug-and-play solutions. They fail to realize that

Copyright © 2001 Adolph, Bramble All Rights Reserved

a pattern is often part of a larger, more comprehensive language. In neither case does the reader get a clear indication of what they might encounter in the book, nor do they get the connection to quality that we are after.

Despite these reservations about the word "patterns," we chose to write our handbook using the pattern form rather than as simple heuristics or rules. A guideline or rule says only "do this", without exploring the alternatives and their ramifications, while the pattern form supports that discussion. Equally important, a pattern introduces phrases in a vocabulary, phrases that allow people to shorten long discussions. Often, that phrase is all that is left of the pattern in the person's mind - which is quite fine. People who have read the Design Patterns [GAM1995] book simply say, "*Visitor*," and understand the tradeoffs and issues that entails. We hope the readers of this book will be able to simply say, "**CompleteSingleGoal(132)**", "**ExhaustiveAlternatives(140)**," or "**TwoTierReview(76)**" and similarly understand the corrections and discussions involved.

There is one final relationship between the entries in this handbook and Christopher Alexander's original pattern language: the individual entries do not stand alone, but lead to each other. We keep finding a remarkable similarity between Alexander's discussion of the relationship between cities, single buildings, and building components, and our own discussions of sets of use cases, single use cases, and the components of use cases. We discovered, for example, that considering only the entries below the level of a single use case, we were still missing critical signs of use case quality. We needed to consider the larger level of discussion to find the missing entry (for example, this is how we identified **UserValuedTransactions(110)**). While this similarity between our work and Alexander's is not at all critical to the use of this handbook, we trust that *aficionados* of Christopher Alexander's pattern languages will enjoy the similarities.

1.5 How Should I use this Pattern Language

Patterns can be very beneficial to a project when used correctly. However, it's not always easy to use them in the right way, especially when you don't understand them. Common misconceptions about patterns include:

Patterns offer a complete methodology in and of themselves. Patterns are supplements that fill in the gaps of our knowledge with solutions to specific problems, rather than give us the complete picture. However, some people mistakenly believe that patterns tell them everything they need to know about a subject. For example, some instructors go so far as to base their Object-Oriented Design courses on the book, Design Patterns, instead of a formally defined methodology. However, these patterns offer solutions to real problems encountered in Object-Oriented development, and as such are more diagnostic in nature, i.e. try this solution when you have that problem. [COP1998]

Copyright © 2001 Adolph, Bramble All Rights Reserved

Using patterns guarantees success. In his book Patterns of Software [GAB1996] Richard Gabriel recounts how Christopher Alexander, discovered that several architectural projects using his pattern language failed to produce the better, 'living' buildings he envisioned. Instead, the resulting buildings were no different or any better than other buildings, even though the architects believed that they were radically different. Alexander was convinced "that they failed because the geometry of the buildings was not as different from the standard modern geometry as it needed to be to generate the quality." In other words, in these cases, using his patterns made little if any visible difference. Much of the blame he felt lay in the process. The people controlling the process, the lenders, zoning commissions, etc. were not using the pattern language, yet influenced a lot of control over the project. Gabriel goes on to claim that these findings hold for software development, stating that "The structure of the system follows the structure of the organization that put it together, and to some extent, its quality follows the nature of the process used to produce it."

Patterns offer new solutions to old problems. As Linda Rising says, "Patterns are not theoretical constructs crated in an ivory tower, they are artifacts that have been discovered in more than one existing system." [RIS1998] Patterns are essentially a documentation mechanism that captures general, tried and true solutions to common, re-occurring problems. Accordingly, patterns rarely present new ideas or leading edge research, but rather document solutions that have proved to be effective in many different situations and environments. In fact, experienced people reading a pattern language for the first time should be struck by the feeling that they have seen some of these solutions before.

Patterns are applicable in all situations. A pattern is a solution to a problem within a context [COP1996]. The key word here is context, the idea being that a pattern applies only within a well-defined area. The patterns in this book present solutions that carefully balance several competing forces within the problem space. Sometimes however, a particular force becomes more important and takes on special meaning. For example, an organization writing use cases using company sensitive information might want to hide some details, or even actors, from their customers. Or, a company describing a system that relies on several well-defined and complicated business rules which everyone involved in the project needs to understand might want to include these rules in their use cases. In this case, it might be more important for the company to publish their business rules in their use cases than it is that their use cases be simple and easily understood. In both instances, these organizations need to balance the forces involved, to determine the advantages of following a specific guideline. In these situations, our recommendations are not necessarily the best, and you may need to tailor them to better fit your needs or even ignore them altogether.

Copyright © 2001 Adolph, Bramble All Rights Reserved

So don't think of this pattern language as a complete methodology for writing use cases. Instead use it to fill in the gaps in your knowledge, to evaluate the quality of your use cases, or to augment your particular use case writing process. Treat it as a set of guidelines to help you write better use cases, or to evaluate their quality. Take each of our guidelines with a grain of salt. Evaluate them, and determine if they apply to your use cases and your situation. They will apply in most instances, because they describe common techniques for everyday situations. But the world won't come to an end if you decide not to follow a particular guideline if you feel you have a good reason for avoiding it. Disasters are more likely to occur if you avoid using a guideline you should clearly follow, or try to force one to work when the situation clearly indicates otherwise.

1.6 What is he Use Case Pattern Form?

Pattern aficionados like to refer to the template they use to write patterns as a "form". Like all standards, everyone seems to have their own and of course we're no different. Each one of our patterns is presented using a template or form that is based on Christopher Alexander's form used in his book A Pattern Language. [ALE1977] This form is frequently referred to as the *Alexandrian* form.

Our form includes the following sections:

- The Pattern Name
- A Picture
- The Context
- The Problem
- A Metaphoric Story
- The Forces Affecting the Problem
- The Solution
- Some Examples

1.6.1 Stepping Through a Sample Pattern

The best illustration is the real thing. Consider the pattern **UserValuedTransactions(110)**

Copyright © 2001 Adolph, Bramble All Rights Reserved

UserValuedTransactions(110)

Pattern Na

You have established a **SharedClearVision(95)** of the project and have defined a **ClearCastOfCharacters(105)** who need services from the system.

Contex

A system is deficient if it cannot deliver services that are valuable to its users and does not support the goals and objectives specified by the system vision.

Probler Stateme

A few years ago I bumped into a colleague of mine who was working for a new startup that was building a radio advertising distribution network. The idea was simple; advertisers were using couriers to distribute tapes of their advertisement to radio stations, and even with overnight delivery, it could be two to three days before a new advertisement was actually playing over the radio. My colleague's company had built a continent wide private network to distribute advertising spots to radio stations nearly instantaneously. Furthermore, with their proprietary protocols, and compression hardware they could guarantee quality of service to their clients.

Metaphor Story

It seemed like a license to print money. Offer a service that was cheaper, faster, and more reliable than what was currently available, and the world will beat a path to your door. This approach resembled a page right out of Marketing 101. But the market never developed, and eventually the company succumbed to the inevitable and was taken over.

A while later I mentioned this story to a radio DJ at a local station. "Doesn't surprise me that they failed", he said analyzing the business case, "radio is a local media, almost no one does national campaigns".

A well-written set of use cases clearly and accurately describes the essential actions that a system provides. This information allows customers to preview a system before it is built, and determine whether it offers the kind of services that they find valuable.

Force

A set of use cases should capture the fundamental value added services the users and stakeholders need from the system. An organization commissions the development of a system because that system will bring a benefit to the organization. Use cases allow them to inspect a system before it is built, so that they can verify that it is what they want, request changes, or decide that it doesn't meet their needs. Use cases should describe the kinds of things that users find valuable so they can present the system in its best light. A system that does not deliver needed valuable services to its actors is deficient, can lose money, and will sully the reputation of the development organization.

Copyright © 2001 Adolph, Bramble All Rights Reserved

It is relatively easy to identify low-level transactions while it can be difficult to identify useful services. It is usually easier to describe the individual routine transactions that a system may provide than it is to discover what the user really wants to do with the system. Doing it the easy way often leads to the so-called 'CRUD' (Create, Read, Update, and Delete) style use cases. It is not unusual to see use cases with names like "*Create Employee Record*", "*Read Employee Record*", or "*Delete Employee Record*". While such use cases may be technically correct, they do not capture what is valuable to the user. Is creating an employee record important, or do we they really want to "*Hire Employee*"?

Use cases need to be relatively stable because they form "anchor points" for the rest of the product development process. Constant changes to the use cases can ripple through the rest of the development process, creating havoc for the developers and significantly increasing the cost. To keep this cost low, we want to write each case at a level high enough to insulate it from inconsequential changes, otherwise the writers will constantly be updating their use cases every time someone changes some trivial detail. Worse, the readers will have trouble understanding the use cases, because their meaning is constantly changing.

Readers want to easily see how the system will meet its purposes (see ***SharedClearVision(95)****.* Just like a picture is worth a thousand words, a use case should allow someone to get a quick system overview. But even pictures can be hard to understand when they are too complex or abstract. Concise use cases that stick to the point are easier to read than long flowery ones.

People tend to work at too high or too low a level. People tend to use excessive detail when describing things they understand, or find interesting. Conversely, they tend to gloss over details they don't understand or find boring. Use cases should be somewhere in the middle, containing enough information to adequately describe system behavior, without describing it in great detail ("*what*" versus "how"). If we write them at too high of a level, then they are not useful to the system developers, because they do not describe the system in enough detail. However, if they contain too much detail, then it is difficult for people to understand the system from the 50,000-foot level. In the words of Ian Graham [GRA1997], use cases should only contain necessary but essential information.

Therefore:

Capture the smallest set of goals that identify the valuable services the system must deliver to the actors to satisfy its statements of purpose.

Solutio

Ideally, a set of use cases should contain all of the information necessary to depict a system but no more. Each use case should describe some unique, essential service that is valuable to at least one user or stakeholder.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Use the **ClearCastOfCharacters(105)** and **SharedClearVision(95)** to identify those services that the system should provide. Define as many valuable services as you can for each actor in your cast of characters that help them reach a goal. Being unable to identify any service for an actor may indicate that the actor might not represent a valid system user and you should consider removing them from the cast. Conversely, if you identify a service that doesn't map to an actor in your cast, it may indicate that you have not identified all of the actors.

For each service that you identify, ask "What value does this service provide to the user or stakeholders?" Get rid of those services that fail to add value to the system. You don't want to waste valuable time writing use cases or implementing code for a feature that no one will use or has an interest in.

Users and stakeholders prefer to see the bottom line rather than an itemized list of 'CRUD' style services, so examine each service and determine whether each service stands by itself or is part of a larger, more valuable service. Fold those services that cannot stand by themselves into more comprehensive ones that address one key objective, and eliminate duplicates. A client booking an airline reservation is interested in getting a good flight at a good price. They don't care how many times the system updates its databases or files as the travel agent books a seat.

Write use cases around these goals. While you want to minimize the number of use cases in your collection, each use case should be a cohesive unit that describes one and only one key concept between an actor and the system, a **CompleteSingleGoal(132)**. Describe this collection in sufficient detail to adequately describe its purpose, yet be high level enough so as to be insulated from simple changes.

This singleness of purpose does not prevent a use case from addressing more than one goal, as long as the use case is cohesive and achieves a unified purpose. For example, a high-level use case can reference several subordinate use cases in an **EverUnfoldingStory(117)**, but these use cases must work together to accomplish a common purpose.

Figure 1-1 Sample Pattern

Copyright © 2001 Adolph, Bramble All Rights Reserved

As this example shows, a pattern contains several sections, including:

1.6.1.1 The Name

Each of our patterns begins with a name in bold text, a noun phrase that emphasizes a common characteristic of the solution being proposed. A good name sets the tone for the pattern and should evoke a picture of its solution in your mind. The name becomes part of our vocabulary for discussing the signs of quality that good use cases possess, or those that are missing from a poor use case. For example, a colleague can criticize a use case model because the services offered are too low level and therefore offer no **UserValuedTransactions(110)** value to the users:

> *Jan*: Look Bob, this is not going to help me, use cases called Display Form and Read File don't tell me anything about what this system actually does. These are not **UserValuedTransactions(110)**.
>
> *Bob*: So what are some **UserValuedTransactions(110)** for our users?
>
> *Jan*: How about *Apply for Mortgage*, and *File Supporting Documentation*?

The name of our example, **UserValuedTransactions(110)**, hopefully brings an image to mind of a service that is valuable to someone, the kind of service for which someone will say, "Yes this helps me do my job", or "This is something for which I would pay real money".

Patterns frequently refer to other patterns by name. **UserValuedTransactions(110)** makes several references to **SharedClearVision(95)**, **ClearCastOfCharacters(105)**, **CompleteSingleGoal(132)** and also **EverUnfoldingStory(117)** during the course of its discussion.

1.6.1.2 A Picture

Each pattern contains a picture that is intended to provide a visual metaphor for the pattern. For example, the visual metaphor for **UserValuedTransactions(110)** is a line of people waiting in line for some service. Illustrations provide a nice touch to the pattern as well as underscore the pattern's intent.

1.6.1.3 The Context

A problem doesn't occur in a vacuum. Certain conditions must hold for a problem to be consequential. This section provides the context that makes the problem relevant. It describes

Copyright © 2001 Adolph, Bramble All Rights Reserved

the boundaries that constrain the pattern and the arena in which its solution is pertinent. It also describes how this pattern relates to other patterns in the language [ALE1977] and specifies which, if any patterns are prerequisites to this one.

The context for **UserValuedTransactions(110)** is:

> You have established a **SharedClearVision(95)** of the project and have defined a **ClearCastOfCharacters(105)** who need services from the system.

This statement tells us that we need to know who the actors are before we can find out what they need the system to do for them. It is pointless to start describing the services that a system is supposed to provide, if we don't know who needs them. Before you can identify any **UserValuedTransactions(110)s**, you must understand the project's **SharedClearVision(95)** and know who will be using the system, i.e. its **ClearCastOfCharacters(105)**. If you do not know these, then you cannot possibly determine which services are valuable.

1.6.1.4 The Problem Statement

Each of our patterns describes a problem that people often experience when writing use cases. Our problem statements consist of one or two sentences in bold type that describe what can happen when a use case fails to meet a certain standard and reflect the risks associated with not following that standard.

The problem statement for **UserValuedTransactions(110)** is expressed as:

> **A system is deficient if it cannot deliver services that are valuable to its users and does not support the goals and objectives specified by the system vision.**

It informs you that you need to write use cases that meet the user's real needs if you wish to sell them your system. While this assertion appears to state the obvious, we shall soon see that several factors exist which cause people to write use cases that fail to address the user's need.

We could have written the problem statement as "How do you find the fundamental services a system offers?", except that expressing the problem as a question does not really express the problem's consequences. It does not tell us why the pattern is significant, nor does it describe the consequences of ignoring it. Therefore, we write our problem statements to describe what would happen if your use case model does not follow this guideline.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.6.1.5 The Metaphoric Story.

Some patterns describe simple problems, while other address complex, hard to understand issues. Yet simple and complex are relative terms that depend on the reader's experience. We include either a metaphoric story or a lightweight case study to make the pattern easier to understand and to provide you with an intuitive feel for the problem and its solution. These stories usually have nothing to do with use cases, or even software development, but serve to illustrate the pattern in a practical, easy to understand manner. While analogies are not normally part of the Alexandrian form, we believe they provide a good synopsis of the pattern's intent.

In this example, the metaphoric story describes a company that developed a product that very few customers found valuable. It emphasizes the problem statement by demonstrating what can happen to a product when its intended users don't find it to be particularly useful.

1.6.1.6 The Forces.

This section outlines the various factors comprising the problem, and the tradeoffs between them that complicate and constrain the solution. Pattern writers refer to these tradeoffs as "*forces"* because they will push or pull you in different and sometimes competing directions as you attempt to solve a problem. In our forces section, we describe the important tradeoffs that you, as the use case writer must resolve for the specific situation that you are facing. Each force is written in a specific format. The first statement, written in italics, summarizes the force. The remainder of the paragraph(s) describes the force and its tradeoffs in more detail.

So what are the forces that we are trying to balance when finding **UserValuedTransactions(110)s**? Briefly, they are:

- *A set of use cases should capture the fundamental value added services the users and stake holders need from the system*
- *It is relatively easy to identify low-level transactions while it can be difficult to identify useful services.*
- *Use cases need to be relatively stable because they form "anchor points" for the rest of the product development process*
- *Readers want to easily see how the system will meet its purposes (see **SharedClearVision(95)**.*
- *People have a tendency to work at too high or too low a level*

Copyright © 2001 Adolph, Bramble All Rights Reserved

The first force in this example states the obvious fact that we want to know what basic services the system offers to its users. Yet this information is important, because it helps us grapple with the next force, which if not countered, can lead to us writing ridiculously small use cases. Without other guidance, people will always take the path of least resistance, and it is usually very easy to describe low level system services.

At first glance it might seem absurd that someone would intentionally write use cases that describe how a system offers useless services to its users (perhaps the antithesis of a use case is a useless cases). But this is where the forces come into play. These are the competing tradeoffs that complicate the problem, which if taken too far can lead to sub-optimal solutions. The purpose of a pattern is to provide instructions to bring these forces into an optimal balance for your particular situation. The forces are the things that if taken too far will make us do something absurd, like writing use cases that are useless to the users.

Why do we go through these lengths to explain the problem in this manner? Because we want you to understand the richness of the problem, and gain an in depth understanding of the tradeoffs that push most people to the solution. Few problems have a one-size-fits -all solution, and the better your understanding of the problem's tradeoffs, the better you can adapt the solution to your specific situation, or determine that the solution is not appropriate for your needs.

1.6.1.7 The Solution.

This section presents a common, well-tried solution that balances the competing forces and reflects the characteristics of well-written use cases. The essence of the solution is written in bold text and follows a "therefore" in the pattern. The bold text summarizes the solution that attempts to bring the forces into balance. A discussion follows the solution explaining it further and identifying other patterns that complete this one.

The essential solution for our **UserValuedTransactions(110)** patterns is:

> Therefore:
>
> > **Capture the smallest set of goals that identify the valuable services the system must deliver to the actors to satisfy its statements of purpose.**

This solution aims to solve the problem within the constraints imposed by the forces. "*Capture the smallest set of goals...*" keeps the goal set small because large sets of goals tend to lead to smaller, implementation oriented use cases that are not valuable to the user. "…*that the system must deliver to the primary actor in order to meet its statement of purpose*." Implies that we

Copyright © 2001 Adolph, Bramble All Rights Reserved

should focus on the actors, and define transactions that are meaningful to them, rather than system or implementation details.

The solution may often seem familiar, or even obvious, but that is what patterns are all about. A pattern should not be some new grand revelation but rather a piece of advice that your grandmother could have told you (assuming Grandma wrote use cases, but then, hey, Dilbert's™ mom is apparently a telecommunications engineer). Our patterns capture the tried and true experience of those who have had success writing use cases. Our contribution is that we have given a name to that knowledge, and packaged it with other like pieces of knowledge and experience.

1.6.2 The Examples

Each pattern has one or more examples demonstrating either the benefit of implementing the solution recommended by the pattern or the consequences of what happens when you don't. We based as many of these examples on live projects as we could, however, we sanitized and simplified many of them because real use cases are often long and can be quite complicated, especially the ones demonstrating bad practices. Many of our examples follow a group of developers from a national travel agency as they write some use cases for their new product, the "*Wings Over the World*" travel reservation system. We based these "*Wings*" examples on our experiences and many of them are composites of real people, events, conversations, and use cases.

1.7 Organization of the Pattern Catalog

Our pattern catalog consists of thirty-one patterns, organized into two broad categories of either process patterns or structural patterns. Structural patterns describe the basic components of use cases, how they should be organized, and offer criteria for judging their use. Process patterns describe the characteristics of proven use case writing practices, and offer criteria for measuring the quality of the writing process.

- These two broad categories are further broken down into sub-categories of related patterns.

There are three sub-categories of process patterns:

- team organization, patterns for judging and improving the quality of how the use case team is organized,
- methodology, patterns for judging and improving the quality of the methodology the team follows to create use cases, and

Copyright © 2001 Adolph, Bramble All Rights Reserved

- editing, patterns for judging and improving the quality of the individual use cases as the underlying requirement's change and the writers knowledge grows.

There are four sub-categories of structural patterns:

- use case sets, patterns for judging and improving the quality of a collection of use cases,
- use cases, patterns for judging and improving the quality of an individual use case,
- scenarios and steps, patterns for judging and improving the quality of a use case scenarios, and the steps in the scenario, and
- use case relationships, patterns for judging and improving the quality of the structuring relationships between the use cases in a collection.

Each chapter in the following catalog addresses one sub-category.

1.7.1 Process Patterns

Individuality and organizational cultures make it difficult to define a universal process for writing use cases. Instead, you have to do what "feels right" for your organization. But "feels right" is hard to quantify, as it depends on a host of variable factors. While it is not the purpose of this book to recommend a specific use case writing process, we have identified several good characteristics of effective use case writing processes, and offer guidelines in several areas to help you improve your own process. The process patterns in our language express the characteristics of effective use case writing processes, offering guidelines in several areas to help you improve your own process. These patterns cover three topics: 1) the composition of the teams writing use cases; 2) techniques for creating a set of use cases; 3) techniques for editing existing use cases into better ones.

1.7.1.1 The Team

Group dynamics are an important, but often overlooked aspect of use case development. The personal interactions between the writers can affect the resulting use cases as much as the techniques that are used to identify and write them. This section investigates the people issues associated with use case development, and outlines several techniques for optimizing writing teams, enabling them to produce better use cases.

Writing **PreciseAndReadable(152)** use cases requires both a **BalancedTeam(54)** (balanced skills and personalities) and a **ParticipatingAudience(50)**. The sponsors, developers, usage, and domain experts all contribute to the work and review it. However, too many writers soon spoil the plot, and so a **SmallWritingTeam(47)** should be used for any one writing task.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.7.1.2 The Methodology.

Following the right development methodology is critical to writing quality use cases. This methodology doesn't have to be elegant or "high powered", but it does need to cover all the bases. For developing use cases, good process means balancing discovery versus writing, and content versus need. You don't want to write use cases so quickly that you overwhelm the writers as they struggle to learn the system, nor do you want to be constantly rewriting or discarding your previous work. At the same time, you need to progress at a reasonably quick pace, so that your developers can begin building the system. You only want enough content to adequately describe your system, but you don't want to waste time writing any more than that. This section investigates process issues, and offers some advice for improving yours.

The **SmallWritingTeam(47)** integrates its work with a **TwoTierReview(76)**, where an inner circle of colleagues representing different specialties first reviews and adjusts the work, before passing it to a large group with representatives from all stakeholders, including customers.

While we do not advocate any specific process for creating use cases, we find that effective groups work **BreadthBeforeDepth(63)**, naming many use cases before partially expanding some, and completing the main success scenario before investigating failure handling, achieving a **SpiralDevelopment(66)** of the use case set.

The effective team understands when it is **QuittingTime(71)**. Rather than getting bogged down in long arguments about cosmetic issues, they allow a certain amount of **WritersLicense(80)**, recognizing that trying to enforce identical writing habits or petty standards soon stops adding economic value to the endeavor.

Not every project team needs the same volume of detail to accomplish its mission, and so we see the need for **MultipleForms(83)** of use cases. Indeed, these forms may each find its appropriate moment on the same project!

1.7.1.3 Editing.

Use cases can become prematurely outdated because the underlying requirements are highly unstable and subject to change. Use cases are highly dynamic, and will metamorphose as your understanding of the system evolves. Behavior that made sense at the start of the writing process may no longer make sense as you discover more about the system through research and talking to customers, resulting in a collection of obsolete or fragmented use cases. This section describes several common techniques that people use to improve the quality of their use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

During the writing process, the team will periodically find themselves either with large, complex and hard to read use cases, or lots of small, insignificant ones. They should **RedistributeTheWealth(206)** of the large ones to smaller ones, and **MergeDroplets(211)**, folding the too-small ones into others. They may eventually discover that some are simply irrelevant; those they can **CleanHouse(216)**.

1.7.2 Structural Patterns

We have identified four basic levels of use case structure: Sets of Use Cases, Use Cases, Scenarios, and Steps. Use case sets describe the behavior of a system and consist of individual use cases, each of which describes some useful service an individual actor needs. Each use case is a collection of scenarios which when taken together describe all the different ways an actor can either reach or fail to reach a specific goal. Individual scenarios consist of steps, each describing an action that an actor or the system must take to move the primary actor closer to their goal.

Use cases often interact with other use cases in the same set. We have identified patterns for structuring some of these relationships. These relationship patterns describe techniques for handling repetitive or excessively complex behavior.

1.7.2.1 Use Case Sets

Use Case Sets are collections of use cases and related information, organized in a usable manner as a use case model. A set contains system level information about a product, including its actors, its boundaries and the relationships between its members. This level is primarily organizational, as it describes key characteristics of the collection rather than specific behavior. People working at this level often refer to individual use cases by name and ignore their contents.

The most important thing about use cases as a set is that they should reflect a **SharedClearVision(95)** for a system with a clear and **VisibleBoundary(101)**. The use cases are collectively structured with higher-level use cases referencing lower-level use cases in an **EverUnfoldingStory(117)** that shows a **ClearCastOfCharacters(105)** interacting with the system to achieve their goals. While the goals that get described sit at various levels, the crucial and interesting ones describe a **UserValuedTransactions(110)**, in which the primary actor accomplishes a goal that he views as a primary service of the system under discussion.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.7.2.2 Use Cases

An individual use case illustrates how actors can use a system to meet a particular goal, showing all of the appropriate paths that they might take to get there, as well as those situations that could cause them to fail. This level is still organizational in nature, providing order and structure so that the reader is able to easily identify and follow the different paths through the use case as they trace the actor's progress towards their goal. It also serves a focal point for related material.

Each use case contains a collection of successful and unsuccessful scenarios that describe the various situations that an actor is likely to encounter when attempting to achieve their goal. The failure cases are especially important, because they describe the various error conditions that can happen and the actions necessary to resolve them.

A single use case describes the pursuit of a **CompleteSingleGoal(132)**, and should have a descriptive **IntentionRevealingName(144)** that gives the reader an idea of its purpose. Each use case structures the multiple ways it can achieve or abandon its goal as a **ScenarioPlusFragments(136)**, with a collection of scenario fragments describing what happens under differing conditions. A complete use case considers **ExhaustiveAlternatives(140)**, so the developers are not surprised with an unexpected situation late in development.

To serve its purpose of satisfying the sponsor, the users, the developers, and the writers strive to make the use case **PreciseAndReadable(152)**, one of the arts of use case writing, and an achievable one. One aspect of this is to remove performance requirements, data formats and ideas for the user interface from the use case text, and document them separately as **Adornments(147)**. This practice keeps the use case robust with respect to shifting technologies and user interface designs, yet clean of unnecessary, confusing clutter.

1.7.2.3 Scenarios and Steps

Scenarios describe a single and complete sequence of events within a use case that an actor follows as they attempt to achieve a particular goal, and results in either success or failure. While scenarios describe behavior, they are still somewhat organizational in nature because they provide structure to a series of steps, which combine to form a coherent and purposeful description. This provides the reader with a systematic view of a particular action sequence.

Each scenario fragment after the main success scenario describes the behavior of the actors under some **DetectableConditions(168)** (detectable to the system under discussion). Part of the readability of attractive use cases is **LeveledSteps(173)**, keeping all the steps at about the same level of detail.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Steps describe single actions within a scenario and detail the interchange between the system and actor as they act out a use case. Each step, depending on its goal level, adds some amount of clarity to its scenario, and represents a singular action that either the system or the actor takes as they interact.

Each step should make distinct **ForwardProgress(164)** toward the goal. Since the user interface details and other design decisions appear as **Adornments(147)**, you can write each step in a **TechnologyNeutral(177)** manner, to the greatest extent possible. Lastly, each step should make the **VisibleActorIntent(161)**, so that the readers always can tell who is doing what.

1.7.2.4 Relationships

Use cases occasionally share some common behavior, and when they do, it is efficient to re-use existing text rather than repeating the same sequence of events each time they are needed. Ivar Jacobson defined the concepts of includes, generalizes, and extends to handle these situations. Unfortunately, everyone seems to have their own ideas as to what these terms mean. This section describes how people successfully use these concepts to improve their use cases.

People have developed a variety of overly complex mechanisms for using the *includes*, *extends*, and *generalizes* relationships. Some of these mechanisms work well, others just make a confusing situation worse. Simplicity seems to be the best course. The simplest and most natural relationship is to move the **CommonSubBehavior(186)** to a sub-use case referenced by the others via the '*includes'* relationship when a common set of actions reoccurs in several use cases. When a single event can interrupt the flow of a use case multiple times, then the writers should document those **InterruptsAsExtensions(191)**. If a given alternative begins to dominate the use case then you should consider a **PromoteAlternative(196)**, promoting that alternative to an extension use case. We have not seen enough good examples of generalizes to have found a pattern for it.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.8 Supplement: A Brief Tutorial on Writing Use Cases

Let's start our tutorial by saying that anything that has behavior is an *actor*. This convention allows us to refer equally easily to people, computer programs and companies, without worrying about which category of actor is playing the role at that moment. A use case, then, describes the way in which a particular system under discussion (SuD), an actor in its own right, interacts with other actors.

To describe the many complicated interactions that a system will have over its lifetime, we link any one use case with the goal of an actor who wants something from the SuD at that moment, and describe all the ways that the system may come to deliver or abandon the goal of that "primary actor".

Then we structure the writing in an interesting way: First of all, we describe how the actors behave in a simple situation in which the goal gets achieved. After that, we name all the conditions under which the behavior is different, and describe the different behavior that ensues, always bearing in mind that sometimes the goal will succeed and sometimes it will fail. These are called *extensions* or *alternate courses* of behavior within the use case.

We can see now that the use cases above were just fragments, since they only described the simple case of goal success (what some people call the "happy day" scenario). The complete use case is too long to put in here, but looks essentially like this:

Copyright © 2001 Adolph, Bramble All Rights Reserved

Use case - *Register for courses* (use case with extensions)

Primary actor: Student *System under Discussion:* Course Enrollment System

Level: User Goal.

1. Student requests to construct a schedule.

2. The system prepares a blank schedule form

3. The system pulls in a list of open and available courses from the Course Catalog System.

4. Student selects up to 4 primary course offerings and 2 alternate course offerings from the available offerings.

5. For each course, the system verifies that the Student has the necessary prerequisites and adds the Student to the course, marking the Student as "enrolled" in that course in the schedule.

6. When the Student indicates the schedule is complete, the system saves the schedule.

Extensions:

1a. Student already has a schedule:

System brings up the current version of the Student's schedule for editing instead of creating a new one.

1b. Current semester is closed and next semester is not yet open:

System lets Student look at existing schedules, but not create new ones.

3a. Course Catalog System does not respond:

The system notifies the Student and terminates the use case.

4a. Course full or Student has not fulfilled all prerequisites:

System disables selection of that course and notifies the Student.

Use Case 1-3 *Register for courses* **(use case with extensions)**

People sometimes find that a briefer way of writing is desirable (e.g., very small projects, and project in which the use cases are merely being used to estimate work effort, not specify the system). In other situations, a more exact way of writing is needed (military contract outsourcing, large distributed projects, life critical systems). It is important to recognize that there is no one, best format for use cases, but that the amount of detail to put into a use case depends on the project and the team at hand, and the purpose to which the use case writing is to be put.

Copyright © 2001 Adolph, Bramble All Rights Reserved

While it is all very well to say that a use case describes the behavior involved in trying to achieve a goal, the difficulty is that goals exist at many levels. Any one goal is achieved by achieving sub-goals. For example, I might have a goal to have my children attend school in a rich section of the city. To achieve that goal, I have to earn enough money to buy a house in that school district. To do that, I need to close some big business deals, so my next sub-goal is to win some particular contract. To do that, I may decide my next sub-goal is to win over a particular decision maker, and so I take him to lunch for a discussion.

Each of these goals can be described using a use case, although I have not yet specified a particular SuD for them. Continuing the sub-goals, I find I need cash to take him to lunch, and so I go to a local cash-dispensing machine, where my goal is to get some cash. My first subgoal, now directly related to the cash machine as a system under discussion, is to identify myself, to which end I have subgoals to enter my card into the machine, type in my identification number and hit the Enter key. (People who know Alistair know that he can make almost any goal end up in going to an ATM to get cash!).

All in all, I could write any of the following use cases within the framework of the information given so far: Find enter key, Authorize user, Enter ATM card, Get cash, Win contract, Buy a too-expensive house, Get kids into good school.

This capacity of use cases to describe goals at all levels is wonderful, but confusing to use case writers. Different writers describe different levels of goals, some staying high (Get cash), and some staying at terribly low levels (Authorize user, Enter ATM card). For most systems, it is critical to identify the goal level in which the system contributes direct, take-home value to the primary actor (see the entry **UserValuedTransactions(110)**. This we refer to as “User Goal” (see Figure 1-2). Then write additional use cases for higher and lower level goals as needed. These we refer to as “Summary” and “Subfunction”, respectively. Figure 1-2, adapted from Writing Effective Use Cases [COC2001], illustrates goal levels by describing some of the deals necessary to get the kids into a better school.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-2

Goal levels for sending you kids to a better school

The use cases are read and used by two very different groups of people: the end users or business experts, who often are not versed in the technical and implementation difficulties, and the programmers, who need very precise answers to their questions in order to make the computer program work properly. It is not obvious that any form of writing can satisfy both groups of people, but use cases have shown themselves as fitting this narrow range. The art of use case writing is to get the precision in writing without overwhelming the non-programmer business reader.

To accomplish this task of getting the use cases correct and precise but still readable, the writing team *must* contain at least one person with a background in programming to get the required accuracy and precision of description, plus at least one person with deep knowledge of the business rules that the system must enforce, plus at least one person with intimate knowledge of how the system will actually be used. In other words, producing a set of use cases is not a job for one person, or even people with the same job description. It is a team effort, requiring people

Copyright © 2001 Adolph, Bramble All Rights Reserved

with different backgrounds and even personalities. When this team does its job well, the result is readable *and* precise, and suits its purpose well.

This book is not an introduction to use cases, rather it is a handbook about how to write meaningful, high-quality ones. Therefore, we are providing only a brief summary of use cases in this section. If you want to learn more about use cases in general, we recommend Alistair Cockburn's *Writing Effective Use Cases* [COC2001]. More discussion of use cases is available at the web site *www.usecases.org.* You may refer to these sources for introductory, template, and tools-descriptive material, and continue reading this book to improve your ability to detect and discuss the signs of quality in your use case writing.

Copyright © 2001 Adolph, Bramble All Rights Reserved

2 The Team

All Hands On Deck!

It's 5:00pm on a sunny West Coast afternoon. You're just closing the last of your files and powering down your workstation, when you boss calls you over to her office and tells you, "Good news, we won the '*Wings Over the World*' contract".

This is great news. *Wings Over the World* is a national travel agency that is modernizing its aging software infrastructure. This project is potentially worth millions to your company and you are the principal consultant on the bid. You might finally make full partner now.

The boss shakes your hand. "You did well; they really liked your presentation, and they want you to take point on this one".

Your boss continues, "They're really excited about this use case stuff you told them about, and they want to make this a "real all hands effort". You hear the sharp "sproing" of a red flag snapping up. "They really want to fast track this, so they've lined up about a dozen people for their use case team and they want you to be part of it". Before you can open your month, your boss cautions you, "Oh, by the way, most of their travel agents haven't completely bought into the new system concept yet, and they don't want us disturbing them, so you are only to work with their business analysts and programmers in their IT group while gathering the requirements."

Now you hear a rapid succession of "sproings" as more red flags snap up. Things are not looking good; you have a large and probably unfocused team. The team is developer heavy with few domain experts and no actual users. Finally, you have little or no access to users and other important stakeholders, so you can only guess at what they really want. Use cases were never intended to solve these problems.

2.1 Team organizational issues

Human behavior is an important, yet often overlooked aspect of use case development. The personal interactions between the writers can affect the resulting use cases as much as the techniques used to identify and write them.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Unfortunately many development organizations seem oblivious to the influence team organization has on creating good use cases. Instead, they treat the effort as a necessary evil that anyone, experienced or otherwise, can do. Management may not understand or support the concept, or the developers themselves might consider this task to be busy work that they must do before they can start on the "real work" of writing code. Even when an organization embraces use cases enthusiastically, it can fail to dedicate the resources necessary to do a good job.

Team size is the most critical factor influencing the quality of the use cases. Not only is it inefficient to have numerous people defining the system, the compromises made to align numerous and competing system visions may lead to unsatisfactory results. Each use case writing team should be a **SmallWritingTeam(47)**, where the number of writers on any use case is limited to two or three people to prevent inefficiencies or design by committee

While you want to keep the use case writing team small, you still want to involve as many of stakeholders in the writing process as possible to ensure the use cases meet all of your stakeholders' needs. However, talking with external customers can present an enormous challenge to a project. Some companies don't want outsiders to access their rank-and-file employees, and even internal organizational boundaries can get in the way, to the detriment of the final product. **ParticipatingAudience(50)** resolves the competing issues of managing team size while including all of the target audience in the use case development process.

Team composition is another factor influencing the quality of use cases. An unbalanced team, consisting of similar, like-minded individuals, will examine a system from only one point of view, frequently that of the software developers rather than that of the users. Unbalanced teams are often inefficient, and can wander in search of an elusive goal. Including a domain expert or expert user can keep the team on track, but it can also inhibit a team from original thinking when the expert is overpoweringly dominant. Effective teams are **BalancedTeam(54)s**, containing a complimentary mixture of skills, expertise, and even personalities, where each participant provides a distinct view of the system being modeled.

Copyright © 2001 Adolph, Bramble All Rights Reserved

2.2 SmallWritingTeam(47)

<missing caption>

You have a **SharedClearVision(95)** for a system and are organizing teams to write use cases.

Using too many people to write a use case is inefficient, and the compromises made to align the many different points of view may result in a less than satisfactory system.

Musicians have a saying, “Two guitarists, three songs”, implying that anytime two people are playing guitars together, they somehow end up simultaneously playing three different songs. Their point is that musicians, especially during practice sessions, like to reward their band mates with impromptu solos unrelated to what anyone else is playing. The more instrumentalists in the group, the more likely these “treats”, as the musicians compete to play their favorite song. Many groups limit their number of guitarists to two or three to avoid these problems.

You can experience similar problems when writing use cases, because everyone writing them wants to contribute their two cents worth and be heard.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Use cases require the views and expertise of diverse groups. A set of use cases must capture the domain knowledge and experience of the people who will use the system, yet be precise enough for the development team to use as an input for system design. Large systems can be especially difficult to describe, because they can support many different kinds of users performing many different kinds of activities. In some situations, the number of people who will be needed to properly contribute information and review the use cases can grow as large as 15 or even 50 people.

Getting a large group of people together is difficult. The more people that must meet, then the more difficult it is to find an agreeable time for everyone to meet. People who work on other projects or work in other geographic locations can't just pop down the hall for a quick discussion. Meeting with external customers presents the biggest challenge, because their company may discourage such meetings or forbid them outright.

In theory, the more people you have working on use cases, the faster you can finish writing them. Throwing a lot of people at a problem is a common reaction to meeting tight schedule pressure. It is especially tempting with use cases because many people think that writing them is easy, and that anyone can do them. This approach works well when there is more work than people, but can make things much worse when a team is already adequately staffed.

