Chapter 1

What Is a Quality Use Case?

The hardest single part of building a software system is deciding precisely what to build.

—Frederick Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering"

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-style "The system shall . . ." functional requirements. After reading these convoluted documents, many of us have often gone back to the customers 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 use cases?

- First, use cases give us a semiformal 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 enough just to tell a story. The story had to have a purpose or, in Jacobson's words, "yield a result of measurable value to an individual actor of the system" (Jacobson 1995, p. 105).

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 choose, how can you determine if the requirements are correct? How do you find redundant stories? How do you spot holes in the stories? Some structure is necessary; otherwise, people's creativity will work at cross-purposes.

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

- 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, describing such requirements 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 (1992) and Larman (2002) 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 remarked in *Writing Effective Use Cases* (Cockburn 2001, p. 205), "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 1.1 Use Case Horror: Example of a Poorly Written Use Case

Register for Course (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
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.

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 design choices. Those design choices are made later, after the use cases have been written and reviewed. The initial design choices are often changed during development, still satisfying the overall requirements. Use case experts

universally warn against including the user interface design inside use cases. Doing so is costly because it adds writing and reviewing time, and it makes the requirements document longer—and 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** (p. 133) and **TechnologyNeutral** (p. 167) 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 well what the system will contribute to the lives of the end consumers of the system. The **CompleteSingleGoal** (p. 118) pattern describes how properly to structure use cases to avoid this problem.
- *Using a use case for non-behavioral information.* Sometimes writers are 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 writers will produce immensely detailed use cases describing the completion of a user interface form, with each field in the form getting one or two lines of description. A much better approach is to create an **Adornment** 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, formulas, or state machines—or 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. 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.) The pattern **LeveledSteps** (p. 153) 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 guess 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** (p. 118) and **ExhaustiveAlternatives** (p. 129) 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 example of Use Case 1a. One could argue that such minor writing errors don't matter, but on all but the smallest projects 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 **ActorIntentAccomplished** (p. 158) 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 and poor quality use cases. Authors and 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 example in Use Case 1.1.

Use Case 1.2 Main Scenario for a Well-Written Use Case

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.

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 for creating high-quality use cases.

We want to capture guidelines that can help us write good use cases and 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 describing the characteristics of a good-quality use case. Put another way, these are characteristics that signify 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, and some apply to the set of extensions or to the use case itself. More patterns 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 their use cases. These patterns are best considered as a diagnostic tool, and should be of great use in 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 (1977, 1979) on *pattern languages*. His patterns capture just the information we need, and in a highly read-

able 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 recurring themes *patterns.*

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" (Alexander 1977, p. x). 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 five-hundred-page book to describe what a pattern is. The one-sentence version might be this: "A pattern is a three-part rule that expresses a certain relationship between a certain context, a problem, and a solution" (Alexander 1979, p. 247). 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* (Gamma et al. 1995). 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 magic bullet for all software development.

It is time to get patterns back to their appropriate place: as signs of quality, and as strategies. When used as a sign of quality, a pattern expresses what is present in a well-formed example (of whatever is being discussed). For instance, when lighting a room, Alexander's *Light on Two Sides of Every Room* states that people are instinctively drawn to rooms that have natural lighting on two or more walls (Alexander 1977). An example in this book, for use cases, is **VerbPhraseName** (p. 122), where 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 *Design Patterns* is *Visitor,* which describes how to traverse complex data structures without changing the class of the object being traversed (Gamma 1995). An example from software project management is Cockburn's

Gold Rush: When you don't have time to capture requirements completely and then design the program carefully, do them in parallel, carefully monitoring communication and rework issues (Cockburn 1998).

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 designs 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 a pattern is often part of a larger, more comprehensive language. In neither case do readers 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 a person's mind—which is quite fine. People who have read *Design Patterns* simply say "Visitor" and understand the trade-offs and issues the pattern entails. We hope the readers of this book will be able to simply say, **"CompleteSingleGoal"** (p. 118), **"ExhaustiveAlternatives"** (p. 129), or **"TwoTierReview"** (p. 64) 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** [p. 95]). 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. Here are some common misconceptions about patterns.

