Top Ten Use Case Mistakes

February 2001

"Use case driven" means writing the user manual first, then writing the code. This practice reinforces the fundamental notion that a system must conform to the needs of the users, instead of your users conforming to the system.

by Doug Rosenberg and Kendall Scott

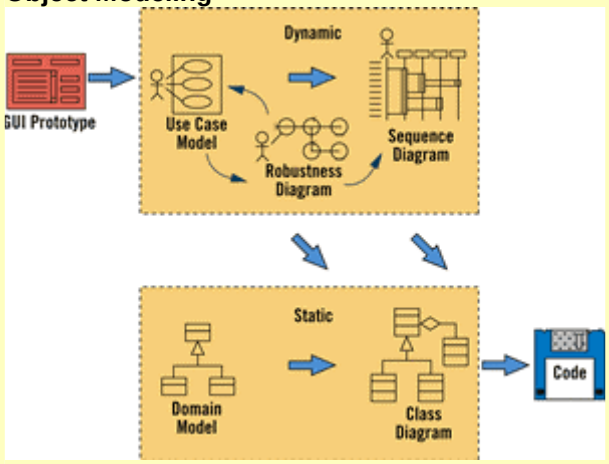
Read part 1: Driving Design with Use Cases
Read part 2: Driving Design: The Problem Domain

Welcome to the third in a series of five articles that provides a prepublication look at the annotated example from the forthcoming book, *Applied Use Case Driven Object Modeling* (Addison-Wesley, 2001; tentatively scheduled for April). We're following the process detailed in our first book, *Use Case Driven Object Modeling with UML* (Addison-Wesley, 1999), as we dissect the design of an Internet bookstore. In this article, we show common mistakes, and then explain how to correct them.

Within the ICONIX process, one of the early steps involves building a use case model. This model is used to capture the user requirements of a new system (whether it's being developed from scratch or based on an existing system) by detailing all the scenarios that users will perform. Use cases drive the dynamic model and, by extension, the entire development effort.

Figure 1 shows where use case modeling resides within the "big picture" of the ICONIX process.

Figure 1. The "Big Picture" for Use Case Driven Object Modeling



The diagram portrays the essence of a streamlined approach to software development that includes a minimal set of UML diagrams and some valuable techniques that take you from use cases to code quickly and efficiently.

The Key Elements

The task of building use cases for your new system is based on immediately identifying as many as you can, and then establishing a continuous loop of writing and refining the text that describes them. Along the way, you will discover new use cases, and also factor out commonality in usage.

You should keep one overriding principle in mind during your effort to identify use cases: They should have strong correlations with material found in the system's user manual. The connection between each use case and a distinct section of your user guide should be obvious. It reinforces the fundamental notion that you are designing a system that will conform to the viewpoints of the users. It also

provides a convenient summary of what "use case driven" means: Write the user manual, then write the code. If you're reengineering a legacy system, you can simply work backward from the user manual.

Once you have some text in place for a use case, it's time to refine it by making sure the sentences are clear and discrete, the basic format of your text is noun-verb-noun, and the actors and potential domain objects are easy to identify. You should also update your domain model—the subject of our previous article, "Driving Design: The Problem Domain" (Jan. 2001)—as you discover new objects and expand your understanding of the objects you'd previously found. And, it's important to determine all possible alternate courses of action for each use case wherever possible, an activity which should take up the majority of the time.

You can use several mechanisms to factor out common usage, such as error handling, from sets of use cases. This is usually effective, because breaking usage down to atomic levels will ease the analysis effort and save you lots of time when drawing sequence diagrams. Whether you use UML's generalization and *includes* and *extends* relationships, or OML's *invokes* and *precedes* relationships, which we recommend in our book, your goal should be a set of small, precise, reusable use cases.

You should feel comfortable proceeding to the next phases of the development process when you've achieved the following goals:

- You've built use cases that together account for all of the desired functionality of the system.
- You've produced clear and concise written descriptions of the basic course of action, along with appropriate alternative courses of action, for each use case.
- You've factored out scenarios common to more than one use case, using whichever constructs you're most comfortable with.

The Top 10 Use Case Modeling Errors

Contrary to the principles we just discussed are a number of common errors that we have seen students make when they're doing use case modeling on their projects for the first time. Our "top 10" list follows.