Teams that are too large become inefficient. Teams having too many members tend to bog down over minor matters. Over-staffing a team can waste its members' time by minimizing their participation and giving them too little to do. As a result, the team either loses interest and slacks off, or they tend to overproduce, creating an artifact with too much detail, and redundant or unnecessary information.

Large teams can lead to design by committee and feature bloat. Large teams are especially vulnerable to design by committee where the project assumes a life of its own. "Group think" prevails and the team loses focus and strays from the vision, resulting in a model bearing little resemblance to the intended system. Members add all kinds of unnecessary features, resulting in a model that fulfills all of the team members' desires without satisfying anyone.

Therefore:

Restrict the number of people refining any one work product to just two or three people. Use a TwoTierReview(76) process to include more people

Copyright © 2001 Adolph, Bramble All Rights Reserved

The fewer the number of people who are responsible for the creation of the use cases the better. It is easier for a small team to meet, communicate and reach a consensus. A team of two or three people is small enough to keep out of each other's way, yet broad enough to write meaningful use cases. You can use several **SmallWritingTeam(47)s**, however, you should appoint one person as the use case architect and give them the responsibility for maintaining a consistent vision through all the use cases.

At the same time, try to represent different viewpoints on each team. Create **BalancedTeam(54)s**, using people with different backgrounds, skills, and personalities so that the teams don't fall into the rut of looking at the use cases the same way. Diversity is especially important for projects aimed at a widespread audience, because it is likely that the various users will use the product in many different ways.

You can maximize participation, while avoiding large teams, by using **ParticipatingAudience(50)** and **TwoTierReview(76)s** to involve more people in the process. These two patterns describe methods for consulting many different people, without cluttering the use case writing process. These practices allow different interest groups to focus on the details they consider important, while shielding them from details raised by other stakeholders with different interests.

Examples

Facilitating Large Groups: No Design without Representation

In his article "Use Case Blue", Andy Kraus [KRAU1996] describes the effort involved facilitating a large group that was collecting use cases. He recounts six weeks of sessions, using an average of 15 to 20 participants, eliciting requirements for replacing the Automated Regional Justice Information System (ARJIS). It required the services of various people from 23 police departments and 16 individual contributors. While smaller teams might have been more desirable, business conditions and the complexity of the system forced them to use the large group approach. During the course of their endeavor, they held meetings with eight to 25 people in attendance. While fewer people would have been more manageable, Andy claims "Only by having representation of the diverse ARJIS membership in the room at the same time could we flush out nuances to the requirements caused by the differing philosophies and operating procedures among the members." But such an effort took its toll, as Andy also claimed that they encountered all of the problems that they were told to expect from having such a large group. For example, "without knowing in advance what the real system actors were, it was impossible for us to predict which users would need to come to which sessions", which they solved by holding a kickoff planning session to derive a schedule for future sessions.

Copyright © 2001 Adolph, Bramble All Rights Reserved

This effort was successful, but required a lot of effort to manage. However, some groups find that this type of approach requires too much energy to be effective.

Large Groups and Frequent Checkpoints

We know an expert in the field who was involved in a similar situation with a company that had arranged for an expert facilitator and 14 people (ten field representatives and four business analysts) to write use cases for an extended period of time, eventually yielding around three dozen use cases. Things started out slowly, as the group had to listen to people speak their opinion one at a time.

Fortunately, the group had the flexibility to adjust their procedure as needed. They devised a branch and join process, using their group meetings for brainstorming sessions and administrative issues. They divided into small groups, based on whom was best qualified to work on what use case. If only one person was qualified for a particular topic, he got someone to help. Once a group finished a draft of a use case, they passed it around for review, with the others penciling their comments on the same sheet. These reviews were especially effective because they localized all of the reviewer's comments to one sheet of paper, keeping the reviewers aware of the previous reviewer's thoughts, and making it easier to organize and apply their suggestions to the use cases once the review was complete.

The facilitator was somewhat uncomfortable with this distributed approach. However, the group consistently chose to do the work in small teams, as they found it much easier and far more efficient. Moreover, they greatly increased their throughput and decreased the duration of the effort by working on seven use cases at a time rather than one.

Copyright © 2001 Adolph, Bramble All Rights Reserved

2.3 ParticipatingAudience(50)

<<missing caption>

You are a **SmallWritingTeam(47)** creating use cases.

You cannot satisfy stakeholders' needs without their input and feedback.

Buying stock in a company makes you a part owner, and entitles you to attend stockholder meetings and vote on company issues. The company's board of directors is legally required to represent their shareholders' best interest, which can be difficult, because companies may have millions of stockholders. It is impossible for the board to listen to everyone. To alleviate this problem, companies allow their stockholders to vote by proxy, where the stockholders can assign their right to vote to someone else. This way, one person can represent hundreds or even thousands of people at the meeting, keeping it small. However, a stockholder who disagrees with the direction the company is going can attend the meeting and voice their sentiments.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Writing use cases is similar because it is not always possible or practical to involve everyone you need every step of the way. Stakeholder input is invaluable because your goal is to develop a system that pleases your stakeholders, and they often know far more about their needs than you can possibly ever know. To effectively model the system, you must discover what they need, know, and want, and allow them to have a voice in the process. The true art is to know when and how to do this without having to involve them in every step along the way.

Many different groups have a vested interest in a set of use cases. Developers are not the only people depending on a set of use cases. Large development organizations consist of many different groups with different responsibilities, styles, and needs. Each of these different groups relies on the same use cases to complete their portion of the product development process. Each one of these groups has its own use for the use cases and looks at the use cases differently.

If the customers don't like it, they won't buy it. Ultimately, customers are the final authority on your product, because they are the ones who have to pay for it. While you may think that your product is fabulous, they may hate it. They don't care how hard it was for you to build, or how much time you spent building it, they just care that it does what they want. If your product doesn't meet their needs, then it is a failure.

It is usually impractical to include your entire target audience in the use case development process. Several factors make it difficult to talk to everyone interested in your product. Large projects require the services of far more people than you can consult. Often, stakeholders reside in different geographical areas, making them extremely difficult to locate or talk with. Other companies may not like you talking with their employees, or your own company may want to keep your project under wraps for competitive reasons. In the case of off-the-shelf products, you don't really know who your stakeholders are, and can only guess.

Development organizations don't adequately represent end users. A common problem in software development is that the developers often assume that the users share their perspective of the system. Unfortunately, developers are often isolated from their customers, and don't really understand their needs. As a result, the systems that they develop are often hard for everyday users to use because they target the wrong audience. For example, a group of high-powered developers might prefer to use an interface requiring regular expressions as input. However, regular expressions are too abstract for most people, who would become extremely frustrated after trying to use this interface. Even other highly skilled users might not appreciate the more

Copyright © 2001 Adolph, Bramble All Rights Reserved

difficult interface, because they want an easy-to-use product that doesn't require much thought or a large learning curve.

Therefore:

Actively involve your customers and internal stakeholders in the use case development process when possible.

Include your customers in the use case development process from the start. If you are developing use cases as an individual effort, consult with your customer often and seek their input. If working as a team, then try to include a customer representative as a team member when you don't have a clear understanding of the product. Their presence provides you with a wonderful opportunity to learn firsthand what it is that they really want.

This kind of relationship makes it easier for you to keep your customers aware of any potential issues that may arise, and obtain their assistance immediately, rather than through a time-consuming formal process. Most important, cooperation can foster a spirit of understanding as well as give them a sense of ownership in the use cases, improving the chances that they will accept the final product.

While the customer is the most important stakeholder, they are not the only one. Internal groups such as quality assurance and documentation often depend on your use cases, and it is important that you keep them in the loop and consult with them as well. Treat all stakeholders as customers, and consider including internal stakeholders on your use case development teams when they can provide valuable insight into the product.

Try to represent the user's viewpoint when they are unavailable. Sometimes the users are inaccessible. Many organizations will not allow outside companies access to their employees, and even internally departmental boundaries can limit access. Their reasons may be valid or frivolous, but don't fret or give up when you can't talk directly to the users. Instead, use other techniques to elicit this information. The book Software for Use [CON1999] describes many practical techniques for obtaining user information without directly contacting users. These techniques include utilizing user surrogates such as domain experts and trainers, consulting with members of non-user groups such as marketing or technical support, or relying on indirect sources such as manuals, records, or even questionnaires. While not as straightforward as direct user participation, these techniques offer a valid alternative.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Reviews, especially well structured **TwoTierReview(76)s** that make good use of the customer's time, provide a good vehicle for sharing use cases with customers, and can even allow you access to those customers and users that you can't see otherwise.

Examples

Soliciting customer input has been a key part of Requirement's Elicitation for years. Yet surprisingly, some teams don't do it. We know of one team that focused entirely on some complex requirement specifications, didn't talk to the end users, and ended up writing over 500 use cases before they threw in the towel. The project was shelved.

The Job Flow Engine

We find it amazing that people don't take the time to investigate the user's side of a project, because this technique is so helpful. One of the authors was part of a small project team developing a job flow engine to control the execution of some specialized programs across a set of distributed work stations. The team interviewed many potential users, discovered what they needed and documented as use cases their various processes, both manual and automated. The resulting product was successful because it provided exactly the needed services, and allowed the users to correctly automate some complicated time-consuming processes. Without the use cases, the result wouldn't have been nearly so positive.

It should come as no surprise that soliciting customer input is a common practice in many industries, and was well established long before the first modern computer.

Making Movies

Customer consultation has been part of the movie making process for decades. MGM sponsored the Marx Brothers on cross-country Vaudeville tours prior to filming A Night at the Opera and A Day at the Races, who used the audience as proving grounds for much of their material. If a joke bombed, they either changed it until it worked, or they dropped it. They fine-tuned those lines that were somewhat funny, and kept those skits that brought down the house for their films. Consequently, these two films rank among their all time best. However, MGM stopped sponsoring these tours after A Day in the Races, and the quality of the Marx's subsequent films dropped significantly. [MAR1972 pp.174-175]

To The Moon

In some instances, the **SmallWritingTeam(47)** absolutely depends upon audience participation. For example, consider a trip to the moon. The Apollo Saturn V spacecraft contained over two

Copyright © 2001 Adolph, Bramble All Rights Reserved

million separate working systems, yet severe weight and fuel constraints prevented NASA from sending more than three people on the spacecraft. Obviously, there was no way that three people could comprehend that many systems, let alone be proficient enough to troubleshoot them when problems arose. To get around this problem, NASA assembled a large group of experts, simulators, computers, and communications equipment to assist the flight from the ground. Likewise, some use cases can be so complex that they cannot be written without vital information from the audience.

Copyright © 2001 Adolph, Bramble All Rights Reserved

2.4 BalancedTeam(54)

Production. C-47 transport planes. These women mechanics at the Long Beach, California, plant of Douglas Aircraft Company are learning some of the fine points of the mighty engines that power the C-47 transport plane. The versatile C-47 performs many important tasks for the Army. It ferries men and cargo across the oceans and mountains, tows gliders and brings paratroopers and their equipment to scenes of action.

Palmer, Alfred T., photographer.

CREATED/PUBLISHED
1942 Oct.

You are organizing **SmallWritingTeam(47)s** to write use cases.

Using teams of similar, like-minded individuals to develop use cases can result in a set of limited, narrowly ranged use cases that do not satisfy everyone's needs.

Copyright © 2001 Adolph, Bramble All Rights Reserved

A musical quartet consists of four people singing different parts that blend together to create a unified sound. You would defeat the purpose of a quartet if you chose four singers to sing the same part. Instead, you would have singers with different ranges who could harmonize together, because it is this arrangement that gives a quartet its distinct and rich sound.

Writing meaningful use cases requires the creative input of several people having different viewpoints, otherwise the results tend to be flat and narrowly focused. A homogeneous team often fails to appreciate the other stakeholders' needs, and produces inadequate use cases as a result.

Groups of individuals sharing a common technical background tend to focus on the same limited issues. This tendency is not surprising, as each member is trained to consider the same issues and use the same techniques to solve a problem. While their approach provides satisfactory solutions, it doesn't encourage the out-of-the-box thinking needed at this stage of product development. Although the members of this group are usually more specialized than most of their clientele, they often fail to think about the system from their user's viewpoint, which can be dangerous for a number of reasons.

Each profession uses its own specialized vocabulary, which people outside of that profession do not understand. This tendency results in documents that can be very confusing for people with other backgrounds. Even if an outsider understands the generalities of this terminology, they usually fail to grasp its subtle implications and can overlook or misinterpret its key points.

Teams need different expertise at different moments. Teams often need subject matter experts at the start of a new project. Later, as they become more familiar with the subject, they need them less, but they may need to know more about other technical areas as the project evolves to address the needs of new and different users.

Good teams are balanced so that each team member's specialties compensate for some other team member's weakness. Teams often fail without this balance. For example, athletic teams are highly specialized. Consider a championship baseball team, which usually contains great pitchers, solid hitters, and players with good defensive skills. Very few, if any, of the pitchers are expected to be good hitters, as their job is to challenge the opposing team's players and prevent them from hitting the ball. It is the job of the other eight players on the team to backup the pitcher by making good defensive plays and to score runs. Teams that are strong in all three of these areas routinely beat mediocre teams with one or two exceptional players.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Similarly, a team containing members with complementary skills is better suited for defining use cases that will support many different user types, because they are more likely to address more issues. This balance is especially critical in today's world with its explosive technical growth, where no one individual can even begin to keep up with the latest technologies. Software development improves as we improve people's personal skills and improve the team's collaboration effectiveness.

Therefore:

Staff the team with people from different specialties to champion the interests of the stakeholders in the development process. Make sure the team contains both developers *and* end users.

Ignore the tendency to use only your best programmers or software engineers for writing all of the use cases. While domain experts and expert users can be essential for the more critical use cases, be careful about putting them on a team that they will dominate. Instead, use a healthy mix of articulate people with varying backgrounds and viewpoints. If you are writing use cases for several different organizations, try to represent the more important ones on some of your teams, especially when most of your potential system's users are non-technical, and use **ParticipatingAudience(50)** to fill in the gaps. If your product has significant legal issues, consider involving someone from your legal department, and so forth. Try to include people who think in low-level details as well as those who function at the big picture level. Look for abstract thinkers and problem solvers, artists and technicians. The team must be diverse enough to provide combined field, corporate and technical viewpoints.

Try to include people who are good at maintaining focus and who know when to quit to prevent feature creep and help the **SmallWritingTeam(47)s** avoid diving into bottomless pits. Otherwise, the more detail-oriented people will inevitably write low-level use cases that attempt to resolve the system's technical issues rather than describe its behavior and how people will likely use it.

Examples

Lack of Diversity

Many organizations fail to achieve a satisfactory level of diversity because they simply assign software developers to write use cases. Unfortunately, this approach often yields use cases that are too detail-oriented, or ones that describe solutions rather than behavior.

Copyright © 2001 Adolph, Bramble All Rights Reserved

One of the authors participated in a group that was writing essential use cases (high-level use cases capturing the essence of a system's behavior) describing a voice messaging dictionary system. Within a matter of minutes, the team began identifying low-level use cases describing the interactions between the dictionary's database attributes. When told that these issues were too technical in nature to be considered "essential", the team disagreed, stating that this information was indeed high level. These people weren't novices; all of them were experienced and knowledgeable use case writers, but they couldn't help themselves as they were problem solvers by nature, and honestly believed that they were operating at an abstract level.

The point is, teams of highly specialized teams tend to produce use cases like this:

USE CASE 1: *Provision a Cross-Connect* (ENGINEERING CENTRIC VERSION)

Primary Actor:	**Network Administrator**: Person provisioning **Network Elements**.
Secondary Actor:	**Network Element** Device being provisioned.
Level:	**User Goal**
Preconditions:	Network Element is accessible and user has a valid account on it.
Success Guarantee:	Cross-Connect is "active".
Main Course:	1) The use case begins when the **Network Administrator** activates account on **Network Element** (ACT-USER:TID:USERID:::PASSWORD). 2) The **Network Element** establishes a session. 3) The **Network Administrator** enters ENT-STSnC:TID:FROMAID::::IS; (or ENT-T3:TID:AID::::IS; if it is on a DS3) to establish the source STS. 4) The **Network Element** establishes the "From AID" STS. 5) The **Network Administrator** enters ENT-STSnC:TID:TOAID::::IS; (or ENT-T3:TID:AID::::IS; if it is on a DS3) to establish the sink STS. 6) The **Network Element** establishes the "To AID" STS. 7) The **Network Administrator** enters ENT-CRS-STSnC:TID::FROMAID,TOAID:::; to establish the cross connection. 8) The system establishes the cross connection.

Copyright © 2001 Adolph, Bramble All Rights Reserved

9) The user logs off of the **Network Element** (CANC-USER:TID:userid:;

10) The use case terminates when the **Network Element** terminates the session.

Use Case 2-1 Use Case Horror: *Provision a Cross-Connect* **(ENGINEERING CENTRIC VERSION**

However, this use case is confusing because it contains fragments of TL1 code, which is a special language for interacting with telecommunication equipment. Most software developers would find this use case perfectly acceptable, especially if they didn't know TL1, because it would tell them how to provision the device. Unfortunately, it leaves those stakeholders who don't understand TL1 in the dark. Worse, the use case is only valid for TL1 implementations, and isn't general enough if you suddenly have to use other protocols. The following version is much more general and easier to understand:

USE CASE 1*: Provision a Cross-Connect* (USER FRIENDLY VERSION)

Primary Actor: **Network Administrator**: Person provisioning **Network Elements**.

Secondary Actor: **Network Element** Device being provisioned.

Level: **User Goal**

Preconditions: Network Element is accessible and user has a valid account on it.

Success Guarantee: Cross-Connect is provisioned and viable.

Main Course:

1) The use case begins when the **Network Administrator** logs on to the **Network Element**.

2) The **Network Element** verifies the user and begins a session.

3) The **Network Administrator** determines the size of the cross-connection, and sets up the endpoints accordingly.

4) The **Network Element** establishes the endpoints.

5) The **Network Administrator** defines the cross-connection.

6) The **Network Element** establishes the cross-connection.

7) The user logs off of the **Network Element**.

Copyright © 2001 Adolph, Bramble All Rights Reserved

8) The use case terminates when the **Network Element** terminates the session.

Use Case 2-2 *Provision a Cross-Connect* **(USER FRIENDLY VERSION)**

This version is better because it is easier to read, and supports multiple protocols. You could still include the TL1 samples for the implementers, but as **Adornments(147)** (Chapter 5) at the end of the use case, or in a separate document.

Copyright © 2001 Adolph, Bramble All Rights Reserved

2.5 Tradeoffs and Collaborations

Using small balanced writing teams and controlling audience participation is an easy way to improve the quality of your use cases. While not difficult, organizing such teams does require forethought and balancing several competing forces involving team size, audience involvement, and skills. There is no one perfect mix that always works and it is up to each development organization to balance these patterns to determine what team arrangements will work best with their particular circumstances.

We cannot say enough about the importance of involving your customers in the use case development process. One of the most significant benefits of use cases is that they model how the system delivers measurable value to its users. Being unable to talk to the customer, whatever the reason, increases your risk, so you should find some other way to gather this information. Larry Constantine offers excellent advice on dealing with these kinds of circumstances in his book "Software for Use" [CON1999].

While these three patterns complement each other well (see Figure 2-1), they do create some tension with **ParticipatingAudience(50)** and **BalancedTeam(54)** trying to enlarge the group and **SmallWritingTeam(47)** trying to shrink it. However, the intention behind **SmallWritingTeam(47)** is to keep the process manageable. **ParticipatingAudience(50)** provides a mechanism for many other people to contribute to the use case process without overwhelming it and therefore avoids the problems of excessively large teams. **ParticipatingAudience(50)** also helps a group become a **BalancedTeam(54)** by involving people with different backgrounds and viewpoints to influence the use cases without increasing the development team's size. The diversity these two patterns provide helps steer the teams towards using a common vocabulary, rather than a discipline specific one.

Figure 2-1

Relationships between the Team Organization Patterns

Copyright © 2001 Adolph, Bramble All Rights Reserved

To illustrate these points, consider what kinds of teams would be best suited for writing use cases for a project involving pharmaceutical manufacturing. These kinds of systems must meet exacting standards for purity and precision, as well as rigorous legal and accounting requirements. A pharmaceutical company manufacturing controlled substances must be able to account for 100% of the controlled substances they use, or they face unwanted visits from the Drug Enforcement Agency.

First, you would want to set up **SmallWritingTeam(47)s**, because you want the writers to exercise tight control over their use cases, and write with a high degree of precision. But, you also need a high degree of **ParticipatingAudience(50)**, probably more than usual, to verify that the resulting product meets all required standards and complies with the law. In this case, **BalancedTeam(54)** has the greatest impact on the results because of the complexity of the problem and the number of technologies involved. Most likely, you would want to staff your teams with combinations of software developers, pharmacists, biochemists, and even lawyers, depending on the needs of the various use cases.

The biggest benefit of using **BalancedTeam(54)** in this instance is that it forces the writers to use a common, understandable terminology in their use cases. Commonality is especially important in this situation because each of the disciplines involved in developing this product has its own exclusive and highly specialized vocabulary. **ParticipatingAudience(50)** helps counter-balance this problem to some degree, but even then one audience will likely have trouble with another's terminology, which is often incomprehensible to those outside of the profession. Software developers talk about stacks, queues, XML, Bridges and Visitors, while the pharmacists would describe chemical reactions such as "de-hydro-halogenation" and use complicated chemical formulas. Lawyers also have a reputation for obtuse writing; imagine reading use cases like "the Line Supervisor actor shall henceforth be know as the party of the third part", etc.

While important, team organization is only a small part of the picture. A team also needs to follow a good process to be able to identify and write quality use cases, which is the subject of the next chapter.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1 The Process

Just Enough To Be Dangerous

It's Monday morning and you've had a great weekend with the family. On Friday afternoon your boss told you that you would lead the use case effort for your new client, the national travel agency, "*Wings over the World*". Now you're having your first phone conversation with Karen, Wing's Chief Information Officer, since doing your original presentation.

You: "I just got the news on Friday that we're going ahead with this. Can you tell me what you have done since my presentation three weeks ago?"

Karen: "Well, I was really impressed with your explanation of how use cases will help us out on this project. We've always had a lot of problems gathering our functional requirements and this sounds like just the ticket for us. I wanted to get my people started quickly, so I brought in an instructor for a couple of days to teach a use case course. All I can say is, people are really excited by this. Anyway, the instructor gave them a great template that covers everything you can think of in a use case, from soup to nuts. I've got everyone on the team now working on the use cases and they're just turning these things out a mile a minute."

Once again you hear that familiar "sproing" sound of red flags popping up. Templates are wonderful you think, but can they ever be misused and abused. After a short pause, Karen confirms they are having problems.

Karen: "I have a big favor to ask you. I know you're not scheduled to be here until next week but is it possible that you could come sooner?"

More "sproings".

Karen: The way you explained the use cases, and the way our course instructor explained them, we thought they would be easy to write. But, obviously we're doing something wrong because we're running into some problems. We're really anxious to get you in here to help us out."

You: "What kind of problems?"

Karen: "Well you know it's no secret that we've had methodology problems on past projects. There are a lot of cowboys here. Anyway, Ahmed, our architect on this project is really insistent that we do things right this time, and that means we go by the book and follow the use case template that our instructor gave us. Well, people are having all kinds of problems filling in the pre-conditions, post conditions, triggers, goal statements and all that. While most of us are really happy to be trying out these use cases, some of the old hands are getting a bit frustrated. I was hoping you could maybe go over all the fields in the template?"

Copyright © 2001 Adolph, Bramble All Rights Reserved

You don't know whether to laugh or cry. But right now you know that last thing you want to do is sound condescending.

You: "It's a bit early in the process to be writing detailed use cases; we should just try to get the big picture first, then refine the use cases."

Karen: "Well, Ahmed said we needed detailed specs, otherwise there was just too much that would be left to the programmers. But I think I have to agree with you on this one, because Ralph and Sitra got into a big fight last week."

You: "A fight? Over what? I thought they worked well together."

Karen: "Not really a fight, more of a programmer's territorial dispute. Ralph is our billing expert and Sitra is our RAPIER expert. Ralph wrote a billing use case that overlaps and in some situations contradicts Sitra's use case for the RAPIER interface. Both of them have spent about two solid weeks writing their use cases and neither wants to throw their work away."

You: "What about the agent reps, what do they think about all this so far?"

Karen: "I'm a little bit worried about their buy in to this project because all they've done is complain. They've always complained that our specs are too complex or that they can't find what they're looking for. Well, all right, we're trying to go the distance for them by writing use cases and they're still complaining. Not only that, but some of travel agent's reps are even refusing to show up for our JAD sessions and reviews. They keep saying there are too many reviews and that they've got better things to do than correct our spelling."

You: "If I can swing it I'll try to hop tonight's Redeye and be there tomorrow and we can sort this out."

Karen: "I really appreciate your dedication. I knew there was a reason we selected you. Oh, and one last thing…"

You: "What is it?"

Karen: "Well, when do we know we're done? Ahmed is really insisting that we go by the book, and Ralph is really getting impatient and saying we should just get on with the job."

After you hang up, you immediately call travel and have them book your tickets for tonight's Redeye. You decide you better get there now and get this project back on track before these guys dig themselves any deeper into the hole they have started.

Wouldn't it be wonderful if there was one well-defined process for writing use cases? Then we could all learn the one universal methodology and know that everyone else clearly understands our ideas. Unfortunately, writing good use cases is highly personalized; everyone has their own style, and every organization has its own way of doing things, based on its culture and business

Copyright © 2001 Adolph, Bramble All Rights Reserved

needs. After all, it is these differences that give a business their competitive advantage. What is good for one group is not necessarily good for another.

While there is no one-size-fits-all process for creating use cases, we have captured common elements of good processes in the process patterns. It is important to note that these patterns do not constitute a process, i.e. they don't follow one after the other. Rather, they make statements that are true about the process. For example, **TwoTierReview(76)** describes inner reviews followed by an outer review. This pattern will be found in organizations using slightly different processes for writing their use cases.

An effective process creates use cases in a **BreadthBeforeDepth(63)** manner, identifying potential use cases before describing them, adding details in a controlled manner and achieving a **SpiralDevelopment(66)** of the use case set. One of the biggest mistakes that people make while developing use cases is writing them sequentially. They start developing one use case, taking whatever time they need to write it, and don't start on the next one until the first one is finished. This approach has two drawbacks: 1) it is easy to get bogged down on difficult use cases at the expense of the remaining ones; and 2) facts uncovered during the development of later use cases often necessitate a rewrite of the earlier ones, wasting the original effort.

Different groups within a development organization should have the freedom to choose the format and level of detail for their use cases that best meets their needs, instead of being required to use some "official" style. An effective process allows the writers to choose from **MultipleForms(83)**, and determine the appropriate level of detail. The group should use the same format for all the use cases in the same project to avoid confusion, but it is their decision to make, based on the project's complexity and the audience's need.

One of the most difficult questions involving use case development is when to stop. The effective team knows when it is **QuittingTime(71)**. Rather than getting bogged down in long arguments about cosmetic issues, they allow a certain amount of **WritersLicense(80)**, recognizing that trying to enforce identical writing habits on everyone doesn't add economic value to the endeavor. Reviews are a better way of identifying errors and inadequacies, however, a traditional review process can often be ineffective for use cases because they can require a large, diverse group of reviewers with competing needs. To get around this problem, **TwoTierReview(76)** describes how to stage reviews, in which those closest to the use cases review their correctness and completeness before turning them over to the larger group to review from a system standpoint.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.1 BreadthBeforeDepth(63)

Blasting crew going into ore vein with horizontal drill. Mahoning pit, Hibbing, Minnesota.

Vachon, John, 1914-1975, photographer.

CREATED/PUBLISHED
1941 Aug.

You are starting to write use cases with a **SharedClearVision(95)** and a **ClearCastOfCharacters(105)**.

You will not make timely progress or create coherent use case sets if you waste energy writing detailed use cases sequentially.

Its hard to imagine building a house without an architect's sketch or drawing that shows us the big picture of how the completed house will look, and how it will relates and influence its surroundings. The architect may also provide interior sketches to help us visualize the interior of the house. From these sketches an architect will begin to create the detailed plans for the house.

Copyright © 2001 Adolph, Bramble All Rights Reserved

But homes have not always been designed in such a breadth first manner. Back in the "good old days", there was no big picture, and people living on the frontier literally built their houses one room at a time. Materials were scarce, and people didn't have the time to build large homes, so they just made do. Later, as they prospered and their families grew, they added more rooms onto the house, again one at a time. The results of this process are interesting; if you visit one of these homes today you will notice unusual features such as exterior walls on the inside, or bedrooms you can only enter through other bedrooms. The tour guides at Abraham Lincoln's house in Springfield, Illinois will tell you that the second floor "floats"; it is not anchored to any supporting walls because someone added it on later.

You can write better use cases if you have a good idea of what they will look like when you are finished. Unfortunately, you usually don't know that much about a system when you start. Developing use cases is a process in which you are continually learning and reevaluating your model. Fortunately, there are some things that you can do to help guide you along the way, and provide order to your process.

Requirements gathering is a process of discovery. It starts with a vague notion of the desired system, which makes it impossible to get the details correct the first time around. In most cases the people who commission the software system do not know exactly what they want and are unable to articulate their needs. The use case writing process is an iterative one in which you must define, analyze and refine your underlying requirements in order to resolve conflicts and remove ambiguities from your model. The writing team will naturally write more suitable use cases once they better understand the system.

People are drawn into writing the details early. It is always difficult to begin a new project, and thinking about the whole, poorly understood system can be overwhelming. Trying to ramp up quickly is hard, and getting your head around a complex system is much akin to boiling the ocean. Developers often feel uncomfortable about spending the time necessary to comprehend a whole system as they feel that they are wasting too much time, and should be producing something tangible instead. When people are not sure how to proceed, they have a tendency to dive into the nitty-gritty details and start specifying detailed requirements about the small parts of the system that they understand, even if it is not very important to the big picture. Unfortunately, project managers encourage this tendency because they measure progress based on concrete deliverables and complete use cases fit the bill perfectly.

People lose energy or get bogged down in too much detail. A typical use case may have up to four general elements: actors and goals, main success scenario, failure conditions and failure handling. It is easy to get drawn into pursuing too much detail when you are writing the narrative

Copyright © 2001 Adolph, Bramble All Rights Reserved

of the use case, as it is tempting to examine the alternatives. But whenever people follow a step with too many details, for example, its failure conditions and failure handling, they tend to lose their focus and energy describing all the possible extensions conditions.

It is advantageous to get an overview early. If you start with an outline and then write just the essence of each use case, you give each stakeholder the opportunity to offer correction and insight early in the process. It also permits the work to be prioritized and split across multiple groups and releases, increasing parallelism and improving productivity.

The more detailed in your early endeavors, the more you will have to change them as you learn more about the system. Most projects are short on time and energy so continually changing the same use case is discouraging and stressful. The participants may start questioning their involvement and lose confidence in their use cases altogether.

Therefore:

Conserve your energy by developing an overview of your use cases first, then progressively add detail, working across a group of related use cases.

Start writing your use cases by outlining them before adding detail. Identify candidate use cases by associating a meaningful goal for each actor, defining a use case for every combination (**UserValuedTransactions(110)**). Using the goal, derive an **IntentionRevealingName(144)** for each use case. Once you feel that you have defined a fairly complete set of use cases, work through your set, refining it, combining equivalent use cases and eliminating unnecessary ones (see **MergeDroplets(211)**, **RedistributeTheWealth(206)**, and **CleanHouse(216)**. The resulting set of system goals provides a fundamental, shared understanding for the stakeholders and helps reduce the amount of refactoring required later on.

Once your overview is complete, develop the outlined use cases in an incremental manner, increasing the precision as you learn more about the system. Avoid the tendencies to complete an entire use case all at once or to write them one at a time. Instead, develop your use cases in a series of **SpiralDevelopment(66)** cycles, incrementally adding detail and reviewing the effect the new detail has on the model, until you determine that it is **QuittingTime(71)**. Expand your use cases into **ScenarioPlusFragments(136)** by describing the main success scenario of each use case, naming the various conditions that can occur, and then fleshing out some of the more important ones. Again, be ruthless about eliminating non-essential pieces from the collection. Add more detail to the fragments, creating a list of **ExhaustiveAlternatives(140)** in the next cycle.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Templates are an essential use case development tool that provide organization and a common look and feel to the use cases. They are especially valuable when doing incremental development, because templates can serve as placeholders which allow the writers to organize their use cases while they are still missing large pieces. Simple or lower priority use cases don't always require the same amount of information as the more critical ones, so you can use **MultipleForms(83)** to document each use case with an appropriate level of detail and precision.

Sometimes you do need to finish some use cases before others. In that case, take an incremental approach based on project priority. Identify the most important use cases, such as those that define the project, or those that need to be done first, based on business or technical issues. Prioritize these as you start writing, and work them in groups for the aforementioned reasons. You may work on other use cases as well, but not at the expense of the most important ones.

BreadthBeforeDepth(XXX) provides order to the use case writing process. The overview lets you see the big picture without getting bogged down in details. Working at a high level makes it easier to sketch in the system and its boundaries, as well as verify that your use cases provide complete coverage of all of the system's functionality. This high level view also allows you to start project planning and enables you to prioritize your work and develop the most important use cases first, and do the less important ones later.

Working across a group of related use cases makes it easier to realign the use cases and eliminate the ones you determine to be unnecessary before expending much effort on them. Fredrick Brook's classic quote "Be prepared to throw one away…because you will" [BROO1995] is appropriate for writing use cases. Often a team learns more about a system's requirements as the project progresses, and they may have to dramatically alter or even throw away earlier work. The sooner you can identify unnecessary use cases, the better, because it is wasteful and demoralizing to throw away highly detailed use cases on which someone has spent a lot of energy.

Examples

This pattern is closely related to the following one, **SpiralDevelopment(66)**. Therefore, we will defer offering an example until we have discussed it.

Breadth Before Depth and UML by Dan Rawthorne

The UML's Use Case Model is an overview of the functionality of the system. It does a very good job of documenting this functionality at the lowest possible level of precision. Therefore, using a

Copyright © 2001 Adolph, Bramble All Rights Reserved

diagram during requirements gathering prevents use case writers from diving into the details too early.

It is tempting to put too much information on the diagram. At this early stage of development it is probably enough to just show the actors and use cases, and be very sparing with the use case relationships. Better yet, avoid the temptation to prematurely introduce use case relationships.

This pattern was influenced by Rick Ratliff's Breadth First pattern from the OOPSLA 98 Use case patterns workshop.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.2 SpiralDevelopment(66)

Spiral fire escape, St. Louis, Missouri.

Vachon, John, 1914-1975, photographer.

CREATED/PUBLISHED
1940 May.

You are writing use cases in a **BreadthBeforeDepth(63)** manner.

Developing use cases in a single pass is difficult and can make it expensive to incorporate new information into them. Even worse, it can delay the discovery of risk factors.

"Hind sight is 20-20", "I wish I knew then what I know now" and "If only I could turn the clock back" are all-too-common expressions that people use when they realize that they could have done better if they had more information available to them earlier.

Writing use cases iteratively gives us a type of hindsight in that we can easily go back and scrap or redo them if we discover that they are not working out. We may lose some of our work, but

Copyright © 2001 Adolph, Bramble All Rights Reserved

much less than we would have lost had we simply written complete use cases one at a time. More importantly, we can identify and confront potential problems sooner using the iterative approach.

It can take a long time to understand a system's behavior. Learning is a linear process, and the more complicated the topic, the longer it takes to understand it. Students may like to "cram" before exams, but this technique only temporarily crams facts into their head, without giving them an understanding of the subject matter. Gaining a thorough understanding of a complex topic requires examining it piece by piece, using trial and error to learn its intricacies.

Delays are expensive. Requirements gathering is critical to a product's success, but it is only a one part of the overall project. Many projects have tight deadlines and many people can be depending on your use cases, so you need to finish your use cases as soon as possible so that they can ramp up to full speed.

The requirements are likely to change as a result of our examining them. Requirements are frequently volatile, and likely to change as we learn more about them. There almost seems to be a Hiesenberg-like uncertainty principle that applies to requirements, analyzing a requirement tends to change the requirement. A requirement that once seemed cast in concrete becomes obsolete upon further analysis. As you examine one requirement, you are likely to uncover information about others, and often discover that several requirements are wrong or missing. Another challenge is separating wants from needs. Things will eventually stabilize, but only after much analysis and re-work.

The cost of requirements mistakes is high. An argument that is sometimes used to justify the waterfall software development lifecycle is the downstream cost of fixing a requirements mistake. Such a mistake caught during the requirements analysis phase may cost only a nominal dollar to fix, the same mistake caught in the design phase, $2. The cost quickly escalates the later the mistake is discovered, implementation $4, test $20, deployment $68. [BOE1981]. No wonder there is a desire to get the requirements right the first time up front.

If you dive in too quickly, you will overwhelm others. Use case development is usually a learning process for everyone involved. People require time to learn, and while you may understand the system, if you go too quickly, you may leave everyone else behind. While it is important to finish your use cases in a timely manner, it is more important that the whole organization has a chance to learn the system well, as they will be carrying most of the development load.

Therefore:

Copyright © 2001 Adolph, Bramble All Rights Reserved

Develop use cases in an iterative breadth-first manner, with each iteration progressively increasing the precision and accuracy of the use case set.

Working along **BreadthBeforeDepth(63)**, pause when you have listed the actors and their goals, and work with that list for a while. Use the list to set up the project plan, estimate work, prioritize the use cases' value, even to help set up the development teams.

Continuing with **BreadthBeforeDepth(63)**, choose a working subset of the use cases to expand, and pause again when you have a set of main success scenarios, to review the purpose of the system. Take the opportunity at this moment to review the use cases. See whether you need to **MergeDroplets(211)**, or **CleanHouse(216)**, or in any other way improve the structure. You may find yourself revising the list of actors and goals at this point.

Finally, completing **BreadthBeforeDepth(63)**, you will probably find yourself revising your list of use cases one more time when you start working out the extension handling behavior for your use cases. It often happens that new use cases turn up at this point. For example, when writing use cases for the simple ATM system, people usually don't think about what happens when there is a communications break between the ATM and its host computer. While writing down the details for handling that situation, it slowly occurs to people that when communications come back up, the ATM will have some specific work to do. Handling that system condition will force the team to create a new use case such as "Reestablish communications," which did not exist before.

SpiralDevelopment interacts with **BreadthBeforeDepth(63)**. Imagine working to build all the use cases **BreadthBeforeDepth(63)**, without any plan to revise the use cases along the way. Experience shows us, however, that as people add to the use case, they uncover new situations for which they have to write use cases, detect new similarities across use cases, and find new and better ways to structure the use case set. **SpiralDevelopment(XXX)** advises the team that they will have to pause and regroup their work, and it indicates where to pause to review their work.

Key to the successful use of iterative development, of which **SpiralDevelopment(XXX)** is an example, is knowing when it is time to stop. Stop as soon as you are sure that your use cases are good enough to meet your stakeholders' needs, so that you can avoid the law of diminishing returns. **QuittingTime(71)** provides a set of criteria that you can use to determine that point.