- *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; they do not 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—that is, try this solution when you have that problem (Coplien 1998).
- *Using patterns guarantees success.* In his book *Patterns of Software,* Richard Gabriel (1996) 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 nor 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" (Gabriel 1996, p. 59). In other words, in these cases, using his patterns made little if any visible difference. He felt much of the blame lay in the process. The people controlling the process—the lenders, the zoning commissioner, and others—were not using the pattern language, yet they wielded a lot of control over the project. Gabriel goes on to claim that these findings hold for software development: "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" (Gabriel 1996, p. 59).
- *Patterns offer new solutions to old problems.* As Linda Rising (1998, p. 10) says, "Patterns are not theoretical constructs created in an ivory tower, they are artifacts that have been discovered in more than one existing system." Patterns are essentially a documentation mechanism that captures general, tried-and-true solutions to common, recurring 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 (Coplien 1996). 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 sensitive company 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 that 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 rather than make their use cases 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.

So don't think of this pattern language as a complete methodology for writing use cases. Instead, treat it as a set of guidelines to help you fill in the gaps in your knowledge, evaluate the quality of your use cases, or augment your particular use case writing process. 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 the 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 presented in *A Pattern Language* (Alexander 1977). 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 Statement
- A Metaphoric Story
- The Forces Affecting the Problem
- The Solution
- Examples

Stepping through a Sample Pattern

The best illustration is the real thing. Consider the pattern **UserValuedTransactions** (p. 95) shown here as Figure 1.1.

