December, 2009

testing experience

The Magazine for Professional Testers

Standards – What about it?

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

Advanced Software Test Design Techniques: Use Cases

by Rex Black

The following is an excerpt from my recently-published book, *Advanced Software Testing: Volume 1*. This is a book for test analysts and test engineers. It is especially useful for ISTQB Advanced Test Analyst certificate candidates, but contains detailed discussions of test design techniques that any tester can—and should—use. In this third article in a series of excerpts, I discuss the application of use cases to testing workflows.

Use Cases

At the start of this series, I said we would cover three techniques that would prove useful for testing business logic, often more useful than equivalence partitioning and boundary value analysis. First, we covered decision tables, which are best in transactional testing situations. Next, we looked at state-based testing, which is ideal when we have sequences of events that occur and conditions that apply to those events, and the proper handling of a particular event/condition situation depends on the events and conditions that have occurred in the past. In this article, we'll cover use cases, where preconditions and postconditions help to insulate one workflow from the previous workflow and the next workflow. With these three techniques in hand, you have a set of powerful techniques for testing the business logic of a system.

Conceptually, use case testing is a way to ensure that we have tested typical and exceptional workflows and scenarios for the system, from the point of view of the various actors who directly interact with the system and from the point of view of the various stakeholders who indirectly interact with the system. If we (as test analysts) receive use cases from business analysts or system designers, then these can serve as convenient frameworks for creating test cases.

Remember that, with decision tables, we were focused on transactional situations, where the conditions—inputs, preconditions, and so forth—that exist at a given moment in time for a single transaction are sufficient by themselves to determine the actions the system should take. Use cases aren't quite as rigid on this point, but, generally, the assumption is that the typical workflows are independent of each other. An exceptional workflow occurs when a typical workflow can't occur, but usually we have independency from one workflow to the next. This is ensured—at least for formal use cases—by clearly defined preconditions and postconditions, which guarantee certain things are true at the beginning and end of the workflow. This acts to insulate one workflow from the next. Again, if we have heavy interaction of past events and conditions with the way current events and conditions should be handled, we'll want to use state-based testing.

The model is less formal than what we've seen with decision tables and state-based testing. Indeed, the concept of a "use case" itself can vary considerably in formality and presentation. The basic idea is that we have some numbered (or at least sequential) list of steps that describes how an actor interacts with the system. The steps can be shown in text or as part of a flow chart.

The use case should also show the results obtained at the end of that sequence of steps. The results obtained should benefit some party, either the actor interacting directly with the system or some other stakeholder who indirectly receives the value of the results.

At the very least, the set of steps should show a typical workflow, the normal processing. This normal processing is sometimes called the primary scenario, the normal course, the basic course, the main course, the normal flow, or the "happy path". However, since things are not always happy, the set of steps should also show abnormal processing, sometimes called exceptions, exceptional processing, or alternative courses.

Approaches to documenting use cases that are more formal cover not only typical and exceptional workflows, but also explicit identification of the actor, the preconditions, the postconditions, the priority, the frequency of use, special requirements, assumptions, and potentially more. The formal approach might also entail the creation of a use case diagram that shows all the actors, all the use cases, and the relationship between the actors and the use cases.

Now, an assumption that I'm making here—in fact, it's an assumption implicitly embedded within the ISTQB syllabi—is that you are going to receive use cases, not create them. If you look at both the Advanced and Foundation Level syllabi, they talk about use cases as something that test analysts receive, upon which they base their tests. So, rather than trying to cover the entire gamut of use case variation that might exist in the wild and wooly world of software development, I'm going to talk about using basic, informal use cases for test design, and talk about using more formalized use cases for test design, and leave out too much discussion about variations.

So, assuming we receive a use case, how do we derive tests? Well, at the very least, we should create a test for every workflow, including both the typical and exceptional workflows. If the exceptional workflows were omitted, then you'll need to figure those out, possibly from requirements or some other source. Failing to test exceptions is a common testing mistake when using informal use cases.

Creating tests can involve applying equivalence partitioning and boundary value analysis along the way. In fact, if you find a situation where combinations of conditions determine actions, then you might have found an embedded, implied decision table. Covering the partitions, boundaries, and business rules you discover in the use case might result in two, five, ten, twenty, or more test cases per work flow, when you're all done.