Examples

Wings Over the World

We have all heard the expression, “can’t see the forest for the trees” which describes one of the mistakes made by the use case team in this installment of the *Wings Over the World* story. Many use case teams divide the system up and assign portions to different use case writers. Each use

Copyright © 2001 Adolph, Bramble All Rights Reserved

case writer goes off and independently begins to write detailed use cases. Even though requirements may be vague, ambiguous, inconsistent, missing, or just plain wrong, the use case writers attempt to write very detailed use cases. Detail becomes a substitute for accuracy and disagreements often break out between the writers as they try to resolve their differences. In this installment of the *Wings Over the World* story, a source of friction between Sitra and Ralph is that Sitra's work makes Ralph's work obsolete. Worse, Ralph has invested weeks of effort into writing his use case, and is not very happy about having to scrap that work.

Never under estimate the benefits of the big picture and incrementally developing and re-adjusting your use cases. When someone invests a lot of effort into creating a use case (or any other project artifact for that matter) they are reluctant to discard it, even when it really is unnecessary or incorrect. If *Wings Over the World* had followed the solution recommended by **BreadthBeforeDepth(63),** what would their work products look like?

First they could start with something simple such as an actor list that simply enumerates the actors and the names of use cases they use. For example, a fragment of the *Wings Over the World* actor list may look something like this:

Actor	Use Case
Agent	Reserve Flight Book Flight Cancel Flight Reservation Request Upgrade Open Passenger Profile Close Passenger Profile
Airline	Cancel Flight Discount Flight

Figure 1-1 Sample Actor List for Wings Over the World.

An actor list is not very precise, but it is accurate to the extent that we know the functional requirements for the system. The name for each of the use cases gives us some idea of what the goal of the use case is. Creating an actor list does not require a lot of effort, making it easier to change or drop proposed use cases. This list also helps us try to keep the *goal level* of each use case the same. It is easier to see at a glance if there are use cases that do not fit in with the others. In this example, *Open Passenger Profile* or *Close Passenger Profile* stand out because

Copyright © 2001 Adolph, Bramble All Rights Reserved

their goal level appears to be lower than the other candidate use cases. With little effort these are quickly pruned out from the use case set or merged into other use cases.

Once the writers have finished the list, they may try to write simple, one paragraph stories describing the main success scenario for the use case. These are often called *Briefs* or *High Level Use Cases*. For example:

Reserve Flight **Actor**: **Agent** The agent specifies a client's desired travel itinerary. The system searches for a suitable flight routing and presents it to the agent. The agent selects a routing that best matches the client's desires. The system verifies availability and then reserves the seats. The system calculates the fare price for the class of service requested by the agent.

Use Case 1-1 Brief for *Reserve Flight*

Book Flight **Actor**: **Agent** The agent retrieves a client's reservation and books the flight. The agent examines the aircraft seat map and selects the client's preferred seats in the aircraft. The agent enters the client's payment information and system books and assigns the seats. The system prints the tickets.

Use Case 1-2 Brief for *Book Flight*

Briefs allow us to refine the actor list and incrementally increase the precision of our use cases. These briefs tell us which actions the use case author believes are essential for reserving and booking flights. From them, we can tell that reserving a flight does not include paying for the flight or even selecting a seat, that is part of *Book Flight*. Briefs do not specify detailed steps, alternatives, pre-conditions, or post conditions. Rather their intention is to quickly cover the broad scope of the system, i. e. its breadth, before we dive into the depths of the details. Briefs are easy to write, available for early review, and easy to fix. They allow you to detect and correct problems quickly, before they gain momentum and become ingrained in the system.

Once we have defined a broad system view, then we can expand a subset of the briefs, and begin to fill in their details with more detailed step by step descriptions of the main success scenario, alternative courses of actions, pre-conditions and post conditions. It is important to note that we only define a subset of the briefs. Perhaps each use case writer takes one brief and simply enumerates the alternatives. Then, with this additional information added to the use case,

Copyright © 2001 Adolph, Bramble All Rights Reserved

the team may once again review the use case to determine what affect it may have on the other candidate use cases.

We call this technique "*Surface and Dive*", where we take one or just handful of use cases and dive down into the details once we have developed the broad scope of the system. Once at the bottom, we surface again to see what affect the detailed use cases have had on the rest of the use cases. We carefully adjust the model, select another subset of briefs, and dive down into the details of the use cases. Using this approach, even Sitra and Ralph may be able to come to a consensus.

*A More Literary Example***.**

Dictionaries and encyclopedias contain immense amounts of information, which their publishers are constantly revising. But the volumes are so large that the publishers don't want to start new editions from scratch and completely rewrite them every few years. Instead, they are written iteratively, reusing as much of the previous edition as possible, adding new words or articles as needed, and modifying or eliminating the obsolete entries. This method considerably simplifies the writing process, and greatly reduces the time to market. As an added bonus, the publisher can resell much of the same information every few years to the same customers.

SpiralDevelopment and UML Models – by Dan Rawthorne

It is especially important to realize that you do not need to develop your Use Case Models all at once. As you are iteratively improving your use case set you should be iteratively improving your use case diagrams as well.

Also, there is no requirement that all your use cases belong on the same diagram. Your diagrams must be understandable. As you progressively increase the quality of your use case set you may find the need for a number of different models that focus on different things, such as:

- Diagrams based on actors

- Diagrams based on similar functionality

- Diagrams based on abstraction level

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.3 MultipleForms(83)

<<Caption and credits missing>>

You are writing use cases in a **BreadthBeforeDepth(63)** manner.

Different projects need different degrees of formality in their work, and various people have different preferences for the template. Requiring everyone to use the same use case template is counterproductive.

Taste in music is a highly personal thing and what is good for one person may not be acceptable for another. Some people like rock and roll, some like country, and never the twain shall meet. People will refuse to buy a particular album because they see it in the wrong section, or think it might be the wrong genre. When I was growing up, it was common for older people who would never buy a Beatles album ("it's not music") to snap up Perry Como or The Boston Pops' renditions of their songs without a second thought. Music companies take advantage of these differences. They don't snub any type of popular music, but instead package each album to maximize sales in its particular market segment.

Copyright © 2001 Adolph, Bramble All Rights Reserved

People and projects are different. Everyone is unique, and has their own way of looking at the world. Our personalities, experiences, and training shape our individual natures, and as a result, each of us sees things differently from other people. Similarly, each development organization has its own people, history and culture that differentiate it from any other group.

Different projects have different needs. Projects may be large or small, simple or complex, and represent either new functionality or a remake of an existing system. For these projects, use cases might describe a business process, focus discussion about a future software system, describe functional system requirements, or document a system's design. You could develop your project locally, or be distributing the work over several teams in several geographical areas. Each of these factors affect the amount of formality you need in your use cases, and the decision as to what is the most appropriate form for your use cases must be made on a project-by-project basis.

You want to avoid getting too caught up in unnecessary precision and rigor, costing your project a lot in expended time and energy. This risk needs to be balanced against the potential damage of missing key requirements. A format that is too lax will cause you to omit important details, but a format that is too rigid can create a lot of needless busy work that adds nothing to the qualities of the model.

Different teams want different amounts of formality. Different methods for documenting use cases exist to fulfill different needs. At one extreme, an organization may require very formal use cases that rigorously define all possible system behaviors, while at the other end of the spectrum, an organization may require very simple use cases. Any of these approaches can be correct, as long as they adequately describe what the implementers and stakeholders need to know with a level of detail consummate with their needs.

Using a common form within an organization facilitates communication. A standard use case form or template makes it easier for people to know what information to put in their use cases as well as what information to expect from one. More importantly, because this form is usually well defined, it is easier to use, and easier to understand the purpose of each of the use case's components.

While templates are useful, some people have an insatiable urge to fill in all the boxes in one go. This depth-first approach can sap your energy and cause you to lose focus. Worse, it can waste a lot of time by forcing you to do unnecessary rework revising and refactoring your use cases as your learn more about the system. It is important to realize what needs to be completed at the desired level of precision.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Therefore:

Select the format based on the risks associated with the project and the preferences of the people involved.

Be open-minded when choosing a format for your use cases, and don't fall into the rut that all use cases should look alike. Use cases are only a tool for describing a system's behavior to your stakeholders and developers, and therefore only need to meet their needs for precision and content. Any more information is unnecessary.

Base your template selection on the purpose and level of risk associated with each use case. For critical use cases, write more formalized use cases containing a generous amount of information, and pay close attention to **ExhaustiveAlternatives(140)** and **PreciseAndReadable(152**) . For less critical or better understood ones, be less formal and use less detail. An **IntentionRevealingName(144)** title may be sufficient for use cases describing well known behavior.

While it can be tempting to use different templates on the same project, it is not a good idea, as it is way too hard for a team to wade through multiple templates. Choose a standard template for any one project, and stay with it. An advanced practitioner team may find it appropriate to write with differing amounts of formality and rigor for different use cases in the same set, but they should still use the same template to avoid confusion.

Examples

Batch Job Distribution

I helped my team write use cases for a system that would distribute large "batch" jobs across a set of workstations and personal computers. During the course of the project, we identified approximately 70 use cases covering a wide range of system behavior. Because we had a strong understanding of the system from working on its predecessor, we only annotated the ten or so use cases that described the various ways the users would use the system. We left the rest blank because they described familiar behavior (i.e. job monitoring, logging onto and off of the system, etc.). These 60 "textless" use cases were very valuable to us, as we used them to define the system servers and the boundaries between them. We could have annotated them, but our team was very small, and this effort would not help our already tight schedule. However, the annotated job flow use cases were very useful because they significantly described interesting behavior that we would have to understand when we built the system. We referred to these use cases often, both with customers and in design meetings. Accordingly, we were able to build a successful system with a small team in a short period of time.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Use Cases for the Workstation Monitor Server

Actor	Description
Administrator	Person monitoring and controlling job control system.

Use Case	Description
Login -	Log Administrator onto server. *
Logout –	Log Administrator off of server. *
Set Monitor Parameters –	Allow Administrator to specify boundaries and precision of items being monitored.
Select Monitor –	Chose something to monitor (i.e. through put, processes, wait queue, etc.)
Monitor Workstation –	Collect the performance statistics for the jobs using the workstation.
Display Results –	Display the performance results that the user selected according to the Administrator's specified preferences.

* Some methodologists debate whether login and logout in general constitute valid use cases. We defined them in this case because we knew we would have to implement functionality for logging users on and off the server.

Use Case 1-3 Use Cases for the Workstation Monitor Server

Purchasing for a Business

The key question is how much information does your audience need? The following use cases, taken from Writing Effective Use Cases [COC2001], demonstrate different forms for the same use case. The first use case is appropriate when describing a high level view of the process, or in those instances when the stakeholders already possess a good understanding of the feature.

USE CASE 4: *BUY SOMETHING* **(CASUAL VERSION)**

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

Copyright © 2001 Adolph, Bramble All Rights Reserved

and sends the goods to Requestor. The use case ends when the Requestor marks the 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, while raising the price sends it back to Approver.

Use Case 1-4 *BUY SOMETHING* (CASUAL VERSION)

Often your use cases require more detail or formality, as demonstrated by the next use case.

USE CASE 4: *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.

Copyright © 2001 Adolph, Bramble All Rights Reserved

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

Frequency of use: 3/day

Channel to primary actor: Internet browser, mail system, or equivalent

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?

Use Case 1-5 *BUY SOMETHING* (FULLY DRESSED VERSION)

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.4 QuittingTime(71)

<<caption and credits missing>>

You have written a set of use cases in a **BreadthBeforeDepth(63)** manner.

Developing a use case model beyond the needs of the stakeholders and builders is wasteful of resources and delays the project.

Engineering history is full of anecdotal stories of heavily over-specified and over-designed systems. The military has often borne the brunt of these stories because they have a tendency to rigorously over-specify the requirements for any acquisition. There are always stories of hammers and toilet seats procured by the military costing tens and sometimes even hundreds of times the price of their off the shelf cousins. While these incidents make entertaining news stories, they point to the real problem of over-specification.

We specify requirements to reduce the risk of creating the wrong system for the stakeholders, and many methodologies prescribe rigorous requirements specification procedures that must be closely followed. However, this rigorous approach to requirements specification does not take advantage of shared experience or common sense among the project participants. Consider

Copyright © 2001 Adolph, Bramble All Rights Reserved

buying a hammer: most people simply walk into a hardware store and say, “I need a hammer”, trusting that the salesperson knows what they mean. If they want to be really precise, they might say, “I need a claw hammer”.

In contrast, military procedures may require the procurement officer to disregard common shared knowledge and say “I need a MIND - Manual Impact Nail Device - The MIND shall have an impact face between 15 to 20 millimeters in diameter and must be made from dropped forged tungsten steel. The MIND shall also have an integrated NRU - Nail Recovery Unit - consisting of twin 30-millimeter carbon steel prongs, set apart in a V formation with a maximum gap spacing of 5 millimeters. Lastly, the MIND shall have a human machine interface of oak or equivalent hardwood, etcetera.” Needless to say, it becomes easy to see how this practice delays simple acquisitions and escalates costs.

Good use cases are balanced, describing essential system behavior while providing only the necessary details about the interactions between a system and its users. They are complete enough so that their readers can easily understand what the system does, but they are not formal tomes that describe the system to the nth degree. But knowing when to quit writing use cases can be difficult because this decision involves balancing some complex forces.

A paralyzing fear of overlooking important requirements encourages builders and stakeholders to prolong the requirements gathering activity. Because it is much easier to fix requirements than it is to fix code, many developers will drag their heels, and not start designing a system until they are absolutely sure that their requirements are correct. Some developers believe that the later a requirement error is found in the project, the greater the cost associated with rectifying the problem Others fear the potential embarrassment and loss of stake holder confidence if they overlook an important requirement.

Many technical professionals place an unwarranted high priority on model formality. Technical professionals are trained and indoctrinated in problem solving techniques that require formal, precise descriptions. Formality is important from their point of view because it provides a “liberating form” for guiding and channeling the thought process, and it also minimizes ambiguity and omissions.

Ambiguity can destroy a project. It was common engineering school dogma in the 1970s that a slight miscalculation of less than one tenth of one percent would have caused the Apollo spacecraft to miss the moon completely, and send the astronauts into eternal orbit. Small details matter, and leaving the interpretation of vague requirements to the implementers’ imagination can lead to missing functionality as well as some highly innovative features that take the product into unforeseen areas that nobody wants. Stopping writing use cases too soon is an easy way to introduce all kinds of ambiguity into a system.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Most people can work with a fair level of ambiguity. Many organizations possess what is often referred to as their "core competencies". These represent the shared knowledge and experience of the people within the organization that gives them their advantage over their competitors. It is this knowledge and experience that helps the organization reduce the risk associated with projects. For example, when a carpenter tells an apprentice to get a hammer and some nails, he probably does not have to explain the full details of what kind of hammer and what kind of nails to fetch. The context of the situation in which they are working and the shared background help the apprentice understand the specific needs of the situation.

Over-specification of requirements may lead stakeholders into believing the requirements are more accurate than they actually are. In high school science many of us were taught the importance of experimental error. For example, we could not write into our laboratory notebooks that the mass of a specimen was 10.3482727 grams if our scale could only measure to a tenth of a gram of accuracy. We did not want to imply more precision in the result than we could actually measure. Stated another way, telling detailed lies does not make them any more accurate or truthful. Requirements are the same in that there is a tendency to assume detailed requirements are more accurate (truthful) than less detailed requirements. Writing detailed requirements will hide the uncertainty associated with the requirements.

The cost of a mistake is quite small if it is caught early, and often the cost of not moving ahead is exorbitant. Human beings make mistakes, and have misunderstandings in the best of circumstances. The art is to quickly know that a mistake has been made, and exploit the knowledge gained from the mistake. Mistakes caught early in the project are less costly to fix, and often delaying a project until all requirements are known can be dramatically more expensive.

Therefore:

Stop developing use cases once they are complete and satisfactorily meet audience expectations.

You need to know what your goals for writing use cases are before you can determine if your use cases are complete. It is important to be clear to all involved why you want to write use cases in the first place, as well as what problems and risks you need them to resolve.

To determine if your use cases are complete, ask the following questions:

1) Have you identified and documented all actors and goals?

2) Has the customer, or someone representing the customer, acknowledged that the use case set is complete, and that each use case is readable and correct?

Copyright © 2001 Adolph, Bramble All Rights Reserved

3) Can your designers implement these use cases?

If the answer to any of these questions is no, then chances are you need to do more. Take advantage of the core competencies of the organization and the shared knowledge of the stakeholders to flesh out your use cases. Always keep in sight what the stakeholders need to see in the model. Once the stakeholders agree that the use cases adequately reflect their vision of the system, you are close to being finished, but it is important to satisfy all three questions.

Don't fuss too much about quality, and avoid excessive attention to adornments. There comes a point beyond which further development of the use case model will not yield any more understanding about the system. At this point you should exercise your **WritersLicense(80)**, "put your pencil down" and move on.

You can always add more details later, as you learn more about the system. If moving on reveals there are ambiguities and uncertainties in the model, do not hesitate to pick up the pencil again and exploit the newfound knowledge. Be careful to consider the consequences to the entire system. Don't add it all at once to completion, but do it in an organized manner (i.e. **BreadthBeforeDepth(63)**), applying this information throughout the set of use cases where appropriate.

Examples

Wings Over the World

In this installment of the *Wings Over the World* story, it was hinted that Ahmed and Ralph are arguing over the completeness of the use cases. Ahmed argues that the use cases cannot be complete until all details are filled in. Ralph argues that is a waste of time and that they should just get on with it. Who is right?

"How do you know when you're done?" is one of the toughest questions to answer on a project. There are many wonderful text book answers that offer absolute answers to this question. But unless you have the financial resources of a federal government you usually must push something out the door long before the completion criteria recommended by the textbooks. Like any real company, our fictitious *Wings Over the World* operates in a very competitive world and needs to deploy systems quickly.

QuittingTime(71) is about balancing the risk of delaying the project with the risk of incomplete requirements. Software engineering textbooks often place a very high-risk premium on incomplete requirements because they assume the development organization has little or no expertise in their problem domain, nor is there some level of trust between project members. But

Copyright © 2001 Adolph, Bramble All Rights Reserved

all of these factors usually exist in a company. The term “core competency” was coined in 1990s to describe the advantage knowledge and trust gave to a company. Fast moving agile development methodologies such as XP take this to the extreme[1] where they take advantage of corporate knowledge for a high level of development agility. For example, XP requires a customer representative be part of the development team to quickly resolve requirements shortcomings.

What the *Wings Over the World* story never makes clear is what the purpose of the use cases actually is, and who is going to use them. For example if the intention is to outsource the development of the system, then the use cases require a great deal more rigor and completeness. The slow pace of communications between a client and their vendor would increase the risk in any of the ambiguities in the requirements. On the other hand, if the system is being developed in house, the developers have easy access to users and domain experts, and the developers take advantage of that close relationship, then they can tolerate a great deal more ambiguity.

Wings Over the World has decided to outsource a portion of the development of their new system. This decision implies that we need more rigor because of the higher potential for misunderstandings and the slower rate at which discrepancies are found in this type of situation. It is likely that the outsource vendor will have a template they use for writing use cases and that template will be one part of a methodology used by consulting company. But they should be cautious that they do not fall into the trap of filling in every detail in the template for the sake of doing so.

The **BreadthBeforeDepth(63)** approach facilitates an early **QuittingTime(71)**, because it does not demand that every use case be written to a pre-defined level of detail. In fact, it may, in some cases, be acceptable to leave some use cases as briefs because adding more detail does not add any new knowledge – the problem and solution are well known in the organization. This is an approach that relies on trust.

[1] We apologize for this obviously extreme pun, but we just could not help ourselves.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.5 TwoTierReview(76)

[Untitled].

CREATED/PUBLISHED
[between 1935 and 1945]

You have been writing use cases in a series of **SpiralDevelopments(66)**, and you believe that it is **QuittingTime(71)**.

Many people may need to review the use cases. This is an expensive, time-consuming proposition.

Representative democracy is a system of government in which citizens elect people to represent them in a deliberative body. These representatives study the issues as they arise, and vote on them, based on their understanding of the subject and their constituents' feelings. If someone has a concern about an issue, they can talk to their representative about it, potentially influencing their decision. While the system isn't perfect, it is certainly more efficient than having millions of people debating every issue or voting on each and every decision, big or small. While the

Copyright © 2001 Adolph, Bramble All Rights Reserved

representatives make the decisions, the voters have the final say, as they can vote those people out of office that they feel are ineffective or who have failed to represent them.

This system provides a good model for reviewing use cases, especially those intended for large, diverse groups of stakeholders. It allows every interested group to have their say as to the use case's contents, without overwhelming everyone, or requiring excessive time.

Reviews are necessary to verify and validate writing and content. Reviews are a good way to validate the correctness and completeness of a piece of work early in the development process. People tend to believe their work is better than it is really is, often believing that the quality of work is proportional to the effort spent on it. But inadequate, inaccurate work can detract the audience from a document's purpose, and your stakeholders expect professionalism in business documents. It is harder for writers to catch their own mistakes because they are familiar with their work, and tend to draw inferences from it that aren't necessarily there. People reading a document for the first time are more likely to notice errors because the errors distract them from understanding it.

The stakeholders have a vested interest in the use cases. Many different groups have a vested interest in a set of use cases, and depend upon them to help them do their work. It is in your best interest as a writer to consult with them early and often, to simplify your effort and minimize the amount of rework you need to do to produce acceptable use cases that adequately address their issues.

It is expensive, tiring and slow to involve everyone in the writing process. Too many people tend to get in each other's way, are inefficient, and require a lot of effort to coordinate. A **SmallWritingTeam(XXX)** helps keep the process manageable, on track, and tends to cut down on feature creep.

If only a small writing team is doing the review, not all stakeholders' interests are incorporated. It is very difficult, if not impossible for a small writing team of two or three people to represent the views of a large diverse audience. A **SmallWritingTeam(47)** without **ParticipatingAudience(50)** doesn't have the experience or the diverse knowledge base to understand or represent all of the stakeholders' views on a large project. It is likely that they will miss key parts of the system without this help.

Reviews can be expensive, tedious and time consuming. For a review to be effective, the participants need to invest a great deal of time and energy. The time spent on frequent or long reviews, with many people, quickly adds up to significant manpower. Therefore, we need to be judicious when conducting reviews, by only scheduling them when needed, and requiring as few people as absolutely necessary. Also, while some people like reviewing development material,

Copyright © 2001 Adolph, Bramble All Rights Reserved

others despise it. You can allow all interested parties to attend, but keep the required attendance to a minimum.

Therefore:

Hold two types of review. The first is done by a smaller, internal team, possibly many times. The second is done by the complete group, perhaps just once.

First, review the use cases internally to verify their readability, implementability, precision and accuracy. These “inner” reviews can be informal desk-reviews, formal meetings, or a combination of both. Any kind of review is appropriate as long as it allows the reviewers to catch errors and verify that your use cases are sufficient as far as they are concerned. One of the purposes of these initial reviews is to eliminate the “noise” caused by spelling, grammatical, and formatting and technical errors, which when left uncorrected are distracting.

You may need to hold several of these inner reviews when the system is large or overly complex. Because people tend to lose interest in detailed discussions outside of their own area of interest, consider holding separate group reviews for different functional areas when formally reviewing use cases aimed at a large, disparate customer base. That way, each group of stakeholders can review the use cases in depth from their particular point of view without distraction.

At the end of these inner reviews, the teams are asserting that it is **QuittingTime(71)**, and that the use cases are complete, correct, and as implementable as they need to be at this point. The use cases are then ready for the bigger group to check.

Hold at least one meeting with the complete group once the use cases pass internal muster, to review the system as a unified whole. Trust the first tier of reviews to validate the internal workings of the system, so that the second tier can focus on how the pieces fit together.

The definition of "complete group" varies by project. It should be all the people who review the requirements before development gets too far underway. In some cases it is just the development team, sometimes developers plus an executive, sometimes it is the business analysts and the lead programmers, sometimes it is users, executives and the entire programming team. The purpose of the “outer” reviews is to determine:

- is this really the appropriate thing for the developers to spend time building (business value check)?

Copyright © 2001 Adolph, Bramble All Rights Reserved

- is this correct as a specification? (Are the business rules correct, and does it leave open the proper allowed variations in implementation. Does it lock down the important decisions - does it identify the appropriate set of open issues that can be handled later?)

- can the developers really build it?

Examples

Wings Over the World

In this installment of the *Wings Over the World* story many of the stakeholders are grumbling about the frequent reviews and some are even refusing to participate. Is it necessary that every review be an all hands effort? Of course not! **SpiralDevelopment(66)** requires regular re-assessments of the use cases to both verify the use cases are a fair representation of the functional requirements and that structurally the use cases exhibit the signs of quality advocated by the patterns listed in this book. The majority of stakeholders are only interested in reviewing the use cases to verify that they protect their interests. While all the stake holders or their representatives want their say and want to participate in the project, they certainly don't want to spend long hours in formal and tedious use case meetings each week to review minor changes.

The Programmer Who Cried Review

Once upon a time there was a programmer who was designing a new system for large company. It was an important system and would impact the jobs of many people. The programmer wanted to make sure the interests of all the people affected by the system were protected, so soon after he finished a draft of the system vision he cried out "Review!". The moment he cried out all the people came running to the conference room because they knew that the last person there would have to take minutes. For the next two hours they reviewed the draft of the system, taking down action items. At the end of the meeting all the people thanked the programmer for allowing them to verify that the vision addressed their interests.

The programmer took all the action items and incorporated them into the second draft of the system vision and once again he cried out "Review!". Just as quickly as before all the people came running into the conference room not wanting to be the one who had to take minutes. For the next two hours all they reviewed the updated draft. Some people grumbled this time that they had better things to do than sit in a review, but still most of the people thanked the programmer for allowing them to verify the vision addressed their interests.

The programmer took all the action items from the second review and incorporated them into a third draft of the system vision and once again he cried out "Review!". As quickly as before all the people came running into the conference room not wanting to be the one who had to take

Copyright © 2001 Adolph, Bramble All Rights Reserved

minutes. For the next two hours all they reviewed the updated draft. This time most of the people grumbled that they had better things to do than sit in a review and that they did not want to be called into another review unless there was something more than just document structure that had changed.

The programmer took all the action items from the third review and incorporated them into a fourth draft of the system vision and once again he cried out “Review!”. But this time no one came running to the conference room, not even the interns. “Review, review!” cried out the programmer. Still no one came. So the programmer decided to baseline the system vision anyway.

Unfortunately, this situation let an important feature fall through the cracks, and the company’s CEO was embarrassed at a live demo when it was not there. The programmer ended up in the call center gulag because he cried review too often.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.6 WritersLicense(80)

<<caption and credits missing>>

You have conducted **TwoTierReviews(78)** on your use cases.

Excessive emphasis on style issues unnecessarily impedes the effort of writing use cases.

Each year, film and television companies produce movies and TV programs based on true stories. Most of these productions take a creative license with the story to make it more appealing to a wider audience. “The Serpent and the Rainbow” [Wes Craven 1988] was based on Wade Davis’ book about a medical research looking for a new anesthetic in Haiti. When asked about the liberties the movie had taken with his story, Wade Davis simply replied, “it’s not exactly the story I wrote, but then I doubt that my story would have opened on 1700 screens.”

We should allow some freedom in our use case style to better suit each unique system and audience.

We would like a consistent writing style to simplify reading. Familiarity can be reassuring when trying to understand something new. An inconsistent style can confuse the readers and make it difficult to find information by forcing them to readjust to each use case. It can be especially frustrating to try finding some information that appears in a different place in each use case.

It is costly and impractical to get everyone to write exactly the same. Writing is highly individualistic, and everyone has their own personal style. You could revise the use cases until they are similar, but this practice wastes time, energy, and money, and leads to sending the

Copyright © 2001 Adolph, Bramble All Rights Reserved

manuscript around and around and around. There comes a point when the law of diminishing returns takes effect, and the results aren't worth the effort. Style guides are more useful, but only solve part of the problem, as it is impossible for them to document every potential issue. A common style is "frosting on the cake", but not worth substantial extra effort.

There is value in getting the use cases into development sooner. The sooner the developers have the use cases, the sooner they can start developing the product, and the sooner they can finish it. Every delay in moving use cases into development can cause project slippage, which costs money. It doesn't matter if the use case's style contains some minor variances; the developers can still use them.

We still need to meet the basic completion criteria of a use case being readable, logically correct, and detailed enough for the developers. The purpose of use cases is to provide information that accurately describes a system in an easy-to-read format. Use cases should correctly document enough knowledge so that the people depending upon them can easily find enough information to help them do their jobs, whether they are working with the big picture, or they are implementing some of the low-level parts of the system.

Therefore:

Small differences in writing style are inevitable. Once a use case passes the tests for QuittingTime(71), the writer can claim "writer's license" on small stylistic differences.

Write each use case so that it passes the following tests:

- it follows the organization's writing template and basic style;
- it is logically correct;
- it is readable to the end evaluators;
- it is precise enough for the implementers to use.

Once the use cases meet these tests, allow the author to have final say on smaller stylistic matters. Small differences in writing style are up to each use case writer; they can claim "writer's license" to make any changes they think appropriate, while ignoring other style changes they feel don't add any value to the manuscript, or aren't worth the effort involved.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Style guides can be useful within a limited context, for a specific purpose, by describing a common "look and feel" for your use cases, and informing the writers about what you expect in your use cases. Just be sure to treat them as guides rather than a set of iron clad rules. They should be flexible enough to promote individual creativity, and not hinder the exchange of information.

Examples

Wings Over the World

In the *Wings Over the World* installment, the Chief Architect Ahmed is suggesting a rigid style manual that specifies the rules for writing use case descriptions. While his intentions are good, they defeat the utility of a use case. Use cases are useful to the project participants because they are semi-formal artifacts. They have enough structure so that the readers can ferret out the inconsistencies and gross ambiguities in the requirements, but not at the expense of making them unreadable to someone who is not expertly trained in the methodology.

All project participants have gone through a basic one-day use case course, so they do know what a use case is about. But can you imagine trying to get a travel agent or marketing manager to understand pseudo code? For requirements to be useful, they must be well understood by those who must review those requirements.

Use case writers frequently abuse templates. In our opinion, templates represent a good checklist of items that may be included in the final use case, not a prescription for everything that must be in the use case. One of our personal peeves is usually the pre-conditions section of a use case. Easily 90% of the use case pre-conditions that we have seen state something to the effect of "user is logged on". While this may be valid, does it really add value to the use case? We are of the opinion that writer should be able to take license to determine what helps the audience understand the use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.7 Tradeoffs and Collaborations

Following the right development methodology is critical to writing quality use cases. This methodology doesn't have to be elegant or "high powered", but it does need to cover all the bases. For developing use cases, good process means balancing discovery versus writing, and content versus need. Although it would be nice to have one, there is no one-size-fits-all process for creating quality use cases. Writing use cases is highly personalized; everyone has their own style, and every organization has its own way of doing things, based on its culture and business needs. Yet, while there are multiple ways of writing quality use cases, the better processes share several common elements. The Process Patterns presented in this chapter describe the characteristics of effective use case development, and are intended to help you create your own process.

Figure 1-2 shows the relationships that exist between these patterns. Be aware that this diagram is not a flow chart, and does imply order. The dotted arrows represent the "helper" relationship, where the pattern at the pointer end of the arrow helps the pattern at the other end do its thing. For example, knowing when it is **QuittingTime(71)** helps you better understand **SpiralDevelopment(66)**, and **WritersLicense(80)** is a practice that helps you reach **QuittingTime(71)**.

Figure 1-2

Relationships between the Process Patterns

Copyright © 2001 Adolph, Bramble All Rights Reserved

These patterns are closely related to each other, each describing a different aspect of use case development. As a result, they tend to work together rather than conflict with each other, and we don't really see any tradeoffs between them. **BreadthBeforeDepth(63)** and **SpiralDevelopment(66)s** are very tightly coupled, and form an efficient basis for developing use cases. Working across a set keeps the set balanced, and keeps you aware of the relationships between its various members. If you determine that two use cases really address the same issue, or that some are unnecessary, you can react sooner in the writing cycle, minimizing the amount of unnecessary work. But this method by itself can be overwhelming, unless you do it in reasonably sized chunks, developing the use cases incrementally in a series of cyclic expansions. Writing use cases is a discovery process, where you often learn things effecting several use cases while researching others. Breadth first development is an efficient way to keep from getting ahead of yourself on any particular use case, and from revising it continuously as you learn more about the system.

Another advantage of this iterative approach is that it allows an organization to determine when their use cases are 'good enough' for their purposes, and stop writing them before reaching the point of diminishing returns. Yet to stop with any degree of confidence, writers need to have some well-defined criteria. The pattern **QuittingTime(71)** says you can quit writing use cases when you have determined that you have identified and documented all actors and goals, the customer has approved the use cases, and you believe that the developers can implement the system from these use cases. Three other patterns also offer advice for writing 'good enough' use cases. **MultipleForms(83)** recommends formatting use cases in a style that is appropriate to the project and the audience's need, instead of using a company-wide standard form. Everyone writing use cases for a particular project should use the same template, but which one should be their own decision. **WritersLicense(80)** states that small, stylistic details are not that important, and that you can stop writing once you have addressed all of the system issues, even if some stylistic issues remain. Lastly, **TwoTierReview(76)** proscribes an efficient review process that helps verify that the use case collection is correct and complete, and also facilitates **ParticipatingAudience(50)**.

Process is only half the battle of writing high-quality use cases. Effective use cases are organized into a structure that clearly identifies certain key features of the system and allows readers of various backgrounds to easily use the various pieces with a minimal amount of effort. This structure is the subject of the next few chapters.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1 The Use Case Set

Redeye

You seem to have a knack for doing it to yourself. No one asked you to hop the Redeye, no one told you to fly to the East Coast. You just blurted out to *Wings Over the World's* CIO, "sure I can be there tomorrow, I'll just see if I can book tonight's Redeye". Its now hour three of a five hour red eye flight east to *Wings Over the World's* corporate office where at nine in the morning you're suppose sit down with their requirements team and help them dig themselves out of the hole they're falling into. There is urgency because in your view they are starting to develop a bad case of silver bullet syndrome, believing the use cases you hooked them on represent their ticket out their requirements woes. That's why you scrambled to get on the Redeye, deal with the problems early and set their expectations before this case of silver bullet syndrome becomes terminal to the project. Like any project, this one is experience a lot of birthing pains. The first thing you noticed was that not everyone was clear on why *Wings* was taking on this multi million dollar project.

During your last conference call with Ahmed, their chief architect, all he could gush about was how excited he was opening their system to the World Wide Web. "Everyone is looking forward to learning IBM's Websphere, JSP, and EJB," he said. You don't recall *Wings Over the World's* management caring one wit about the implementation technology. Rather they were more interested in reducing ticketing costs and opening new revenue streams. Excitement about new technologies and methodologies is admirable – that has gotten you excited about this project too, but there already seems to be a conflict between tech team's vision and the rest of the management's vision

The red flags really started popping up with a chorus of "sproings" when you read the first use cases from *Wings Over the World*. First major problem is that everyone seems to have a different idea about what belongs in the system and more importantly, what does not. Some of the programmers have already started writing use cases for managing customer complaints. You recall that only an interface to a CRM system was ever discussed, the CRM system itself was supposed to be out of scope. When you suggested that these features might be out of scope, you got the classic, "but we may need it in the future". At the other extreme some of the programmers assumed that the scope of the system was just the *Wings Over the World* wholesale ticket operation, and that none of the flight planning features are part of the system.

The spill over from a lack of a clear boundary is actors all over the place. It looks like someone simply took all of the possible *Wings over the World's* job descriptions and declared them to be actors. Most of the jobs have huge overlaps in responsibilities. You see use cases

Copyright © 2001 Adolph, Bramble All Rights Reserved

with names “Booker: Find Flight”, “Agent: Find Flight”, and “Consultant: Find Flight”. The only difference you can see between these use cases is just who is trying to book the trip. Worse, without knowing the scope of the system, its seems that they have simply assumed that everyone in the company will need to use the new system to do their job. That’s probably the source of all those customer service use cases that you think belong outside of the system. You get a little restless smile as you read other use cases. CRUD must be one of the most aptly defined acronyms ever, because it best describes so many of the use cases you have seen. Easily seventy percent of the use cases are of the form *actor* create data, *actor* read data, *actor* update data, *actor* delete data. None of these use cases tell you why the actor is performing that action or what they hope to accomplish from it. How can they ever help you discover if the system is going to do anything meaningful or satisfy its stakeholders’ expectations?

The Use Case Set patterns are the signs of quality for a well-written set of use cases. These signs of quality are not directly attributable to a single use case, rather, they describe the emergent properties of all the use cases taken together. These patterns answer the question, “What are the signs of quality that are visible in a good set of use cases?”

First and foremost, the use case writers must have a **SharedClearVision(95)** of the system’s purpose and goals. Why are you building this system, and does everyone know and share the same vision? The need for a vision goes beyond use cases because if project participants are unclear about its’ purpose, they are likely to assume one, correctly or incorrectly. A clear, well-defined vision significantly simplifies development by enabling everyone working on the system to understand what it is that they are trying to do, and why they are doing it. Everyone involved with the system must share the same vision to better enable them to work in unison.

The vision must be bounded by a **VisibleBoundary(101)** that limits the scope of the system. This boundary clearly delineates between what belongs in the system, and what does not. What are the responsibilities of the system and who are the people, and other systems that will interact with the system?

Together, a **SharedClearVision(95)** and a **VisibleBoundary(101)** should provide readers with a good grounding for understanding the project’s vision and scope.

The **VisibleBoundary(101)** helps us discover who or what can interact with the system, who on the outside requires services from the system, as well as who on the outside helps the system provide these services. We call these people or things actors in use case terminology, and we need to know who they are, what they want from the system, and what they offer to it. These actors are the **ClearCastOfCharacters(105)**. The cast includes

Copyright © 2001 Adolph, Bramble All Rights Reserved

people, organizations, computer systems or hardware. This list provides a very concise and useful picture of the system, defining the initial negotiating point between the user representative, the financial sponsor, and the development group.

What are the services the system must offer the actors if the system is to realize the objectives and goals specified in the **SharedClearVision(95)**? These services are the **UserValuedTransactions(110)**, the services that yield measurable value to the actors of the system.

We structure our use cases with higher-level use cases referencing lower-level use cases in an **EverUnfoldingStory(117)**. This organization allows the reader to focus on the system at varying levels of details, depending on their need.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.1 SharedClearVision(95)

Lower Manhattan seen from the S.S. Coamo leaving New York.

Delano, Jack, photographer.

CREATED/PUBLISHED

1941 Dec.

You have decided that writing a set of use cases will help you and your stakeholders better understand a system's functional requirements.

The lack of a clear vision about a system can lead to indecision and contrary opinions among the stakeholders and can quickly paralyze the project.