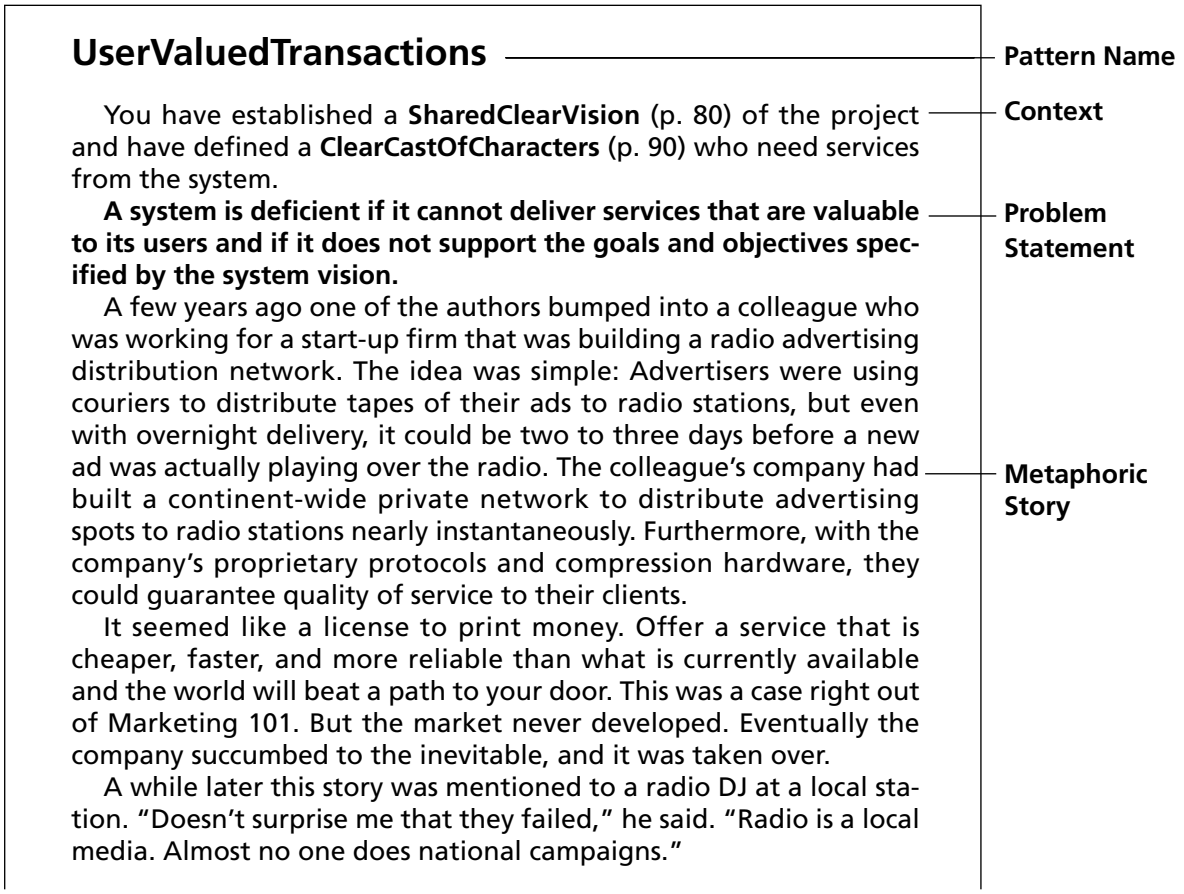

Figure 1.1 *A sample pattern (continued)*

Forces

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.

A set of use cases should capture the fundamental value-added services that users and stakeholders need from the system. An organization commissions the development of a system because that system will return some benefit. Use cases allow the organization's project team 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, but 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 "CRUD" (Create, Read, Update, and Delete). 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 does the user 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 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 will be constantly changing.

Readers want to see easily how the system will meet its goals (see ***SharedClearVision***). Just as a picture is worth a thousand words, a use case is worth a thousand pages of system specifications. 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.

(continued)

Figure 1.1 *A sample pattern (continued)*

People tend to work at a level that is either too high or too low. 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 describe system behavior adequately, without describing it in great detail ("what" versus "how"). If we write them at too high a level, then they will not be useful to the system developers, because they do not describe the system in enough detail. However, if use cases contain too much detail, then it is difficult for non-programmers to understand the system from their very high level. In the words of Ian Graham (1997), use cases should contain only necessary but essential information.

Therefore:

Identify the valuable services that the system delivers to the actors to satisfy their business purposes. — **Solution**

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** and **SharedClearVision** 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. Each service must help at least one actor reach a goal. Being unable to identify any service for an actor may indicate that the actor might not represent a valid system user; you may need to remove that actor 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 users 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 cares about.

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 one 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,

(continued)

Figure 1.1 A sample pattern (continued)

and then eliminate duplicates. A client booking an airline reservation is interested in getting a good flight at a good price. The client doesn'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** (p. 118). Describe this collection in sufficient detail to adequately convey its purpose, yet make the use case at a high enough level 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** (p. 102), but these use cases must work together to accomplish a common purpose.

Figure 1.1 *A sample pattern*

As this example shows, a pattern contains several sections, including the ones described next.

The Name

Each of our patterns begins with a name in bold text, usually 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** 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**.
>
> BOB: So what are some **UserValuedTransactions** for our users?
>
> JAN: How about *Apply for Mortgage,* and *File Supporting Documentation*?

The name of our example, **UserValuedTransactions**, 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** makes several references to **SharedClearVision** (p. 80), **ClearCastOfCharacters** (p. 90), **CompleteSingleGoal** (p. 118), and **EverUnfoldingStory** (p. 102) during the course of its discussion.

A Picture

Each pattern contains a picture that is intended to provide a visual metaphor for the pattern. While not shown in this example, the visual metaphor for **UserValued-Transactions** is a line of people waiting to place a bet with a bookie. Their willingness to stand in line to pay money illustrates that they believe this action to be worthwhile. The fact that other people may feel otherwise about this service does not detract from the image because any one person is likely to value only a portion of the services that a system offers. Illustrations provide a nice touch to the pattern as well as underscore the pattern's intent. (We obtained most of our photos from the Library of Congress American Memory Collection, because we feel that they provide a pleasant, unified visual theme.)

