Fast Track UML 2.0

KENDALL SCOTT

Apress™

ISBN (pbk): 1-59059-320-0

Printed and bound in the United States of America 12345678910

Technical Reviewer: Doug Holland

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Kylie Johnston

Copy Editor: John Edwards

Production Manager: Kari Brooks

Proofreader: Katie Stence

Compositor: ContentWorks

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

CHAPTER 4

Use Cases

THIS CHAPTER DESCRIBES THE PRIMARY MEANS by which you can use the UML to capture functional requirements. You express these requirements in terms of the specific actions that external entities and the system perform in executing required and optional behavior.

Actors and Use Cases

An *actor* represents one of the following things:

- A role that a user can play with regard to a system
- An entity, such as another system or a database, that resides outside the system

The UML notation for an actor is a stick figure with a short descriptive name, as shown in Figure 4-1.

Figure 4-1. Actors (primary notation)

You can also show an actor using a stereotyped class or a user-defined icon (see Figure 4-2 for examples of both).

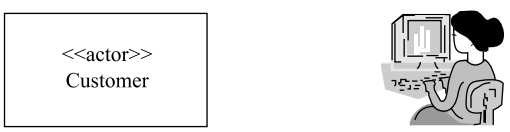

Figure 4-2. Actors (secondary notation)

NOTE The name of an actor should *not* be that of a particular person; instead, it should identify a role or set of roles that a human being, an external system, or a part of the system being built will play relative to one or more use cases. Note also that a single physical entity may be modeled by several different actors and, conversely, a given actor may be played by multiple physical entities.

A *use case* is a sequence of actions performed by an actor and the system that yields an observable result, or set of results, of value for one or more actors.

The standard notation for a use case is an ellipse combined with a short name that contains an active verb and (usually) a noun phrase (see Figure 4-3).

Figure 4-3. Use cases (primary notation)

The name of a use case can appear either within the ellipse or below it.

The text of a use case describes possible paths through the use case. Two kinds of flows of events are associated with use cases. These are as follows:

- The *main flow of events* (sometimes referred to as the *basic course of action*) is the sunny-day scenario, the main start-to-finish path that the actor and the system follow under normal circumstances. The assumption is that the primary actor doesn't make any mistakes, and the system generates no errors. A use case always has a main flow of events.

- An *exceptional flow of events* (or *alternate course of action*) is a path through a use case that represents an error condition or a path that the actor and the system take less frequently. A use case often has at least one exceptional flow of events.

Each unique execution of a use case represents a *use case instance.*

A use case can be associated with one or more *subjects.* For example, the subject may be a collaboration (see the section "Collaborations" in Chapter 1), and the use case would express the conditions that something interacting with that collaboration would need to meet in order to gain full access to the services that the collaboration offers. (Note that a subject can also be associated with more than one use case.) A subject may also "own" a use case, which is represented using the notation shown in Figure 4-4.

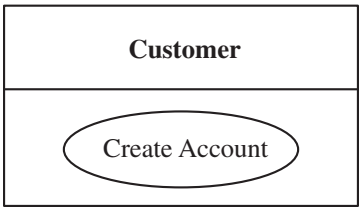

Figure 4-4. Subject owning use case

Actors and use cases appear on use case diagrams, which are described later in this chapter.

Qualities of a Good Use Case

The following guidelines have proven useful in producing tight, easy-to-understand use cases in a variety of contexts:

- **Use active voice, and speak from the actor's perspective.** For some reason, engineers and other technically inclined people tend to rely heavily on passive voice: "The connection is made," "The item is selected by the customer," and so forth. You should always express use cases in active voice. After all, you wouldn't expect to see a user manual written in passive voice, and there's a direct correlation between use cases and user manual text. (The most significant difference is that the latter is written in what's called the second-person imperative, with an unspoken "you," whereas use case text is written in the third person, in terms of specific actors and the system.)

- **Use present tense.** Requirements are usually written in the future tense: "The system shall do this and that," "The throughput of the system shall meet the following parameters." Each sentence of a use case should appear in the present tense: "The Customer selects the item," "The system makes the connection." This includes the text for alternate courses. (For example, "If the Customer selects a different item, the system comes to a grinding halt.") Keeping all the text in a consistent form makes it easier for its readers to trace the different paths through the basic course and alternate courses.

- **Express your text in the form of "call and response."** The basic form of your use case text should be "The [actor] does this" and "The system does that." The actor may do more than one thing consecutively, and the same holds true for the system, but the text should reflect the fact that the actor performs some action and the system responds accordingly. There shouldn't be any extraneous text.