In a special session before the U.S. Congress, President John F. Kennedy gave the challenge "…that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to earth." These words became a mantra to NASA and much of the nation. The scope of the project was enormous, and presented many technological, manufacturing and management challenges. Inspired by his dream, the United States spent over 25 billion dollars (that's close to 100 billion by today's inflated standards) and used the services of several hundred thousand people for eight years. The well-focused vision was realized on July 20, 1969, as people all over the world watched

Copyright © 2001 Adolph, Bramble All Rights Reserved

Neil Armstrong step onto the moon and utter the famous words, “That’s one small step for [a] man…”

Time pressures can encourage people to start developing a system prematurely, basing their work on erroneous assumptions. Pulling them back together again can be expensive. Often, the builders simply enumerate the system’s basic services, creating use cases without any regard to their purpose or understanding their value to the actor requiring the service. This approach fails to accurately reflect the user’s needs and can paradoxically result in features that are unnecessary or deficient. Many of the so-called CRUD (Create, Read, Update, and Delete) use cases originate from a lack of a clear vision.

Builders have a natural tendency to expand the scope of the system. Without some guiding principle that can be used as criteria for evaluating and scrubbing features, there is a danger the development team will expand the scope of the system. The vision provides a mechanism for removing vague, ambiguous or poorly defined requirements from the scope of the project. It is an objective filter for scrubbing requirements, and helps to define what is in or out of the system’s scope.

Stakeholders have competing visions. There are competing visions of what are the needs and responsibilities of actors may be, and often very little knowledge to help select the most appropriate vision of the system. Most projects are under time pressure and cannot simply evaluate countless number of alternatives. They must begin to limit the solution space quickly. If there is no clear vision, then project participants may substitute their own vision for the project, which may conflict with the corporate mission. For example, without a vision the software developers may create their own vision for the product that emphasizes scalability and flexibility in the software architecture. While these are highly desirable architectural characteristics, they may conflict with the corporate need to quickly and cheaply deploy a system.

People in the trenches don't know what the goal of the project is or why they are doing it. An interesting experiment on larger projects is to go around and ask the participants why they think the project is important. If you’re fortunate, most people will understand how their specific piece of the project contributes to the overall project. But does anyone understand what the goal of the project is beyond make money or reduce costs? What does the system allow its users to do that they couldn’t do before? Why is this important? Simply, is it clear to all project participants why someone is willing to pay money for the system?

People don't communicate. Despite the best efforts to encourage communications by tearing down private offices and replacing them with cubicle hell there are many barriers to informal

Copyright © 2001 Adolph, Bramble All Rights Reserved

communications in any organization. The larger the organization the stronger these barriers are. When there are strong barriers to informal communications, then the formal communications channels must be strengthened.

Therefore:

Prepare a statement of purpose for the system that clearly describes the objectives of the system. Ensure that this vision supports the mission of the organization, and freely distribute it to everyone involved with the product.

Include:

- the objectives of the system,
- the problems that the system is solving,
- the problems the system is not solving,
- who the stakeholders are, and
- how the system will benefit the stakeholders.

Maintain consistency in the vision by having a small team of people responsible for creating the vision (**SmallWritingTeam(47)**). A characteristic of many successful projects is the appointment of a single person who champions the vision of the system and is responsible for maintaining the consistency of that vision. This responsibility usually falls on someone in a marketing role such as the product manager. This person must also take an active role in ensuring that all members of the development team share the same interpretation of the vision.

Validate the vision and seek support for the vision by obtaining advice from those who will be affected by the system (**ParticipatingAudience(50)**).

Strengthen and constrain the vision by clearly specifying what is outside of the system and what is part of the system (**VisibleBoundary(101)**).

Ensure the vision is supportive of the stakeholders' mission and does not work at cross-purposes. A truly excellent system vision will fail if it defines a vision that works against the organization's mission.

Finally, when there are changes to the vision, ensure that all project participants are immediately aware of the changes.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Examples

Wings Over the World

There are few things short of an extinction event size comet hitting the Earth that can derail your project as quickly as the lack of a shared clear vision. Most modern methodologies emphasize the need for a vision statement or vision document. Unfortunately it seems that when someone produces a vision statement, they do so only to satisfy the requirements of the methodology or to fulfill a contractual obligation with the client. It seems that the vision is rarely shared or clear.

Typical of this is the experience of our long-suffering consultant asking Ahmed "why are you building this system?" Ahmed answers by itemizing the technology that's being used to implement the new system. On further probing, the Ahmed only offers "well to allow people to book their travel on-line."

It is hard to believe the company's directors would launch a project simply to "allow people to book travel reservations online[1]." Why is this feature important to the mission of the company? Does the company see itself as a market leader and is trying to increase market share by offering its clients the option of booking online? Is it playing catch up with its competitors in an effort to preserve market share? Or is it trying to reduce the cost of bookings in a highly competitive market place?

By not ensuring that all project participants understand the objectives for the system, the directors are creating a void into which others may start to fill in with their own agendas. Technical teams are notorious for this. In this installment of the *Wings* story, Ahmed is envisioning a high performance, scaleable web based system. There is nothing sinister in this practice, its just human nature. Maybe Ahmed's vision aligns with the needs of the corporate vision. On the other hand, what if the company is playing catch-up and needs to deploy a system quickly and cheaply to stay abreast of its competition. Then Ahmed's vision may run counter to the corporate needs. Without a **SharedClearVision(95)** the, development team will create one based on what they believe is important. Developers are trained to believe that reliability, scalability, and expandability are the important characteristics of a high quality system. With out a clear vision for the system, they will bias their design efforts towards these goals rather than the goals that are important to the organization.

What is an appropriate vision statement for the *Wings Over the World* project?

[1] Maybe this was acceptable during the dot com bubble, but it certainly is not the case anymore - hopefully.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Wings Over the World endeavors to maintain its image for innovation by increasing access to our travel services and by offering new and innovative services that are unmatched by our competitors. Specifically:

- Increase brand awareness of Wings Over the World with the creation of a public website.
- Increase market share by 15% and lower the cost of booking tickets by letting 30% of our clients book online.
- Open the Wings Over the World travel system to independent travel agents by offering premium quality booking services and therefore create a new revenue stream.
- Etc.

Figure 1-1 Wings Over the World Vision Statement

Now we have something and while we will not claim it's perfect, we see what the corporate vision is for the system. This will help structure our thinking as we begin the process of discovering the use cases.

Fortunately for Ahmed, this corporate vision seems to support his technology vision.

Automated Railway Signaling

The Vehicle Control Center (VCC) guides automated trains for city mass transit systems. The original software for this system was nearly 25 years old, and it was reaching the point where it was difficult to find replacement parts for the its aging 16 bit mini computers, or find the programmers who could – or even wanted to - program them. Furthermore the capacity of the system was fast approaching its design limits. With rapid growth in the mass transit market place, it was time to consider replacing this tried and true legacy system.

Replacing legacy systems is fraught with risks. According to Fredrick Brooks the second system is the most dangerous system a man can ever build, because it provides an opportunity to release all their pent up desires to fix the perceived shortcomings in the first system [BROO1995]. Naturally this desire translates into "gold plating", adding those features of little value that make the product look better. However, the VCC replacement project organizers controlled this desire with a well-defined project vision statement.

The purpose of the project is to lower costs associated with the maintenance and modification of the Vehicle Control Center by:

- Porting the system from aging 16 bit mini computers to modern 32 machines.
- Re-coding the original assembly code in a high level language to make it easier to recruit developers.
- Adopting a development methodology that complies with Railway Industry requirements.

Copyright © 2001 Adolph, Bramble All Rights Reserved

- Quadrupling the capacity of the system.
- Incorporating new features identified in reference document DOC-XXXX.

No changes shall be made to the system's fundamental architecture or algorithms.

Figure 1-2 Vision Statement for the Automated Railway Signaling System

This vision statement clearly stated what the project's participants could and could not do, effectively preventing gold-plating. Possibly the most important statement in this vision was the last statement. Rigorous enforcement of the last sentence in the vision led to the success of the project [ADO1996].

Mobile Dispatching

Contrast the Vehicle Control Center legacy replacement project with another one whose aim was to replace a mobile dispatch system. The circumstances surrounding the decision to replace this aging legacy system were similar to those for the Vehicle Control Center, however, the results though were much different. A year after the project's inception, it was cancelled without delivering anything of worth [ADO2000].

The project began with a well-defined vision statement:

Replace the existing mobile dispatch system with one that:

- lowers software maintenance costs
- supports a relational database,
- supports new client features as specified in document DOC-XXXX.

Figure 1-3 Vision Statement for the Mobile Dispatching System

The "lower software maintenance costs" clause was probably the spark that contributed to our explosive downfall, because management and development understood it differently. Management believed that it meant reducing the time it took to perform standard system customizations for a client. The developers, however, felt that it meant completely re-writing of the dispatch system using leading edge technology. While both sides had a clear vision it wasn't a shared one, or even close.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.2 VisibleBoundary(101)

Sign on city limits of Franklin, New Hampshire.

Locke, Edwin, photographer.

CREATED/PUBLISHED
1937 Sept.

You have a **SharedClearVision(95)** for a system.

The scope of a system will grow in an uncontrollable manner if you do not know its boundaries.

The late 1950s was probably the golden era of the so called “B” films, and one of the films that is near and dear to all B film connoisseurs is the 1958 science fiction film, “The Blob”. In this movie that launched Steve McQueen in his Hollywood acting career, a group of teenagers discover an alien creature from outer space that has no form or shape, and grows rapidly as it absorbs everything and everyone around it. Like all movies of this genre, the

Copyright © 2001 Adolph, Bramble All Rights Reserved

teenagers attempt to warn a doomed town of the danger it faces, but no one believes a bunch of kids until its too late.

Few other films have been used as extensively as a metaphor for the dangers of uncontrolled or unmanaged growth. “The Blob” has been used to describe everything from civic problems such as urban sprawl, to poor software architecture [Bro1998].

Different people have different points of view regarding where a system’s boundary is. Some stakeholders such as business owners are often more interested in how the business runs, than in the software. Developers are often more interested in the software than in the business. For the stakeholders the system is the business and the environment is the world in which the business operates, with customers, suppliers and competitors on the outside of the system. Resources such as the sales staff and customer service clerks are inside or part of the business. On the other hand, the developers may regard the sales staff and customer clerks as users and therefore outside of the system.

Poorly defined boundaries lead to scope creep. Many of the stakeholders have “pet problems” they want solve and hope that a new system will resolve for them. Sales and marketing staff often load features onto a system in the hopes of capturing customer interest. Developers love to enhance or add features to even well-defined systems. Poorly defined or undefined boundaries exacerbate these problems. Without a clear definition of where the system boundary is to guide them, then everyone will try to add feature they deem even slightly necessary. But these extra features often go beyond the useful mandate of the system usually don’t add any value to the system. If anything, they make the system more complicated and less useful.

Imprecise and conflicting objectives early in the project often make it difficult to determine the boundary of a system. Early in the project there may be insufficient information to identify what belongs to the system and what is part of its environment because the system is large, or its purpose is not clearly defined.

People have the perception that defining a boundary is unnecessary. Programmers often feel that analysis activities only get in the way of the real work (coding), and don’t provide them with any real value. This perception is especially true in the startup world, where companies live on a shoestring budget, and their very livelihood is heavily dependent on the time it takes them to get their product to market. But this approach can quickly result in unnecessary and missing features, inadequate products, and excessive costs.

Therefore:

Copyright © 2001 Adolph, Bramble All Rights Reserved

Establish a visible boundary between the system and its environment by enumerating who and what interacts with the system

The **VisibleBoundary(101)** limits and supports the **SharedClearVision(95)** by:

1. specifying what external systems and personnel the system must collaborate with, and by
2. specifying what resources the system has available to it to accomplish its purpose.

In the early stages of development, the visible boundary may be fuzzy because the vision for the system may not be clear. Start by putting a stake in the ground and document the system boundary based on the information you have available to you. As you develop your use cases and learn more about the system, you can refine and sharpen the system boundary i.e. **SpiralDevelopment(66)**.

Examples

Wings Over the World

A context diagram is a useful tool for documenting the visible boundary between a system and its environment. It is a simple diagram consisting of a circle and boxes on a sheet of paper, with lines and arrows extending between the circle and boxes. But that simple diagram contains a lot of information. The circle represents the system. The boxes, or terminators as they are called, represent anything that is outside of the system. The lines running between the boxes and system represent the flow of data between the system and its terminators. In short, it shows what the system interacts with within the environment.

Both the context diagram and its UML equivalent the deployment diagram are simple, effective tools for identifying potential actors and use cases, because they help us focus on the things that interact with a system while ignoring the services that those things require.

In addition to simplicity, the benefit of using these kinds of diagrams is that they are based on information that is readily available early in the development cycle. With use cases, you must find actors and the services they need. On the other hand, with context or deployment diagrams, you only need to find the physical things that the system will talk to. You do not need to know what services they will require or deliver to the system.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-4 Context Diagram for Wings Over the World

This diagram provides an excellent starting point for building the use case set. To find actors we ask questions like, "What are the different roles terminators such as Customer, Bank, and Ticket Wholesaler play with respect to the system?" The diagram also identifies invalid actors. For example, Figure 1-4 shows that the system is the company, and travel agents are part of the system. Therefore, travel agents cannot be actors, and any use case written where the travel agent is considered an actor will be inappropriate.

Just as important, this diagram shows what does not interact with the system. For example, the *Wings Over the World* system does not interact with independent hotels or car rental agencies. According to this diagram, actors and use cases for independent car rental and hotels are outside of the system's scope.

The type of interactions that are outside of the system's scope can also be documented. For example, while this system does interact with customers and with the customer assistance system, it has no business getting involved in any direct interactions between the customer and customer assistance. A use case describing a scenario where the customer phones customer assistance to complain about an airline or the service they received is outside of the system's scope.

Copyright © 2001 Adolph, Bramble All Rights Reserved

VisibleBoundary(101) and UML Models by Dan Rawthorne

The context diagram shown above is a lovely thing, can be drawn with standard UML, and shows the environment of the system being developed. I love drawing context diagrams and putting them up on the wall. When coupled with the use case diagrams you have drawn as you do **SpiralDevelopment(66)** and **BreadthBeforeDepth**, you have scoped the system's functionality from both the inside and the outside. The context diagram shows the scope as it looks from the outside, while the use case diagrams show the functional scope from the inside.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.3 ClearCastOfCharacters(105)

<<caption and credits missing>>

You have a **SharedClearVision(95)** of the system's purpose and have established a **VisibleBoundary(101)** that specifies who and what is outside of the system.

If we only analyze the users of the system rather then the roles those users play with respect to the system or the stakeholders interests in those roles, then we can miss important system behavior or introduce redundant behavior.

An actor is a person who plays a role on the stage or in movie. Some actors are so good at their craft that it is frequently hard to believe that the actor is not the role they play. Actors frequently get type cast into a role they play. For many people it is difficult to separate Clint Eastwood from his role as "Dirty Harry" or Sean Connery from "James Bond". Seeing actors in radically different roles, such as Clint Eastwood as a caring sensitive man in Bridges of Madison County is often an experience nearing culture shock.

What often really demonstrates to us the craft of acting is when one actor plays several radically different roles in the same play or movie. In Stanley Kubrick's classic cold war movie, Dr. Strangelove, Peter Sellers plays three distinct roles, the calm rational Group Captain Mandrake who is trying to prevent a nuclear holocaust, the ineffective and milquetoast president of the United States, and the very mad scientist Dr. Strangelove. In

Copyright © 2001 Adolph, Bramble All Rights Reserved

acting, it's the role that we are interested in, not the person. In the movie we don't follow Peter Sellers, we follow Group Captain Mandrake, the President of the United States, or Dr. Strangelove. Each of these characters has their own traits and develops in different ways throughout the movie, and each role contributes the development of the movie's plot.

Just like actors in a movie or on the stage, users may play many roles with respect to a system. If we do not understand all the roles a user plays, then we cannot understand the system.

A system must satisfy its user's needs and protect the interest of its stakeholders if it's to be useful. If the system is not to be deficient in its requirements then we must find all the services its users need. A development organization can suffer a damaging loss of goodwill when it fails to provide the services that the users need versus what they want.

Tying services to users may make the system so rigid that services cannot be re-used. Different users may have overlapping sets of responsibilities. Users can play many different roles with respect to the system and many different users will actually share similar roles with respect to the system. If we only consider the different users then we may create a system with different interfaces for similar operations. Failure to discover overlapping responsibilities and to separate them from the users will lead to bloated rigid systems

The narrow focus or point of view of subject matter experts can cause us to miss all the services a user requires. Subject matter experts are quite vocal on the topics they find of interest and often have little interest in areas outside of their specific expertise. Unless we have a mechanism to probe and elicit from the subject matter experts all the services the required from the user, the system will be deficient. (See Center A case study in this pattern's examples)

Focusing on users may also bog us down with implementation details rather than understanding of the services the user needs. Engineers love to solve challenging detailed problems because that's why they became engineers in the first place. It is what engineers do best. If we begin to focus on the user rather than the roles they play with the system then we run the risk of prematurely introducing implementation details into our analysis (the "*how*" rather than the "*what*"). The user is something concrete that our engineers can grab onto immediately and begin solving the problem of how should the system communicate with the user. How should we lay out the UI? What happens when the user asks for a non-existent part number? What is not so easy is to try to pull from thin air the services the system must offer the user.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Many people find it easier to comprehend concrete concepts rather than abstract ones. It is difficult for many people to work with abstract concepts. A comment heard frequently "If I can't point to it, then I don't want to know about it". A person, machine, or even another piece of software is something that is easy to point to and identify. The roles that a person, machine, or piece of software plays is much more abstract and therefore more difficult for people to visualize. Furthermore, abstract concepts may sometimes mask fuzzy or imprecise thinking.

Time pressure and expediency encourage us only to analyze the users. Once the boundaries of the system have been identified, the users are relatively easy to find. After all it is easy to identify users by name and job description. It is relatively simple to point to a box at the end of a cable that must communicate with your system. It takes more time and effort to find out that that box and person play the same roles.

Therefore:

Identify the actors and the role each plays with respect to the system. Clearly describe each of the actors the system must interact with.

Analyze the system by focusing on the set of services the system must offer the users. You should at least know who the users are because the **SharedClearVision(95)** gives you a statement of the system's purpose and the **VisibleBoundary(101)** gives you a glimpse of the scope of the system.

For each user identify and list the services they require from the system. Try to organize the services into cohesive sets and name those sets. Look carefully for users that have overlapping sets of services and collect those services into sets. Examine the sets of collected services to see if they are complete and are not missing services. These sets are the actors of the system and represent the roles entities play with respect to the system.

Give each actor a clear crisp noun or noun phrase name. If you cannot provide a good name for the actor then the odds are you have not identified an actor. Write a description for the actor specifying the services they require or offer to the system.

In the early development phases it can be difficult to find the roles of the actors. Finding the roles of the users is often an emergent characteristic of creating the use cases. So don't get bogged down if you cannot find all of the roles right away, because they should naturally emerge as part of the **SpiralDevelopment(66)** cycle.

Examples

Center A: The Museum of Contemporary Asian Art

Copyright © 2001 Adolph, Bramble All Rights Reserved

During the design of the *Center A* use case set, our subject matter experts focused on the activities of the museum curator, and specifically the curator's scholarly activities such as the search and acquisition of art for the museum. Our subject matter experts were either artists or curators themselves. What we missed and had to recover from was that the curator also has a tedious day to day job of simply running the museum. This role consists of the very mundane tasks of fighting the board of directors for funding, receiving bids from contractors, authorizing purchase orders, of hiring and firing. In larger museums some curators have only a scholarly role, while others have the more of the management role. Had we missed this management role, our system would have been grossly deficient. Yet this role of the curator as a manager was largely hidden because we only analyzed the curator's scholarly role, the only part of the curator's job that was of interest to the subject matter experts (see **BalancedTeam(54)**).

The Pharmacy Receptionist

In a pharmacy, there are three main actors, the receptionist, the pharmacy technician and the pharmacist. Any one of these three actors may greet a patient and take their prescription. Either the pharmacy technician or the pharmacist may fill the prescription, and only the pharmacist can check and authorize the prescription. If we only analyze the users of the pharmacy system then we run the risk of not seeing the overlaps in responsibility between the receptionist, pharmacy technician, and pharmacist. In this project we came very close to creating the use case model shown in figure 1-6 before we noticed the similarities in the receptionist roles.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-5 Receive Patient Use Case when all users are treated as different actors

Actor List

Receptionist:	The receptionist welcomes patients to the pharmacy. They take the patient's prescription.
Pharmacy Technician	The pharmacy technician assists the pharmacist by filling prescriptions, but is not authorized to dispense the prescription to the patient. The pharmacy technician may also welcome the patient to the pharmacy and take the patient's prescription.
Pharmacist	The pharmacist fills and dispenses prescriptions for patients. Only the pharmacist can check and authorize the prescription The pharmacist may also welcome the patient to the pharmacy and take their prescription.

Figure 1-6 Actor List for Pharmacy System

Copyright © 2001 Adolph, Bramble All Rights Reserved

This model nearly came about because we had not notice the shared job roles and we nearly lost the opportunity to simplify the system by capturing a common role. We need to understand and model role(s) played by the system's users rather than the model the user. In the worse case we could end up designing separate reception interfaces for the receptionist, the pharmacy technician, and the pharmacist without realizing that these three users play common roles and could share the same interface.

Figure 1-7 Receive Patient with actor defined as a role of Receptionists

Actor List

Receptionist:	The receptionist welcomes patients to the pharmacy. They take the patient's prescription. Anyone in the pharmacy can play the role of the receptionist.

Figure 1-8 Revised Pharmacy System Actor List

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.4 UserValuedTransactions(110)

Bookies taking bets at horse races. Warrenton, Virginia.

Wolcott, Marion Post, 1910- photographer.

CREATED/PUBLISHED
1941 May.

You have established a **SharedClearVision(95)** of the project and have defined a **ClearCastOfCharacters(105)** who need services from the system.

A system is deficient if it cannot deliver services that are valuable to its users and does not support the goals and objectives specified by the system vision.

A few years ago I bumped into a colleague of mine who was working for a new startup that was building a radio advertising distribution network. The idea was simple; advertisers were using couriers to distribute tapes of their advertisement to radio stations, and even with overnight delivery, it could be two to three days before a new advertisement was actually playing over the radio. My colleague's company had built a continent wide private network to distribute advertising spots to radio stations nearly instantaneously. Furthermore, with their proprietary protocols, and compression hardware they could guarantee quality of service to their clients.

Copyright © 2001 Adolph, Bramble All Rights Reserved

It seemed like a license to print money. Offer a service that was cheaper, faster, and more reliable than what was currently available, and the world will beat a path to your door. This approach resembled a page right out of Marketing 101. But the market never developed, and eventually the company succumbed to the inevitable and was taken over.

A while later I mentioned this story to a radio DJ at a local station. "Doesn't surprise me that they failed", he said analyzing the business case, "radio is a local media, almost no one does national campaigns".

A well-written set of use cases clearly and accurately describes the essential actions that a system provides. This information allows customers to preview a system before it is built, and determines whether it offers the kind of services that they find valuable.

A set of use cases should capture the fundamental value added services the users and stakeholders need from the system. An organization commissions the development of a system because that system will bring a benefit to the organization. Use cases allow them to inspect a system before it is built, so that they can verify that it is what they want, request changes, or decide that it doesn't meet their needs. Use cases should describe the kinds of things that users find valuable so they can present the system in its best light. A system that does not deliver needed valuable services to its actors is deficient, can lose money, and will sully the reputation of the development organization.

It is relatively easy to identify low-level transactions while it can be difficult to identify useful services. It is usually easier to describe the individual routine transactions that a system may provide than it is to discover what the user really wants to do with the system. Doing it the easy way often leads to the so-called 'CRUD' (Create, Read, Update, and Delete) style use cases. It is not unusual to see use cases with names like "*Create Employee Record*", "*Read Employee Record*", or "*Delete Employee Record*". While such use cases may be technically correct, they do not capture what is valuable to the user. Is creating an employee record important, or do we they really want to "*Hire Employee*"?

Use cases need to be relatively stable because they form "anchor points" for the rest of the product development process. Constant changes to the use cases can ripple through the rest of the development process, creating havoc for the developers and significantly increasing the cost. To keep this cost low, we want to write each case at a level high enough to insulate it from inconsequential changes, otherwise the writers will constantly be updating their use cases every time someone changes some trivial detail. Worse, the readers will have trouble understanding the use cases, because their meaning is constantly changing.

Copyright © 2001 Adolph, Bramble All Rights Reserved

*Readers want to easily see how the system will meet its purposes (See **SharedClearVision(95)**.* Just like a picture is worth a thousand words, a use case should allow someone to get a quick system overview. But even pictures can be hard to understand when they are too complex or abstract. Concise use cases that stick to the point are easier to read than long flowery ones.

People tend to work at too high or too low a level. People tend to use excessive detail when describing things they understand, or find interesting. Conversely, they tend to gloss over details they don't understand or find boring. Use cases should be somewhere in the middle, containing enough information to adequately describe system behavior, without describing it in great detail ("*what*" versus "how"). If we write them at too high of a level, then they are not useful to the system developers, because they do not describe the system in enough detail. However, if they contain too much detail, then it is difficult for people to understand the system from the 50,000-foot level. In the words of Ian Graham [GRA1997], use cases should only contain necessary but essential information.

Therefore:

Capture the smallest set of goals that identify the valuable services ~~that~~ the system must deliver to the actors to satisfy its statements of purpose.

Ideally, a set of use cases should contain all of the information necessary to depict a system but no more. Each use case should describe some unique, essential service that is valuable to at least one user or stakeholder.

Use the **ClearCastOfCharacters(105)** and **SharedClearVision(95)** to identify those services that the system should provide. Define as many valuable services as you can for each actor in your cast of characters that help them reach a goal. Being unable to identify any service for an actor may indicate that the actor might not represent a valid system user and you should consider removing them from the cast. Conversely, if you identify a service that doesn't map to an actor in your cast, it may indicate that you have not identified all of the actors.

For each service that you identify, ask "What value does this service provide to the user or stakeholders?" Get rid of those services that fail to add value to the system. You don't want to waste valuable time writing use cases or implementing code for a feature that no one will use for or has an interest in.

Users and stakeholders prefer to see the bottom line rather than an itemized list of 'CRUD' style services, so examine each service and determine whether each service stands by itself or is part of a larger, more valuable service. Fold those services that cannot stand by

Copyright © 2001 Adolph, Bramble All Rights Reserved

themselves into more comprehensive ones that address one key objective, and eliminate duplicates. A client booking an airline reservation is interested in getting a good flight at a good price. They don't care how many times the system updates its databases or files as the travel agent books a seat.

Write use cases around these goals. While you want to minimize the number of use cases in your collection, each use case should be a cohesive unit that describes one and only one key concept between an actor and the system, a **CompleteSingleGoal(132)**. Describe this collection in sufficient detail to adequately describe its purpose, yet be high level enough so as to be insulated from simple changes.

This singleness of purpose does not prevent a use case from addressing more than one goal, as long as the use case is cohesive and achieves a unified purpose. For example, a high-level use case can reference several subordinate use cases in an **EverUnfoldingStory(117)**, but these use cases must work together to accomplish a common purpose.

Note: In **BreadthBeforeDepth(63)**, the actor names and the names of the user valued transactions make up the original depth to which use cases should be taken before **SpiralDevelopment(66)** is applied.

Examples

Wings Over the World(1)

What are **UserValuedTransactions(110)** for *Wings Over the World*? Where would our analyst look to find them? The vision statement is a great place to start because it is suppose to state the objectives for the system and if the system is going to be successful, then its use cases need to support those objectives. The vision statement for *Wings Over the World* is:

> Wings Over the World endeavors to maintain its image for innovation by increasing access to our travel services and by offering new and innovative services that are unmatched by our competitors. Specifically:
>
> - Increase brand awareness of Wings Over the World with the creation of a public website.
> - Increase market share by 15% and lower the cost of booking tickets by letting 30% of our clients book online.

Copyright © 2001 Adolph, Bramble All Rights Reserved

○ Open the Wings Over the World travel system to independent travel agents by offering premium quality booking services and therefore create a new revenue stream.

Figure 1-9 Wings Over the World Vision Statement

What are some of the valuable services the system may offer to support this vision? Some examples are:

- Book Trip
- Search for Flights
- Promote Vacations
- Create Trip Itinerary
- Update Trip Itinerary
- Delete Trip Itinerary

But are all of these services valuable? *Book Trip*, *Search for Flights*, and *Promote Vacations* are the kinds of things that successful travel agencies do to stay in business. But *Create Trip Itinerary*, *Update Trip Itinerary*, *Delete Trip Itinerary* are probably not representative of those goals a user would consider valuable. Each is a part of a larger service. *Book Trip* involves *Create Trip Itinerary*, *Change Booking* includes *Update Trip Itinerary*, and *Cancel Booking* involves *Delete Trip Itinerary*. These changes reduce the list to the following services.

- Book Trip
- Change Booking
- Cancel Booking
- Search for Flights
- Promote Vacations

Does this small list mean to say that a complex online travel website may only have ten or fifteen use cases? At the 50,000 foot level there will only be a few summary use cases. We can then unfold these use cases into their sea level use cases later. But what is most important is that we know the important and user valued transactions.

Wings Over the World (2)

Many use case writers make the mistake of creating a use case for each user interface form or external protocol message [LIL1999]. This technique results in the creation of many small

Copyright © 2001 Adolph, Bramble All Rights Reserved

use cases that add miniscule value to the primary actor. Following such a strategy could result in the following set of use case for booking an airline ticket.

Figure 1-10 Forms Base Use Case Set

Each of these use cases describes the steps required to obtain a piece of information from some form and process it. Each use case may have a well-defined goal, but the goal is only a small step in booking an airline ticket for a customer. If we tried this approach for a larger system with many forms, we would end up with hundreds of small use cases, and we would lose the story context that is so valuable to our understanding the system.

It is easy to lose the context when writing these kinds of use cases, even in this small example. Are these three use cases all that is done to book a ticket? How do you know these are related to booking a ticket?

Furthermore, these use case do not have goals that reflect the actor's point of view. This style of goal encourages writing use cases from the system's perspective rather than the actor's perspective. Note how each use case is named "*Get XXX*", and would most likely describe how the system acquired information from the actor, rather than how the actor used the system to perform a useful task.

The transaction that is valuable to the travel agent is not getting payment information, rather it is booking a ticket. *Get Customer Information*, *Get Flight Information*, and *Get Payment*

Copyright © 2001 Adolph, Bramble All Rights Reserved

Information are really steps in the *Book Airline Ticket* use case. The goal of the travel agent is booking a ticket and making a sale, not gathering payment information.

Figure 1-11 Booking an airline ticket as a valuable transaction.

If the steps *Get Customer Information*, *Get Payment Information* etc. represent important and complex behavior, you can write as lower level use cases (**EverUnfoldingStory(117)**). *Book Airline Ticket* now gives a context into which these use cases fit, and makes it easier to comprehend the use case set.

The Cruddy Mortgage Origination System

A banking client was developing their next generation mortgage origination system. This was the first project they were trying to apply use cases and after their first cut they had come up with use cases such as:

- *Create Customer*
- *Update Customer*
- *Delete Customer*

The *Create Customer* use case described the steps a loan officer took to add a customer to the customer database. The *Update Customer* use case described the steps to update customer information, and *Delete Customer* described the necessary steps to … well you get the general idea. In short they were well on their way to building a cruddy use case set, which was an accurate description of behavior, but was not useful description. For example, how is creating a customer a goal of measurable value to the loans officer? Or updating the customer? Or deleting the customer? The question we want to ask here is why. Why are you creating a customer? Or why are you updating the customer or why are you deleting the customer?

When asked about creating the customer, the loans officer replied, “well that’s part of what I need to do when I open a new customer account, I have to create a customer”. Poof, the light bulb goes on in everyone’s head, creating a customer is part of a very valuable transaction

Copyright © 2001 Adolph, Bramble All Rights Reserved

called *Open Customer Account*. Now that's a goal that has measurable value to the loans officer.

Now what about that *Update Customer* use case? Still talking to the loans officer we find out that updates are done in two situations, one are simple corrections to the customer account such as change of address, or misspellings, and the other is to add documents that support the customers request for the loan. So the two valuable transactions that come out of this are *Correct Customer Information* and *Acquire Supporting Documents*.

Finally, why is a customer deleted from the database? Well says the loan officer, they're never quite really deleted, rather after we have either approved or declined the loan, we archive the customer account. So out of this comes *Archive Customer Application*.

Our three cruddy use cases now become the much more valuable four:

- *Open Customer Account*
- *Acquire Supporting Documents*
- *Correct Customer Account*
- *Archive Customer Account*

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.5 EverUnfoldingStory(117)

Reduction gears. Pratt and Whitney Aircraft Corporation. East Hartford, Connecticut.

Collier, John, 1913- photographer.

CREATED/PUBLISHED

1941 Sept

You have a **SharedClearVision(95)** for a system and are writing a description for a use case, and you wish to supply more detail for the steps in a use case or supply a higher level of abstraction to understand the context of the use case.

The number of steps needed to describe the behavior of the system exceeds both the memory and the interest of the various types of readers.

Imagine for a moment if maps could only be drawn at one scale. If the scale for the map was very fine, say one centimeter equals one kilometer, then we could probably create a useable

Copyright © 2001 Adolph, Bramble All Rights Reserved

street map for a city. But then maps of the world would be totally unusable. For example a globe built to this scale would be some 20,000cm in diameter[2]! If we changed from a fine scale to a very coarse scale, say 1cm equals 100 kilometers, then we can build a very serviceable globe, but we could never build a useable street map. This is why maps are drawn to a variety of different scales, such that the users of the map can easily use the map for the information they require. Fine scale street maps for finding an address in a city and coarse scale world maps for finding a city on the planet.

Each use case can have many readers and many uses all requiring different levels of detail. Some use case readers are interested in system architectural issues, and need a set of use cases to help them understand the big system picture. Other readers are interested in business issues, and need a set of use cases to help them understand the business layer, and issues about system installation and testing. Still other readers may be interested in implementing the system, and need the use cases to help them understand the complex interactions the system provides. Each of these readers needs to be able to obtain a different level of detail from the same set of use cases to do their job. Accordingly, the use cases must contain enough information to satisfy all of them. If, for example, the use cases contain only high level system descriptions, then the subject matter experts will not have the information that they need.

Use cases need to show how every function contributes value to the business. Every part of a system needs to directly add value to the business, or support a process that does. Unnecessary functionality is expensive, adds extra cost to the system and hinders performance. Accordingly, each use case needs to demonstrate this value to the reader, so that they can easily determine whether the functionality is required or not. It is much cheaper to eliminate unnecessary functionality early in the product development cycle, preferably in the requirements phase, as this pruning prevents the unnecessary allocation of precious resources during subsequent cycles.

It is, however, confusing to write at multiple levels. Writing at different levels requires an understanding of these levels. Not only must the writer understand how the system should operate at different levels, but they must understand the appropriate domain requirements and what the readers need to be effective at each level.

Writers need a principle to follow for an organizing the requirements. Readers need an easy way to navigate through the use case set. Simply presenting the use cases en masse can

[2] That's about 12 miles in diameter for the metrically challenged.

Copyright © 2001 Adolph, Bramble All Rights Reserved

overwhelm them as they attempt to make sense out of the model. Good organization simplifies this task for the reader by allowing them to easily locate and obtain the information they need. But good organization doesn't happen in a vacuum. It requires the writers to define comprehensive principles so that the readers can understand what they are trying to accomplish.

Therefore:

Organize the use case set as a hierarchical story which can be unfolded to get more detail, or folded up to hide detail and get more context.

Just as maps are drawn at different scales to reveal or hide details, use cases can be written at different levels of abstraction to hide or reveal details. A good set of use cases contains several cohesive levels of related use cases that completely describe the system in different levels of abstraction, where each subordinate level closely resembles the level above it, albeit with greater detail.

A good way to describe unfolding use cases is to include several subsets of use cases in your collection, where each subset describes the system at a different level of detail. In his book "*Writing Effective Use Cases*" Alistair Cockburn suggests three levels of use cases:

Summary Level Use Case	is one that takes multiple user goal sessions to complete, possibly weeks, months, or years.
User Goal Use Case	satisfies a particular and immediate goal of value to the primary actor. It is typically performed by one primary actor in one sitting.
Subfunction use case	satisfies a partial goal of a user goal use case or of another subfunction. Its steps are lower level subfunctions.

The following diagram, adapted from Alistair's book, illustrates the relationships between these levels. The higher-level use cases provide the context for the lower level use cases. Or more simply, higher-level use cases answer the question "why is the actor doing this" for lower level use cases. The lower level use case answers the question for the higher-level use cases "how is this going to happen."

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-12 Use Case Goal Levels [COC2001]

Examples

Wings Over the World

The system developer's desire to have detailed specifications is a force driving us towards detailed use cases. But detailed specifications are long-winded and confusing to most people outside of the development community. If the use cases only describe the system from the programmers' level, they won't meet the needs of the stakeholders who also need them to understand the system.

This was one of the problems our consultant for *Wings Over the World* ran into. In our story the analyst was told that the *Wings Over the World* development teams were "just tearing into the use cases, and that that had already written a 30 page use case for booking a flight." We don't know about how you like to spend your time on a project, but reading a 30 page use case is not our idea of a fun and productive activity. Consider the following use case fragment:

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-13 Book Flight

Book Flight:

Level: User Goal

Main Success Scenario:

1. This use case begins when a customer contacts the travel agency and requests a flight.
2. The customer proxy captures the customer's trip origin and destination.
3. The customer proxy looks up the airport codes for the origin and destination.
4. The customer proxy captures the preferred departure times for the customer.
5. The customer proxy captures the customer's preferred class of service.
6. The system requests from the airline proxy a list of the flights available that match the customer's preferences.
7. The system retrieves the customer's profile to get their preferred airline and frequent flyer number.
8. The system sorts the list of available flights first by the customer's airline preference and those that are closest to the customer's preferred departure times.
9. Etc. etc for the next thirty pages.

Use Case 1-1 Use Case Horror: Thirty page *Book Flight* use case.

Would you want to read these steps for thirty pages? One solution to this problem is to maintain two parallel sets of use cases, one set written at a very detailed level required by the programmers, and a second set written at a much higher level of abstraction, much like a corporate report and its executive summary. Unfortunately, this approach has several problems. First the use case set remains relatively flat and without any context explaining why a given use case add value to the business. Second, the scope of these use case is often ambiguous, whether they are modeling the business or a supporting system. In our

Copyright © 2001 Adolph, Bramble All Rights Reserved

example, we have both business level functions and system level functions in one use case. To get around the problem of the scope many use case writers add a pseudo actor they call the proxy. Finally, it is highly unlikely that the parallel use cases will remain parallel for very long.

An approach that we and other use case writers have had success with is to create a hierarchy of related use cases, organizing them as stories within stories that unfold as you need to discover more detail.

Figure 1-14 Book Flight

Book Flight

Level: User Goal

Main Success Scenario

1. This use case begins when a customer calls and requesting a flight.
2. The customer describes their flight needs by specifying their origination, destination, travel dates, and preferred departure times
3. The system looks up all flights that match the customer's travel preferences and presents the travel options to the customer.
4. The customer selects a flight.
5. The system builds a flight itinerary for the customer.
6. The reserves the flight for the customer.
7. The customer provides a credit card number and charges the price of the flight against it.
8. The system issues the ticket to the customer.