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 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 (Alexander 1977) and specifies which, if any, patterns are prerequisites to this one.

The context for **UserValuedTransactions** is:

> You have established a **SharedClearVision** (p. 80) of the project and have defined a **ClearCastOfCharacters** (p. 90) 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**, you must understand the project's **SharedClearVision** and know who will be using the system—that is, its **ClearCast-OfCharacters**. If you do not know these, then you cannot possibly determine which services are valuable.

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. The statement also reflects the risks associated with not following that standard.

The problem statement for **UserValuedTransactions** is expressed this way:

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

This problem statement 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 that 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 convey 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 did not follow this guideline.

The Metaphoric Story

Some patterns describe simple problems, while others 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. Although analogies are not normally part of the Alexandrian form, we believe they provide a good synopsis of the pattern's intent.

In the example in Figure 1.1, 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.

The Forces Affecting the Problem

This section outlines the various factors that affect the problem, and the trade-offs between them that complicate and constrain the solution. Pattern writers refer to

these trade-offs 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 trade-offs 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 trade-offs in more detail.

So what are the forces that we are trying to balance when finding **UserValuedTransactions** (p. 95)? Briefly, they are:

- *A set of use cases should capture the fundamental value-added services the users and stakeholders need from the system.*
- *It is relatively easy to identify low-level transactions, but 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 see easily how the system will meet its goals (see* **SharedClearVision***).*
- *People tend to work at a level that is either too high or too low.*

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 case). But this is where the forces come into play. These are the competing trade-offs that complicate the problem, which if taken too far can lead to suboptimal 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 to these lengths to explain the problem in this manner? Because we want you to grasp the richness of the problem and gain an in-depth understanding of the trade-offs 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 trade-offs, the better you can adapt the solution to your specific situation, or determine that the solution is not appropriate for your needs.

The Solution

This section of the pattern 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** pattern is this:

Therefore:

Identify the valuable services that the system delivers to the actors to satisfy their business purposes.

This solution aims to solve the problem within the constraints imposed by the forces. "Identify the valuable services . . ." limits the goal set to use cases that benefit the users instead of smaller, implementation-oriented use cases that are not particularly valuable to the user. ". . . that the system delivers to the actors to satisfy their business purposes" implies that we 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 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.

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 many of these examples on live projects, but 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 examples on our experiences and many of them are composites of real people, events, conversations, and use cases.

1.7 Organization of the Pattern Language

Our pattern language consists of thirty-one patterns, organized into two broad categories: development patterns and structural patterns. Development patterns describe the characteristics of proven use case writing practices, and offer criteria for measuring the quality of the writing process. Structural patterns describe the basic components of use cases, explain how they should be organized, and offer criteria for judging their use. These two broad categories are further broken down into sub-categories of related patterns.

There are three subcategories of development patterns:

- Team organization—patterns for judging and improving the quality of how the use case team is organized
- Process—patterns for judging and improving the quality of the methodology the team follows to create use cases
- Editing—patterns for judging and improving the quality of the individual use cases as the underlying requirements change and the writer's knowledge grows

There are four subcategories 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 use case scenarios, and the steps within those scenarios
- 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 remainder of the book addresses one subcategory.

Development 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. Although 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 development. The development patterns in our language offer 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) the techniques for creating a set of use cases, and (3) techniques for editing existing use cases into better ones.

The Team

Group dynamics is 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 of our book 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** (p. 138) use cases requires both a **BalancedTeam** (p. 39) (balanced skills and personalities) and a **ParticipatingAudience** (p. 35). The sponsors, developers, usage experts, and domain experts all contribute to the work and review it. However, too many writers soon spoil the plot, and so a **SmallWritingTeam** (p. 31) should be used for any one writing task.

The Process

Following a good process is critical to writing quality use cases. This process 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 describe your system adequately; you don't want to waste time writing any more than that. This section of the book investigates process issues and offers some advice for improving yours.