Remember that I said that a use case has a tangible result. So, part of evaluating the results of the test is verifying that result. That's

above and beyond verifying proper screens, messages, input validation, and the like as you proceed through the workflow.

Note the coverage criterion implied above: At least one test per workflow, including both typical and exceptional workflows. That's not a formal criterion, but it's a good one to remember as a rule of thumb.

What is our underlying bug hypothesis? Remember in decision tables we were looking for combinations of conditions that result in the wrong action occurring or the right action not occurring. With use cases, we're a bit more coarse-grained. Here, we are looking for a situation where the system interacts improperly with the user or delivers an improper result.

E-commerce purchase: Normal workflow

1. Customer places one or more Items in shopping cart
2. Customer selects checkout
3. System gathers address, payment, and shipping information from Customer
4. System displays all information for User confirmation
5. User confirms order to System for delivery

Exceptions

- Customer attempts to checkout with empty shopping cart; System gives error message
- Customer provides invalid address, payment, or shipping information; System gives error messages as appropriate
- Customer abandons transaction before or during checkout, System logs Customer out after 10 minutes of inactivity

Figure 1: Informal Use Case Example

In Figure 1, we see an example of an informal use case describing purchases from an e-commerce site, like the rbcs-us.com example shown for decision tables in the earlier article.

At the top, we have the web site purchase normal workflow. This is the happy path.

1. Customer places one or more Items in shopping cart
2. Customer selects checkout
3. System gathers address, payment, and shipping information from Customer
4. System displays all information for User confirmation
5. User confirms order to System for delivery

Note that the final result is that the order is in the system for delivery. Presumably another use case having to do with order fulfillment will describe how this order ends up arriving at the customer's home or place of business.

We also see some exception workflows defined.

For one thing, the Customer might attempt to checkout with an empty shopping cart. In that case, the System gives an error message.

For another thing, the Customer might provide an invalid address, payment, or shipping information. On each screen—if we're following the typical e-commerce flow—the System gives error messages as appropriate and blocks any further processing until the errors are resolved.

Finally, the Customer might abandon the transaction before or during checkout. To handle this, the System logs the Customer out after 10 minutes of inactivity.

Now, let's look at deriving tests for this use case. In Figure 2, you see the body of the test procedure to cover the typical workflow. (For brevity's sake, I've left off the typical header and footer information found on a test procedure.)

#	Test Step	Expected Result
1	Place 1 item in cart	Item in cart
2	Click checkout	Checkout screen
3	Input valid US address, valid payment using American Express, and valid shipping method information	Each screen displays correctly and valid inputs are accepted
4	Verify order information	Shown as entered
5	Confirm order	Order in system
6	Repeat steps 1-5, but place 2 items in cart, pay with Visa, and ship international	As shown in 1-5
7	Repeat steps 1-5, but place the maximum number of items in cart and pay with MasterCard	As shown in 1-5
8	Repeat steps 1-5, but pay with Discover	As shown in 1-5

Figure 2: Deriving Tests Example (Typical)

As you can see, each step in the workflow has mapped into a step in the test procedure. You can also see that I did some equivalence partitioning and boundary value analysis on the number of items, the payment type, and the delivery address. Because all of these selections are valid, I've combined them. Note that space-saving approach of describing how to repeat the core steps of the test procedure with variations, rather than a complete re-statement of the test procedure, at the bottom.

In Figure 3, you see the body of the test procedure to cover the exception flows. You can see that I use equivalence partitioning on the points at which the customer could abandon a transaction.

This is a good point to bring up an important distinction, that between logical and concrete test cases. For the ISTQB exam, you'll want to make sure you know the Glossary definitions for these terms. For our purposes here, we can say that a logical or high-level test case describes the test conditions and results. A concrete or low-level test case gives the input data to create the test conditions, and the output data observed in the results.