Use Case 1-2 Shortening *Book Flight* with EverUnfoldingStory(117)

Now we need more details we unfold the story and follow a more detailed of the use case set.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-15 Book Flight

Find Flight

Level: SubFunction

1. This use case begins when a customer contacts the travel agency and requests a flight.
2. The travel agent captures the customer's trip origin and destination.
3. The travel agent looks up the airport codes for the origin and destination.
4. The travel agent captures the preferred departure times for the customer.
5. The travel agent captures the customer's preferred class of service.
6. The travel agent confirms that the customer's preferences are correct.

Copyright © 2001 Adolph, Bramble All Rights Reserved

7. The system requests from the airline reservation system a list of the flights available that match the customer's preferences which is presented to the travel agent.

Use Case 1-3 Lower Level Use Case *Find Flight*.

Someone who is only interested in understanding the steps involved in booking a flight and the possible alternatives (e.g. flights unavailable, bad credit, restricted fares, etc.) can read the *Find Flight* use case and see if the right process is captured. They can unfold *Find Flight* and look at the lower level use cases if they wish to know the details of how we find a flight, or build an itinerary.

The benefit of this approach is that it maintains the thread of the use case goals from the system vision to the lowest level use case. It helps us answer the question, why is this use case here? For example, why is there a *Construct Itinerary* use case? Because it is an essential step for the *Book Flight* use case which is **UserValuedTransactions(110)**.

Many use case writers implement this solution by using the includes and extends relationships. In UML terms, when you reference a lower-level use case from a higher-level use case, the higher-level use case includes the lower-level one. That is, the steps from the lower-level one could, in principle, be placed directly into the higher-level one. Figure 1-16 shows how they will structure their use case set with includes to show how lower level use cases are related to a higher level use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-16 Use Case Horror: Distorted include Association for EverUnfoldingStory

A lot of people like this approach because it provides a very satisfying drawing that clearly reveals a relationship between *Book Flight* and the other use cases. However, use case sets constructed in this manner can be confusing because of the myriad of different interpretations people have for includes and extends. We believe the application of includes and especially extends should be limited (see **InterruptsAsExtensions(191)**, **CommonSubBehavior(186)**, and **PromoteAlternative(196)**).

Center A The Museum For Contemporary Asian Art

Center A is an example of a use case set with the **EverUnfoldingStory(117)** pattern. At the highest level is the use case *Engage Art*.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-17 An Artistic Rendering of *Engage Art*

For those of you who are mystified by an artist's hand drawn use case we present a more traditional UML diagram.

Figure 1-18 Engage Art

Copyright © 2001 Adolph, Bramble All Rights Reserved

Engage Diversity.

The museum creates a forum in which the interests and cultures of everyone on the planet may be discussed.

Level: Summary

1. Audience bemoans the lack of diversity in society.
2. curator develops a mission
3. curator conducts research for an event by creating forums to seek the input of the audience.
4. curator produces event
5. curator hosts event
6. Audience participates in the event.

Use Case 1-4 *Engage Diversity*

Copyright © 2001 Adolph, Bramble All Rights Reserved

The *Engage Diversity* use case provided the context for *why* a museum of art exists in our society. From this we could create a set of lower level use cases that described *how* the museum would accomplish this.

Figure 1-19 Center A Use Case Model

You will note in the figure that *Center A* took some liberties with the UML notation for use cases. Being artists, they were not too impressed with Ivar Jacobson's stick figures for actors. They were far more comfortable with the model of hat that represented a role that someone played. What is truly exciting about the Center A model is that it has with stood nearly two years of debate and argument. The stakeholders have found it to be an excellent description of how a museum of art operates.

EverUnfoldingStory and UML Models by Dan Rawthorne

We recommend that you have only one level of abstraction on each use case diagram. If you insist on drawing something like the diagram shown in Fig 4-14, then do the relations

Copyright © 2001 Adolph, Bramble All Rights Reserved

between abstraction levels with a <<trace>> relationship. The <<trace>> relationship is a UML pre-defined stereotype of a dependency (dashed arrow) that is used specifically to trace between different abstraction layers in a model. The following diagram shows how Fig 4-14 can be shown to reduce the confusion that an <<includes>> relationship will cause.

Figure 1-20 Book Flight

Of course, if your drawing tool will not allow you to use the <<trace>> relationship as shown, then you must do what you can. Just keep in mind that using an include relationship may cause confusion… and at least you know what it should look like.

In the textual representation of the "lower-level" use cases you may want to show the traceability if the route through the **EvenEnfoldingStory** is not enough. For example, the use case "Reserve Passage" may look like the following.

Reserve Passage

Traces to: Book Flight, Book Passage

1. The Agent …

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.6 Tradeoffs and Collaborations

Properly organizing your use cases can bring a lot of bang for your buck. An efficient, well thought out structure makes it much easier for everyone, readers and writers alike, to use your use case. Organization requires effort, and everyone writing use cases needs to fully understand the system's purpose, what it does and does not do, who will use it, and what its boundaries are, so that they can accurately describe these facts to their audience. Furthermore, they should organize their use cases so that their audience can easily follow them and understand their value. While it requires some effort to write this way, structuring use cases is fairly straightforward and quickly pays for itself.

The first level of organization is as a set or collection (see Figure 1-21), and we want to organize its contents in a friendly, easy to use manner that allows the user to see the system as a whole, then shift their focus to its individual pieces. A good way to do this is to describe the system as an **EverUnfoldingStory(117)**, which consists of several complete subsets of use cases that present the system in increasing levels of detail. Each of these subsets can stand alone, and allow the reader to examine the system from their particular level of precision, yet each subset expands into the next level of detail, allowing the reader to 'zoom-in' on the parts they find interesting. This style allows some readers the ability to examine the system from the 50,000-foot level, as well as focus on some of the low-level details of the system as well.

Creating an **EverUnfoldingStory(117)** requires the writers to have a **SharedClearVision(95)** of the system, so that they can clearly describe its purpose, and inform their audience as to what the system will and will not accomplish. To successfully reveal this vision, the writers need to clearly answer three questions. We have identified three patterns to help them do this. First, we need to know who will use the system. We need a **ClearCastOfCharacters(105)**, describing everyone's role in the system. Second, we need to know what actions the users expect the system to do, actions that are **UserValuedTransactions(110)**. Lastly, we want to know where these actions are meaningful to the user; we want to know the system's **VisibleBoundary(101)**.

These three helper patterns are tightly coupled. If the system's boundary changes, then its cast of characters will grow or shrink as more actors move inside or outside of the boundary. The converse also holds; adding or removing actors from the cast will expand or decrease the system's scope and shift its boundary accordingly. Changing the cast of characters also

Copyright © 2001 Adolph, Bramble All Rights Reserved

impacts the need for services; new actors require new services and removing actors usually reduces the number of services required.

Figure 1-21

Relationships between the Use Case Set Patterns

Sets contain use cases. The next chapter describes the characteristics of good use cases within the context of a set as an **EverUnfoldingStory(117)**.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1 The Use Case

Precise But Unreadable

After flying through the night, you've finally arrived at *Wings Over the World's* corporate headquarters in New York. You're not feeling too bad considering your lack of sleep and you take it as a good omen that you were actually able to hail a cab during a rainstorm.

After the initial meeting with the Chief Information Officer, you decide to sit down with their chief architect Ahmed and step through his 30-page use case that you spent most of the night reviewing. He's been with *Wings* for three years, and graduated from Cornell about four years ago. You remember meeting him during your sales presentation weeks ago; he's got a keen mind and a wonderful supply of terrible jokes. In a way you feel guilty tearing his meister work apart, knowing that he must have spent two weeks writing it. But Ahmed is the type who likes to learn, and while a little hurt that he's wasted so much effort, you see the excitement growing in his eyes as he learns.

You: "Ok if there was just one problem that I could single out it would be the inclusion of the business rules into the *Change Seat* use case. Easily seventy percent of the steps here relate to the rules for seat assignment and upgrades. Just taking those steps out shrunk the use case down to 10 pages.

Ahmed: "But in the travel industry, the rules are everything, that is what our reviewers will be looking for. I have to capture that information and the use case is perfect for that because it shows the circumstances for when the rule is applied, and I can write alternatives that describe what happens when the rule is not complied with.

You: That's true, but the use case should anchor this information rather than incorporate it. What we can adorn the use case with business rules, constraints and everything else. The information is still there, it's just that it won't hide the essence of the use case.

Ahmed: OK I like that, that's great, what else?

You: The other thing that we have to look into is the goal of this use case, because it seems that there it really addresses two separate goals.

Ahmed: What do you mean?

You: The use case begins when the traveler either wants to change his assigned seat or upgrade to the next level of service. That seems like two distinct goals to me.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Ahmed: On the surface it may, but when we constructed an object oriented analysis model of the seat change we discovered that an upgrade is just a seat change – to a nicer one mind you - but still just a seat change. So we thought if we used the *Change Seat* use case to do the upgrade then we could re-use the *Change Seat* components when we build the system.

You: That's not what we're trying to do here. We just want describe the black box behavior of the system. How that system implements the behavior that will come out in analysis later. Our concern right now is just trying to capture user needs and stake holder interests, you know that stuff I told you about the **UserValuedTransactions(110)**? Trying to combine the two goals into one use case really bloats it up and makes it much more complicated and difficult to comprehend. Right now, half your alternatives are based on whether the seat change is for an upgrade request. The upgrade seat should in my opinion be separate use case.

Ahmed: Even so, there would be a lot of steps in common between the *Change Seat* and *Request Upgrade*

You: Well if that really is the case, then we can factor out the common courses of actions into a use case that is included by both. We have a principle for do that which we can discuss later. One last thing, why are you using OCL (Object Constraint Language) in your descriptions? Is there anyone here besides you who can read this?

Ahmed: Well Ralph and I are having a bit of a disagreement. He really likes the use cases, but thinks that writing all details are just busy work and the use cases he's written are pretty ambiguous. He's assuming that the only people who'll read his use cases are billing experts. I'm trying to explain to him that precision is important because we're outsourcing a large part of the project to you guys, and the more precise I can be, the easier its going to be for you, and the less chance there'll be any miscommunications. So I thought I would use OCL in my use cases as an example that Ralph and the others can follow.

You: I'm probably the only one my group who can read OCL and your travel agents certainly won't be able to. How are you going to get them to review these use cases?

Ahmed: Well I thought we could have a few seminars and teach them OCL.

You: You're already having a hard time getting the agents onboard for this project, do you think making them learn OCL is going to help you?

A use case is a story that describes how an actor uses the system to obtain a specific measurable benefit. Like all stories, a use case can be a good story that clearly expresses the author's vision and is well received by the audience. Or it can be confusing, obscure the author's vision and be rejected by its audience.

Copyright © 2001 Adolph, Bramble All Rights Reserved

What are the properties that make a use case easy to read and comprehend? Like a good story, a use case should not have multiple plots or plot fragments. Rather it should address a single and complete goal that supports the system vision, which we call a **CompleteSingleGoal(132)**. The actor will either succeed in reaching this goal, or fail to completely reach it.

One difference between a use case and a story is that a use case has more than a single story line. There may be numerous alternative ways the actor can reach, or fail to reach the goal. The use case must describe all the different courses of action possible as **ExhaustiveAlternatives(140).**

How should we present all these different story lines in a use case? Should we exhaustively write every possible alternative as its own story? Or should we try to write the use case story like a computer program with nested ifs, loops and other complex logic? Neither approach is satisfactory because people find the exhaustive scenarios redundant, and most people prefer not read anything that looks like program logic. The favored approach is to structure all the different alternatives as **ScenarioPlusFragments(136)**, where we write the expected path for achieving the **CompleteSingleGoal(132)** as a simple single story line called the scenario, and then write each alternative as an incremental addition to that story, i.e. the fragments.

All good stories should have a good catchy title, and a use case is no different. The title should be verb phrase, or **IntentionRevealingName(144),** that reminds the readers of the use case's goal.

Good stories do not distract their readers with background information or references. Rather, they include these details as supplements, such as the list of references or an appendix. The same is true of use cases; we should not distract the reader with technological details, business rules and constraints. Rather we can use supplemental specifications to adorn the use case – **Adornments(147)**.

Finally, all good stories must be readable. A use case is suppose to be a work of non-fiction and serves as the basis for comprehending a system. Therefore it is important that it be **PreciseAndReadable(152)**.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.1 CompleteSingleGoal(132)

[Untitled].

CREATED/PUBLISHED
[between 1935 and 1945]

You have started to identify some of the actors as a **ClearCastOfCharacters(105)** and the services that those actors require as **UserValuedTransactions(110)**.

Improper goals will leave the writers deciding where one use case ends and another begins.

Copyright © 2001 Adolph, Bramble All Rights Reserved

One of the most painful events in American history was the Vietnam War, where tens of thousands of young lives were lost, and many more tens of thousands more lives ruined. The Vietnam War made a powerful and proud nation look inward and doubt itself. There are many armchair generals who are willing to offer numerous theories as to why the U.S failed to succeed in the Vietnam war, but a consensus always seems to form around a lack of a clear moral purpose.

The U.S. had never declared war on Vietnam, its involvement started with sending advisors to assist the South Vietnamese. And slowly the U.S. was drawn into the quagmire that had previously swallowed France. Battles were fought, some we're won, and others were lost. But it always seemed that they were fragmentary, with no goal.

Use cases that fail to address any goals are tangential and don't add value to the system. Those that fail to completely address one goal are insufficient, and leave the developers no choice but to look for other use cases, or guess at some parts of the system, while those that address too many goals tend to be too complex.

To build the correct system we need to understand how an actor achieves some accomplishment of value. A story is organized into chapters and in each chapter; we want to show the hero/heroine getting past some obstacles and achieve some particular accomplishment related to the overall story. In the same way, each use case should describe one significant accomplishment of interest to the primary actor.

We want to control complexity by partitioning our use cases into pieces that make sense to the stakeholders. A use case should describe a manageable unit of work. It should be a self-contained, logically cohesive unit that describes or references all relevant facets of a particular behavior. At the same time, it should be small enough so the reader can grasp its meaning, and not be overwhelmed by size or detail. We must consider several factors for controlling the complexity of our use cases, dealing with time and size.

> *The duration of the actor goals may vary.* An actor can have some goals on the minute-by-minute level, some on the one-sitting level, and some that take multiple days to accomplish. Use cases developers often miss this observation. Many people tend to focus on goals that an actor can complete rather quickly in a short session, which can result in small fragmented use cases. Worse, they can complete overlook services that are important to the actor.
>
> *Excessively large use cases can bury the stakeholder in an avalanche of details and obscure the purpose of the use case.* The longer a use case, the longer it takes to

Copyright © 2001 Adolph, Bramble All Rights Reserved

write, and the harder it is for the reader to keep all the details straight. Use cases resembling Russian Novels are bad because they are too complex, contain too much information, and provide too many alternatives. The readers and even the writers can lose sight of the use case goal, implement the wrong service, and emphasize the wrong characteristics of the system, resulting in additional features of no value to anyone. Once again, the purpose of the use case set is to convey how the system brings value, rather than exhaustively describing everything.

Large use cases can inhibit reuse. Both size and detail make it hard to reuse basic features. Larger use cases can hide important system properties, making it hard to identify reusable services. Larger use cases tend to contain implementation details because writers have a tendency to substitute details for completeness. Such an approach is likely to contain solutions rather than address stakeholder needs and leads to rigid, narrowly defined transactions that are too precise to reuse in other contexts. It is simply easier to reuse smaller pieces than larger ones.

Excessively small use cases will describe only a fragment of some accomplishment of value. If writing use cases like Russian romantic novels is not appropriate, then what about writing them as partial stories? Unfortunately, this type of use case will only cover a portion of the overall behavior of the system, so the reader will not see its overall purpose. This style of writing results in cliffhanger endings which are wrong because each use case needs to represent a "complete usage of value" to the actor.

The narrow knowledge of Subject Matter Experts may push us towards small use cases. Experts often have a narrow focus so they tend to propose those use case transactions with which they are familiar. Many experts favor smaller transactions because they are well versed in some particular process, but don't necessarily understand the purpose or goal behind that process. Yet these transactions only partially satisfy the actor's need, and often require one or more additional transactions to satisfy the goal of the actor.

Therefore:

Write each use case to address one complete and well-defined goal. That goal may be at any level in the EveryUnfoldingStory.

Select and name the primary actor goal that you wish to highlight. This goal may require a few seconds, days, a weeks, or months to accomplish, and can be at any level. Being unable

Copyright © 2001 Adolph, Bramble All Rights Reserved

to come up with a suitable name may indicate that you do not have a goal after all. Good characteristics of a goal are:

- it is associated with a well-defined actor;
- it is valuable to the actor or the stake holder on whose behalf the actor is working
- it is consistent with other goals that you have identified for the system at this level. If the goal is not appropriate for the level of abstraction that you are working at consider re-organizing the use case model as an **EverUnfoldingStory(117)**;

Describe the different ways the primary actor can either reach or fail to reach this goal. To accomplish this, structure the use case as a **ScenarioPlusFragments(136)**, with a main success scenario describing the nominal case. This is followed by a set of fragments describing all reasonable alternatives that can affect the user as they attempt to achieve their goal.

A use case should leave the system in a well-known state when it ends, where the primary actor either fully obtains their goal or it is as if the use case never happened. Use cases should not contain cliffhanger endings, rather the initiating and terminating events should be clear and unambiguous to the actors.

Occasionally, your use case may need to address more than one goal. This can be done without violating the spirit of **CompleteSingleGoal(132)**, as long as the use case is cohesive and achieves a unified purpose. It is much better, however, to write the use case to address a higher-level goal that encompasses sub-goals, as well as any other issue associated with the higher level goal.

Examples

Wings Over the World

In this installment of the *Wings Over the World* story, our consultant reviewed the *Change Seat* use case that had been written by Ahmed. Its goal was to change a traveler's seat, either as a normal exchange or an upgrade.

We can immediately see that this use case really addresses two goals, obtain a different seat and upgrade a seat. While Ahmed may have correctly argued that a seat upgrade is just the same as a seat change from a system point of view, we want to write use cases from the actor's perspective. For most air travelers, an upgrade is a very different goal than a seat change.

Combining these two goals will likely lead to long, complex alternatives that can obscure the differences between them. You can capture any steps that are truly common to both in a "uses" (UML <<include>>) use case to avoid redundancy (see **CommonSubBehavior(186)**).

Copyright © 2001 Adolph, Bramble All Rights Reserved

Select Exit Row Seat is another example of a poor use case goal. It says:

> Select a seat in a designated exit row for the traveler.

The qualification in this example should make us question whether this statement represents a complete goal, or just a fragment of another. Most airlines will not assign a seat in a designated exit row until the passenger checks in and airline staff can confirm that the passenger is physically capable of opening the exit door in an emergency. Most likely, this use case should be an alternative to *Change Seat*.

A frequent cause for writing fragmented use cases is that many writers tie each of their system level use cases to specific interface details, with one use case per user interface form. This practice frequently results in numerous system focused use cases, such as *Capture Traveler's Desired Itinerary*, *Present Itinerary to Traveler*, *Capture Traveler's Preference*, *Capture Traveler's Personal Information*, and *Capture Traveler's Payment Option*. Do these examples truly reflect a useful goal for a Traveler, or are they all just fragments of the goal *Book Flight*? The use case is inappropriate for capturing user interface details. Such information should supplement a use case as an **Adornments(147)**.

Writing use cases around goal fragments leads to the creation of numerous closely related use cases without any structure tying them together. In this example, if you were reviewing twenty or more use cases, would it be obvious to you that *Change Seat* and *Select Exit Row Seat* are closely related? Probably not and you would lose an opportunity to simplify the system and make it more comprehendible by consolidating closely related behavior.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.2 IntentionRevealingName(144)

Restaurant, Sikeston, Missouri.

Vachon, John, 1914-1975, photographer.

CREATED/PUBLISHED
1940 May.

You have identified a use case associated with a **CompleteSingleGoal(132)**.

Meaningless generic names will not set reader expectations or provide a focal point that people may conveniently refer back to.

Names convey meaning. A picture may be worth a thousand words, but descriptive names can say much, much more. Just the name "The Grand Canyon" instantly invokes vivid images of colors, plateaus, rock formations, mules, and snow-covered vistas accompanied by the gentle strains of the Grand Canyon Suite in the listener's mind.

Businesses recognize the value of a good name, and will spend millions of dollars researching good ones. They want something 'snazzy' that instantly catches people's attention and draws them to their product. They also want to avoid names with negative connotations that hurt their product's image. A popular business school example of a bad

Copyright © 2001 Adolph, Bramble All Rights Reserved

product name was General Motor's attempt to introduce the Chevrolet Nova into Spanish speaking countries. Unfortunately, Nova sounds like the Spanish for "no-go".

Using descriptive names for your use cases is a good practice, because they accurately reveal the intention of each use case.

The name sets the tone and association for the audience and can provide a focal point for the writer. A name should reveal the use case's intention, and reflect the **CompleteSingleGoal(132)** that the actor is trying to achieve. A meaningful name that adequately describes a use case's purpose can give the reader more insight into a use case than several paragraphs of text. It should offer a preview of things to come, and be descriptive enough to stand by itself, allowing the reader to work at the set level without having to delve into the contents of the use cases comprising the set.

A descriptive, goal based name can also help the writers better understand the essence of the use cases that they are writing, and constantly remind them of the goals that they are trying to accomplish.

An appropriate name provides a handle the use case. One of the benefits of software patterns is that the names of the various patterns convey sufficient meaning to become part of the development nomenclature. For example, developers can use the terms "Visitor", "Bridge" or "State" to describe fairly complex components to their audience with very few words. In the same way, a meaningful name can adequately describe a use case so that the audience doesn't need to read it to use it.

Appropriate use case names allow you to see the big picture and work on the whole set. People working with a system at the 50,000-foot level don't care about the low-level details of the individual use cases. In fact, these details make it hard for people to work at this level because they quickly obscure the abstract concepts that they are trying to understand or explain. Meaningful use case names assist those working at this level by revealing intent of the various use cases to the audience. They don't have to stop and read each one, but can instead focus on the system as a whole, without distraction.

Therefore:

Name the use case using an active verb phrase that represents the goal of the primary actor.

Begin each name with an active verb that describes the use case's goal, and follow this verb with a phrase describing its object. Be terse, yet descriptive enough to capture the use case's

Copyright © 2001 Adolph, Bramble All Rights Reserved

essence. For example, a good name for a use case describing paying an accident claim is "Pay Claimant".

Choosing a good name for a use case is important, because it conveys the use case's purpose to the audience. Descriptive names are easy to work with and can significantly improve the cohesiveness of your use cases. If you have trouble coming up with a name, then you should reconsider your use case.

Examples

Insurance Claims

Use Case Naming Horrors

Consider the following names for some insurance use cases.

1. Main Use Case
2. Claim Process
3. Use Case 2
4. Process Stuff

It's nice to know that *Main Use Case* is apparently the important one, but what does it do? This kind of name is okay for those who know what it does, but it doesn't help anyone who is unfamiliar with the system. Names like this are not evocative of the use case goal and force the audience to read most of the use cases to determine whether they are interested in each one or not.

Claim Process appears to be a better name, but what does it imply? This use case name doesn't define how the system handles all claims, or how users enter claims, nor does it tell what kind of claim the use case defines. Not only does it force the reader to read the use case to determine its purpose, but they must also look through the whole set to find the ones about the claims in which they are interested.

Use Case 2. Believe it or not, some of us have seen use cases named *Use Case 1*, *Use Case 2* before. Of course, no one had any idea as to what they covered, but at least knew there were at least two of them.

Process Stuff. Someone actually used this one too. This use could be about anything, but at least it is portable.

Good Use Case Names

1. File Accident Claim

2. Approve Property Damage Claim

3. Report Fraudulent Claim to Police.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Each of these names is descriptive, and conveys some meaning to the audience so that they don't have to read every use case in the collection to understand it. They also serve as a convenient reference so that the readers can easily locate only the ones in which they are interested.

You can number use cases for organizational purposes. For example:

Use Case 42: *Approve Property Damage Claim*

allows us to use descriptive names and organize our use cases numerically. Very short comments are also acceptable. If the use case you are describing drives the rest, they you could describe it as:

Use Case 1: *Book Airline Reservation (Main)*

We still refer to it as *Book Airline Reservation*, but we also know that it is the first use case in our collection, and it is the main one as well.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.3 ScenarioPlusFragments(136)

<<caption and credits missing>>

You are writing a use case description for the primary actor's **CompleteSingleGoal(132)**.

The reader must be able to easily follow the path through the specific scenario or story that they are interested in, otherwise they are likely to become frustrated or miss important information.

A delightful style of children's book is the "choose your own adventure" story. These books draw the reader into the story by asking them to choose the hero's course of action. Do you heed the warning of the old gypsy and avoid the dark path into the forest, or do you venture into the forest to seek the legendary magic amulet? The reader must choose one alternative. If you heed the gypsy warning then the reader turns to page 59 to continue the thread of the story. If you choose to boldly venture into the forest, then the reader turns to page 112. One choice advances the hero closer to their goal while the other may lead to failure and death.

A "choose your own adventure" story is a good style for writing use cases, because it makes it easy for the readers to follow the various paths in a use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

An interesting use case needs to capture alternatives to the main success scenario. Many different things can happen as an actor attempts to attain a goal. Things don't always go right. Sometimes, if the system is prepared, it can detect the problem and take the steps necessary to rectify it. Other times, however, things can be screwed up too much to continue, and the best course of action is to stop. The system should be able to handle these situations gracefully, and to do that its developers need to know all of the things that they can reasonably expect to go wrong before they build the system.

The difference between a use case and a "choose your own adventure" story is that the adventure story has no "main" success story line, so the reader must make the choices at various moments while reading the book. Unlike the story, a use case is a description that should reveal the crisscrossing structure, not hide it. There are several ways we can show this structure:

> *Writing every alternative as a complete story will obscure the differences between story variations*. We could write every possible variant of the plot as an entire story. This has the advantage the reader can simply read any complete variation of the story from top to bottom. However, there will be a great many of these to read, and at some point the reader will have trouble telling just the difference is between, say, plot variant 17 and plot variant 18. The two variants might be almost entirely similar, perhaps differing in only two sentences. At that moment, the structure becomes a drawback, since the reader will get more confused over time, instead of less.
>
> *Separating every variant out also makes the writer's life difficult*. Any change in the story line usually affects a number of the variants, and the writer must find and change every variant, trying to be consistent and accurate in the change. This is both tiring and error-prone, so we want to keep the number of variants to a minimum.
>
> *A large number of "if statements" will clutter the story*. A second alternative is to put "if" statements into the story, which shortens the writing. Programming languages are constructed this way, and many people have been trained to write in the form, "If such and such is the case, then so and so happens."
>
> If our story had only one of these "ifs", then this would indeed be a practical way to proceed. However, most of the stories that get written into use cases have a large number of variations, half-a-dozen up to several dozen, sometimes "if" within "ifs". This structure quickly gets out of hand, and the reader is soon unable to follow the storylines. It is important that every reader, from end user to executive to programmer, be able to understand the crisscrossing story, and so the "if... then..., but if... then..." storyline structure is no longer recommended.

Copyright © 2001 Adolph, Bramble All Rights Reserved

> *People cope well with incremental complexity*. There is a surprising third alternative that has shown itself to be effective. People seem to be quite adept at modifying and adding complexity to a base concept a little at a time. We often see people describing a simple concept, and then saying, "Well, actually, it isn't that simple. This can also happen…", adding a twist to the story. The listeners to these explanations are able to build up a complex understanding, a small piece at a time. We can use this idea to convey the intertwining story line. [COC2001 – pg 87]

The main success scenario needs to be clearly identified. If all threads are written as scenarios, then which scenario is the most important one? The readers and the builders need to know which scenario is the main success scenario that you expect the primary actor to typically seek.

Therefore:

Write the success storyline as a simple scenario without any consideration to possible failures. Below it, place story fragments that show what alternative condition might have occurred.

The main success scenario should describe how the primary actor accomplishes the goal in a straightforward manner. It doesn't have to be the shortest possible path or the only possible successful path, but it should be the normal desired path for reaching the goal, i.e. the one the users are most likely to follow.

Create a header for each scenario fragment after the main success scenario. First, describe the **DetectableConditions(168)** that cause the actor to take the branch. Then describe what happens in this branch, and how it ends, either rejoining the main story, or ending in failure.

Each scenario in a use case should have a clear purpose. Write each scenario in the use case as a set of **VisibleActorIntent(161)** steps, making **ForwardProgress(164)** towards the use case's **CompleteSingleGoal(132)**.

Usually, the fragment ends in a fairly obvious way, by retrying the step that preceded the branch, fixing up the step so that it can rejoin the next step in the main success scenario, or simply failing altogether. On occasion, it is necessary to explicitly name where the story picks up again.

Occasionally, some of the fragments become long and complex enough to obscure the rest of the use case. At this point it would be useful to extract the alternative and put it into an extension use case (**PromoteAlternative(196))**.

Note: In terms of **BreadthBeforeDepth(63)**, this pattern identifies these stopping points:

Copyright © 2001 Adolph, Bramble All Rights Reserved

- Main Success Scenario
- Naming the conditions
- Finishing some of the fragments.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Examples

Auto Insurance Claim Handling

Use Case: *Get paid for car accident*

PRIMARY ACTORS:	**Claimant** –	Accident victim making claim.
	Insurance Company	Company insuring **Claimant**
	Agent	**Insurance Company** representative processing claim.
Level:	Summary	

Main success scenario

1. **Claimant** submits claim with substantiating data.

2. **Insurance Company** verifies **Claimant** owns a valid policy

3. **Insurance Company** assigns **Agent** to examine case

4. **Agent** verifies all details are within policy guidelines

5. **Insurance Company** pays **Claimant**

Extensions:

1a. Submitted data is incomplete:

1a1. **Insurance Company** requests missing information

1a2. **Claimant** supplies missing information

2a. **Claimant** does not own a valid policy:

2a1. **Insurance Company** declines claim, notifies **Claimant**, records all this, and 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, and terminates proceedings.

4b. Accident violates some minor policy guidelines:

Copyright © 2001 Adolph, Bramble All Rights Reserved

4b1. **Insurance Company** begins negotiation with **Claimant** as to degree of payment to be made.

Use Case 1-1 *Get paid for car accident*

(from Writing Effective Use Cases [COC2001])

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.4 ExhaustiveAlternatives(140)

[Untitled].

CREATED/PUBLISHED
[between 1935 and 1945]

You are writing a use case as a **ScenarioPlusFragments(136).**

A use case may have many alternatives. Missing some means the developers will misunderstand the system's behavior and the system will be deficient.

We had just moved, and this was our first winter in our new home. For two days it snowed, dumping nearly three feet of heavy, wet snow. Even with chains, there was no way my car would get out of the driveway, and I was forced to begin shoveling. After an hour I had managed to clear my driveway, feeling quite satisfied with my accomplishment. That is, until I saw the municipal snow plow barreling down the street, pushing a wall of snow in front of it, directly into my freshly shoveled driveway. Swearing mightily, I cleared the snow away, and then I noticed the wall of snow that the plow had pushed into my neighbor's driveway. He was an older gentleman, and I thought it would be neighborly to clear the snow out of his driveway.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Halfway through clearing his driveway, I heard the rumble of machinery down the block. I thought, “Oh no, not again, the plow’s coming back to finish me off”. Then I saw that the city was using a small front-end loader to clear the snow the plow had pushed into people’s driveways! I didn’t know whether to laugh or cry, because I could have just waited warm and cozy in my house for this plow to come by.

Later, when I told my story to my neighbor, he laughed and apologized profusely, “I’m sorry, I forgot to tell you that the city clears the driveways after the plow comes through.” I had gone out, got cold and soaking wet, because I did not know that the alternative of just sitting and waiting was available to me.

A good use case describes all important alternatives, so that the developers can properly address potential problems, protecting the users from unpleasant surprises.

Investigating variations sometimes results in the discovery of new actors and use cases. Some variations differ so radically from the use case’s main course that they introduce a new dimension of functionality into the system, which should stand alone as a separate use case when it is sufficiently complex or significantly different from its base use case. Even when the variation doesn’t require new use cases, it can introduce new actors into the system by interacting with previously undefined users or systems in the course of its normal events.

Developers need to know how to handle errors. Error handling and exception processing comprise the bulk of most software-based systems. Certain classes of errors are well known and easy to handle, but a significant number of errors are hard to predict and difficult to detect. For example, it is easy to detect missing data, but not so easy to detect problems with boundary conditions or subtle interactions between seemingly unconnected fields. It is difficult for a system’s developers to identify potential errors because they are working under time constraints, and good error finding requires “out of the box” thinking. Developers are focused on using the system correctly, and often fail to consider cases of incorrect system use, because these actions involve “breaking the rules”. But users do not share this constraint and often, through unfamiliarity, necessity, or experimentation, attempt to use the system in ways the developers never considered. A robust system must be able to handle these types of situations without failing. But the only way to handle these errors is to identify them before building the system, so that the system knows how to identify and handle them. One of the most important facets of the analysis process is identifying potential error situations that the system is likely to encounter, and developing uniform policies for handling them.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Schedule pressure limits the time developers can spend identifying variations. Identifying variations requires time, creativity, and out of the box thinking and it can be mentally tiring thinking about many of them and keeping them straight. Developers often fail to see the value in this exercise, especially when many of the differences are so miniscule that they appear to be meaningless, and they are experiencing intense schedule pressure. But this kind of thinking is backwards, as this information is absolutely essential for estimating the scope of a project, as variations represent a significant, and possibly majority portion of the effort involved, and is needed to help determine an accurate schedule for the project.

Some of the variation handling polices need significant investigation. Identifying errors is often the easy part. Handling these errors can be much more difficult, and often requires significant time and effort. It is easy to decide that certain input fields should allow only integers, but it requires a coordinated policy and potentially complex components to efficiently implement this feature for tens or hundreds of similar fields. Moreover, other input fields may have somewhat similar constraints, and it is highly desirable to use the same mechanism to handle these fields as well. But designing this kind of feature requires much time and effort.

Having information about the variations helps developers build a robust design. The better the developers understand a system, the easier it is for them to design and build it. Even when using iterative development processes, it is often much cheaper and simpler to build a system when its developers are aware of error conditions at the beginning of the development process. Having this knowledge allows them to seamlessly incorporate exception handling into their system architecture, and handle these conditions in an effective manner, providing a much more robust and user-friendly system. They can still add functionality later on, but it is hard to seamlessly incorporate these add-ons into the system's architecture, and the result it is often not as well structured as it would have been if they were part of the original system.

Therefore:

Capture all failure and alternatives that must be handled in the use case.

Once you have identified all of your use cases and their main courses, identify as many variations from the main course as you can. Capture all of the variations that you want the system to handle, yet be selective. These variations should be **DetectableConditions(168)** that result from the user using the system differently, or from error conditions. Eliminate all the variations that begin with conditions that the system cannot or does not have to detect, and then merge together all the variations that produce the same resulting behavior.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Document the remaining variations in the use case using **ScenarioPlusFragments(136)** format, listing the appropriate **DetectableConditions(168)** as the first step of each scenario or fragment.

Note:

This is a good point in time, when doing **BreadthBeforeDepth(63)**, to get all the alternative conditions that will force different paths, before spending much time on the extension handling. The list of conditions will act as a table of contents of issues that need to be researched over the following days.

While working **BreadthBeforeDepth(63)**, this is the third place at which to pause and examine your work. (The first is after you have the actors and **UserValuedTransactions(110)** named. The second is after the main success scenario has been written).

Example

Email Access

The following example demonstrates how to itemize a use case's alternate courses. Instead of defining a separate use case for each of the alternate and error courses, a use case itemizes each as a deviance from the original. This format nicely encompasses those situations where the user performs several separate actions, such as saving some email, forwarding some email, and replying to still other email.

Use Case: *Access Email*

PRIMARY ACTORS: **Reader** – Person reading email.

LEVEL: **USER GOAL**

Main success scenario

1. User initiates use case by logging on to the system and requesting email.
2. Server displays user's email screen.
3. User and server repeat the following sequence indefinitely.
4. User selects email.
5. Server display email, and marks it read.
6. User selects "quit" option.
7. Server updates user's email account information and logs off user.

Alternate Courses

User saves email

4. User selects "Save Email" option.
5. Server marks email as unread.

Copyright © 2001 Adolph, Bramble All Rights Reserved

User replies to email

4. User selects “reply” option.

5a. Server prompts user for a reply.

5b. User enters reply and instructs server to send email.

5c. Server sends email to address of message’s originator.

User forwards email to another user

4. User selects “forward Email” option.

5a. Server requests address.

5b. User supplies information and instructs server to send email.

5c. Server forwards email to specified address.

Error Courses

Invalid Login

1. User supplies incorrect login id or password.

2. Server displays error message and initiates a new login sequence.

Use Case 1-2 *Access Email*

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.5 Adornments(147)

<<caption and credits missing>>

You are writing the use case description as a **ScenarioPlusFragments(136)**.

The inclusion of non-functional requirements in a use case can quickly clutter and obscure the use case.

Okay, I will admit that I am an opera fan[1]. I enjoy listening to people shout at each other in a foreign language while wearing strange costumes. Most people say they don't understand opera but the stories behind the operas are often fairly simple, usually something along the lines of young romantic love thwarted by meddling parents or the patriarchy. An adequate synopsis of an opera can usually fit on two pages of a program.

[1] Of course it is up to you to guess which one of us is the opera fan!

Copyright © 2001 Adolph, Bramble All Rights Reserved

Of course there is a great deal more information about an opera that assists your understanding of the opera. For example, what are some of the critical interpretations of the opera? Or, what were the historical influences on the composer that led to the creation of the opera? The answers to these questions are never included in the synopsis because they would clutter and obscure the synopsis. Rather, separate articles in the program answer these questions.

A use case is similar to a program synopsis because it offers an understandable explanation of a complicated production. You should be able to read a use case and understand how a system delivers value to its actor without worrying about user interface details, data storage details, or other non-functional requirements.

The purpose of a use case is to clearly express the functional requirements of a system. A use case should show what a system does, not how it does it. It should provide a clear, concise description of the services that a system provides its users in plain, everyday language [Jac1992]. A reasonably intelligent person, regardless of their technical background, should be able to pick up a use case and quickly understand the functionality that it describes.

We often discover non-behavioral information while researching functional requirements. It is very common to collect all kinds of valuable information when gathering system requirements. Some portions of this collected data, such as the types of actions to perform or the number of users to support, form a basis for the resulting requirements. Other portions, including such items as sample reports, example GUIs or generated data files, don't directly map into the requirements, but instead serve to illustrate and clarify them.

Inclusion of non-functional requirements in the use case is distracting. Excessive detail makes use cases harder to comprehend and forces the reader to wade through more material and grapple with issues that they don't need to understand. Therefore, we don't want to include this extra information in the use case proper. The most common type of unnecessary information is user interface details, which can quickly bloat an otherwise simple use case into an incomprehensible monster. However, a detailed description of how the system captures and validates customer-shipping information usually does not help your understanding of a use case describing how a customer makes a purchase.

But we do not want to lose information that aids in understanding the use case or is valuable to the developers. Just because some data describes specific details or is highly technical doesn't mean that it isn't valuable. Many of these non-functional requirements offer insight into the use case and could well be information that the developers need to implement the

Copyright © 2001 Adolph, Bramble All Rights Reserved

system. Because much of this information is either specific or relevant to a use case, we need to keep this information close to the use case so that the readers are aware of its existence. While this information can be captured in separate documents we will lose a lot of the context that comes from the close association of the non-functional requirements with the use case.