Although we do not advocate any specific process for creating use cases, we find that effective groups work **BreadthBeforeDepth** (p. 48), naming many use cases before partially expanding some, and completing the main success scenario before investigating failure handling, achieving a **SpiralDevelopment** (p. 52) of the use case set.

The **SmallWritingTeam** (p. 31) integrates its work with a **TwoTierReview** (p. 64), 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.

The effective team understands when it is **QuittingTime** (p. 68). Rather than getting bogged down in long arguments about cosmetic issues, team members allow a certain amount of **WritersLicense** (p. 73), 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** (p. 58) of use cases. Indeed, these forms may each find its appropriate moment on the same project!

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 undergo metamorphosis 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 for improving the quality of use cases.

During the writing process, team members will periodically find themselves with either large, complex, and hard-to-read use cases or lots of small, insignificant ones. They should **RedistributeTheWealth** (p. 204) of the large ones to smaller ones, and **MergeDroplets** (p. 209), folding the too-small ones into others. They may eventually discover that some are simply irrelevant; to deal with those, they can **CleanHouse** (p. 213).

Structural Patterns

We have identified four basic levels of use case structure: (1) sets of use cases, (2) use cases, (3) scenarios and steps, and (4) relationships. 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 that, 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 his or her (or its) 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.

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** (p. 80) for a system with a clear and **VisibleBoundary** (p. 86). The use cases are collectively structured with higher-level use cases referencing

lower-level use cases in an **EverUnfoldingStory** (p. 102) that shows a **ClearCastOfCharacters** (p. 90) interacting with the system to achieve their goals. While the goals that get described sit at various levels, the crucial and interesting ones describe **UserValuedTransactions** (p. 95), in which the primary actor accomplishes a goal that he views as a primary service of the system under discussion.

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 easily to identify and follow the different paths through the use case as they trace the actor's progress toward his goal. It also serves as 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 his 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** (p. 118), and should have a descriptive **VerbPhraseName** (p. 122) 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** (p. 125), with a collection of scenario fragments describing what happens under differing conditions. A complete use case considers **ExhaustiveAlternatives** (p. 129), so that the developers are not surprised with an unexpected situation late in development.

In order to satisfy the sponsor, the users, the developers, and the writers strive to make the use case **PreciseAndReadable** (p. 138), one of the arts of use case writing, and an achievable goal. 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** (p. 133). This practice keeps the use case robust with respect to shifting technologies and user interface designs, yet clean of unnecessary, confusing clutter.

Scenarios and Steps

Scenarios describe a single and complete sequence of events within a use case that an actor follows as she attempts 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** (p. 148) (detectable to the system under discussion). Part of the readability of attractive use cases is **LeveledSteps** (p. 151), keeping all the steps at about the same level of detail.

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** (p. 162) toward the goal. Since the user interface details and other design decisions appear as **Adornments** (p. 133), you can write each step in a **TechnologyNeutral** (p. 167) manner, to the greatest extent possible. Last, each step should make the **ActorIntentAccomplished** (p. 158), so that the readers always can tell who is doing what.

Relationships

Use cases occasionally share some common behavior, and when they do, it is efficient to reuse existing text rather than repeat 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 his or her 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 **CommonSub-Behavior** (p. 176) to a sub–use case referenced by the others via the *includes* relationship when a common set of actions recurs 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** (p. 182). If a given alternative begins to dominate the use case, then you should consider a **PromotedAlternative** (p. 190), promoting that alternative to an extension use case. While we have not seen enough examples of generalization to create a pattern, our colleague Dan Rawsthorne has contributed the pattern **CapturedAbstraction** (p. 198) which suggests when to use generalization.

1.8 Supplement: A Brief Tutorial on Writing Use Cases

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 discussed so far were just fragments, since they described only 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 Use Case 1.3.

Use Case 1.3 *Register for Courses:* Use Case with Extensions

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.

People sometimes find that a briefer way of writing is desirable (for example, for very small projects, and projects 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 (such as military contract outsourcing, large distributed projects, and 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 of use case writing.

