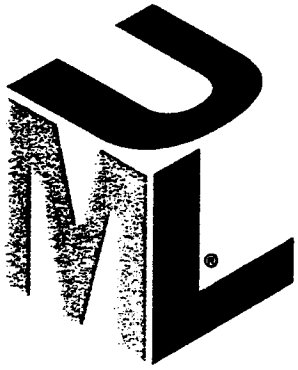
Second Edition covers through version 1.3 OMG UML Standard

UML DISTILLED SECOND EDITION

A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE

MARTIN FOWLER
WITH KENDALL SCOTT

Foreword by Grady Booch, Ivar Jacobson, and James Rumbaugh



Chapter 3

Use Cases

Use cases are interesting phenomena. For a long time, in both object-oriented and traditional development, people used typical interactions to help them understand requirements. However, these scenarios were treated very informally—always done but rarely documented. Ivar Jacobson is well known for changing this with his Objectory method and associated book (his first one).

Jacobson raised the visibility of the use case to the extent that it became a primary element in project development and planning. Since his book was published (1992), the object community has adopted use cases to a remarkable degree. My practice has certainly improved since I started using use cases in this manner.

So what is a use case?

I won't answer this question head-on. Instead, I'll sneak up on it from behind by first describing a scenario.

A **scenario** is a sequence of steps describing an interaction between a user and a system. So if we have a Web-based on-line store, we might have a Buy a Product scenario that would say this:

> *The customer browses the catalog and adds desired items to the shopping basket. When the customer wishes to pay, the customer describes the shipping and credit card information and confirms the sale. The system checks the authorization on*

the credit card and confirms the sale both immediately and with a follow-up email.

This scenario is one thing that can happen. However, the credit card authorization might fail. This would be a separate scenario.

A **use case**, then, is a set of scenarios tied together by a common user goal. In the current situation, you would have a Buy a Product use case with the successful purchase and the authorization failure as two of the use case's scenarios. Other, alternative paths for the use case would be further scenarios. Often, you find that a use case has a common all-goes-well case, and many alternatives that may include things going wrong and also alternative ways that things go well.

A simple format for capturing a use case involves describing its primary scenario as a sequence of numbered steps and the alternatives as variations on that sequence, as shown in Figure 3-1.

Buy a Product

1. Customer browses through catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

Alternative: *Authorization Failure*
At step 6, system fails to authorize credit purchase
Allow customer to re-enter credit card information and re-try

Alternative: *Regular Customer*
3a. System displays current shipping information, pricing information, and last four digits of credit card information
3b. Customer may accept or override these defaults
Return to primary scenario at step 6

Figure 3-1: *Example Use Case Text.*

There is a lot of variation as far as how you might describe the contents of a use case; the UML does not specify any standard. There are also additional sections you can add. For example, you can add a line for preconditions, which are things that should be true when the use case can start. Take a look at the various books that address use cases, and add the elements that make sense for you. Don't include everything, just those things that really seem to help.

An important example of this is how you divide up use cases. Consider another scenario for on-line purchase, one in which the purchaser is already known to the system as a regular customer. Some people would consider this to be a third scenario, whereas others would make this a separate use case. You can also use one of the use case relationships, which I'll describe later.

The amount of detail you need depends on the risk in the use case: The more risk, the more detail you need. Often I find that I go into details only for a few use cases during elaboration, and the rest contain no more than the use case in Figure 3-1. During iteration, you add more detail as you need it to implement the use case. You don't have to write all of the detail down; verbal communication is often very effective.

Use Case Diagrams

In addition to introducing use cases as primary elements in software development, Jacobson (1994) also introduced a diagram for visualizing use cases. The **use case diagram** is also now part of the UML.

Many people find this kind of diagram useful. However, I must stress that you don't need to draw a diagram to use use cases. One of the most effective projects I know that used use cases involved keeping each one on an index card and sorting the cards into piles to show what needed building in each iteration.

Figure 3-2 shows some of the use cases for a financial trading system.