Therefore:

Create fields within the use case template that are outside the scenario text to hold the auxiliary information that is useful to associate with the use case.

Non-functional details such as business rules, user interface sketch, external interface protocols, data validation rules, and even outstanding issues (so called TBDs – To Be Determined) can be added to the use case in a supplementary section. This keeps non-functional information from cluttering up the basic use case but also keeps the information associated with the relevant use case.

Examples

Wings Over the World

In this installment of the *Wings Over the World* story, our consultant is explaining to the Ahmed the chief architect the consequences of including the business rules in the use case. A fragment from Ahmed's *Change Seat* use case may appears as follows:

1. Traveler selects preferred seat.
2. The system verifies a seat has been assigned to the traveler.
3. The system verifies the seat is not in a designated exit row.
4. The system verifies the ticket is a full price economy ticket, class Y, or is a business class ticket class J. If not the system verifies, the ticket is either class L, M, or N, and the traveler is a platinum member of the frequent flyer program. The system verifies the seat assignment was not issued as part of a companion ticket. If not the system verifies…
5. The system yada yada yada and the use case terminates.

Use Case 1-3 Use Case Horror: *Book Flight* with Business Rules

This is a simplified fragment, and we should remember that airlines tend to have Byzantine rules regarding changes to tickets. We have seen use cases where the inclusion of the business rules have bloated what should be a simple use case into a thirty-page monstrosity. A *Change Seat* use case may be as simple as the following use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Change Seat

Level: User Goal

Main Success Scenario

1. This use case begins when the traveler specifies they wish to change their seat assignment.
2. The system verifies the traveler is eligible for a seat change.
3. The traveler specifies their flight number and day of departure.
4. The system displays the current seat map for the aircraft.
5. The traveler selects their preferred seat.
6. The system verifies the seat is still available.
7. The system releases the traveler's existing seat, and assigns their chosen seat.
8. The system confirms to the traveler their new seat assignment.

Alternatives:

2a: Traveler is not eligible for seat change

2a1 The system…

Business Rules:

Seat Re-assignment Eligibility

1. Unconditional seat re-assignment is allowed for full fare economy, and business class tickets, ticket classes J, and Y
2. Unconditional seat re-assignment is allowed for discount fare codes, L, M, and N if passenger is a platinum tier frequent flyer.
3. Ticket classes L, M, N may request seat re-assignment 72 hours prior to departure time to Premium Economy seats.
4. Ticket classes K, may request seat re-assignment 24 hours prior to departure time to Premium Economy seats.
5. Award tickets, class C, may not request re-assignment.

etc.

Use Case 1-4 *Change Seat* **Adorned with Business Rules**

Copyright © 2001 Adolph, Bramble All Rights Reserved

In this revised version of the *Change Seat* use case, we only test to see if the Traveler is eligible for a seat change. In a separate section of the use case – or even in a separate document – we capture the eligibility rules for seat changes. There are a several of advantages to this approach:

1. The reader can clearly see the flow of events because the use case description is not cluttered with the business rules.
2. It reduces the number alternatives in the use case. There may be dozens of rules regarding whether a traveler is eligible for a seat change. If we try to capture all of these rules as part of the use case, we risk specifying an entry in the alternatives section for each situation where we can fail the rule. But in this example, the net result of failing any of the rules is the same, i.e. the seat is not reassigned, so we have only one alternative – **DetectableConditions(168)**.
3. Whenever the business rules change, we do not have to update the use case scenario.

One of the most frequent mistakes use case writers make is that they believe the use case replaces all other tools for system specification. Business rules, data formats, and user interface navigation have no place in the use case description. But these items are important and valuable pieces of information that we want to associate with the use case. While this information should not be part of the use case description, it can be written up in supplementary documents that are anchored by the use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 1-1 Supplemental Specification Adorning a Use Case

Adapted from Writing Effective Use Cases [COC2001]

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.6 PreciseAndReadable(152)

Subcontracting. Passaic home workshop pool. Jack Vida, of Passaic, New Jersey, inspects precision lenses he ground for military use in his home workshop. In addition to this work, Mr. Vida employs four men to turn out machine tools for aircraft production as a member of a subcontract pool organized by the Howe Machinery Company of Passaic.

Liberman, Howard, photographer.

CREATED/PUBLISHED
1942 Mar.

You are writing the text for a use case as a **CompleteSingleGoal(132)**.

Use cases that are too complicated for non-technical readers, or too imprecise for developers, are deficient and likely to result in poorly built, inadequate systems.

Beginning guitarists quickly learn that it is difficult to determine how to play a song correctly from traditional sheet music, as it is possible to play any given note or chord at a number of different locations on a guitar. Traditional musical notation shows what notes to play, but it can't always show where or how to play them. For this reason, many guitarists prefer a specialized notation called tablature, which is very precise and easy to learn, demonstrating where and how to play a lick or riff in such a manner that a novice can try it. The lick may be very complex, and require much practice, but any player can at least see how it is done

Copyright © 2001 Adolph, Bramble All Rights Reserved

In a similar way, a use case should be written with an easy-to-understand precision that meets the knowledge level of most members of the use case audience.

A use case should be readable by both the stakeholders and the developers. Any use case can have several different audiences with varying needs and technical abilities. Higher level audiences, such as customers, can get turned off if the use cases are too difficult to read or too technical to easily follow. Likewise, lower level audiences like developers will not use them if they contain too much "fluff", or fail to adequately describe the system. Well-written use cases address the needs of several different audiences without unduly favoring any.

Developers tend to add detail and solutions. This tendency is natural, and hard to control, as people believe extra details add clarity. Unfortunately, these details are often unnecessary, adding little value to a use case. Worse, they can complicate it, making it more difficult for the reader to understand. Problem solutions are especially unnecessary in use cases, because the people writing use cases are still trying to understand and describe system behavior. They are hardly in a position to propose meaningful designs, as they haven't full determined the problems associated with this behavior. Well-written use cases describe the essential characteristics of a system's behavior without specifying technical details or favoring a specific implementation.

Non technical stakeholders are likely to miss necessary considerations. Non-technical people have difficulty understanding highly technical documentation. Those without an understanding of the problem domain are unlikely to make the proper inferences from it, as they are unaware of all of the nuances of the problem domain. Well-written use cases specify the issues involved with a system's behavior and spell out the consequences of its various actions.

We want to encourage a dialogue between the stakeholders and the developers to ensure accuracy of the requirements. One of the greatest benefits of use cases is that they make it easy for stakeholders, many of whom are non-technical, to understand a system's behavior, and determine if it is what they really want. If not, they can work with the system's developers to better describe the system so that it meets their needs. Unfortunately, developers often ignore this benefit because they are in a hurry to build the system. But, use cases are meaningless to the system developers until they correctly describe a system that the users really want. Well-written use cases contain a consensus of people having different system view's (**ParticipatingAudience(53)**).

Dual-models are unacceptable because they can become disjoint and are difficult to maintain. Having different sets of requirements documents for the customer and the

Copyright © 2001 Adolph, Bramble All Rights Reserved

developers is project nightmare. Projects are under too much pressure to maintain two sets off documentation. It is inevitable that these will documents will become disjoint and difficult to maintain. . Not only is updating multiple copies of a document unnecessary, tedious and time consuming; it becomes unworkable on most projects. Even worse, multiple versions of design documents unnecessarily introduce ambiguity. When someone refers to a use case, how do you know which one they are referring to? Are your tests based on what the developers did, or what the customers want?

Therefore:

Write the use case to be readable enough so that the customers bother to read and evaluate it, and precise enough so that the implementers understand what they are building.

Every use case you write should accurately and fully describe a **CompleteSingleGoal(132)**, without being so verbose that the audience cannot read it, or so high-level that it fails to communicate enough information for them to adequately understand it.

The cardinal rule for writing is "know your audience". Determine who needs these use cases, and write each one using their terminology in a style that they can easily understand. A less technical audience needs to understand the rationale for each action you describe, so write each step with **VisibleActorIntent(161)**. The correct level of precision and readability presents a moving target, so it is important that you understand your audience's needs and abilities when writing each one.

Do not substitute detail and verboseness for precision. A good general rule to use is "Never expect your audience to fill in the gaps". Use plain language text to describe essential system behavior. Include only that information necessary to describe the system's behavior, but describe the behavior clearly and precisely enough so that an uninformed reader can fully understand the consequences of the behavior, without missing important events. This precision not only benefits the customers, but the developers as well, as they should not have to make guesses when building the system.

Use **ExhaustiveAlternatives(140)** to describe all situations that the actor can reasonably expect to encounter, as well as what actions the system should take to resolve them. The resolution should be complete enough to give the developer a good handle on what to do, but not so detailed that it spells out every action to the nth degree; it must be **TechnologyNeutral(177)**.

Examples

Copyright © 2001 Adolph, Bramble All Rights Reserved

Wings Over the World(1) Readable but Imprecise

Ralph is the *Wings* billing expert who believes that formality in writing use cases is just busy work that is delaying getting down to the real work of building the system. Frustrated with the formality advocated by Ahmed, he wrote his own version of the *Change Seat* use case.

Use Case: *Change Seat*

PRIMARY ACTORS: **Traveler** – Person flying on airplane.

LEVEL USER GOAL

Change Seat

Actor: Traveler

Main Course

The traveler specifies their seat preference and the system re-assigns them the new seat.

Alternatives:

Traveler is not eligible for a seat change.

Traveler is informed that a seat change is not available.

Use Case 1-5 Informal Version of *Change Seat***.**

This use case is written as a brief or high level use case. It uses free flowing prose to describe the exchange between the traveler and the system. In a tight knit work group, this level of precision may be suitable, after all you do not want to waste a lot of energy writing detailed specifications if you do not have to (**BreadthBeforeDepth(63)**, **QuittingTime(71)**). On the other hand, the *Wings* project is being outsourced and this brief does not provide enough precision for a distant team to correctly implement this feature.

Wings Over the World(2) Precise but Unreadable

While imprecision results in ambiguity, excessive precision results in unusable use cases. Ahmed the Chief Architect had used UML Object Constraint Language (OCL) in his use case description. OCL is a formal language for specifying object constraints and is intended for situations in which natural languages were too imprecise for specifying constraints on objects. He has used OCL to describe the rules for seat change eligibility:

```
t : Traveler
f: Flight
        t.seatAssignment->notEmpty and
        t.ticketClass = #J or t.ticketClass = #Y
        t.ticketClass is in set and f.departureTime < 24 hour
```

Copyright © 2001 Adolph, Bramble All Rights Reserved

Ahmed's suggests that using OCL will reduce ambiguity in the use cases. There is justification for this point of view because *Wings Over the World* is outsourcing the project, and the more precise the specifications are then the less opportunity there is for misunderstanding on the part of the developers. Some software developers believe that use of formal languages in specifications can greatly reduce the potential for ambiguity in the specifications.

But this level of precision is too complex for most readers to follow[2]. Often the real problem may not be the different interpretations of the requirements that stake holders and desingers may have as a result of using natural language. Rather, the greater risk may be that the project's stake holders do not fully comprehend the formal language or worse are intimidated by it. Therefore we may not know if we have the correct specification in the first place. We may in fact precisely capture the wrong specifications because no one has the ability to clearly understand the language, and few people are willing to admit they cannot understand a spec.

A good specification is one that you can present to the stakeholders that they can read and clearly tell you "No, this is completely wrong, you do not understand my operation. Let me tell you how things really work here…" Although your description was incorrect, it was clear enough for the stakeholder to comprehend and tell you that it was wrong. On the other hand, if the stakeholder tells you in a hesitant manner after reading the specification, "Well, ok, I guess so. You're the expert in this stuff after all", then you are heading for trouble. This response can indicate that the stakeholder did not understand the specification and is withdrawing, leaving you holding the bag.

It has been our observation that higher levels of formality in specifications give the developers a false sense of security that there is less opportunity for problems resulting from ambiguity. Nothing can replace a good on going dialog with the stakeholders.

---

[2] David Parnas reportedly once quipped on the topic of formal specifications "I like writing these but I really hate to read them!"

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.7 Tradeoffs and Collaborations

As its name would suggest, the use case is the key component of use case modeling. Its purpose is to illustrate how a system enables an actor to meet a particular goal by showing all of the appropriate paths that they might take to get there, as well as those situations that could cause them to fail. It is primarily organizational in nature, providing order and structure so that the reader is able to easily identify and follow the different paths, or scenarios, as the actor progresses towards his goal.

UserValuedTransactions(110) tells us to capture the smallest set of goals that deliver all necessary services to the actors. A well-written use case provides a story describing how the system helps the primary actor completely reach a particular goal, a **CompleteSingleGoal(132)**. The most important factor to consider in this regard is granularity. A use case that addresses multiple goals becomes awkward, confusing to read, and hard to develop. A use case that addresses a partial goal will likely force the reader to wade through multiple use cases to follow a single thread of thought, making it easier for them to miss important actions.

There are several factors to consider when writing **CompleteSingleGoal(132)s**, and we have identified several patterns (Figure 1-2) to help. A well-written use case contains a main success scenario and an orderly, well-structured collection of fragments. We call this style **ScenarioPlusFragments(136)**. The scenario describes a singular and complete sequence of events that the actor follows as they attempt to achieve some goal, and results in either success or failure. The fragments are a list of **ExhaustiveAlternatives(140)** that describe any and all plausible alternative situations that the system can reasonably expect to encounter when attempting to meet this goal. This list is important because it enumerates the situations that the developers need to handle, and helps them to quantify effort involved.

Aim to keep these various pieces simple and free of clutter. If we have some information that doesn't really belong in the use case but we feel is still valuable, then we append it to the use case as an **Adornments(147)**.

A meaningful use case name is also important. Every use case should have an **IntentionRevealingName(144)** that gives the reader a hint of the use case's contents, and provides a word association, similar to a pattern name, that the audience can use to describe the use case's purpose in the course of subsequent conversations. The name is tightly coupled to the **CompleteSingleGoal(132)**. If the goal changes, you should consider changing the name as well, or if you want to change the name, you need to verify that the goal is correct. If you have trouble coming up with a name, then you should question whether your use case really represents a **CompleteSingleGoal(132)**.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Finally, a use case should be **PreciseAndReadable(152)**, written so that it is tailored to its audience. Every use case should accurately and completely describe some behavior, but at the same time it should not contain so much detail, nor be so verbose, that the audience cannot read it. The correct level of precision and readability present a moving target, so it is important that you know your audience when writing each one.

These guidelines are important because they describe a structure for organizing the details associated with multiple complex scenarios. Use cases without this structure can be very cumbersome, forcing the users to jump between seemingly unrelated sections of text, or to read volumes of tediously repetitive sections that seem to differ little from each other. **CompleteSingleGoal(132)** and **ScenarioPlusFragments(136)** are important factors in writing quality use cases. The organization of the individual scenarios is just as important, and is the topic of the next chapter.

Figure 1-2

Copyright © 2001 Adolph, Bramble All Rights Reserved

1 Scenarios and Steps

Telling a Good Story

There are few cities in the world where you can still get a terrific corned beef sandwich like you can in New York. Despite the great outdoors life style the West Coast offers, it still has not learned the fine art of the deli lunch. It almost makes what you have had to endure for the last twenty-four hours worthwhile. A Red Eye flight from the coast, spending the morning tearing poor Ahmed's use cases apart, and now you are about to do the same to poor Sitra, *Wings Over the World's* resident RAPIER expert. Sitra is much like Ahmed, probably less than three years out of school, very smart, very eager to try out new technology. It also appears that Ahmed has warned her about "the consultant".

Sitra: Ahmed said that you shredded his use case.

You: Shredding wasn't my intention, we shortened it and made it more useful.

Sitra: That's what Ahmed said also, so I shouldn't be afraid of this. He was really happy with the results and he says that you can really help us.

You: OK I was looking at your *Reserve Flight Segment* use case and it has a lot of the same issues that I discussed with Ahmed such as the inclusion of business rules and excessive precision. I see Ahmed talked you into using OCL as well. The intention to make the use cases more precise is good, but it is inappropriate. There are also a few other things, for the most part this use case is quite low level – most of what this is a description of the physical protocol between you and RAPIER.

Sitra: That's true, and most of my other use cases are like this one, they describe a transaction with RAPIER. But that's important because just about everything we do must go through RAPIER, and my subsystem provides a wrapper around it.

You: Fair enough, but consider this, *Reserve Flight Segment* is really a single step in the *Book Flight* use case.

Sitra: Yes, but it's a lot more complicated than that, the request can be rejected for a number of reasons, or even fail, and I thought the use cases were such a nice mechanism for capturing that.

You: I agree, but a big problem with this use case is that the steps are all at varying levels of abstraction. Some are at a business level for example, you have one step here– traveler selects flight – and then other steps are down to almost the program level – system formats RAPIER request header with request type set to booking. Really what we would like is the steps in a use case to all be at the same level, what I call **LeveledSteps(173)**. That way you

Copyright © 2001 Adolph, Bramble All Rights Reserved

do not have to do mental calisthenics jumping back and forth between the different levels of abstraction. The essence of *Reserve Flight Segment* is to send a message to RAPIER to make a reservation on a single flight. RAPIER is either able to make the reservation or it is unable to do so. The essence of this use case is about three or four steps.

Sitra: Yes that may be true but it is a lot more complicated than that, you have to first set up a transaction with RAPIER, then format a request, send the request, decode the response. There are literally dozens of return codes…

You: OK, but that is not necessarily what we want to capture in a use case. What you have here is 20 pages of prose that as a use case I can distill down into about three or four steps, plus a couple of alternatives. That is the essence of the *Reserve Flight Segment* use case.

Sitra: But what about everything else, that's the meat of the problem. I can't design a RAPIER interface subsystem from four sentences of prose.

You: Of course not, but the same thing I told Ahmed applies here, we can use the use case as an anchor for all the details, business rules, protocol specifications, technical constraints are supplements to the use case. I tend to call these **Adornments(147)**. Ninety percent of this use case is a protocol description. Maybe you want to draw a state transition diagram that would more succinctly describe the protocol. Also, what you have done, is tied this use case to the RAPIER implementation. Really the use case should be **TechnologyNeutral(177)**, after all we're trying to capture the system essence, not its implementation. I would publish the RAPIER interface as a separate external interface document and then reference it from the use case.

Sitra: So that way if we want to work with another airline reservation system, our model is not tied to one specific implementation?

You: That's right, the essence is still there, but if you have to change protocols then only that portion of the model is affected. Also this approach will reduce the number of alternatives your use case has. Right now you have some twenty alternatives, most of them based on the return code that you receive from RAPIER. But from what I can see there are really only three outcomes, either one, you were able to reserve the segment, two, you could not book the segment because space is unavailable, or three you could not book the segment because of a system failure. That's it. But you have twenty alternatives. Ahmed had the same problem because he had incorporated the business rules into his use case, and had an alternative for each of the different failures. We have a principle called **DetectableConditions(168)**, which states that if the action you take as a result of a set of different conditions is the same, then they are the same condition. This really helps reduce the complexity of the use case by eliminating redundant alternatives.

Copyright © 2001 Adolph, Bramble All Rights Reserved

The scenario and step patterns capture the signs of quality for writing good stories about systems. A sign of a quality story is that it stays focused, steadily advances towards its climax and does not distract the reader with arbitrary or optional plot twists. It does not bore its readers with long descriptive narratives that don't add any value to the plot.

Like any good story, it must always be clear in a use case step who is speaking and acting. **VisibleActorIntent(161)** recommends that each step clearly describe what is being accomplished and who is responsible for accomplishing it. Furthermore, it's not enough just to know who is speaking or acting, but each step in a use case scenario must make **ForwardProgress(164)** towards the primary actor's goal. Steps that don't advance towards the goal are meaningless, and don't belong in the use case, as they can only distract the readers.

Wishes, fantasy, and magic are all part of good fictional stories but they are certainly not part of good use cases. A system must operate in the real world and **DetectableConditions(168)** recommends that each scenario be something that can actually happen or be detected by your system, as opposed to listing all conceivable alternatives.

While technology is part of the real world, technology or implementation dependencies are not part of a good use case. Each step and scenario should be **TechnologyNeutral(177)**. Issues such as technology constraints or recommendations, user interface requirements, and business rules are supplemental as an **Adornments(147)** (see Chapter 5), and not part of the use case proper.

Finally, we should never ask a reader to cope with a mixture of complex and trivial steps written at different levels of abstraction. Good use case descriptions consist of scenarios with **LeveledSteps(173)**, where each step is similar in scope.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.1 DetectableConditions(168)

Washington, D.C. The U.S. Weather Bureau station at the National Airport. Wind direction and velocity are measured by these devices. The instruments are known as wind vanes and anemometers.

Driscoll, Fred, photographer.

CREATED/PUBLISHED
1943 Oct.

Copyright © 2001 Adolph, Bramble All Rights Reserved

You are structuring the multiple alternatives for a use case that can happen under differing condition as a **ScenarioPlusFragments(136)**.

Writers always wrestle with how many and which conditions to include.

Playing the stock market became very popular during the dot com bubble of the late 1990s. Every investor had the same dream – timing it so that they bought their stock at its absolute lowest price, and selling it at its peak, becoming enormously wealthy in the process. Many investors developed complicated methods for determining just where the market was in its cycle to help them achieve this goal, but all of these techniques shared a flaw. It is absolutely impossible to detect if the market has either peaked or bottomed out until well after the fact, when it is too late to act.

A system cannot handle events that it cannot detect. If the system cannot detect an event, then it cannot take the appropriate steps to deal with it. Any code that is written to handle an undetectable event is wasted because the system will never know to run it. It is possible that this condition may be a form of another detectable condition, but can lead to repetition. For example, if you are writing a use case for the computer operating a car, losing a key, having a bad starter, or no gas in the tank all lead to the same condition, the car will not start. While the resolution of each situation is different, the onboard computer doesn't care – it just knows that the car won't start.

The developers need to know what conditions to detect. Effective use cases provide the stakeholders and developers with a complete picture of a system's behavior, enumerating all of the conditions that they can reasonably expect the system to encounter, and a brief description of their resolution. Otherwise, the readers may not realize that a particular event can happen, or if they do, they are left to guess at its resolution, or take the time to track it down. We want to capture every reasonable possibility, so that the system developers know what the business rules are that they need to incorporate.

Fear of overlooking an important alternative encourages developers to specify irrelevant conditions that cannot be detected by the system) If the people writing the use cases miss a condition, either the developers or the end users of the system will likely bump into those conditions later. And researching detailed business rules late in development is expensive. Discovering a forgotten condition after the system has put into service is even more expensive. This, plus a natural tendency to feel that a thicker requirements document is

Copyright © 2001 Adolph, Bramble All Rights Reserved

better, all tend to cause beginning use case writers to enumerate every circumstance imaginable as alternatives the system should handle.

Many apparently different conditions can lead to writing unnecessary alternatives. Writing, reading, and reviewing a use case document is an expensive and tiring business. You don't want to write more than you have to, and it makes no sense to write scenarios that cannot happen. Also, if you write the same condition in several ways, the developers may implement what should be one handling procedure several different ways. Both situations can lead to code bloat.

Therefore:

Include only detectable conditions. Merge conditions that have the same net effect on the system.

Each scenario in a use case must begin with an action that the system is able to detect. Otherwise, the system will never be able to execute the scenario, and you are only wasting valuable time writing it and risk introducing features that unnecessarily complicate the system. Being unable to come up with a detectable condition to initiate a scenario may indicate that the scenario doesn't belong in the use case.

If you find a scenario that begins with a non-detectable condition ask, "What general condition is this scenario trying to handle?" If you can identify the condition, then use it to initiate the scenario, otherwise, eliminate the scenario because it can't happen. Be careful, however, because a non-detectable condition quite often is a special case of a more general detectable condition, and several apparently different conditions may mask the same basic set of actions. While a use case should capture all of the conditions that the system must be able to handle while resolving a specific goal **ExhaustiveAlternatives(XXX)**, it should capture only those, without duplication.

Examples

The ATM JAD Session

There are many things that can wrong when a customer accesses an automated banking machine. Lets eavesdrop on a group of ATM developers brainstorming alternatives in a JAD session:

> "What are the situations that can get in your way while getting money from an ATM?"
>
> "Once, when I bent down to tie my shoelaces, before I straightened up again, the machine had both pushed out the money and sucked it back in again!"

Copyright © 2001 Adolph, Bramble All Rights Reserved

"I was picking up my crying baby, and the same happened!"

"It ran out of ink on me, and I couldn't read the receipt!"

"I accidentally put in my library card!"

"There was some gum stuck in the card reader!"

"I got all the way there and forgot my card!"

"I got there and the power was out!"

"I was in the middle of using it and the power went out!"

The note taker at the board says, "Right. So I'll set up different scenarios in this use case, one for bending down and tying shoelaces, and one for attending to babies."

At that, someone quickly pipes up, "Wait a minute, the machine can't tell why they didn't take the money. All it can tell is that that they did not take the money within a certain time period."

This speaker is right. It makes no sense to write down a condition that the system can't detect. In the cases above, all the system can detect is that time went by and the money did not get taken.

"What's the difference between sliding a card in upside down and sliding one in that is badly scratched?"

"None. Put them together: Card unreadable."

While you are at it, remove any conditions that the system does not actually have to handle. Each unnecessary condition that gets written into the requirements document increases the cost of the final system. Those that really must be detected must be written down, of course. But sometimes people get so creative and detailed that they include things that really don't need to be included.

"Hey! What about when the ink runs out?"

"It's expensive to build a low-ink detector. How about we just have someone check the machine once a day to see that the receipt is still printing well enough to be read, and otherwise wait for someone to complain?"

"Hey, what about when the printer accidentally pushes the receipt crooked and it bends back up into the machine and the person can't get it?"

"They don't need it."

Copyright © 2001 Adolph, Bramble All Rights Reserved

Finishing off the ATM example above, suppose that the main success scenario is written as follows:

USE CASE: *Withdraw Cash* **(Fast path option)**

Level: User Goal

Main success scenario:

1. Customer inserts card.
2. ATM verifies card as readable and legitimate.
3. Customer enters PIN and selects "fast cash."
4. ATM sends account, PIN and cash withdrawal request to the main banking computer system (BCS), which returns an OK for the withdrawal.
5. ATM issues $40 in two $20 bills, the customer's card and a receipt, logs those actions, and send to the BCS confirmation that the cash was given.
6. Customer takes the cash, card and receipt.
7. ATM resets for the next customer.
8. The administrator logs off the system.

Extensions: (*only the conditions are presented here)*

*a. Sudden power failure detected during operation:

1a. Card jams in reader:

2a. Unreadable card:

3a. Time exceeded on PIN entry or function selection:

4a. BCS is unreachable (off-line or line broken):

4b. BCS refuses transaction:

4c. Insufficient cash in cash dispenser:

5a. ATM detects misfeed of number of bills (too few or too many):

5b. Cash jams in dispenser:

5c. Customer does not remove cash within time limit:

5d. Customer does not remove card within time limit:

Use Case 1-1 Withdraw Cash

Wings Over the World

In the previous chapter (Chapter 5) we saw the problem of Ahmed placing the business rules in the use case. His *Change Seat* use case was apparently thirty pages long because he had incorporated the business rules for checking for eligibility in the use case. He had then written alternatives for the failure of each business rule. But from the point of view of the Traveler, they were either able or unable to change their seat. They don't care why.

Copyright © 2001 Adolph, Bramble All Rights Reserved

In this chapter's installment of *Wings Over the World,* Sitra ran into the same problem. She had incorporated the technology details into her *Reserve Flight Segment* use case and for each of the different RAPIER failure codes she thought she would have to create an alternative in the use case. This approach would lead to the creation of a large number of different alternatives. However, it turns out, there are really only three outcomes, either RAPIER was able to reserve a seat, or it failed because either there were no seats available, or there was a system failure.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.2 LeveledSteps(173)

<image: long set of steps going up (Sacre Coeur?). even better is extra long steps> Image to come.

You are writing the steps of a use case as a **ScenarioPlusFragments(136)**.

Excessively large or excessively small use case steps obscure the goal and make the use case difficult to read and comprehend.

Anything can be described in smaller and smaller steps, until it loses its sense. Consider the following sample describing something as simple as stepping up to the sidewalk:

- She steps to the sidewalk,
- She lifts her foot from the street to the curb,
- She lifts her foot, arcs it through the air and lowers it to the pavement,
- She rocks her foot,
  Disengages her weight and frees the foot to be lifted
  Lifts it in a curve higher than need be,
  Shifts her weight slightly forward,
  Brings the foot heel first down onto the curb,
  Adjusts to support her weight as it moves above,
  Then stands on it

Excessively small steps make a use case long, hard to read, and obscures the "why". The "stepping onto the sidewalk" example demonstrates how excessively small steps make the reading long and tedious. Worse, it hides the intent in the details.

Excessively large steps may bury important behavior. The opposite occasionally happens, that the writer writes at a very high level of abstraction and makes large leaps in the narrative, omitting key actions the developers must know about.

Mixing levels of detail in a scenario is distracting. Occasionally, one must write adjacent steps at different levels of abstraction. Too much of this distracts the reader from what is supposed to be happening and makes correct interpretation of the instructions difficult,

Therefore:

Each scenario should have three to nine steps. Ideally, they are all at similar levels, a level of abstraction just below the use case goal.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Write each step with **VisibleActorIntent(161)** and make the goal of the step at a level of abstraction lower than the **CompleteSingleGoal(132)**. The steps themselves are generally kept at similar levels. If you need to explain how a step is performed, then create a smaller use case that explains the details underlying the step using the **EverUnfoldingStory(117)** to expand from a short description to a fuller description. The step's goal becomes the **CompleteSingleGoal(132)** for a lower level use case.

Examples

A Long and Tedious Use Case for an Online Store

All of the steps in the following online store example are legitimate, but the levels too low. The result will be a very long and tedious document.

USE CASE 15: *Purchase Goods*

Primary Actor: *User* – Customer wanting to make purchase.

Level: User Goal.

Main success scenario:

1. System asks user for first name.

2. User enters first name.

3. System asks for middle initial.

4. User enters middle initial.

5. System asks for last name.

6. User enters last name.

7. System asks for first line of street address.

... (etc.... we'll spare you the details)

Use Case 1-2 Use Case Horror: A *Purchase Goods* **use case with unleveled steps.**

The usual motivation on the part of the use case writer for creating long and tedious scenarios like this example, is to specify all the fields of personal information the system should collect from the user – surname, first name, middle initial, etcetera. Use cases are not interface specification nor are they data dictionaries. Many use case writers unfortunately tend to employ use cases as surrogate interface specifications or data dictionaries and this leads to long tedious use cases. Such information should be expressed as a **Adornments(147)** to the use case. We can level this long and tedious scenario by simply consolidating the data capture steps:

USE CASE 15: *Purchase Goods*

Copyright © 2001 Adolph, Bramble All Rights Reserved

Primary Actor: *User* – Customer wanting to make purchase.
Level: User Goal.
Main success scenario:

1. User specifies their personal information.
2. (some really interesting or useful step rather than enumerating every piece of personal information the user is suppose to enter)

Use Case 1-3 Revised *Purchase Goods* **use case with leveled steps.**

If the reader is interested in knowing what data the system collects as personal information, then they can look up the definition for personal information in the data dictionary.

In general, we find that good writers keep a single scenario to less than 10 steps. It seems that this is the number that readers can read and understand easily.

A Use Case with Excessively Large Steps for an Online Store

What if the steps are excessively large? Consider these steps:

USE CASE 15: *Purchase Goods*
Primary Actor: *Visitor* – Customer wanting to make purchase.
Level: User Goal.
Main success scenario:

1. Visitor enters all the customer, product and purchase information.
2. System presents visitor with final sums, charges credit card and delivers packing list to the shipping department.

Use Case 1-4 Use Case Horror: A *Purchase Goods* **use case with excessively large steps.**

This example meets our less that ten steps rule, but it really doesn't tell us anymore about the system than we could deduce from the name.

We can expand these excessively large or "overloaded" steps by asking, "How is that accomplished?" Breaking the two steps into smaller steps, we now notice that the system was supposed to perform some intermediate actions:

USE CASE 15: *Purchase Goods*
Primary Actor: *Visitor* – Customer wanting to make purchase.
Level: User Goal.
Main success scenario:

Copyright © 2001 Adolph, Bramble All Rights Reserved

1. Visitor enters customer information (name, address, etc.).
2. System retrieves customer's profile information, and presents product search and selection mechanisms.
3. Visitor selects products until satisfied. After each selection, system adds the selected product to the customer's "shopping cart" and presents the ongoing running total of products selected.
4. Visitor selects to purchase the selected items.
5. System presents contents of shopping cart, requests customer's payment information.
6. Customer enters manner of payment and other payment details.
7. System presents visitor with final sums, charges credit card and delivers packing list to the shipping department.

Use Case 1-5 Revised *Purchase Goods* **use case.**

A Use Case that Mixes Large and Small Steps for an Online Store

What if we mix large and small steps? Consider this scenario:

USE CASE 15: *Purchase Goods*

Primary Actor: *User* – Customer

Level: User Goal.

Main success scenario:

1. The system displays the Login screen.

2. *User* enters a username and password.

3. The system verifies the information.

4. The system sets access permissions.

5. The system displays the Main screen.

6. *User* does one or more of the following: *Place Order*, *Return Product*, *Cancel Order*, *Send Catalog*, *Register Complaint*.

7. The system terminates the session and reset the screen.

Use Case 1-6 Use Case Horror: Mixing large and small steps in a use case.

While this scenario has less than 10 steps, there is a vast gulf between the levels of the first steps and the sixth step. It would be better if the first five steps were merged into two or three.

Copyright © 2001 Adolph, Bramble All Rights Reserved

In this example we want to merge the first four or five steps. We ask, "Why were the user and system doing those things? What were they trying to accomplish?" The answer is that the user wanted to log in. With this answer, we can revise the steps as:

USE CASE 15: *Purchase Goods*

Primary Actor: *User* – Customer

Level: User Goal.

Main success scenario:

1. The user logs in to the system.
2. The system presents the available functions to the user.
3. The user does one or more of the following: *Place Order*, *Return Product*, *Cancel Order*, *Send Catalog*, *Register Complaint*.
4. The system terminates the session and resets the screen.

Use Case 1-7 Revised use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.3 VisibleActorIntent(161)

Lancaster, Pennsylvania. Company dance given in Moose Hall by employees of the Hamilton Watch Company so that new employees might get acquainted. Refreshments: coca cola and soda pop during an intermission.

Collins, Marjory, 1912-1985, photographer.

CREATED/PUBLISHED
1942 Nov.

You are writing the steps of a scenario as **LeveledSteps(173)**.

Both readers and developers get confused about a system's behavior if it is not clear which actor has responsibility for performing a step, and is trying to accomplish in that step.

Some time ago I was visiting friends who have young children. Their seven-year-old son was trying to tell me about a movie that he had just seen.

> "It was so cool, these guys search for this monster that is sinking ships, and well they find it, only it's a submarine. They get captured and travel underwater in the submarine. There is this guy and he's really fascinated by the submarine and the submarine captain is this ex prisoner guy and he's got an attitude and he's always mixing it up with this other guy. And then there is this giant squid that nearly eats him, but the other guy harpoons the squid and saves him."

Copyright © 2001 Adolph, Bramble All Rights Reserved

If I had never seen Disney's version of Jules Verne's 20,000 Leagues Under the Sea, I would have had a lot of trouble following this synopsis. I would have been confused as to which "guy" was fascinated by the submarine, who had the attitude, and who saved whom when the Nautilus was attacked by the giant squid.

Writing quality use cases is hard work and it takes a lot of mental energy to create good prose. Often, people have trouble writing **PreciseAndReadable(152)** use cases. Non-programmers tend to write ambiguous steps and miss the meaningful details the developers need to know. Programmers, on the other hand, tend to generously incorporate both design and technology details in their use cases, making them too hard for the non-programmers to understand. Often, programmers are told to write the use cases that they themselves will implement, leading them to write entirely from the system's perspective, leaving out what the other actors will do.

The developers need to know clearly what they are to implement, at what moments the system should wait for input, and when the system should take the initiative. Otherwise, they are left to make their own assumptions about these issues, or spend time tracking down details they should already have. The cost of miscommunication is high, so we aim to write use cases that are general enough for all of the stakeholders to follow, yet precise enough for the developers to use when building the system (**PreciseAndReadable(152)**).

Therefore:

Write each step to clearly show which actor is performing the action, and what that actor will accomplish.

Minimize your writing effort by using a simple format for each step. We recommend a clear sentence with straightforward grammar: actor, verb, direct object, and prepositional phrase. "User enters name and address," or "System notifies user of results." When the actor is visible in each step then it is clear to both the reviewers and the developers which actor must perform the step. This reduces ambiguity and helps protect the project from the problems that arise from miscommunication.

Use active verbs in the present tense to describe how an actor advances towards their goal (**ForwardProgress(164)**). Each step, except those at the lowest detail level, should address a lower level **CompleteSingleGoal(132)**, so that if necessary, you can create an **EverUnfoldingStory(117)** by converting the step's verb phrase into a lower-level use case.

Examples

Copyright © 2001 Adolph, Bramble All Rights Reserved

The Actorless ATM

Having programmers write their own use cases often results in system-oriented scenarios such as:

USE CASE 44: *Withdraw Cash*

Primary Actor: *User* – Account Holder

Level: User Goal.

Main success scenario:

1. Read the ATM card.
2. Validate the card information.
3. Collect the transaction information.
4. Validate the transaction details.
5. Issue the cash and update the account.
6. Reset the system.

Use Case 1-8 Use Case Horror: ATM Description without a VisibleActorIntent

While this scenario is clear to the writer, it leaves questions about what is the customer's part in the transaction. The following, better version of this use case clearly shows what is the actor's intent:

USE CASE 44: *Withdraw Cash*

Primary Actor: *User* – Account Holder

Level: User Goal.

Main success scenario:

1. *User* inserts their ATM card. (versus Read the ATM card.)
2. System reads and validates the card information (versus Validate the card information.)
3. *User* selects transaction and enters transaction details (Versus Collect the transaction information.)
4. System validates transaction details (versus Validate the transaction details.)
5. *User* collects cash and withdraws card. (after all what happened to the *User*?)
6. System and updates the account and resets the system (versus Issue the cash, update the account. Reset the system.)

Use Case 1-9 Improving ATM Description with VisibleActorIntent.

This style clearly shows who is responsible for what action.

In a related style, some people like to write in the passive voice, again leaving out which actor is taking the initiative:

Copyright © 2001 Adolph, Bramble All Rights Reserved

USE CASE 44: *Withdraw Cash*

Primary Actor: *User* – Account Holder

Level: User Goal.

Main success scenario:

1. The card gets inserted.
2. The card information gets validated.
3. The transaction information gets collected and validated.
4. The cash is issued, card returned, cash removed, account debited, screen reset.

Use Case 1-10 Use Case Horror: Description Written in Passive Voice.

Unfortunately, different actors initiate the actions. While the steps are leveled and show forward progress, it is not clear to anyone, other than the writer, who is responsible for doing what. They can only guess.

A final mistaken style of writing is to name many, even technology-specific, movements the user of the system will take:

USE CASE 42: *Access ATM*

Primary Actor: *User* – Bank Customer

Level: User Goal.

Main success scenario:

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

Use Case 1-11 Use Case Horror: Description with Technology Specific Steps.