10. *Don't write functional requirements instead of usage scenario text.* Requirements are generally stated in terms of what the system shall do, while usage scenarios describe actions that the users take and the responses that the system generates. Eventually, our use case text will be used as a run-time behavioral specification for the scenario we'll describe, and this text will sit on the left margin of a sequence diagram. We want to be able to easily see *how* the system (shown with objects and messages) implements the *desired behavior*, as described in the use case text. So, we need to clearly distinguish between usage descriptions (behavior) and system requirements.

9. *Don't describe attributes and methods rather than usage.* Your use case text shouldn't include too many presentation details, but it should also be relatively free of details about the fields on your screens. Field names often match the names of attributes on your domain classes, which we discussed in January's article. Methods shouldn't be named or described in use case text because they represent how the system will do things, as opposed to what the system will do.

8. *Don't write the use cases too tersely.* When it comes to writing text for use cases, expansive is preferable. You need to address all of the details of user actions and system responses as you move into robustness analysis and interaction modeling, so you might as well put some of those details in your use cases. Remember also that your use cases will serve as the foundation for your user manual. It's better to err on the side of too much detail when it comes to user documentation.

7. *Don't divorce yourself completely from the user interface.* One of the fundamental notions of "use case driven" is that the development team conforms the design of the system to the viewpoints of the users. You can't do this without being specific as to what actions the users will perform on your screens. As we mentioned for item number nine, you don't need to talk about fields in your use case text, and you don't want to discuss the cosmetic appearance of your screens; however, you can let your prototypes, in whatever form they take, do that work for you. You do need to discuss those features of the user interface that allow the user to *tell the system to do something*.

6. *Don't avoid explicit names for your boundary objects.* Boundary objects are the objects with which actors will interact. These frequently include windows, screens, dialogs and menus. In keeping with our theme of including ample detail and being explicit about user navigation, we submit that it's necessary to name your boundary objects explicitly in your use case text. It's also important to do this because you will explore the behavior of these objects during robustness analysis (the subject of the next article in this series), and it can only reduce ambiguity and confusion to name them early.

5. *Don't write in the passive voice, using a perspective other than the user's.* A use case is most effectively written from the user's perspective as a set of present-tense verb phrases in active voice. The tendency of engineers to use passive voice is well-established, but use cases should state the actions that the user performs, and the system's responses to those actions. This kind of text is only effective when it's expressed in the active voice.

4. *Don't describe only user interactions; ignore system responses.* The narrative of a use case should be event- response oriented, as in, "The system does this when the user does that." The use case should capture a good deal of what happens "under the covers" in response to what the actor is doing, whether the system creates new objects, validates user input, generates error messages or whatever. Remember that your use case text describes both sides of the dialog between the user and the system.

3. *Don't omit text for alternative courses of action.* Basic courses of action are generally easier to identify and write text for. That doesn't mean, however, that you should put off dealing with alternative courses until, say, detailed design. Far from it. In fact, it's been our experience that when important alternative courses of action are not uncovered until coding and debugging, the programmer responsible for writing or fixing the code tends to treat them in ways that are most convenient for him. Needless to say, this isn't healthy for a project.

2. Don't focus on something other than what is "inside" a use case, such as how you get there or what happens afterward. Several prominent authors, such as Alistair Cockburn and Larry Constantine, advocate the use of long, complicated use case templates.

Spaces for preconditions and post-conditions are generally present on these templates. We like to think of this as the 1040 "long form" approach to use case modeling, in comparison to the 1040EZ-like template that we advocate (two headings: Basic Course and Alternate Course). You shouldn't insist on using long and complex use case templates just because they appeared in a book or article.

1. Don't spend a month deciding whether to use includes or extends. In our years of teaching use case driven development, we've yet to find a situation where we've needed more than one mechanism for factoring out commonality. Whether you use UML's include construct, or OML's invoke and precede mechanisms, or something else that you're comfortable with, doesn't matter; simply pick one way of doing things and stick with it. Having two similar constructs is worse than having only one. It's just too easy to get confused—and bogged down—when you try to use both. Don't spin your wheels.

Figure 2 shows use case text that contains violations of five of the top 10 rules.