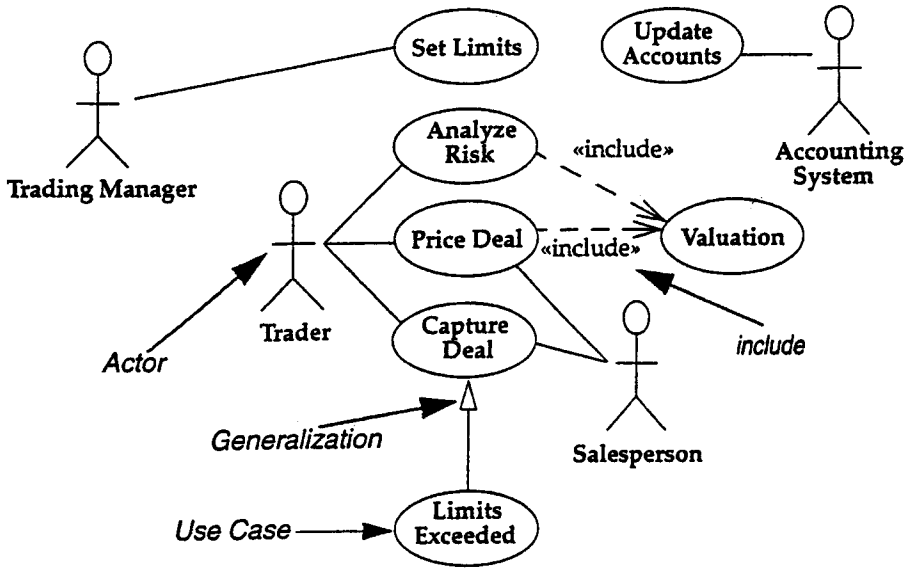


Figure 3-2: *Use Case Diagram*

Actors

An **actor** is a role that a user plays with respect to the system. There are four actors in Figure 3-2: Trading Manager, Trader, Salesperson, and Accounting System. (Yes, I know it would be better to use the word "role," but apparently, there was a mistranslation from the Swedish.)

There will probably be many traders in the given organization, but as far as the system is concerned, they all play the same role. A user may also play more than one role. For instance, one senior trader may play the Trading Manager role and also be a regular trader; a Trader may also be a Salesperson. When dealing with actors, it is important to think about roles rather than people or job titles.

Actors carry out use cases. A single actor may perform many use cases; conversely, a use case may have several actors performing it.

In practice, I find that actors are most useful when trying to come up with the use cases. Faced with a big system, it can often be difficult to come up with a list of use cases. It is easier in those situations to arrive

at the list of actors first, and then try to work out the use cases for each actor.

Actors don't need to be human, even though actors are represented as stick figures within a use case diagram. An actor can also be an external system that needs some information from the current system. In Figure 3-2, we can see the need to update the accounts for the Accounting System.

There are several variations on what people show as actors. Some people show every external system or human actor on the use case diagram; others prefer to show the initiator of the use case. I prefer to show the actor that gets value from the use case, which some people refer to as the primary actor.

However, I don't take this too far. I'm happy to see the accounting system get value, without trying to figure out the human actor that gets value from the accounting system—that would entail modeling the accounting system itself. That said, you should always question use cases with system actors, find out what the real user goals are, and consider alternative ways of meeting those goals.

When I'm working with actors and use cases, I don't worry too much about what the exact relationships are among them. Most of the time, what I'm really after is the use cases; the actors are just a way to get there. As long as I get all the use cases, I'm not worried about the details of the actors.

There are some situations in which it can be worth tracking the actors later.

- The system may need configuring for various kinds of users. In this case, each kind of user is an actor, and the use cases show you what each actor needs to do.
- Tracking who wants use cases can help you negotiate priorities among various actors.

Some use cases don't have clear links to specific actors. Consider a utility company. Clearly, one of its use cases is Send Out Bill. It's not so easy to identify an associated actor, however. No particular user role requests a bill. The bill is sent to the customer, but the customer wouldn't object if it didn't happen. The best guess at an actor here is

the Billing Department, in that it gets value from the use case. But Billing is not usually involved in playing out the use case.

Be aware that some use cases will not pop out as a result of the process of thinking about the use cases for each actor. If that happens, don't worry too much. The important thing is understanding the use cases and the user goals they satisfy.

A good source for identifying use cases is external events. Think about all the events from the outside world to which you want to react. A given event may cause a system reaction that does not involve users, or it may cause a reaction primarily from the users. Identifying the events that you need to react to will help you identify the use cases.

Use Case Relationships

In addition to the links among actors and use cases, you can show several kinds of relationships between use cases.

The **include** relationship occurs when you have a chunk of behavior that is similar across more than one use case and you don't want to keep copying the description of that behavior. For instance, both Analyze Risk and Price Deal require you to value the deal. Describing deal valuation involves a fair chunk of writing, and I hate copy-and-paste. So I spun off a separate Value Deal use case for this situation and referred to it from the original use cases.

You use **use case generalization** when you have one use case that is similar to another use case but does a bit more. In effect, this gives us another way of capturing alternative scenarios.