Here, we can improve the writing by combining the steps, making it **TechnologyNeutral(177)**, and capturing the intent of the actor instead of the technology-specific movements:

USE CASE 42: *Access ATM*

Primary Actor: *User* – Account Holder

Level: User Goal.

Main success scenario:

Copyright © 2001 Adolph, Bramble All Rights Reserved

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

Use Case 1-12 Improved TechnologyNeutral ATM Use Case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.4 ForwardProgress(164)

Detroit, Michigan. Women civilian defense workers marching in full regalia in the Labor Day parade.

Siegel, Arthur S., photographer.

CREATED/PUBLISHED
1942 Sept.

You are writing the steps of a scenario as **LeveledSteps(173)**.

Writers have to decide how much behavior to put into any one step. They easily write too much detail, making the use case long and tiring to read.

Creating a diversion is a simple plot technique in stories to add conflict to what would otherwise be a very simple and straightforward story. Two or more rivals are pursuing a common goal and one of them attempts to slow or stop the other by creating a distraction.

Copyright © 2001 Adolph, Bramble All Rights Reserved

The Star Trek television series frequently used this technique to add conflict to what would otherwise be a simple story. A Star Trek episode would be very uninteresting if all that happened when the Enterprise raced to answer a distress call from a damaged ship is that it rescued the crew and then headed off to the next star base. Either chief engineer Scotty shout his famous “she cannae much mor o’ this” or an opponent such as the Romulans would use their favorite diversion ruse of sending fake distress call that would force the Enterprise to break off and race to the rescue. The story now focused on the efforts of the Enterprise crew to overcome the diversion. The original goal of the rescue mission becomes almost an after thought.

While diversions help create good stories, a good use case should tell a story in a straightforward and simple manner.

Clear, succinct steps lower the mental cost of understanding and evaluating a use case. Each step in a scenario needs to be germane, succinct and clear, to make it as easy as possible for the reader to follow. If you put too much information into a single step, you increase the chance that the reader will lose track of what is happening inside that step, perhaps missing a key business or implementation issue. If you put too little information within a step, you will need more steps to tell the story, making the use case longer and more tedious to read.

A desire for completeness and detail may lead to the inclusion of steps that are tangential to the goal of the use case. People like to add these details because they feel that this extra information improves our understanding of unfamiliar issues. However, tangential steps are more likely to cause the reader to lose focus. A good use case is much like a good movie: well paced and adhering to the plot. Extraneous scenes and tangential steps only serve to clutter the description. Worse, someone may try to implement the tangents, wasting time and benefiting no one. Even when the step adds value to the scenario, unnecessary or overly complex details can distract the reader from the main path.

Therefore:

Eliminate steps that do not advance the actor, simplify passages that distract the reader from this progress

Writing long, complex, or tangential steps unnecessarily diverts the reader, who should instead see a clear progression within a scenario. Provide enough detail in each step to enlighten the reader of its purpose, but not so much so as to describe the step in excruciating

Copyright © 2001 Adolph, Bramble All Rights Reserved

detail. Ensure that the steps stay relevant to the reader and that each step is a **LeveledStep(173)**, containing a "good" or "bite-sized" amount of information.

Many people have trouble reading technical documents because they are complex and contain a lot of unrelated details. Anything you can do to simplify your use cases (short of removing necessary information) will make them easier to use. If there are tangential steps that represent important alternatives then incorporate them as an alternative to the scenario – **ScenarioPlusFragments(136)**. If there are details you believe are important to understanding the story, or are simply too important to lose, and then attach them to the use case as **Adornments(147)**.

Examples

Wings Over the World – A Diversionary Scenario

The following snippet of *Request Upgrade* use case shows a use case with two blatant diversions in the main success scenario:

USE CASE 28: *Request Upgrade*

Primary Actor: *Customer* – Ticketed Passenger

Level: User Goal.

Main success scenario:

1. *Customer* enters their account code for their flight and requests a seat upgrade.
2. If the *Customer* is a frequent flier then the system displays their current mileage and recent flight activity.
3. The system verifies there are seats available for upgrading
4. The system upgrades the *Customer's* seat assignment and the appropriate upgrade certificates are removed from the customer account.
5. The *Customer* may also buy more upgrade certificates while requesting an upgrade.The system issues upgrade receipt to the *Customer*.

Use Case 1-13 Use Case Horror: *Request Upgrade* **with Steps Diverting Forward Progress.**

In this example, steps two and five are alternatives that divert attention away from the real story of how a traveler requests a seat upgrade. Step two is really an alternative to step one, handling those instances where the customer is frequent flier. Step five is an

Copyright © 2001 Adolph, Bramble All Rights Reserved

alternative available to step four when the traveler does not have enough upgrade certificates.

You can still show these diversions, but as alternatives. A better way to write this scenario is

USE CASE 28: *Request Upgrade*

Primary Actor: *Traveler* – Ticketed Passenger

Level: User Goal.

Main success scenario:

1. *Traveler* enters their account code for their flight and requests a seat upgrade.
2. The system verifies there are seats available for upgrading.
3. The system upgrades the *Traveler* seat assignment and the appropriate upgrade certificates are removed from the *Traveler's* account. The system issues upgrade receipt to the *Traveler*.

Alternatives:

1a: *Traveler is a frequent flyer*

1a.1The system displays their current mileage and recent flight activity.

4a: *Traveler does not have enough upgrade certificates*

4a.1 Traveler purchases additional upgrade certificates.

Use Case 1-14 *Request Upgrade* **makes Forward Progress.**

Insurance Claim – Not Enough Forward Progress

Here is a different example in which the diversion is not as blatant as the in the previous example but is still equally distracting. Its steps are not "leveled", and so while there is no explicit diversion as shown in the previous example, there is not *enough* forward progress being made:

USE CASE 4: *CLAIM INSURANCE*

Primary Actor: **Claimant** – policyholder reporting accident claim

Level: Summary.

Main success scenario:

Copyright © 2001 Adolph, Bramble All Rights Reserved

1. Claimant obtains claim form.
2. Claimant enters name and address on claim form.
3. Claimant enters policy information on claim form.
4. Claimant enters accident information on claim form.
5. Claimant photocopies ticket and attaches it to claim form.
6. Claimant photocopies claim form and puts it in file cabinet.
7. Insurance company checks whether claimant owns a valid policy.
8. Insurance company determines that claimant does own a valid policy.
9. Insurance company assigns agent to examine case.
10. Agent verifies all details are within policy guidelines.
11. Insurance company pays claimant

Use Case 1-15 Use Case Horror: *Claim Insurance* **not making sufficient Forward Progress**

The tip-off here is that several steps in a row refer to the same actor performing one action after another. When you see this, consider whether there is enough forward progress being made in each step. Ask, as in getting **LeveledSteps(173)**, "What is the actor trying to accomplish?' and, "Can I find a way to express all this information in a single step?"

In the above example, steps one through five are a necessary part of the process, but waste a lot of the reader's energy by making very little overall progress. It would be more succinct, easier to read, and just as informative to write:

1. Claimant submits claim with substantiating data.

Step six does not bring any forward progress to the use case - indeed, it is superfluous. Users may want to save copies of their claim forms (that is their prerogative). They may also want to post a their insurance agents' pictures on the wall for dart practice, or use them to line a birdcage. Again, that is their prerogative. These actions have nothing to do with the use cases; the insurance company will process their claim regardless. This sentence can be deleted.

Step seven says, "checks whether." This commonly used phrase shows only half of the needed action. The check will either succeed or fail. Sayings “checks whether” implies that the writer must write another sentence saying what the outcome will be. However, a use case step shows *accomplishment*, in other words, that the check succeeds. A failure on the

Copyright © 2001 Adolph, Bramble All Rights Reserved

verification will show up in an extension condition. Therefore, steps seven and eight should be merged, becoming the shorter, clearer:

2. Insurance company verifies claimant owns a valid policy.

At this point, we can revise the above, 11-step scenario into a five-step scenario, in which each step carries its own weight, making distinct **ForwardProgress(164)** towards a specific goal.

USE CASE 4: *CLAIM INSURANCE*

Primary Actor: **Claimant** – policyholder reporting accident claim

Level: Summary.

Main success scenario:

1. Claimant submits claim with substantiating data.
2. Insurance company verifies claimant owns a valid policy.
3. Insurance company assigns agent to examine case.
4. Agent verifies all details are within policy guidelines.
5. Insurance company pays claimant

Use Case 1-16 *Claim Insurance* **Revised to Make Forward Progress.**

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.5 TechnologyNeutral(177)

<image picture of a couple at a car show, guy is excited, woman couldn't care less – or maybe break the stereotype and do it the other way around> No appropriate image yet.

You are writing the steps of a scenario as **LeveledSteps(173)**.

Including technology constraints and implementation details in a use case description increases the complexity and obscure the goal of the use case.

One of the less popular tasks that software developers do is called "porting" a system, which involves taking a software system that exists on one hardware platform, and changing it so that it can run on another. This task is often tedious because it involves redoing the same system several times. One reason that the Java programming language is so popular is that it provides a "Compile-once, Run anywhere"[1] feature that allows programmers to write a program once, then run it on any computer that supports Java. It accomplishes this feat by providing a standardized interface that hides the machine dependent code from the users, so that they can call it without concerning themselves with platform specific issues.

Good use cases should likewise insulate the readers from system specific details, leaving them free to focus on the system's essential behavior.

Many people like to think in concrete terms. While abstraction is key to good design, such thinking is difficult, and requires a thorough understanding of the problem. People cannot abstract things they don't understand, and instead they seek concrete details to improve their knowledge. Only when they feel comfortable with a topic will they begin to think abstractly about it. It is not surprising that people like to add low-level details to use cases, because they are often learning about the system as they write, and these details make them feel comfortable by giving them a better understanding of what they are doing.

Technology is volatile, and including details about specific technologies will cause rework. Technology is constantly changing. Vendors are constantly releasing new versions of their products on a regular basis, as those of us who have bought a top-of-the-line computer that became obsolete within a matter of weeks will attest. In theory, these changes should not present a problem because they are hidden within a product, but unfortunately, many of them subtly and visibly affect the product's behavior. Consequently, uses case describing these

[1] Or compile once, debug everywhere as some Java detractors say.

Copyright © 2001 Adolph, Bramble All Rights Reserved

kinds of details become obsolete whenever the product changes, forcing an organization to either perpetually update their use cases, or use obsolete ones.

Technological details impose improper constraints on future activities. Implementation details can bias the readers to a particular design by masquerading as requirements. Worse, they assume an undue aura of importance because they often require explanatory text. For example, mentioning a file in a use case might force the writers to include extra error and alternative courses for handling the file, turning the use case into one more about file handling than about its primary subject.

Adding technological details increases the cost of both reading and writing use cases. Use cases should capture essential system behavior without delving into its low-level structure. When they leave out technical details, they offer a powerful tool that speaks to a wide range of audiences, technical and non-technical alike. If the readers can easily grasp the system's purpose, then they are free to use their own imagination to flesh out those details, often coming up with a better, more creative solution than the use case implies.

Sometimes technology is a requirement. Product development doesn't occur in a vacuum. Often, a new system must be able to interact with existing systems using specific protocols or access common data using a specific database engine. Unless the product is the first of its kind, it may also have to meet industry or government standards.

Therefore:

Write each use case in a technology neutral manner.

Focus on essential system behavior such as what actions the system performs, or why someone might want to do them. Describe specific actions in neutral ways, avoiding details that might guide the readers towards specific implementations. Be especially cautious about specific terminology, because common words like database, file, or GUI can subtly guide developers towards specific design decisions. Use phrases such as "the user requests information from the system" instead of "the user selects the OPEN button from the FILE pull-down menu". The former allows the developers the latitude to decide how to best store this information, while the latter leads them towards using files and a specific GUI. They may decide a file is best, but they can make that decision at a more appropriate time, when they have a better understanding of the system.

If there are implementation details and technology constraints that are relevant to the use case or help increase comprehension then include these as **Adornments(147)** to the use case.

Examples

Copyright © 2001 Adolph, Bramble All Rights Reserved

Developers constantly violate this guideline by tying their use cases to a particular Graphical User Interface (GUI). Yet this practice constrains the developers, and leads to a situation where the GUI drives use case development. But GUI's only describe a system's look and feel, not its behavior. Tying GUIs to use cases is putting the cart before the horse, because GUI developers should be using the use cases to help design the system interface, not the other way around.

Consider the following use case for submitting a claim to an insurance office.

USE CASE 4: *FILE ACCIDENT CLAIM*

Primary Actor: Claimant – policyholder reporting accident claim

Level: User Goal.

Main success scenario:

1. **Claimant** accesses Accident Reporting System
2. **Claimant** identifies self to system using name and policy number
3. **System** verifies claimant owns a valid policy against policy database.
4. **Claimant** submits claim with substantiating data including name of both parties, investigating officer, and citation number, if applicable
5. **System** logs claim file and policy database, and sends email acknowledgement to the claimant

Use Case 1-17 Use Case Horror: Technology dependent *File Accident Claim* **use case.**

This example contains at least three flaws:

1. It presupposes the system will store policy information in a database.
2. It presupposes the system will store claim information in a data file.
3. It presupposes the system will send email to acknowledge a claim.

In short, this scenario makes some design assumptions. While these may or may not be requirements, they don't belong here. If they are not requirements, then they may prevent the developers from examining and implementing other, more efficient mechanisms for processing the claim.

For that reason, it is better to write the scenario as:

USE CASE 4: *FILE ACCIDENT CLAIM*

Primary Actor: **Claimant** – policyholder reporting accident claim

Copyright © 2001 Adolph, Bramble All Rights Reserved

Level: User Goal

Main success scenario:

1. **Claimant** accesses Accident Reporting System
2. **Claimant** identifies self to system using name and policy number
3. **System** verifies claimant owns a valid policy.
4. **Claimant** submits claim with substantiating data including name of both parties, investigating officer, and citation number, if applicable
5. **System** logs claim and acknowledges its receipt to the claimant
6. **Requestor**: mark request delivered.

Use Case 1-18 A TechnologyNeutral File Accident Claim use case.

This scenario describes the system's behavior without supplying any design information. It lists necessary information such as policy number and name, but it does not specify its format. It also informs the reader that much of this data is persistent, but it doesn't tell them how it is stored. This format allows the developers to make these decisions in a manner that is best for the system.

It is incredibly easy to introduce implementation details into a scenario, and we often do so without realizing it. Just adding two seemingly innocent words completely changes the nature of this scenario.

USE CASE 4: *FILE ACCIDENT CLAIM*

Primary Actor: **Claimant** – policyholder reporting accident claim

Level: User Goal

Main success scenario:

1. **Claimant** accesses Accident Reporting System via Internet
2. **Claimant** identifies self to system using name and policy number
3. **System** verifies claimant owns a valid policy.
4. **Claimant** submits claim with substantiating data including name of both parties, investigating officer, and citation number, if applicable
5. **System** logs claim and acknowledges its receipt to the claimant
6. **Requestor**: mark request delivered.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Use Case 1-19 Use Case Horror: Technology creeps into a use case.

While the phrase "via Internet." seems innocuous enough, it subtly focuses the reader's attention on a web-based solution. But who said that the Internet is the best solution for this system? What about customers without Internet access, either because they don't have access to a computer, or because they suffer an accident that requires lengthy hospitalization? It is quite possible that such clients might not be able to report the accident until they return home. This case is of particular interest to insurance companies, however, because most of them wish to know immediately when their clients are hospitalized in order to control costs. These two simple words could lead the developers to devise a solution that is in conflict with their client's most important goal of reducing costs.

This pattern was influenced by Rick Ratliff Sans GUI pattern from the OOPSLA 98 Use case patterns workshop.

Copyright © 2001 Adolph, Bramble All Rights Reserved

1.6 Tradeoffs and Collaborations

Like the old proverb, “a chain is only as strong as its weakest link”, a use case is only as clear as its most ambiguous step. Organizing steps in scenarios and using a simple standard form for each step lowers the effort required in both reading and writing the step, and the overall use case. The patterns in this chapter (see Figure 1-1) form the baseline for a simple standard form.

Figure 1-1

Relationship between the Scenario and Step Patterns

The two scenario patterns help us write **ScenarioPlusFragments(136)**. **DetectableConditions(168)** tells us that we should only describe those kinds of situations that can really occur as the actors interact with the system. Otherwise, we waste our time writing scenarios that no one needs, or are redundant. Worse, someone may try to implement the unnecessary scenarios, wasting valuable development time. **LeveledSteps(173)** recommends that we balance our use cases so that no one scenario, especially an alternative one, can dominate. Instead, we want to create balanced scenarios, writing them in parallel styles so that the reader can easily follow them, without attaching undue importance to an overly complex one.

But leveling steps involves more than just writing steps similarly, and we have several patterns to guide us. Each step in a scenario must meet three conditions. First, it must add value by clearly making some **ForwardProgress(164)** towards fulfilling the scenario’s ultimate goal. It must demonstrate purpose by making the **VisibleActorIntent(161)**, showing

Copyright © 2001 Adolph, Bramble All Rights Reserved

who is trying to accomplish the action, and what it is that they are trying to accomplish. Lastly, it must be **Technology Neutral**, neither bombarding the user with excessive details, nor implying a design. Steps that fail to meet all three of these conditions only serve to confuse the reader and add ambiguity to the scenario, and should be rewritten or removed from the scenario.

LeveledSteps(173) is similar to **EverUnfoldingStory(117)**, bringing unity and parallelism to scenarios, just as **EverUnfoldingStory(117)** does for a set of use cases. However, they are two different patterns, having different contexts and purposes. **EverUnfoldingStory(117)** balances goal levels for sets of use cases, while **LeveledSteps(173)** balances the content of scenarios. In fact, using **LeveledSteps(173)** makes it easier to write an **EverUnfoldingStory(117)** because it is a fairly simple matter to expand a series of leveled step into a set of lower level use cases.

The three chapters – Use Case Sets, Use Cases, and Scenarios and Steps, describe the structure of well-written use cases. Sometimes, however, our well-written use cases can violate these guidelines for reasons beyond our control. System boundaries can change and goals can shift, causing some use cases to become redundant, obsolete, or address the wrong scope. The following chapters describe what you can do when these situations occur to restore order to your use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

7 Use Case Relationships

But I thought you said these were just stories.

Yitka Dvorak is *Wings Over the World* Director of Marketing and has just dumped a set of use cases on the table in front of you, "I thought you said these things were just simple stories. How am I suppose to figure them out if I have to keep jumping from one story to another?"

You glance at the diagram representing the use cases and immediately see the source of the confusion. It more resembles the plans for an oil refinery's plumbing than functional requirements. The use cases are loaded with *includes* and *extends* relationships, and even a couple of generalizations. Some of them contain a dozen *includes* relationships, and *includes* on *includes*. No wonder Yitka is so upset. You notice the author is Ahmed, and you assure her that you'll take of this.

You go over to Ahmed's cubicle and show him the use cases. "But that's how they show it in the book" he says pointing to a text book on his desk, "and besides these are all standard UML relationships". You explain how many authors tend to overemphasize *includes* and *extends*. You talk about the principle of one stop shopping, which aims to minimize the need for your readers to jump all over the place to read one use case.

"The most important characteristic of a good use case", you explain to Ahmed, " is that the stake holders understand it. Remember my success criteria for a good use case is that the stake holder should be able to come up to us and say, 'No this is not how it works, you guys are completely wrong'. When they can do that, it means they read the use case and understood it. If they come up to us and say 'yeah I guess so, you guys are the expert, or they refuse to read the use cases because they're hard to follow, then we're dead." *Includes* and *extends* should only be used when they simplify things, not make things more complicated and confusing.

It's easy to start a riot at a use case conference. Simply walk up to an open microphone and ask, "What is the difference between *includes* and *extends?"* and then run for cover! How to effectively apply the *includes* and *extends* relationships in use cases is a question that has perplexed both beginning and experienced use case writers.

Includes and *extends* were originally intended to simplify the use case model by collecting common threads in the use case model. Unfortunately, these simplifying relationships seem to have the opposite effect, making the model more complex and difficult to understand. Many factors contribute to this problem. First, the definitions of *includes* and *extends* are ambiguous, leading to a wide variety of interpretations as to what they really mean. Second, software professionals tend to favor formality in the use case model over usability, creating

Copyright © 2001 Adolph, Bramble All Rights Reserved

models that follow all the rules, but are hard to read. Finally, instructors tend to over emphasize these techniques when teaching people to write use cases.

The UML revision task force exacerbated this ambiguity in 1998 when they attempted to bring peace to the warring factions by introducing a new definition of the use case structuring relationships. They replaced the original *uses* relationship with the new <<include>> relationship, made <<extends>> a UML dependency relationship and introduced the generalization relationship. While these changes helped to remove ambiguity in the formal grammar that supports the definition of use cases in UML, this revision has led to even more confusion and ambiguity in the practical applications of the use case structuring relationships.

Consider this: while a programmer may conceptually understand these relationships, try getting the product manager, the lawyer or the accountant who is a stakeholder in the system to understand them. These are intelligent, capable people, who have very little time to learn formal rules to verify their systems. The cardinal rule of writing is "*know your audience"*. In our opinion, the best use cases are semi formal, with just enough structure to be reasonably precise and unambiguous in the way they communicate important concepts (see **PreciseAndReadable(152)**), yet simple enough so that a stakeholder who is familiar with the problem domain can read and understand it. Rigid formality does not help this cause.

We have observed that good use case models tend to use the *includes* and *extends* relationships judiciously. Models that excessively use the *extends* relationship tend to use extensions to describe implementation details, frequently describing alternatives on a user interface. We have seen models where a use case may have dozens of extension use cases, with extensions on extensions. Many people would quickly throw such work back at the writer or worse, simply ignore it.

Furthermore, people frequently tend to use these relationships to create director style use case, in which extension use cases specify the details of each step. The consequence of this technique is that usually the model usually represents the developer's point of view, causing the readers to lose sight of the important actor goal in the details.

We have identified three patterns that help use case writers understand these relationships.

CommonSubBehavior(186) states that you should consider creating an <<includes>> relationship between use cases only when two or more use cases share a common set of actions. The purpose is to consolidate common behavior and simplify your model by reducing redundancy.

InterruptsAsExtensions(191) recommends you consider an extension use case only when an alternative interrupts a number of steps in a scenario. This technique helps simplify the

Copyright © 2001 Adolph, Bramble All Rights Reserved

model by consolidating the related interrupting actions in one use case, rather than dispersing them across several alternative courses of action.

PromoteAlternative(196) suggests that you may also want to use extensions when you have some important alternatives that you may want to emphasize. In this case, you can "graduate" these alternatives into an extension use case so that they stand out. You should use this technique sparingly, so that you don't conflict with the guidelines described in **InterruptsAsExtensions(191).**

We haven't yet to seen many good models utilizing the UML generalization relationship to create a pattern for guiding its use. However, our colleague Dan Rawthorne has provided us with a pattern that suggests an appropriate use for generalization.

Copyright © 2001 Adolph, Bramble All Rights Reserved

7.1 CommonSubBehavior(186)

Women in war. Supercharger plant workers. These women war workers at a Midwest supercharger plant are buffing small delicate airplane engine parts. Note protective head coverings and comfortable slack suits. Allis Chalmers Manufacture Company.

Rosener, Ann, photographer.

CREATED/PUBLISHED
1942 Oct

You are writing use case descriptions as **ScenarioPlusFragments(136)**.

Writing the same steps for different use cases is wasteful and makes it more difficult to see common sub-processes within a use case model.

Maurice Wilkes once gave a lecture on his role in the development of EDSAC, the first stored program computer. Unlike ENIAC, which was programmed from a hardware plug board, EDSAC actually stored its instructions in the same memory that was used to store data, the basis of all modern day computers. Programs were fed into EDSAC using paper tape.

Copyright © 2001 Adolph, Bramble All Rights Reserved

The EDSAC team quickly noticed that while all of their programs solved different problems they frequently employed the same sets of instructions. Punching paper tape for EDSAC was a labor intensive and tedious activity, and people found that re-punching the same instructions over and over again was tedious, error-prone, and a waste of time.

Someone finally took the initiative and began cutting out the common sets of instructions from the paper tapes and storing them in little 35-millimeter film canisters. Then when someone wanted to write a program that used that sequence of instruction, they could retrieve the paper tape, attach it with scotch tape to their program, and then feed the paper tape with their program, and the sub-program into EDSAC. It made programming EDSAC much easier and less error prone.

Rewriting common steps is redundant and increases the risk of inconsistencies or inaccuracies in the model. Different processes may have common steps. . For example, buying or selling stock are two distinct activities, yet each requires a valuation of the being stock traded. Similarly, paying property taxes on a house, selling a house, or buying a house are all different activities that require an assessment of the value of the house. The steps for stock valuation can be re-written for both purchase and sale of stock, but redundancy is never a good thing to have in a model. First, it is simply a waste of time writing and reading the same steps over and over again for each process.

Second, redundancy can lead to inconsistencies in the model, if the commonly used steps must be changed. A frequent source of errors in requirements model results from the need to change a repeated behavior and forgetting to change every instance of that common behavior. For example, let's say the rules for assessing the value of a property are changed. We change the steps in the purchase property use case, but we forget to change the steps in the pay tax use case. We run the risk of building a system that is inconsistent in the way that it assesses property values.

Partitioning individual use cases tends to scatter important behavior and make them harder to understand. "One stop shopping" is a useful requirements writing principle. Simply, a person reading the requirements should be able to get all of the information they need from one document or from one section of a document. They should not have to jump between different sections to understand the requirement.

A classic example of violating this one stop shopping principle is the famous "Joy of Cooking" cookbook mothers traditionally give to their sons when they leave home. The "Joy of Cooking" is probably one of the most complete self-contained cookbooks written. It is also one of the most annoying cook books to follow because for any recipe it forces you to jump

Copyright © 2001 Adolph, Bramble All Rights Reserved

back and forth between different sections of the book to discover how to make the white sauce, how to braise the beef, how to blanch the beans.

The strongest benefit of the use case is that it tells a story that provides the context for understanding requirements. Forcing a reader to jump between different use cases risks losing this benefit.

Misunderstanding the includes relationships leads to its misuse. Programmers are trained to break up large blocks of program logic into more manageable sub routines. But when they apply this technique to use cases, they often include more detail in the "called" use cases than in the callers, effectively turning the "high level" use cases into transaction centers that simply call a number of lower level use cases to perform their tasks.

Again, the problem becomes the high probability that the use case will become too formal and the all important story context may be lost. Many use case books encourage this practice by creating contrived examples in an effort to demonstrate <<includes>> relationships.

In addition, programmers are trained to prefer formality to informality and UML tends to encourage an excessive level of formalism in use case models. Relationships between use cases such as <<includes>>, <<extends>>, and generalization have well defined formal definitions in UML. Unfortunately, there is a great deal of uncertainty about how these formal concepts translate to useful concepts in the everyday world of requirements.

Therefore:

Express shared courses of actions with <<included>> use cases.

Idealy, create <<includes>> relationships between use cases when two or more use cases share a common set of steps. Extract the common steps and place them into their own use case.

Avoid using <<includes>> simply as a mechanism for partitioning a long complex use case into smaller ones, even though they seem more manageable. A long complex use case is often a symptom of one or more of the following problems:

- Scope of the use case is too large (identify each **CompleteSingleGoal**(**132**) and **RedistributeTheWealth**(**206**));
- The steps in the use case are written at too low a level of detail for the goal of the use case or are written at different levels of detail(create **LeveledSteps**(**173**)),
- Non functional requirements such as user interface details have crept into the use case (put them into **Adornments**(**147**)).

Copyright © 2001 Adolph, Bramble All Rights Reserved

Examples

Wings Over the World

A problem commonly found in many use case sets is the "director" use case, that is a use case that depends on several <<include>> use cases.

Figure 7-1 Misuse of <<*include*>> to create a director style use case.

Book Flight:

Level: User Goal

1. The use case begins when the agent selects set travel itinerary. Call *Capture Flight Information* to get the travel itinerary.
2. The agent chooses select flight. Call *Reserve Passage* to reserve a flight segment.
3. The agent chooses book flight, Call *Obtain Payment Information* to get payment choice.
4. Call *Issue Ticket* to print ticket.

Use Case 7-1 Use Case Horror: Director Style *Book Flight* use case which incorrectly applies the includes relationship.

The problem is the included use cases in this case are trivial. They are too simple and small to justify their existence as entire use cases. The included use cases are, to use a visual metaphor, droplets. The repair for droplets is to **MergeDroplets**().

Instead, use **EverUnfoldingStory(117)**.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Book Flight:
Level: User Goal.
1. The use case begins when the agent specifies a travel itinerary.
2. The system searches a set of appropriate flights and presents them to the agent.
3. The agent selects flight.
4. The system verifies space is available on the flight and reserves the seats on the flight.
5. The agent finalizes the booking by supplying payment information.
6. The system books the seats, and issues the ticket.

Use Case 7-2 Revised *Book Flight* **use case.**

The appropriate application of <<include>> is to eliminate redundancy that would otherwise make the set of use cases difficult to understand. If two or more use cases have a set of steps in common, than rather than re-writing those steps we can collect them into a common use case.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Here, both *Book Cruise* and *Book Flight* use *Collect Payment Information*. No matter what kind of trip we are booking, we always collect payment the same way, so why write the same thing out twice?

CommonSubBehaviour and UML by Dan Rawthorne

The include relationship is fairly simple and useful and should be used between use cases at the same level of abstraction. It should not be used to organize use cases in an **EverUnfoldingStory**.

As you can see, the <<include>> relationship is a stereotype of a dependency (dashed arrow). If your tool will not allow you to draw this, use either an aggregation or a <<include>> stereotype of an association - don't use a generalization arrow. I know that the initial <<uses>> relationship was a stereotype of a generalization, but his was just wrong, wrong, wrong… there is no way that the include relationship can logically be rationalized as a stereotype of a generalization. So, use either

or

but not

Copyright © 2001 Adolph, Bramble All Rights Reserved

Copyright © 2001 Adolph, Bramble All Rights Reserved

7.2 InterruptsAsExtensions(191)

<<caption and credits missing>>

Writing the use case description as **ScenarioPlusFragments(136)**,

An alternative that affects more than one step in a scenario can scatter related details throughout a use case, causing the reader to become confused or lose track of important information.

It seems that a lot of Human Resources departments go out of their way to make hiring temporary or contract employees difficult. Most firms have a conventional process they follow when they hire a new employee. Usually a new hire simply fills out a personal data form, then some kind of tax withholding form, signs up for company benefits, assigns their

Copyright © 2001 Adolph, Bramble All Rights Reserved

intellectual property rights to the company and finally specifies where they want their pay deposited.

It often seems that the Human Resources department enjoys taking this paper shuffling process to new heights of complexity for contract employees. Just like regular employees, the contract employee will usually have to fill out most of the same forms but they may have to do some or all of the steps differently. For example, contractors may not be eligible for company benefits, or may be required to invoice the company to receive pay. They may have to negotiate transfer of intellectual property rights. So when hiring a new contractor the human resources department may go through each of its normal steps for hiring, but then modify some of the steps or require additional work.

Multiple diversions or alternatives that interrupt several steps in a scenario can cause the readers to lose track of the path that they are trying to follow, and can indicate problems with the basic scenario itself. The recommended approach for coping with an alternative course of action is to write the alternative as a fragment off the main success scenario (**ScenarioPlusFragments(136))**. This technique works well for most alternatives because they are usually a simple diversion from a step in either the main success scenario or a step in a fragment. In either case, once the diversion is completed the scenario resumes from the point where the diversion took place.

However, what happens when the alternative requires repeated diversions on more than one transaction in the use case? You can write a diversion for each step but then it is easy for readers to lose track of the alternative thread that ties these related diversions together.

Creating extension use cases tends to scatter important behavior and make them harder to understand. "One stop shopping" is an important requirements writing principle. Simply, a person reading the requirements should be able to get all of the information they need from one document or from one section of a document. They should not have to jump between different sections to understand the requirement or the context into which the requirement.

A major benefit of *use* cases is that they tell a story that provides the context for understanding requirements. Forcing a reader to jump between different use cases runs the risk of negating this benefit.

Misunderstanding the extends relationships leads to its misuse. If you ask five use case experts "what does the *extends* relationship mean" you will get ten different answers. *Extends* has historically been likened to an inheritance relationship between use cases and most programmers are trained to interpret inheritance as a specialization. Recently however, the UML Revision Task Force (RTF) decided to change the formal definition of *extends* from

Copyright © 2001 Adolph, Bramble All Rights Reserved

an inheritance relationship to that of a dependency. While this change is interesting to those concerned about the formal structure of the UML, it has made the life very difficult for many analysts and in many ways is irrelevant in real life

In addition, programmers are trained to prefer formality to informality and UML tends to encourage an excessive level of formalism in use case models. Relationships between use cases such as <<includes>>, <<extends>>, and generalization have well defined formal definitions in UML. Unfortunately, there is a great deal of uncertainty about how to pragmatically apply these formal concepts in the everyday world of requirements.

Many authors encourage use case writers to incorporate alternative courses of actions as includes relationships. Many use case courses, articles, and text books tend to encourage the use of extends for expressing alternative courses of action, rather than writing the alternative as part of one use case.

Therefore:

Create an extension use case when an alternative course of action interrupts a number of steps in a scenario.

Create an extension use case by extracting all of the steps that are associated with the thread of the alternatives and place them in an extension use case. For each statement in the extension use case, specify which statement in the original or base use case is being interrupted. This approach has two benefits:

- The thread of an interrupting alternative is kept together and therefore the reader can see the context of the alternative (one stop shopping principle).
- The original or base use case is not cluttered with interrupting alternatives.

Examples

Wings Over the World

The following use case describes how a flight is booked. The main success scenario is relatively straightforward with the agent using the system to build up a travel itinerary for a client, and then reserving it. However, there are extensions to many of the steps in the main success scenario if the client is a frequent flier because frequent fliers get privileges and services that are not available to conventional clients.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 7-2 Book Flight Use Case

Book Flight:

1. The use case begins when the agent specifies a travel itinerary for a client.
2. The system searches a set of appropriate flights and presents them to the agent.
3. The agent chooses select flight.
4. The system verifies space is available on the flight and reserves a seat on the flight.
5. The agent finalizes the booking by supplying payment information.
6. The system books the seats, and issues the ticket.

Alternatives:

2a: Client is frequent flier:

2a1 The system retrieves the client's profile and displays the flights sorted by client's airline preference.

4a: Client is a frequent flier:

4a1 The system offers premium seats for the client.

4b: Seat is not available in ticket category

4b1: The system informs the agent that no seats are available in the client's chosen price category.

4b2: The agent specifies another price preference.

4c: Seat is not available (flight fully booked)

4c1: The system informs agent that no seats are available at all.

4c2: The agent specifies another set of departure time preferences.

4d: Client is a frequent flier and seats are not available.

4d1 The agent puts the client on a priority wait listed for the seats.

5a Client is a frequent flier

5a1: The system verifies client has upgrade coupons in their account.

Copyright © 2001 Adolph, Bramble All Rights Reserved

5a2: The system wait lists the client for upgrade on the flight.

Use Case 7-3 Use Case Horror: Confusing Frequent Flier Alternatives for *Book Flight*.

In this example, it is all too easy for the use case reader (and sometimes even the use case author) to forget that all of the individual frequent flyer alternatives listed in the use case's alternatives section represents the thread of an ongoing alternative course of action.

We can clarify this situation by using an extension use case that collects all of the frequent flyer alternatives together.

Figure 7-3 Book Flight Use Case Diagram with Frequent Flyer Extension.

Now all alternative courses of actions associated with the frequent flier are collected together into one use extends use case.

Book Flight for Frequent Flier Extends *Book Flight* 2a1 The system retrieves the client's profile and displays the flights sorted by client's airline preference. 4a1 The system offers premium seats for the client. 5a1: The system verifies client has upgrade coupons in their account. 5a2: The system wait lists the client for upgrade on the flight. Alternatives: **4d: Client is a frequent flier and seats are not available.**

Copyright © 2001 Adolph, Bramble All Rights Reserved

4d1 The agent puts the client on a priority wait listed for the seats.

Use Case 7-4 Extension Use Case for *Book Flight*

The extension use case keeps all the steps in the Frequent Flier alternatives together, making it easier to follow the Frequent Flier thread alternatives. It is less likely now that the reader (and even the use case author) will lose the thread of the alternatives within the other alternatives.

InterruptsAsExtensions and UML by Dan Rawthorne

According to the UML, this use of the extends relationship is legal and proper. However, some people just don't get it because this use of the extends relationship seems complicated to them. All it is doing is saying "this use case does what that one does, with the following additions…" To some people this sounds like one use case inheriting from another, and in some sense it is. In fact, it can be viewed as a legitimate specialization because the **IntentionRevealingName** of the "child" use case indicates that it is more than the parent, rather than just an add-on. So, if it makes more sense to you or those that need to understand your models, just give in and use a specialization relationship, like

Remember that the important thing is to tell a story that is correct and understandable, not to blindly follow some semantic rules. By the way, the only difference in the textual representation of this specialized use case is to replace the line "Extends *Book Flight*" in the second line with "Specializes *Book Flight*".

Copyright © 2001 Adolph, Bramble All Rights Reserved

Extensions and UML Extension Points

Extension use cases are not independent of their base use case and must reference statements in the base use case. For example, the extension use case *Book Flight For Frequent Flyer* directly references statements in the base use case *Book Flight*. Each of the alternatives and statements in *Book Flight For Frequent Flyer* are direct reference or based on the statement numbering of *Book Flight*. If *Book Flight* is updated and the statement numbering changes, then we will have to re-label and renumber the statements in *Book Flight For Frequent Flyer* as well. The UML extension point feature helps mitigate this problem by providing extension points for the extension use case. Technically, an extension point is a point at which additional behaviour may be inserted. These extension points are like symbolic labels and help make the extension use case more independent of the internal structure of the base use case.

Figure 7-4 UML Use Case Diagram Featuring an Extension Point.

Book Flight:

1. The use case begins when the agent specifies a travel itinerary for a client.
2. The system searches a set of appropriate flights and presents them to the agent. (*frequent flyer a*)
3. The agent chooses select flight.
4. The system verifies space is available on the flight and reserves a seat on the flight. (*frequent flyer b*)

Copyright © 2001 Adolph, Bramble All Rights Reserved

5. The agent finalizes the booking by supplying payment information. (*frequent flyer c*)
6. The system books the seats, and issues the ticket.

Alternatives:

4a: Seat is not available in ticket category

4b1: The system informs the agent that no seats are available in the client's chosen price category.

4b2: The agent specifies another price preference.

4b: Seat is not available (flight fully booked)

4c1: The system informs agent that no seats are available at all.

4c2: The agent specifies another set of departure time preferences.(*frequent flyer d)*

Use Case 7-5 Book Flight Use Case with Extension Points.

Book Flight for Frequent Flier

Extends *Book Flight*

frequent flyer a

1 The system retrieves the client's profile and displays the flights sorted by client's airline preference.

frequent flyer b

1 The system offers premium seats for the client.

frequent flyer c

1: The system verifies client has upgrade coupons in their account.

2: The system wait lists the client for upgrade on the flight.