- **Write your text in no more than three paragraphs.** One of the guiding principles of object-oriented design is that a class should do a small number of things well, and nothing else. Why not adhere to this principle with use cases as well? A use case should address one functional requirement, or perhaps a very small set of requirements, and do it in a way that's obvious to anyone who reads it. Anything more than a few paragraphs, and you probably have a candidate for another use case. (See the section "Organizing Use Cases," later in the chapter, for a discussion of how to break up and organize use cases.) A sentence or two, however, is a signal that you don't have enough substance in your use case. Each use case should be a small, mobile unit that lends itself to possible reuse in other contexts.

- **Name your classes.** There are two basic kinds of classes that lend themselves to inclusion in use case text: (a) those in the domain model (see the section "Domain-Level Class Diagrams" in Chapter 3) and (b) "boundary" classes, which include those windows, HTML pages, and so on that the actors use in interacting with the system. Down the line, it's going to be easier to design against text such as "The Customer changes one or more quantities on the Edit Contents of Shopping Cart page" and "The system creates an Account for the Customer" than against less-specific text (for example, "The Customer enters some values on an HTML page"). Be careful, though, to avoid including design details. You wouldn't talk about, say, the appearance of that HTML page or exactly what happens when the system creates an Account. The idea is to provide just enough detail for the designers to understand what's needed to address the basic requirements spelled out by the use cases.

- **Establish the initial context.** You have to specify where the actor is, and what he or she is looking at, at the beginning of the use case. There are two ways to do this. The first way involves specifying the context as part of the first sentence: "The Accountant enters his or her user ID and password on the System Login window," for example. The second way involves defining a precondition: "The Accountant has brought up the System Login window." Doing this also makes it easier for someone to piece together the larger picture across a set of use cases.

- **Make sure that each use case produces at least one result of value to one or more actors, even if that result is negative.** It's important to remember that a use case can't just end floating in space—something measurable has to happen. Of course, this is generally some positive result: "The actor is logged in to the system," "The system completes the actor's task by updating the database," "The system generates a report." A use case can end on an

alternate course of action, though, so "The system locks the user out of the system and sends an email to the system administrator" is also a viable result, even though it's hardly a desirable one.

- **Be exhaustive in finding alternate courses of action.** A lot of the interesting behavior associated with a system can be nicely captured within alternate courses—and it's a well-established principle that it's a lot cheaper to address this kind of behavior early in a project rather than later. A highly effective, if sometimes exhausting, way to root out alternate courses is to "challenge" every sentence of the basic course. In other words, ask repeatedly, "What can the actor do differently—or wrong—at this point?" or "What can go wrong internally at this point?" Remember two things while you're doing this. First, you don't need to account for generic failure conditions—network down, database inaccessible—within each use case; focus on those things that might happen in the specific context of the use case. Second, remember to take into account not only the novice/unsophisticated user but also the malicious user, the person who tries things he or she shouldn't just to see what might happen.

  There are a couple of good methods for pointing to alternate courses from within the basic course. One surefire way to signal the presence of an alternate course involves using words such as *validates*, *verifies*, and *ensures* within the basic course. Each time one of these words appears, there's at least one associated alternate course, by definition, to account for the system's inability to validate, verify, or ensure the specified condition. For example, the basic course might say, "The system verifies that the credit card number that the Customer entered matches one of the numbers it has recorded for that Customer," while the corresponding alternate course might read, "If the system cannot match the entered credit card number to any of its stored values, it displays an error message and prompts the Customer to enter a different number." Another way to indicate the presence of an alternate course involves using labels for the alternate courses and then embedding those labels in the basic course. For example, an alternate course might have a label A1, and that label would appear in parentheses after the relevant statement(s) in the basic course.

Example Use Cases

Let's look at a couple of sample use cases. We start by walking through how a use case named Log In, associated with an Internet bookstore, might have come into existence.

You first establish where the actor is, and what he or she does to initiate the use case. The steps may be as follows:

1. The Customer clicks the Login button on the Home Page.

2. The use case describes the system's response:

   ```
   The system displays the Login Page.
   ```

The rest of the basic course of action for this use case describes the actions that the actor performs in order to log in to the bookstore, and the actions that the system performs along the way as well:

3. The Customer enters his or her user ID and password and then clicks the OK button. The system validates the login information against the persistent Account data and then returns the Customer to the Home Page.

Remember that the Account class you defined in Chapter 1 (see Figure 1-3) contains three attributes, two of which are `ID` and `password`. As mentioned in the previous section, it's good practice to make explicit connections to domain model classes within use case text; you can certainly extend this principle to the attribute level as appropriate.