As you just saw in Figure 2 and Figure 3, you can easily translate a use case into one or more logical test cases. However, translation of the logical test case into concrete test cases can require additional documentation. For example, what was the maximum number of items we could put in the shopping cart? We'd need

#	Test Step	Expected Result
1	Do not place any items in cart	Cart empty
2	Click checkout	Error message
3	Place item in cart, click checkout, enter invalid address, then invalid payment, them invalid shopping information	Error messages, can't proceed tonext screen until resolved
4	Verify order information	Shown as entered
5	Confirm order	Order in system
6	Repeat steps 1-3, but stop activity and abandon transaction after placing item in cart	User logged out exactly 10 minutes after last activity
7	Repeats steps 1-3, but stop activity and abandon transaction on each screen	As shown in 6
8	Repeat steps 1-4; do not confirm order	As shown in 6

Figure 3: Deriving Tests Example (Exception)

some further information, ideally a requirements specification, to know that. What items can we put in shopping cart? Some description of the store inventory is needed.

Is it cheating to define logical test cases rather than concrete ones? No, absolutely not. However, notice that, at some point, a test case must become concrete. You have to enter specific inputs. You have to verify specific outputs. This translation from logical test case to concrete test case is considered an implementation activity in the ISTQB fundamental test process. If you choose to leave implementation for the testers to handle during test execution, that's fine, but you'll need to make sure that adequate information is at hand during test execution to do so. Otherwise, you risk delays.

So, what's different or additional in a formal use case? Usually, a formal use case contains more information than an informal one. Here, you can see some of the typical elements of a formal use case:

- ID—some use case identifier number
- Name—a short name, like E-commerce Purchase
- Actor—the actor, such as Customer
- Description—a short description of the use case
- Priority—the priority, from an implementation point of view
- Frequency of use—how often this will occur
- Preconditions—what must be true to start the use case normally
- Typical workflow—often like the informal use case, but sometimes broken into two columns, one for the actor actions and one for the system response
- Exception workflows—one for each exception, often also with actor action and system response columns.
- Postconditions—what should be true about the state of the system after the use case completes normally

Notice that you can use some of this information as a test analyst. Some, like the priority and frequency of use, you might not use, except during the risk analysis process. Also, notice that the breakdown on the workflows, especially the exception workflows, is finer-grained, so your test traceability can be finer-grained, too.

Typical workflow	1. System gathers address, payment, and shipping information from Customer 2. System displays all information for User confirmation 3. User confirms order to System for delivery
Exception 1	Customer attempts to checkout with empty shopping cart System gives error message
Exception 2	Customer provides invalid address, payment, or shipping information System gives error message as appropriate
Exception 3	Customer abandons transaction before or during checkout System logs Customer out after 10 minutes of inactivity
Exception 4	Order is active in system

Figure 5: Formal Use Case Example (Part 2)

Figure 4 shows the header information on a formal version of the informal use case we saw earlier. Notice that some of the steps of the informal use case became preconditions. This means that the shopping portion of the use case would become its own use case, allowing this use case to focus entirely on the purchase aspects of the e-commerce site. Notice also that we didn't know about that "logged in" requirement before. That's important information for our testing.

Figure 5 shows the main body of the formal use case, the normal workflow and the three exceptions. Notice the normal workflow is a bit shorter now because some of its steps became preconditions. Also, each exception has its own row in the table. Finally, notice that the postcondition is true only if the normal workflow is ultimately completed.

Conclusion

In this article, I've shown how to apply use cases to the testing of typical and exceptional workflows. We have looked at decision tables as a way to test detailed business rules, state-based methods to test state-dependent systems, and now use cases for workflows. With these three techniques, you can perform a full range of internal business logic testing.

ID	02.001
Name	E-commerce Purchase
Actor	Customer
Description	Allow Customer to complete a transaction by purchasing the item(s) in her shopping cart
Priority	Very high
Frequency in use	25% of Customers, up to 1,000 customers per day
Preconditions	1. One or more items in shopping cart 2. Customer is logged in 3. Customer has clicked on checkout

Figure 4: Formal Use Case Example (Part 1)

Biography

With a quarter-century of software and systems engineering experience, Rex Black is President of RBCS (www.rbcs-us.com), a leader in software, hardware, and systems testing. For over a dozen years, RBCS has delivered services in consulting, outsourcing and training for software and hardware testing. Employing the industry's most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide.