Use cases are a practical way of specifying the behavior of a system from a user’s perspective. As such, they can provide a powerful source for generating test cases. This article describes how to derive these test cases and provides some typical examples.

The Nature of Use Cases

A use case is a sequence of actions performed by a system, which combined together produce a result of value to a system user. While uses cases are often associated with object-oriented systems, they apply equally well to most other types of systems.

Each use case has pre-conditions which need to be met for the use case to work successfully. For example, a withdrawal cannot be made without an open bank account and without a working automated teller machine (ATM).

Each use case terminates with post-conditions, which are the observable results and final state of the system after the use case has been completed. For example, after a withdrawal the account balance is expected to be debited by the withdrawn amount, and the ATM should be available and ready for the next transaction.

A use case usually has a mainstream, most likely scenario, such as the successful withdrawal of funds from an ATM.

The use case may also contain alternative branches, such as the disapproval of a withdrawal because (A) the user’s bank card is unrecognized, (B) there is insufficient cash in the ATM, or (C) there are insufficient funds in the bank account. Use cases may also have shared or common pieces, such as a thread of steps re-used across several use cases.

Use cases and test cases work well together in two ways: If the use cases for a system are complete, accurate, and clear, the process of deriving the test cases is straightforward. And if the use cases are not in good shape, the attempt to derive test cases will help to debug the use cases.

Use Cases and Testing

Traditional test case design techniques include analyzing the functional specifications, the software paths, and the boundary values. These techniques are all valid, but use case testing offers a new perspective and identifies test cases which the other techniques have difficulty seeing.

Use cases describe the “process flows” through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system (that moment you realize “we can’t get there from here!”). They also help uncover integration bugs, caused by the interaction and interference of different features, which individual feature testing would not see. The use case method supplements (but does not supplant) the traditional test case design techniques. In fact, in the following example you will see some merging with the boundary value technique, which is an approach used to identify both valid and invalid variations of the input data driving the use case.
Use cases and test cases work well together in two ways:
If the use cases for a system are complete, accurate, and clear, the process
of deriving the test cases is straightforward.
And if the use cases are not in good shape, the attempt to
derive test cases will help to debug the use cases.

A Use Case Example

Use Case Name: Select Product

Use Case Description
This use case helps to build a purchase order (PO) by adding a new line
item and selecting the product to be ordered. A PO can contain several
line items; each line item on the PO orders a quantity of one specific prod-
uct, from the vendor to whom the PO is addressed. (See Figure 1)

Pre-Conditions
The following pre-conditions need to be fulfilled for the use case to work
as expected: An open PO must already be in existence, with its status set
to the value “in progress,” and this PO must have a valid Vendor ID num-
ber and a unique PO number. The user must have entered a valid user ID#
and password, and be authorized to access the PO and to add or modify a
line item.

The number of line items on the existing PO must not equal twenty-
five. (For this sample, the organization’s business policy limits the number
of lines per PO to twenty-five or less.) The product number must be
unique on each existing line item (another business policy— no duplicate
products within one PO).

The system must be able to access the correct product list for the
vendor.

Post-Conditions
After the use case has executed, the expected final state of the system is
as follows: The same PO must still exist and still be open. A new line has
been added to the PO, with the next sequential line item number as-
signed; and the correct product has been selected for this line item (un-
less the use case was aborted).

The system must be left in the correct state to be able to move on to the
next steps after the product selection. (These next steps are to enter a
quantity to be ordered on the same line item, check if sufficient inventory is
on hand for the order, etc. They are outside the scope of this test proj-
ector, except to make sure we have continuity: that “we can get there
from here.”)

The product number cannot duplicate an existing
product number on another line item on the same PO.
The product selected or entered must be a valid one
for this vendor.

Alternatively, the original PO must remain un-
changed from its original state if a product was not se-
lected (for example, because the user aborted the
process).

Error messages are issued, if appropriate, and after
the use case has been exercised the system is waiting in
a state ready to accept the next input command.

Use Case Flow of Events
For brevity, the following use case is not complete. Some
alternatives have been left out. The user actions are la-
beld with an identifier, which begins with “U,” and the
system actions have identifiers beginning with “S.” At
the end of each action below, the identifiers in parentheses
indicate the possible next actions in the use case. For ex-
ample, in response to user action U1, the system can take
either action S1.1 or S1.2. (See Figure 2A, page 35)
<table>
<thead>
<tr>
<th>The user</th>
<th>The system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U1</strong></td>
<td>Requests that a new line item be added to the PO. [S1.1, S1.2]</td>
</tr>
<tr>
<td><strong>S1.1</strong></td>
<td>Adds a new line item, and assigns the next sequential line item number within the PO. [U2.1, U2.2]</td>
</tr>
<tr>
<td>or <strong>S1.2</strong></td>
<td>Alternatively, states that a new line cannot be added. (The PO already contains twenty-five line items.) Exit from use case.</td>
</tr>
<tr>
<td><strong>U2.1</strong></td>
<td>Requests a list of products supplied by the vendor to whom the PO is addressed. [S2]</td>
</tr>
<tr>
<td><strong>S2</strong></td>
<td>Provides the product list for this vendor. [U3.1, U3.2]</td>
</tr>
<tr>
<td>or <strong>U2.2</strong></td>
<td>Alternatively, enters the specific product number directly into the line item. [S3.1, S3.2]</td>
</tr>
<tr>
<td><strong>S3.1</strong></td>
<td>Verifies that the product number selected or entered is a valid one. Displays the description of the product for visual verification by the user. [U4]</td>
</tr>
<tr>
<td>or <strong>S3.2</strong></td>
<td>Alternatively, rejects the entered product number. [U2.1, U2.2]</td>
</tr>
<tr>
<td><strong>U3.1</strong></td>
<td>Selects the desired product from the vendor’s list. [S3.1, S3.2]</td>
</tr>
<tr>
<td>or <strong>U3.2</strong></td>
<td>Alternatively, decides there is no suitable product available from this vendor to meet the need. The user aborts the process. Exit from use case.</td>
</tr>
<tr>
<td><strong>U4</strong></td>
<td>Confirms that the right product has been selected or entered, by comparing the displayed response to what the user had intended to order. [S4.1, S4.2]</td>
</tr>
<tr>
<td><strong>S4.1</strong></td>
<td>Verifies that the product number (either selected via the list or entered directly) does not duplicate any of the product numbers on the other line items for the same PO. Exit from use case.</td>
</tr>
<tr>
<td>or <strong>S4.2</strong></td>
<td>Alternatively, if the product number is a duplicate, rejects it. [U2.1, U2.2]</td>
</tr>
</tbody>
</table>

### How to Derive the Test Cases

Normally, an important precursor to deriving test cases is to first carefully review the use case for correctness and completeness. The use case developers need to address any ambiguities, inconsistencies, and omissions that we might find. I won’t cover that review in this article, except to show how the process of creating tests can uncover further problems.

Now that we have the use case, how do we analyze it to generate test cases? The process is relatively straightforward. **Remember, simple engineering is great engineering.**

Our test cases will traverse paths through the use case. Test design consists of picking those paths. The technique is similar to that used in white-box testing, where we work with the flow chart of the software. See Boris Beizer’s *Software Testing Techniques* for such white-box test design techniques.

Where do we start? First, look for the mainstream path through the use case, the one most likely to be used. In this example, it is the process of adding a new line item to a PO and selecting the right product from the list. We need to make sure this scenario works correctly: in other words, we need to write a test case for it. The test case, in all its glory, should look something like this:

### A Test Case Example

**Test Case Name:** Select Product — Normal Mainstream Process  
**Use Case Name:** Select Product  
**Use Case Path to Be Executed:** [ U1 -> S1.1 -> U2.1 -> S2 -> U3.1 -> S3.1 -> U4 -> S4.1 ]  
(See Figure 2B, next page)  
**Input Data:** Product ID# 554875 (Bahamian Sardines, 6 ounce size).  
**Initial Conditions:** PO # 6135527 is already open and displayed for Vendor ID# 42296. User ID# 443 is authorized to work on this PO. Seven line items already are attached to this PO.  
**Test Steps:**  
Ensure that the initial conditions have been met before proceeding.  
Follow the steps through the use case as listed above.  
Compare the actual result to the expected result below.  
Write the test case ID#, date and time, and actual results in the test log.  
**Expected Results:** PO # 6135527 is still open and displayed for Vendor ID# 42296. User ID# 443 is still authorized to work on this PO. Eight line items are now attached to this PO. New line item has been established for Bahamian Sardines, 6 ounce size. New line item has been assigned line item number 8. System is ready to proceed to the next activity (which is setting the quantity of Bahamian sardines to be ordered).
After we have the first test case down, what do we do next? We need to look for the major alternative paths that the user could take through this use case. Since in the real world the number of alternatives is usually overwhelming, we need to select only the most fruitful ones, based on likely frequency of use, risk, and complexity. We need to generate test cases for each of these paths too.

A Negative Test Case Example
We also need to think about what could go wrong. Invalid input data is a good source of negative test cases. In addition, what if the pre-conditions of the use case have not been met? The different ways in which the pre-conditions could be violated are a fruitful source of negative test cases. There are actually two possibilities for negative testing:

1. Creating test cases to exercise legitimate paths through the use case which include invalid user actions. This is no different than exercising a path that contains any other kind of action. That’s what I’ve done in the sample test case below, where a user-entered product number is rejected (step S3.2 in the use case).

2. Creating test cases by trying inputs not listed in the use case. That includes deliberately violating the use case pre-conditions, as well as unexpected erroneous input at any step. For example, what if the PO status is not “in progress” but instead has another status such as “closed”? Can we still add a new line item to this PO?

To create a negative test case, we might decide to exercise step S3.2, where the system rejects an invalid product number and prompts the user for another one. At that point, the user might decide to abort and exit until finding out what the correct number is.

But there’s a problem: there’s no way to exit directly after step S3.2. You must first enter a correct product number or request a list of products. That’s needlessly annoying. The use case is in error. Perhaps the system works correctly despite the use case. Or perhaps both the use case and the system are wrong. Whichever is true, one possible fix is to amend step S3.2 of the use case as follows:

S3.2 Alternatively, rejects the entered product number. [U2.1, U2.2, U3.2]

The process of analyzing a use case always seems to uncover flaws in the use case itself. (This provides the important side benefit of allowing the test professional to feel virtuous... and even a little smug.) These use case flaws need to

Our test cases will traverse paths through the use case.
Test design consists of picking those paths.
The technique is similar to that used in white-box testing, where we work with the flow chart of the software.
be resolved as soon as they are found, since the resolution may change the use case and thus change the test cases.

Here’s a negative test case for the use case, which will work even if the flaw is not fixed:

Note that this test case does not exercise a complete path through the use case. There still needs to be an exit strategy after step S3.2, such as [U2.1 -> S2 -> U3.2].

### Risk Prioritization

If you fully analyze this short use case, you can generate dozens of test cases.

The number of test cases we could build is usually overwhelming: unless the use case is life-critical (which is not the situation here), we typically do not have the time to exercise every single test case. In fact, some test cases are just not worth it. If we weigh the cost of building and exercising the test case against the probability of finding a defect—together with the importance of that defect—we will choose not to use the test case.

So how do we prioritize? We need to weight the test cases, based on: (a) the criticality of the condition being tested (how much trouble could we get into), (b) the likely frequency of use of the path or condition (the more it is used in live operation, the more opportunities it has to break), (c) prior defect patterns...
The number of test cases we could build is usually overwhelming: unless the use case is life-critical, we typically do not have the time to exercise every single test case. In fact, some test cases are just not worth it.

What happens if we try to add a line to a PO, but the database is already physically full?
What happens if two users, working at different workstations, simultaneously try to add a line item to the same PO?
So how do we find these other test cases? Conducting a peer review of the test case design would help. We should also balance use case analysis with other techniques; other test case design techniques which can be helpful here include cause-effect graphs, state-transition diagrams, statistical sampling, and orthogonal arrays (but that’s the basis for another story). And, above all, we should remember to apply our good judgement and common sense.

Ross Collard is a consultant who currently is working on software testing and quality assurance projects for AT&T, Cisco, GE, Lucent, and the State of California. He teaches software testing for UC Berkeley. Ross has an MS in computer science from the California Institute of Technology and an MBA from Stanford. He can be reached at rcollar@ibm.net.