As you can see, the basic course addresses the sunny-day scenario: It assumes that nothing will go wrong. However, a good use case accounts for as many alternate paths and error conditions as possible. In this case, there are two kinds of alternate courses that we need to address. These are as follows:

- The Customer can perform some other action before logging in. This leads to the following alternate courses:

  If the Customer clicks the New Account button, the system displays the Create New Account page.

  If the Customer enters his or her user ID and then clicks the Reminder Word button, the system displays a dialog box containing the reminder word stored for that Customer. When the Customer clicks the OK button, the system returns the Customer to the Home Page.

- The system is unable to validate a value that the Customer provided and thus is unable to complete the login process. The following courses of action are possible:

If the Customer enters a user ID that the system does not recognize, the system displays the Create New Account page.

If the Customer enters an incorrect password, the system prompts the Customer to reenter his or her password.

If the Customer enters an incorrect password three times, the system displays a page telling the Customer that he or she should contact customer service.

Note how the reader of a use case can see where each alternate course branches out from the basic course.

Another key principle is that a use case has to end with the system providing some result of value to an actor. The basic course of Log In reflects that the Customer is logged in to the system at the end of the use case. The alternate courses also reflect that result, albeit in somewhat different ways. In two cases, the Customer ends up back at the Home Page, and the assumption is that he or she will proceed with the basic course of action. In another case, the system takes the Customer to a different page; this may not be a desirable option from his or her standpoint, but it's still a result of value. And in two cases, the system displays a new page that the Customer can use to create a new account. (Let's assume that some use case diagram for our bookstore will show that Log In has a connection with a Create New Account use case.)

Our second, more complicated use case is called Write Customer Review. It illustrates more basic principles discussed previously: active voice from the actor's perspective, present tense, a result of value, no more than three paragraphs, and alternate courses of action that reflect different paths.

Basic Course

The Customer clicks the Review This Book button on the Book Page. The system displays a page entitled Write a Review.

The Customer selects a rating for the given Book, types a title for his or her review, and then types the review itself. Then the Customer indicates whether the system should display his or her name or email address, or both, in connection with the review.

When the Customer has finished selecting and entering information, he or she clicks the Preview My Review button. The system displays a Look Over Your Review page that contains the information that the Customer provided. The Customer clicks the Save button. The system stores the information associated with the Review and returns the Customer to the Book Page.

Alternate Courses

If the Customer clicks the Review Guidelines button on the Book Page, the system displays the Customer Review Guidelines page. When the Customer clicks the OK button on that page, the system returns the Customer to the Book Page.

If the Customer clicks the Edit button on the Look Over My Review page, the system allows the Customer to make changes to any of the information that he or she provided on the Write a Review page. When the Customer clicks the Save button, the system stores the review information and returns the Customer to the Book Page.

Organizing Use Cases

The UML offers three constructs for factoring out common behavior and variant paths for use cases. The following subsections describe these constructs.

Include

Within an *include* relationship, one use case *explicitly* includes the behavior of another use case at a specified point within a course of action.

The included use case doesn't stand alone; it has to be connected with one or more base use cases. The include mechanism is very useful for factoring out behavior that would otherwise appear within multiple use cases.

Within Figure 4-5, the Add to Wish List and Check Out use cases include the behavior captured within the Log In use case, because a Customer must be logged in before he or she can add a book to a wish list or make a purchase.

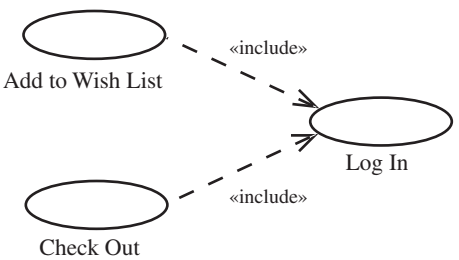

Figure 4-5. Include

Extend

Within an *extend* relationship, a base use case *implicitly* includes the behavior of another use case at one or more specified points. These points are called *extension points*.

You generally use this construct to factor out behavior that's optional or that occurs only under certain conditions. One way to use extends is in creating a new use case in response to an alternate course of action having several steps associated with it.

Figure 4-6 shows that a Customer has the option of canceling an Order in conjunction with checking the status of that Order.

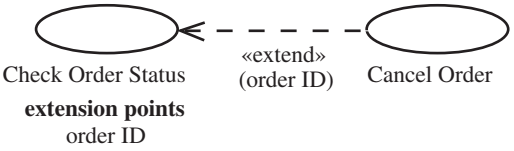

Figure 4-6. Extend

If it's desirable to show an explanation of when a use case extension comes into play, you can show this in a note, as illustrated in Figure 4-7.

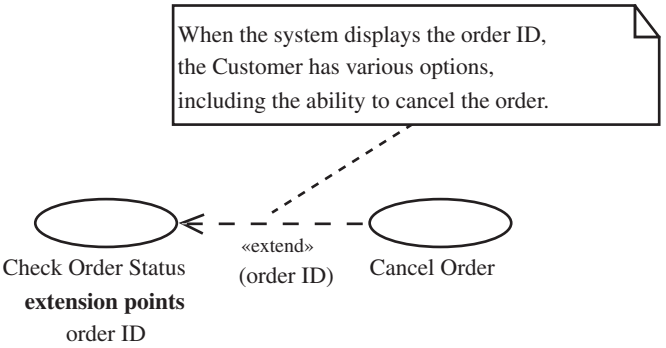

Figure 4-7. Explaining a use case extension

NOTE You have the option of using standard class notation to represent a use case, with the use case ellipse in the name compartment, and to add one or more compartments to the class box to hold relevant information—for example, multiple extension points. See Figure 4-8.

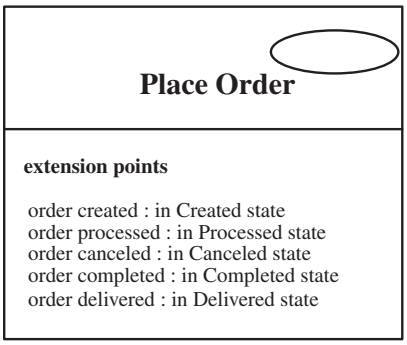

Figure 4-8. Use cases (secondary notation)

Use Case Generalization

Generalization works the same way for use cases as it does for classes (see the section "Generalization" in Chapter 2): A parent use case defines behavior that its children can inherit, and the children can add to or override that behavior.

Figure 4-9 shows use cases that describe three different searches that a Customer can perform, all of which use the basic search technique defined by the Perform Search use case.

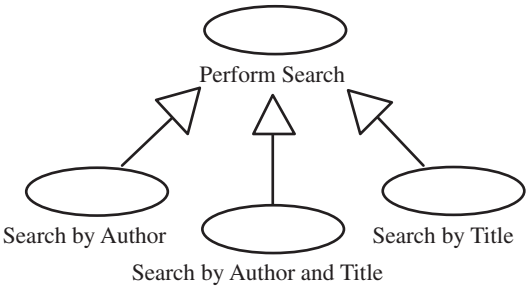

Figure 4-9. Use case generalization

NOTE You can also use generalization with actors. For example, there might be a general Accounting Personnel actor, which has certain privileges, and then Accountant and Accounting Clerk actors that represent more specialized usage privileges than those actors have.

Use Case Diagrams

A *use case diagram* shows a set of use cases and actors and the relationships among them. Figure 4-10 shows an example of a use case diagram.

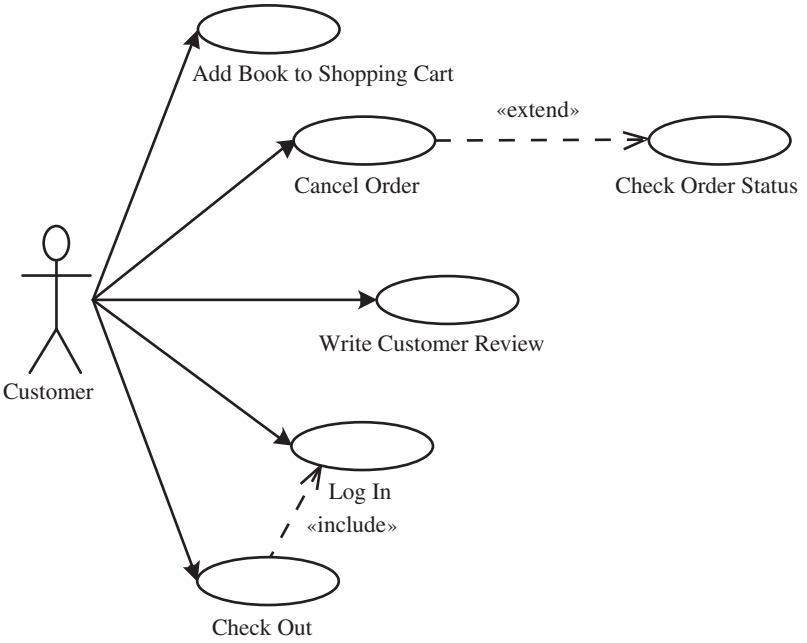

Figure 4-10. Use case diagram

It's a good idea to draw one use case diagram per actor. You can also use different layers of use case diagrams to good effect. For example, you might have an "executive summary" layer that shows only the use cases of most general interest, and then subsequent layers would show more specialized use cases.

Looking Ahead

In the next chapter, you take a look at a UML construct that you can use to help you organize conceptually related elements of your models.