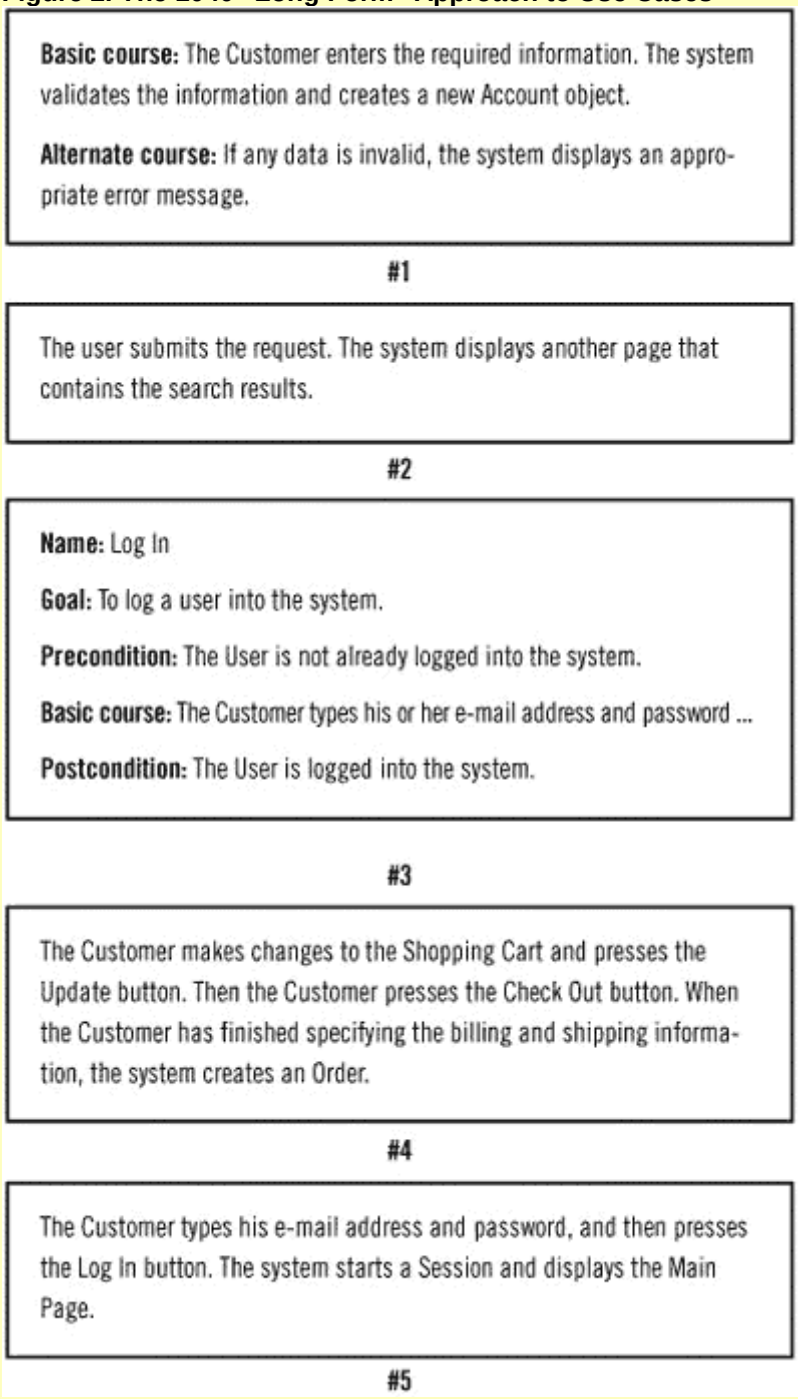
Did you spot the violations?

- Use case one is too terse. There is no reference to what kind of information the customer enters, nor to the page he or she is looking at. The text doesn't explain what is involved in validating the data that the customer entered. And the use case doesn't describe how the customer needs to respond to an error condition.
- Use case two doesn't have explicit names for the relevant boundary objects.
- Use case three reveals how useless it can be to obsess about using a complicated use case template. The name of the use case expresses the goal clearly enough; the content of the basic course will make the stated precondition and postcondition redundant.
- Use case four lacks alternate courses, even though it should be fairly clear from the context that some validation needs to occur, and that there are several possible error conditions (for instance, the system can't find the e-mail address, or the password that the customer entered doesn't match the one that is stored).
- Use case five doesn't specify how the system responds when the customer presses the update button.