In our example, the basic use case is Capture Deal. This is the case in which all goes smoothly. Things can upset the smooth capture of a deal, however. One is when a limit is exceeded—for instance, the maximum amount the trading organization has established for a particular customer. Here we don't perform the usual behavior associated with the given use case; we carry out an alternative.

We could put this variation within the Capture Deal use case as an alternative, as with the Buy a Product use case I described earlier. However, we may feel that this alternative is sufficiently different to deserve a separate use case. We put the alternative path in a special-

ized use case that refers to the base use case. The specialized use case can override any part of the base use case, although it should still be about satisfying the same essential user goal.

A third relationship, which I haven't shown on Figure 3-2, is called **extend**. Essentially, this is similar to generalization but with more rules to it.

With this construct, the extending use case may add behavior to the base use case, but this time the base use case must declare certain "extension points," and the extending use case may add additional behavior only at those extension points. (See Figure 3-3.)

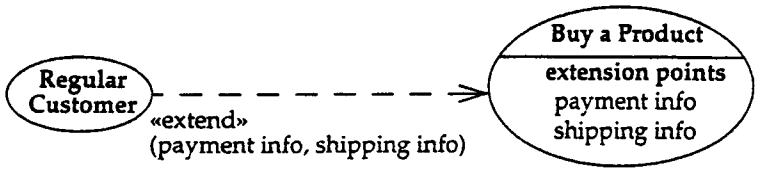


Figure 3-3: *Extend Relationship*

A use case may have many extension points, and an extending use case may extend one or more of these extension points. You indicate which ones on the line between the use cases on the diagram.

Both generalization and extend allow you to split up a use case. During elaboration, I often split any use case that's getting too complicated. I split during the construction stage of the project if I find that I can't build the whole use case in one iteration. When I split, I like to do the normal case first and the variations later.

Apply the following rules.

- Use *include* when you are repeating yourself in two or more separate use cases and you want to avoid repetition.
- Use *generalization* when you are describing a variation on normal behavior and you wish to describe it casually.
- Use *extend* when you are describing a variation on normal behavior and you wish to use the more controlled form, declaring your extension points in your base use case.

Business and System Use Cases

A common problem that can happen with use cases is that by focusing on the interaction between a user and the system, you can neglect situations in which a change to a business process may be the best way to deal with the problem.

Often you hear people talk about system use cases and business use cases. The terms are not precise, but the general usage is that a system use case is an interaction with the software, whereas a business use case discusses how a business responds to a customer or an event.

I don't like to get too bogged down in this issue. In the early stages of elaboration, I lean more toward business use cases, but I find system use cases more useful for planning. I find it useful to think about business use cases, particularly to consider other ways to meet an actor's goal.

In my work, I focus on business use cases first, and then I come up with system use cases to satisfy them. By the end of the elaboration period, I expect to have at least one set of system use cases for each business use case I have identified—at minimum, for the business use cases I intend to support in the first delivery.

When to Use Use Cases

I can't imagine a situation now in which I would not use use cases. They are an essential tool in requirements capture and in planning and controlling an iterative project. Capturing use cases is one of the primary tasks of the elaboration phase.

Most of your use cases will be generated during that phase of the project, but you will uncover more as you proceed. Keep an eye out for them at all times. Every use case is a potential requirement, and until you have captured a requirement, you cannot plan to deal with it.

Some people list and discuss the use cases first, then do some modeling. I've also found that conceptual modeling with users helps

uncover use cases. So I tend to do use cases and conceptual modeling at the same time.

It is important to remember that use cases represent an *external* view of the system. As such, don't expect any correlations between use cases and the classes inside the system.

How many use cases should you have? During a recent OOPSLA panel discussion, several use case experts said that for a 10-person-year project, they would expect around a dozen use cases. These are base use cases; each use case would have many scenarios and many variant use cases. I've also seen projects of similar size with more than a hundred separate use cases. (If you count the variant use cases for a dozen use cases, the numbers end up about the same.) As ever, use what works for you.

Where to Find Out More

A good short book on use cases is Schneider and Winters (1998). Its current printing uses the UML 1.1 relationships—uses and extends—but it remains the best book I've seen on how to apply use cases. My other favorite recommendation is the set of papers at Alistair Cockburn's Web site: **<http://members.aol.com/acockburn>**.

Ivar Jacobson's first book (1994) is the book that started the ball rolling, but it's rather dated now. Jacobson's follow-up book (1995) is still useful for its accent on business use cases.