Alternatives:

Seats are not available.

frequent flyer d

1 The agent puts the client on a priority wait listed for the seats.

Use Case 7-6 Extension Use Case Referencing Extension Points

Copyright © 2001 Adolph, Bramble All Rights Reserved

PromoteAlternative(196)

<<caption and credits missing>>

Writing the Use Case description as **ScenarioPlusFragments(136)**,

Long or complex alternatives can dominate a use case, and appear to be more important than they really are just because they are so prominent.

Normally, the military strictly adheres to a formal, well-defined process for promoting officers. Candidates must meet all of the qualifications for their particular rank, have served the required amount of time, and have a spotless service record. But in wartime, especially during urgent situations, commanders will promote someone, be it an experienced enlisted man or a junior officer to fill a critical need. They refer to these special cases as “Battlefield Promotions”.

Sometimes, an alternative scenario stands out. It suddenly becomes important, and while it doesn’t meet the requirements of a full use case, it fills an important hole in the system, and it makes sense to elevate it to extension case status.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Complex or long alternatives clutter a use case and can obscure other alternatives. A use case can become difficult to write and even more difficult to follow when the alternative is too long or complex, especially when it contains several alternatives itself. Use cases are supposed to be easy to read and understand not complicated engineering documents.

Some problems are very complex, and require complex use cases to adequately describe them. Hard technical problems don't simplify themselves just because we want to simplify our use cases. Some systems can be very complicated and hard to understand. Yet these are the kinds of systems that we need to describe especially well so that everyone working on them will have a clear understanding of how the system is supposed to behave in all reasonable situations.

Partitioning individual use cases tends to scatter important behavior and make them harder to understand. One of the biggest benefits of using use cases is that they tell stories that provide a context for understanding requirements. Forcing readers to jump between different use cases risks negating this benefit, because it violates the "one-stop shopping" rule. Unfortunately, many use case methodologists tend to encourage use case fragmentation by suggesting that people should use "extends" to represent alternatives, going so far as to that it is the proper way to add any new feature to an existing use case model. But we believe that people should be able to get all of the information they need about a requirement from one place, without having to refer to several documents to understand it.

Therefore:

Consider moving complex alternatives that overly dominate a use case into separate extension use cases.

An alternative course of action should never dominate a use case. After all, it's supposed to be an alternative and represents a less common behavior than the main success scenario. The best way for handling alternative courses of action is to write them as fragments based on a main success scenario (**ScenarioPlusFragments(136)**). But sometimes there are alternatives which may require long complex series of steps (e.g. control break processing, and boundary conditions). Under these circumstances you may consider extracting the alternative from its base use case and create a separate use case for it. Use an <<includes>> relationship to link to the new use case.

You need to exercise caution when using this pattern to justify the creation of an extension use case because complex alternatives may indicate that the use case is too detailed or too large (see **EverUnfoldingStory(117)**, **LeveledSteps(173)**, **CompleteSingleGoal(132)**). If

Copyright © 2001 Adolph, Bramble All Rights Reserved

this is not the case, then promoting the alternative to an extension use case is a reasonable course of action.

Examples

Wings Over the World

All of us who have to frequently fly in those small things the airlines pass off for seats appreciate seat upgrades. The following use case is an example of where a seat upgrade may be requested at the time of booking.

Book Flight:

1. The use case begins when the agent specifies a travel itinerary for a client.
2. The system searches a set of appropriate flights and presents them to the agent.
3. The agent chooses select flight.
4. The system verifies space is available on the flight and reserves a seat on the flight.
5. The agent finalizes the booking by supplying payment information.
6. The system books the seats, and issues the ticket.

Alternatives:

4a: Client is eligible for seat upgrade.

4a1: The agent requests an upgrade for the client.

4a2: The system verifies there are seats available for upgrade on the selected flight.

4a3: The system verifies client has sufficient upgrade coupons for upgrade.

4a4: The system changes ticket class to “upgraded”.

4a Alternatives:

4a2a: No seats available for upgrade.

4a2a1: Agent places client on wait list for upgrades.

4a2a2: continue with regular ticket booking at step 4.

4a3a: Client has insufficient upgrade coupons.

4a3a1: Agent buys additional upgrade coupons for client.

Copyright © 2001 Adolph, Bramble All Rights Reserved

4b: Seat is not available in ticket category

4b1: The system informs the agent that no seats are available in the client's chosen price category.

4b2: The agent specifies another price preference.

4c: Seat is not available (flight fully booked)

4c1: The system informs agent that no seats are available at all.

4c2: The agent specifies another set of departure time preferences.

Use Case 7-7 Use Case Horror: Request Upgrade alternative dominates *Book Flight* Use Case.

Notice how the "*Requesting an Upgrade*" alternative obscures the other alternatives, and dominates the use case. Promoting the alternative to an extension use case makes the main use case much simpler to follow.

Book Flight:

1. The use case begins when the agent specifies a travel itinerary for a client.
2. The system searches a set of appropriate flights and presents them to the agent.
3. The agent chooses select flight.
4. The system verifies space is available on the flight and reserves a seat on the flight.
5. The agent finalizes the booking by supplying payment information.

Copyright © 2001 Adolph, Bramble All Rights Reserved

6. The system books the seats, and issues the ticket.

Alternatives:

4b: Seat is not available in ticket category

4b1: The system informs the agent that no seats are available in the client's chosen price category.

4b2: The agent specifies another price preference.

4c: Seat is not available (flight fully booked)

4c1: The system informs agent that no seats are available at all.

4c2: The agent specifies another set of departure time preferences.

Use Case 7-8 *Book Flight* after Upgrade Seat alternative has been Promoted.

Upgrade Seat

Extends *Book Flight* at step 4 when the client is a frequent flyer

1: The agent requests an upgrade for the client.

2: The system verifies there are seats available for upgrade on the selected flight.

3: The system verifies client has sufficient upgrade coupons for upgrade.

4: The system changes ticket class to "upgraded".

Alternatives:

2a: No seats available for upgrade.

2a1: Agent places client on wait list for upgrades.

2a2: continue with regular ticket booking at step 4.

3a: Client has insufficient upgrade coupons.

3a1: Agent buys additional upgrade coupons for client.

Use Case 7-9 *Upgrade Seat* as an extension use case.

Notice how much simpler *Book Flight* is, now that *Upgrade Seat* no longer dominates it. We wish to emphasize that this pattern is intended to solve those situations in which a large complex alternative is obscuring the other alternatives and making the use case difficult to read.

Copyright © 2001 Adolph, Bramble All Rights Reserved

PromoteAlternatives and UML by Dan Rawthorne

This is a classic use of an extend relationship, and what I believe the UML developers had in mind. Note that the extending use case has an **IntentionRevealingName(XXX)** that represents the additional functionality by itself rather than the totality of the extended use case and the additional functionality. This makes it possible for an extending use case to extend more than one base use case, which is an option that is explicitly mentioned in the UML. For example, the "Upgrade Seat" use case could conceivably extend a "Book Train" use case, as well.

Therefore, unlike the case in **InterruptsAsExtensions(XXX)**, it is inappropriate for this relationship to be represented by a specialization, whether stereotyped or not. So, if your tool will not allow an extends relationship, use a stereotyped association, or even an aggregation, but not a generalization - unless you change the name appropriately

.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Tradeoffs and Collaborations

These guidelines improve the organization of the **ScenarioPlusFragments(136)**) by helping us to either reduce complexity or eliminate redundancy in our use cases. But these benefits can be expensive, because adding additional extension use cases can break up the flow of the story, making it harder for the reader to follow it by requiring them to find and reference additional use cases. Therefore, you need to weigh the benefit of reduced complexity against the benefits of better readability when deciding whether to create specialized use cases.

Figure 7-5

Relationships between the Relationship Patterns

These patterns govern extracting information from a use case under different circumstances. **CommonSubBehavior(186)** recommends that you extract behavior only when it is common to several use cases to reduce redundancy and potential inconsistencies. **InterruptsAsExtensions(191)** and **PromoteAlternative(196)s** focus solely on alternative courses within a scenario. **InterruptsAsExtensions(191)** describes creating extension use cases when an alternative effects multiple steps within a scenario, so that the alternative course is easier to follow. Sometimes though, alternatives can be so complex, or contain so much important information, that it makes sense to create a new extension use case as well (**PromoteAlternative(196)s**) so that the alternative doesn't overshadow the use case.

The question is when should you use these constructs? The answer is judiciously, and only when their use adds some demonstrable value to your use cases. Properly used, they can greatly improve the quality and even readability of a set of use cases. The danger is that they are quite easy to abuse, and can cause more problems than they solve. For that reason, you're better off avoiding them when their use is questionable, and only apply them when you can show some benefit from doing so. As a general rule, "If in doubt, don't use them".

Copyright © 2001 Adolph, Bramble All Rights Reserved

These patterns apply to only a few situations that specifically deal with alternative courses and repetitive behavior. You are likely to find other problems with your use cases, such as excessive complexity, which require you to significantly reorganize them. That subject is the topic of the next chapter.

Copyright © 2001 Adolph, Bramble All Rights Reserved

7.3 CaptureTheAbstraction – A Pattern for Applying UML Generalization by Dan Rawthorne

<image of a plane and a train both going from one city to another> No appropriate image yet.

Writing the Use Case description as **ScenarioPlusFragments(136)**,

Trying to document two or more distinct alternatives, neither of which is dominant, is very difficult and confusing in a single use case.

In Europe businessmen are as likely to take trains as planes for business travel. For example, to travel from Frankfurt to Munich by train takes approximately four hours door to door, while it takes three hours by plane. The plane costs slightly more in order to compensate for the time saved.

Sometimes there is more than one dominant scenario for a single use case. Each of them could be its own use case, but they all fill the same hole in the system, meeting the same goals for the same actors. When this happens it makes sense to make an abstract use case that documents the actors and goals, put each scenario in its own use case, and have them inherit from the abstract one.

Complex or long alternatives clutter a use case and can obscure other alternatives. A use case can become difficult to write and even more difficult to follow when the alternatives are too long or complex, and cannot be easily written as a main scenario plus fragments. Use cases are supposed to be easy to read and understand, not complicated engineering documents.

Some problems are very complex, and have several distinct solutions. Keeping the solutions in separate use cases makes writing them easier, but loses the traceability to the problem. Yet this traceability is important so that everyone working on the system will have a clear understanding of how the system is supposed to behave in all reasonable situations.

Having many individual use cases tends to scatter important behavior and makes the behavior harder to understand. One of the biggest benefits of using use cases is that they tell stories that provide a context for understanding requirements. Forcing readers to jump between different use cases to understand solutions risks negating this benefit, because it violates the “one-stop shopping” rule. We believe that people should be able to get all of the information they need about a requirement through one place, without having to explore several documents to understand it.

Therefore:

Copyright © 2001 Adolph, Bramble All Rights Reserved

Consider creating an abstract use case and putting each distinct scenario in its own use case that specializes the abstraction.

A use case is defined by its goal. When more than one use case has the same goal there is the possibility of confusion. Logically, in this situation you have multiple use cases that collectively contain a number of scenarios of the same use case. The first option would be to combine the use cases using **MergeDroplets(XXX)**, but this can't be done if the individual scenarios are complex. Second choice would be to document the dominant scenario in the main use case, and have the secondary ones as extension use cases, using **InterruptsAsExtensions(XXX)** or **PromoteAlternatives(XXX)**. You should only consider using an abstraction if there were no dominant alternative.

You need to exercise caution when using this pattern to justify the creation of an abstract use case because complex alternatives may indicate that the use case is too detailed or too large (see **EverUnfoldingStory(117)**, **LeveledSteps(173)**, **CompleteSingleGoal(132)**). If this is not the case, then putting each distinct alternative in its own use case that specializes an abstract use case is a reasonable course of action.

Examples

Wings Over the World

Assume that retinal scanners are being installed in order to identify agents that want to use the system. If it is also possible to login to the system in the usual way (name and password) then we are in a situation where an abstraction named *Validate Agent* may be appropriate, as follows. (note that italics indicate abstraction in the UML).

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 7-6 UML Generalization Relationship

Note that there is a natural tension between this pattern and **TechnologyNeutral(XXX)**, as these abstractions are often used to eliminate technology variations that lie in the specializations. In fact, it could be argued that this example is just such a thing. However, sometimes discussion of technology is necessary, as it is part of the problem domain and not the solution domain. A general rule is that mentioning technology is permissible if it is an intrinsic part of the goal, but not when it is part of the solution documented in the alternatives.

Capture the Abstraction and UML

This technique is often used graphically when eliciting requirements using **BreadthBeforeDepth(XXX)**. It should be used sparingly, and pains must be taken to assure that both the abstraction and the specializations are at the same level of abstraction. When using this pattern make sure to use the abstraction when you use the **EverUnfoldingStory(XXX)**; that is, refer to "*Validate Agent*" rather than either "*Validate Agent with Login*" or "*Validate Agent with Retinal Scan*".

Copyright © 2001 Adolph, Bramble All Rights Reserved

8 Editing Existing Use Cases

Well it Sounded Like a Good Idea at the Time.

You are sitting with Sitra in one of those tiny closets that *Wings Over the World* calls a conference room. Sitra had been working with Yitka developing the requirements for creating new client accounts. Things seem to have been going well, until Yitka literally tore a strip off of poor Sitra's use cases. Now Sitra has asked you to help her revise them.

Sitra: When I started working with Yitka on the new client account feature, I built this little UI prototype and walked her through each of the forms. She was happy enough about that, but when I wrote up the use cases she just threw them back at me saying these are "like little pieces of a puzzle, there is no story here". I don't think anyone can make that woman happy.

You pick up the use cases from the table and quickly glance at their names, ", *Open Account*, *Fill in Client Personal Details*, *Fill in Client Travel Preferences*, *Save Client Account*,"

You: This looks like you created a use case for each one of the user interface forms that the user fills out to create a new client account.

Sitra: That's exactly what I did. I thought that would be a good way of doing things because I could show someone the user interface form, and then the use case that drives it. I thought a use case was a terrific way to deal with all the different errors someone could get if they filled the form in wrong.

You: Ok, a lot of people run into this problem. Usually, filling in a single user interface form does not represent a **CompleteSingleGoal(132)** for the user. This use case here, *Open Account*, it really has no value unless I go on to also *Fill in Client Personal Details*, *Fill in Client Travel Preferences* and then *Save Client Account*. None of these use cases really stand on their own, and non of them have enough context in them to make it easy for a reviewer to understand.

Sitra: So what should I do?

You: There is nothing wrong with having an ongoing dialog with the actor, and nothing wrong with having a few forms pop up for a use case. This is very good in fact because the use case provides the context for the forms. Think of each of these use cases that you have written as a fragment or droplet if you like, of the real use case that has a useful goal. Call that new merged use case Create *New Client*. We have a guideline called **MergeDroplets(211)** that applies to situations like this.

You notice that Sitra is quickly counting up the steps for the new use case,

Copyright © 2001 Adolph, Bramble All Rights Reserved

Sitra: There'll be almost twenty steps in the main success scenario if we merge all of these use cases together.

You: Yes, so it means we may have to edit the new use case. The odds are that most of the steps are low level steps so if they are still relevant then we may **RedistributeTheWealth(206)** by moving them to lower level use cases.

Analysis is a process of discovery with many false starts and dead ends. What seemed like a good idea at the time may later become redundant or obsolete. Inevitably, some of your use cases will become out-of-date and fail to properly describe the system you are building. None of this is surprising, for two reasons. First, requirement specification is not a precise science, so the requirements underpinning the system are themselves highly arbitrary and subject to frequent change because writing use cases is frequently a revealing process. Ideas that made sense when you started writing your use cases may no longer seem reasonable as you discover more about the underlying system. Second, use cases that span multiple development cycles are likely to fall behind as the system matures and evolves in unforeseen directions. Consequently, your use cases may fail to keep up as your understanding of the system evolves.

Use cases can degrade in several ways. They might contain outdated information left over from the initial attempts at describing the system. Or, they might partition the system in a manner that no longer supports the vision or business case for the system. Concepts that once represented a useful sequence of events may now illuminate only a fragment of some isolated behavior. Conversely, some use cases may be too big or may encompass too much functionality, describing too many actions in a manner similar to Riel's "God Class" problem [Object-Oriented Heuristics].

You have several options available for correcting these problems. You may decide that it wasn't **QuittingTime(71)** after all, and you need to enhance or even write more use cases. On the other hand, you may find that your use cases are reasonably complete, and you just need to spend some time reorganizing them to improve them. Obsolete use cases are the easiest to handle, as you can simply (**CleanHouse(216)**) by discarding them

In his book Refactoring, which deals with refactoring object-oriented software, Martin Fowler refers to indications that software may require refactoring as "*bad smells*" [FOW1999]. One of the "bad smells" we think that you should always be sniffing about for in a use case is excessive length. The main success scenario of a good use case is usually between three to nine steps. A main success scenario containing more than 9 steps may indicate that either the use case has multiple goals or includes lower level details. In either case, you should

Copyright © 2001 Adolph, Bramble All Rights Reserved

RedistributeTheWealth(206), equitably partitioning the larger use case into smaller cohesive ones.

Another “bad smell” can result from use cases that fail to address a **CompleteSingleGoal(132)** and therefore deliver only a fragment of a **UserValuedTransaction(110)** to the primary actor. This bad smell frequently results from either CRUD style use cases or systems where the use case authors have created a use case for each user interface form. [LIL1999]

This section describes three patterns for removing “bad smells” from existing use cases. **RedistributeTheWealth(206)** involves separating the independent portions of an overly complex use case into simpler ones that can stand alone. **MergeDroplets(211)** describes the opposite action of combining fragmented use cases into more complete ones. Neither of these actions is arbitrary, but involves applying some rules so that the resulting use cases are both cohesive and singular in purpose. Lastly, **CleanHouse(216)** recommends eliminating those use cases that do not provide value, or help us see how the actor can achieve a goal.

Copyright © 2001 Adolph, Bramble All Rights Reserved

8.1 RedistributeTheWealth(206)

Children waiting in line for soup given out each night by the city mission, a community chest- financed organization. Dubuque, Iowa.

Vachon, John, 1914-1975, photographer.

CREATED/PUBLISHED
1940 Apr

You are discovering the **UserValuedTransaction(110)s** for the system.

An excessively long use case is unwieldy and hard to use, causing users to become distracted and lose focus

Copyright © 2001 Adolph, Bramble All Rights Reserved

During the 60's and 70's, many corporations formed into conglomerates that operated in many different unrelated lines of business. For example, Canadian Pacific Ltd. operated railways, passenger airlines, shipping lines, and developed real estate. The multi-national oil company Exxon even started a computer division. The business diversity of the conglomerate was seen as a way to protect the interests of shareholders from down turns in a specific market sector. The principle behind these conglomerates boiled down to the prudent proverb "Don't put all your eggs in one basket".

The problem was that the conglomerates' share prices and earnings tended to lag behind those of more narrowly focused corporations. The usual source of the problem was the difficulty corporate managers had in creating a single unifying vision for the company to embrace across all of its diversified divisions. Eventually, in the 90's, most of the conglomerates broke themselves up into separately traded corporations. In most cases, the value of the shareholder assets increased.

The use case's goal must be clear to the stakeholders. The ideal use case is clear, precise and unambiguous because it forms an implicit contract[1] between the developers and the stakeholders. The use cases must paint a clear picture of the system, demonstrating how everyday users will normally use it in language that they can understand.

It is expensive to add new use cases. New use cases don't appear by magic and often require the efforts of several people following a specific process. Someone must spend time sifting through requirements or talking to customers to understand a feature's basic characteristics, and then exhaustively identifying all plausible error conditions and alternative behaviors. One or more persons need to organize this information, and write the text, while others must review it. Additional use cases mean additional development work, as your developers must implement the new feature. All of these efforts take time and cost money, so it makes sense to only write those use cases that provide measurable value.

Excessive detail can make use cases too hard to read and understand. Each use case should address only one goal and clearly describe how the system helps an actor reach that goal (**CompleteSingleGoal(132)**). If it attempts to go beyond this scope, then the readers, and even the writers, can easily lose sight of the important parts of the story line in the tangents and various plot twists. Use cases that describe behavior beyond their prime objective cloud the boundaries between use cases, duplicating or even contradicting them, making the system even murkier and harder to understand.

---

[1] And perhaps even a legal one.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Therefore:

Move a long, unwieldy passage or an overly complex extension to its own use case.

An effective use case will address a **CompleteSingleGoal(132)**. Be especially wary of long use cases. While size by itself may not be a problem, a long main success scenario (say over nine steps) may be a "bad smell" indicating that the use case addresses more than one goal. When you discover that a use case has multiple goals, divide it into separate use cases, one for each goal. How you reorganize depends on the level of those goals and their relationship to other use cases.

Relocate fragments to other use cases. If the stories for these extra goals are fragments of behavior that belongs in another use case, then incorporate them into that use case. **MergeDroplets(211)** by folding these fragments into the appropriate use case, providing that you can do so without violating the other use case's **CompleteSingleGoal(132)**.

Create a new use case If the extra stories describe behavior that both stands alone and fulfils an actor's goal, then create a new use case for that behavior. This approach requires more than just moving the extra pieces to another document and calling it a complete use case, because the result must meet all of the criteria for a use case (**CompleteSingleGoal(132)**, with an **IntentionRevealingName(144)**). It also involves making the new use case complete by extracting related information from the original use case, as well as identifying new alternative scenarios. If the resulting use case(s) are closely coupled, then this may indicate potential relationships between the use cases ((**PromoteAlternatives(196), InterruptsAsExtentions(191), CommonSubBehavior(186)).**

Create a new lower level use case If there are extra story steps that describe lower level behavior then level the use case steps (**LeveledSteps(173)**) by creating a lower level use case with its own **CompleteSingleGoal(132).** Relocate the lower level steps to the new use case (**EverUnfoldingStory(117)**, and reference the lower level use case from the original (this is the <<Includes>> relation)..

Relocate superfluous If these superfluous story fragments were included to help clarify the use

Copyright © 2001 Adolph, Bramble All Rights Reserved

fragments to the case description then save these pieces as **Adornments(147)**.
Supplementary
Specifications

Examples:

Call Processing

Consider the following use case, which provides a simplified description of making a phone call from the viewpoint of the service provider:

USE CASE 42: Make a Phone Call

PRIMARY ACTORS:	**Caller** –	Person making the call.
	Called –	Person being called.
SECONDARY ACTOR:	**Switching Network -**	Equipment handling calls between telephones.
LEVEL:	**User Goal**	

MAIN SUCCESS SCENARIO:

1) **Caller** takes the phone "off-hook".
2) **Caller** dials number.
3) **Switching Network** translates the digits, connects **Caller** to **Called**'s phone and sends Ring signal.
4) **Called** answers the phone (goes off-hook) and talks to **Caller**.
5) **Called** and **Caller** hang up.
6) **Switching Network** logs billing information about the call. If this is the **Caller's** first call of the day, it initializes a daily usage report entry in the customer's file. If it this is a local call, then it logs it in the Service Providers log as well. If it is a long distance, then it records an entry in the appropriate long distance provider's log.
7) **Switching Network** tears down the connection and frees all of components that the call required.
8) The use case ends once the Switching Network frees all of its components used in the call.

Use Case 8-1 Use Case Horror: A use case with multiple goals.

Copyright © 2001 Adolph, Bramble All Rights Reserved

While its length is acceptable, this use case violates **CompleteSingleGoal(132)** because it describes two separate behaviors, making a phone call, and billing the user. Once you detect this situation, your options include creating a new use case, creating a new sub use case, or even including the extra information as **Adornments(147)**. In this case, log billing information appears to fulfill a lower level goal and so a sub-use case is appropriate (**EverUnfoldingStory(117)**).

To **RedistributeTheWealth(206)**, you should:

1. Extract the bill logging information from the use case, and create a new sub-use case with it.

SUB-USE CASE 43 : Log Billing Information

PRIMARY ACTOR: **Caller** – Person making the call.

Supporting ACTOR: **Switching Network** – Equipment handling calls between telephones.

LEVEL: **User Goal**

MAIN SUCCESS SCENARIO:

1) **Switching Network** logs billing information about the call. If this is the **Caller's** first call of the day, it initializes a daily usage report entry in the customer's file. If it this is a local call, then it logs it in the Service Providers log as well. If it is a long distance, then it records an entry in the appropriate long distance provider's log.

Use Case 8-2 RedistributingTheWealth by Creating the Log Billing Information Use Case.

2. Modify step 6 in the Process Normal Call Use Case to refer to this new sub-use case.

USE CASE 42: Process Normal Call

PRIMARY ACTORS: **Caller** – Person making the call.

Called – Person being called.

Supporting ACTORS: **Switching Network** – Equipment handling calls between telephones.

LEVEL: **User Goal**

MAIN SUCCESS SCENARIO:

Copyright © 2001 Adolph, Bramble All Rights Reserved

1)	**Caller** takes the phone “off-hook”.
2)	**Caller** dials number.
3)	**Switching Network** translates the digits, connects **Caller** to **Called**’s phone and sends Ring signal.
4)	**Called** answers the phone (goes off-hook) and talks to **Caller**.
5)	**Called** and **Caller** hang up.
6)	**Switching Network** Logs Billing Information.
7)	**Switching Network** tears down the connection and frees all of components that the call required.
8)	The use case ends once the **Switching Network** frees all of its components used in the call.

Use Case 8-3 Redistributing the wealth to give a use case a CompleteSingleGoal(132).

Copyright © 2001 Adolph, Bramble All Rights Reserved

8.2 MergeDroplets(211)

Blacksmith, Southeast Missouri Farms.

Vachon, John, 1914-1975, photographer.

CREATED/PUBLISHED
1940 May.

You are discovering the **UserValuedTransaction(110)s** for the system.

Use cases that describe tiny or isolated fragments of behavior don't convey enough information to help the readers understand how the system delivers UserValuedTransactions.

Many of us enjoy playing with picture puzzles. There is a wonderful challenge in trying to merge hundreds, sometimes thousands of little picture fragments into a single complete picture. Imagine sitting at the kitchen table with hundreds or even thousands of picture pieces scattered across its top. You carefully compare a fragment of sky to the picture on the box and trying to position it roughly were you think it should be. Slowly, bigger pieces of the puzzle come together. Soon you will have larger fragments of sky, ground, and the castle on the hill. Some serious

Copyright © 2001 Adolph, Bramble All Rights Reserved

puzzle aficionados refuse to even look at the picture on the box; instead they turn all the pieces over and solve the puzzle only by merging pieces without help from matching color or images.

Use cases are not puzzles. A use case represents a **CompleteSingleGoal(132)**, and a reader should not have to hunt down and mentally merge several use cases to see the complete picture.

Partial use cases are incomplete and fail to tell the whole story. An individual use case should present a "complete usage of value" to the actor. Fragments of use cases or partial use cases will cause readers to lose sight of system goals and overlook valuable features. To understand a single and complete behavior a reader is forced to either find other use cases, if they can find them, or attempt to fill in the gaps themselves. This action often results in them missing important features or adding unnecessary ones.

It is best to localize information whenever possible. Distributing the thread of a story throughout a set of use cases makes it very difficult for the readers to follow the narrative. Localizing information about a feature to a use case helps the readers understand the primary actor's goal and the context surrounding the functional requirements.

Developers often implement systems on a per-use case basis. If you distribute actions across multiple use cases, then you increase the likelihood that your developers will miss some features, or that multiple teams will be unknowingly working on the same features, wasting valuable development time, and potentially delaying the project.

It is best to minimize the number of use cases. Every use case you must implement adds to the cost of a project, requiring you to manage, design, write, review and test more code. Every additional feature adds an element of risk to the product, so you want to implement as few use cases as necessary when building a system, especially when time to market is an important factor.

Smaller use cases are easier to understand and use. Ideally, you want to write short, concise use cases that tell the reader everything that they need to know about a system, nothing more, and nothing less. Each of your use cases should adequately describe how the system helps the actor meet one goal (**CompleteSingleGoal(132)**), without unnecessary or technical (**TechnologyNeutral(177)**) details. Otherwise, you run the risk of overwhelming your readers.

Therefore:

Merge related tiny use cases or use case fragments into use cases that relate to the same goal.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Merging use cases requires more effort than simply throwing several steps together and calling the results a use case, because the final product must still contain the patterns for good use case. The best approach is to carefully edit the pieces together into cohesive unit that describes a **CompleteSingleGoal(132)**. If the resulting use cases are too large or contain too much detail, then refine their steps to keep the **VisibleActorIntent(161)** or level them to make them more understandable (**LeveledSteps(173)**, **EverUnfoldingStory(117)**). It is quite likely that you will also have to sift through the alternative courses, removing some and adding others.

How you reorganize partial use cases depends on their presumed goal, and their relationship to other use cases.

Merge fragments to create a new use case.

If several partial use cases describe behaviors that are related to the same goal, then merge them together. Merging partial use cases together especially makes sense when they must run in a specific order, because serial use cases are prone to repeat information. The resulting use case must still meet all of the criteria for a use case **CompleteSingleGoal(132)**.

Merge fragments into existing use cases

If a use case describes behavior that belongs in another use case, then incorporate it into that use case. As with **RedistributeTheWealth(206)**, you should only combine them when you can do it without violating the resulting use case's integrity and still provide a **CompleteSingleGoal(132)**.

Examples:

Call Processing

Let's look at making a phone call again. It can be tempting to break a use case down into smaller, serialized use cases, with each describing a distinct functional action, as this simplified example shows:

Primary ACTORS:	**Caller**:	Person making the call.
	Called:	Person receiving the call.
SUPPORTING ACTORS:	**Switching Network**	Equipment handling calls between telephones.
LEVEL:	**User Goal**	

Copyright © 2001 Adolph, Bramble All Rights Reserved

USE CASE 1: Place Call

Preconditions: **Caller's** and **Called's** phones are on-hook.

Success Guarantee: System has collected all of **Caller's** dialed digits.

Main Course:

1) The use case begins when **Caller** takes phone "off-hook".
2) **Switching Network** sends dial tone.
3) **Caller** dials call.
4) **Switching Network** translates digits
5) **Switching Network** connects call to **Called's** phone.
6) **Switching Network** sends ring signal to **Called's** phone.
7) **Switching Network** sends ring signal to **Caller's** receiver, and the use case terminates.

USE CASE 2: Talk on the Phone

Preconditions: Use Case *Place Call* has sent ring signal to **Called's** phone.

Success Guarantee: Users have finished call.

Main Course:

1) The use case begins when **Called's** phone rings.
2) **Called** answers phone, taking phone "off-hook".
3) **Switching Network** completes connecting **Caller** and **Called**.
4) **Called** and **Caller** talk.
5) **Called** and **Caller** hang up, and the use case terminates.

USE CASE 3: Terminate Call

Preconditions: Call is connected via *Talk on Phone*.

Success Guarantee: **Switching Network** has broken down call, and all components are in the same state they were before call.

Main Course:

1) The use case begins when **Caller** and **Called's** hang up.
2) The **Switching Network** frees all of the components that the call

Copyright © 2001 Adolph, Bramble All Rights Reserved

required.

3) The **Switching Network** Logs Billing Information, and the use case terminates.

Use Case 8-4 Use Case Horror: Set of small use cases for placing a telephone call.

This level of proceduralization is great for writing code, but not use cases. These use cases are incomplete as they only describe part of the call, and all but the first use case depend upon the completion of other use cases. Also, each of these use cases met a sub-goal of our main goal, which is making a phone call. These factors indicate that we should merge these use cases.

A better way to write this use case is:

USE CASE 5: Make a phone call.

PRIMARY ACTORS:	**Caller**:	Person making the call.
	Called:	Person receiving the call.
SUPPORTING ACTORS:	**Switching Network**	Equipment handling calls between telephones.
LEVEL:	**User Goal**	

Preconditions: Caller's and Called's phones are on-hook.

Success Guarantee: **Switching Network has broken down call, and all components are in the same state they were before call.**

Main Course:

1) The use case begins when **Caller** takes phone "off-hook".
2) **Switching Network** sends dial tone.
3) **Caller** dials call.
4) **Switching Network** translates digits
5) **Switching Network** connects call to **Called's** phone.
6) **Switching Network** sends ring signal to **Called's** phone and to **Caller's** receiver (so that they can hear it ringing).
7) **Called** answers phone and talks to **Caller**.
8) **Called** and **Caller** hang up.
9) The **Switching Network** breaks down the connection,

Copyright © 2001 Adolph, Bramble All Rights Reserved

Logs Billing Information, and the use case terminates.

Use Case 8-5 Droplet use cases merged into a new use case.

This use case is much easier for the readers to follow, because it contains all of the information necessary to place, make, and conclude a phone call. The reader doesn't have to sift through several use cases to find the necessary information about this behavior, it is all there, and in order.

Also notice, that in this example at least, the total number of steps required to express the behavior drops once the use cases are combined. The combined use case contains as much information as the three partial use cases, but it doesn't need to repeat the setup data or provide information to help the readers connect the separate smaller use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

8.3 CleanHouse(216)

<<caption and credits missing>>

You are discovering the **UserValuedTransaction(110)s** for the system.

Use cases that don't contribute value to the whole are distracting, and can lead the audience down the wrong path.

Our culture is one of acquisition. Closets, basements and attics throughout North America attest to our ability to accumulate possessions, most of them long forgotten. Our garages are so full we can't park our cars in them. We spend countless hours cleaning and rearranging things, trying to find better ways of storing these belongings so that they aren't in our way, even to the point of renting expensive storage units so we can accumulate even more. Incredibly, we spend more dollars and hours boxing up these items and transporting them across the country when we move, so that we can clutter up our new garages, closets, basements and attics. Much of this stuff we will never use again, but we keep it anyway, because "it might come in handy someday." But do we really want polyester to come back into fashion? Ironically, people usually feel much better after they finally throw it away, and wonder why they didn't do it years earlier.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Similarly, unnecessary use cases become nothing but clutter, distracting the readers from the system's main purpose. Worse, they can waste a lot of time and energy working on something that no one either wants or needs.

Cleaning up old use cases consumes time and intellectual energy. You can't modify or discard active use cases without understanding them or their value to your system. New ones are usually fresh, and relatively easy to understand, but old ones often require work. Relearning takes time, and the older your use cases, the longer it will take because you will likely need to review your entire collection before you can accurately make any changes or decide that you no longer need some of them.

Unused use cases clutter up the workspace and waste intellectual energy. Every use case you create requires some effort to maintain and adds to the cost of a project. The more use cases you have, the harder it is for the readers to keep them straight. A collection of use cases should only contain members that describe meaningful interactions between the various actors and the system. Meaningless features don't add any value to the system; rather they only create unnecessary work for everyone using them. Unnecessary use cases can become legacy items that no one understands, yet everyone is afraid to remove because they might be valuable (Lava Flow Pattern [Brown]), and remain as a fixture for the life of the system.

Removing a use case removes a huge amount of development work and conserves intellectual energy. Every use case that you have to implement requires more time, increases your costs and adds an element of risk to your project. You want to implement only as many use cases as necessary. Every use case that you can eliminate right from the start reduces your overhead significantly.

The consequence of leaving unnecessary use cases in your collection is that someone might try to implement them, wasting valuable development time on something you don't need and delaying your time to market. This possibility is more likely in large development organizations, where the writers and implementers are separated by organizational boundaries, and don't communicate much.

Therefore:

Remove those use cases that do not add value to your system or that are no longer in your active list.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Remove these use cases from your collection as soon as you determine that they no longer contribute value (**UserValuedTransaction(110)s**). You may save any of them you feel are useful as **Adornments(147)**, but discontinue writing them immediately. A general rule for determining whether to discard a use case is when in doubt, throw it out. Don't worry if you throw out what later turns out to be a valuable use case. If it is valuable, then you can always bring it back into your collection later.

Weigh the implementation cost of your use cases against their perceived value, and remove those use cases whose goal is too small compared to their development cost. Some use cases cost the project more than they will ever contribute to it. Consider removing these use cases and saving some money. The chances are good that you will never implement them anyway, as you will be maintaining and enhancing more important features.

Examples

Hospital Claim

Consider the following four use cases for paying medical insurance claims for a physician's services:

Use Case 1: Pay Hospitalization Claim

Use Case 2: Pay Outpatient Claim

Use Case 3: Pay House Call Claim.

Use Case 4: Pay Office Visit Claim

Each of these potential use cases describes separate and distinct situations, each having their own subtlety, expenses and forms. But doctors rarely, if ever, make house calls anymore, so Use Case 3 is impractical. It doesn't belong in the collection because it doesn't contribute any value to the system, even though it might contain interesting information that the other ones don't. It would make a good **Adornments(147)** though, if it contained useful information.

CleanHouse(216) addresses a real problem common to many professions beyond software development. For example, we identified several patterns while writing this book that we eventually decided not to use. Several of them were quite good, and we used them as the foundation for our early research. Unfortunately, they became less valuable as work progressed, or we found better ones, so we discarded them. It wasn't easy, and we had several spirited discussions about some of the patterns, but we felt that the book would be much better without them.

Copyright © 2001 Adolph, Bramble All Rights Reserved

8.4 Tradeoffs and Collaborations

Writing quality use cases can be a complicated process, made even more so by the volatile nature of the underlying requirements. Goals change, and what once represented quality use cases may now be incomplete fragments of behavior or large chunks that address too many issues. We may even have written some use cases that are no longer needed. In short, and often through no fault of our own, some of our use cases fail to meet the standard of a **CompleteSingleGoal(132)**.

The patterns in this chapter are helper patterns for **CompleteSingleGoal(132)**, and provide guidance for editing existing use cases to meet this standard (see Figure 8-1). Each relies on **CompleteSingleGoal(132)**, which states that a use case should completely address one and only one goal at its level, as the standard for determining what belongs in a specific use case. We can use this principle in two ways – determining whether something belongs in a particular use case, and if not, where it belongs. These patterns cover three situations: 1) Breaking up excessively large use cases and reducing them to a manageable size (**RedistributeTheWealth(206)**); 2) Combining incomplete use case fragments into new or existing use cases (**MergeDroplets(211)**); and 3) Eliminating unnecessary use cases (**CleanHouse(216)**). More important, they tell us how to accomplish these actions, and what to do with the left over pieces. Breaking up excessively large use cases involves identifying and removing information that does not pertain to the use case's goal. We don't necessarily want to get rid of this information; we usually want to redistribute it, either by merging it into a new use case, or combining it into an existing one. Combining fragments into cohesive units makes it easier for the readers to understand specific tasks without having to find their own path through the entire system to understand a specific function. Localizing closely related behavior in one use case also reduces the risk that some of the less informed readers might miss some detail that is buried in another, seemingly unrelated use case. Lastly, eliminating unnecessary use cases makes everyone's life easier, by reducing the size and number of use cases to maintain, and eliminating work for the implementers.

We may also decide that some fragments or even whole use cases no longer fit into the scheme of things, yet they contain some information we wish to preserve. We can always treat those cases as **Adornments(147)**, and preserve this information without directly including them in our use cases.

Copyright © 2001 Adolph, Bramble All Rights Reserved

Figure 8-1 Editing Patterns Relationships

One final note. Editing use cases can be rather involved, requiring that you understand both the system and most of the use cases in your collection before you start. This knowledge helps you identify the out-of-place pieces, as well as determine where they belong. Otherwise, you are likely to miss many of the subtleties contained in the system, and make things even worse. If you don't have this experience, take some time to acquaint yourself with all of the use cases you are editing, so that you may understand them better.

Copyright © 2001 Adolph, Bramble All Rights Reserved