Figure 3 shows the use case text with the mistakes corrected.

Our next article will demonstrate how to do robustness analysis in order to tighten up use cases and make it easier to head into detailed design. See you next month.

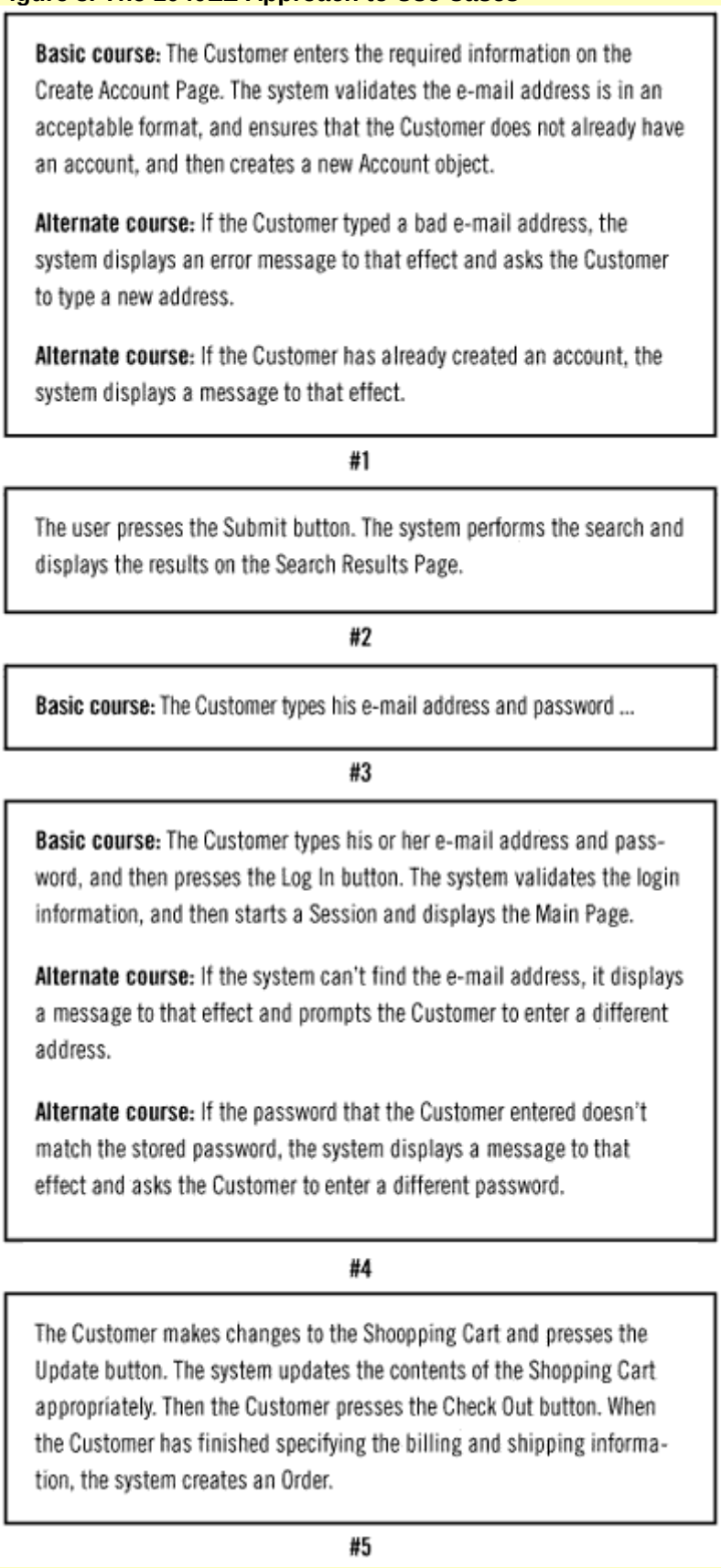
Figure 2. The 1040 "Long Form" Approach to Use Cases



Use case text that contains violations of five of the top 10 rules.

[back to text]

Figure 3. The 1040EZ Approach to Use Cases



The use case text with the mistakes corrected.

[back to text]