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 subgoals. For example, I might have a goal to send my children to 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 subgoal is to win some particular contract. To do that, I may decide my next subgoal 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 subgoals, 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 an SuD, is to identify myself, to which end I have subgoals to insert my card into the machine, type in my identification number, and hit the Enter key. (People who know Alistair have already learned that he can make almost any goal involve 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, Insert 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, Insert 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 **UserValued-Transactions** (p. 95). This level we refer to as *user goal*. Then we write additional use cases for goals at higher and lower levels as needed. These we refer to as *summary* and *subfunction,* respectively. Figure 1.2, adapted from *Writing Effective Use Cases*

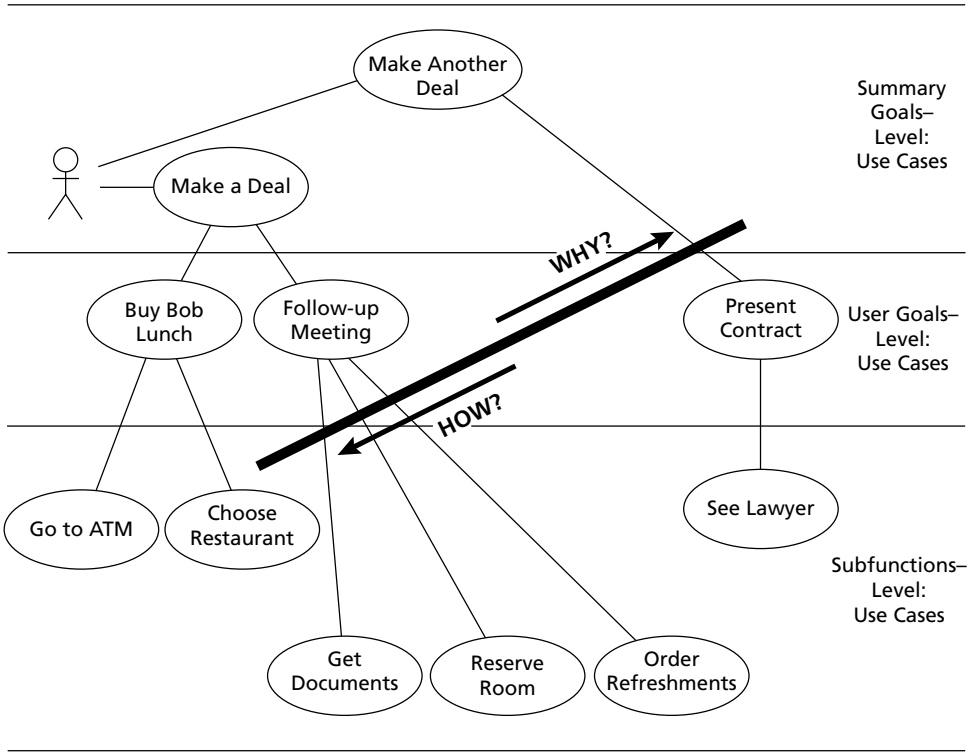

Figure 1.2 *Goal levels for sending your kids to a better school*

(Cockburn 2001), illustrates goal levels by describing some of the deals necessary to get the kids into a better school.

Use cases are read and used by two very different groups of people: (1) end users or business experts, who often are not versed in the technical and implementation difficulties; and (2) 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 create use cases that are correct and precise but still readable, the writing team *must* include:

- At least one person with a background in programming, to get the required accuracy and precision of description

- At least one person with deep knowledge of the business rules that the system must enforce
- 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 one group of people with the same job description. It is a team effort, requiring people with different backgrounds and even different personalities. When this team does its job well, the result is readable *and* precise.

This book is not an introduction to use cases. Rather, it is a handbook about how to write meaningful, high-quality use cases. Therefore, we have provided only a brief summary of use cases in this chapter. If you want to learn more about use cases in general, we recommend Alistair Cockburn's *Writing Effective Use Cases* (2001). 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.