Extended Use Case Test Design Pattern

Intent

Develop a comprehensive application system test suite by modeling essential capabilities as extended use cases.

Context

Extended Use Cases applies when most, if not all, of the essential requirements for the system under test (SUT) can be expressed as use cases. Use cases are not always the model of choice for essential system-scope capabilities. For example, consider a complex simulation which runs for weeks on a supercomputer. The use cases describe the set up and inquiry for a simulation run, but only hints at the algorithms to compute the results and render the results with animated, high-resolution graphics.

It is possible to develop test cases by imagining specific inputs for a use case and corresponding expected results. Some examples of specific use case inputs appear in figure 1. Typically, a *very* large number of specific cases can be enumerated. But unless test points are systematically selected based on use case's implicit constraints and relationships, it is unlikely that all the fundamental relationships will be exercised. Attempts at enumeration can lead to a large but inefficient test suite. Efficient and effective testing requires modeling constraints and relationships and generating enough test cases to be sure they have been exercised.

Fault Model

A use case specifies a family of responses to be produced for specific combinations of external input and system state. The extended use case model represents these relationships as a decision table. This suggests use case implementation faults can be modeled as decision table faults. A use case implementation is therefore susceptible to the same kinds of faults that are possible for combinational logic implementations.

- Domain faults. The code which deals with boundary of a condition is incorrect. See *Invariant Boundary* for details of domain faults.
- Logic faults. The IUT behaves as if *and* logic was specified instead of *or*, etc.
- Incorrect handling of *don't care* conditions.
- Incorrect or missing dependency on states established by other use cases.

Use case implementations are also susceptible to generic system-scope faults.

- Undesirable feature interactions.
- Incorrect output.
- Abnormal termination.
- Inadequate response time.
- Omitted capability.
- Extra capability (a surprise.)

The typical implementation of a use case is a collaboration: a chain of messages between objects and components. We must exercise all collaborations which support the use case to reach a fault. Assuming integration testing has demonstrated basic operability for all the collaborations in the use case, then if a fault exists, it must be sensitive to some combination of input and state. Exercising the combinations of conditions and boundary values of the operational variables is the most effective approach to triggering a fault. Propagation impendence depends on the particular structure of the collaboration and its components.

Strategy

Test Model

An extended use case includes the following information:

- A complete inventory of operational variables.
- A complete specification of domain constraints for each operational variable.
- An operational relation for each use case.
- The relative frequency of each use case. This value is optional. It is used to support *Allocate Tests by Frequency*.

Extended use cases may be prepared with any OOA/D technique which calls for use cases. Operation variables are analyzed to construct an operation relation. Extended use cases model input output relationships with a decision table.

Test Procedure

With extended use cases in hand, a test suite is developed in four steps. The Establish Session use case for a simplified automatic teller machine (ATM), shown in figure 1 is used to illustrate.

1. Identify the Operational Variables. Operational variables are the factors which vary from one scenario to the next and determine significantly different system response. Operational variables include the following.

- Explicit inputs and outputs.
- Environmental conditions which result in significantly different actor behavior.
- Abstractions of the state of the system under test (e.g., ATM state: out of service, out of cash, ready, etc.)

These variables are used to construct specific test cases. The operational variables must include all variables which are explicitly part of the interface which supports the use case. If additional variables are necessary to account for all the distinct responses that may be produced, then operational variables that represent system state and relevant environmental factors should also be modeled.

For the Establish Session use case, four variables determine the ATM's response to a Bank Customer's inserting a card and entering a PIN: the PIN encoded on the card, the PIN entered by the customer, the response from the customer's bank, and the condition of the customer's accountat the bank.

Use Case	Actor	Possible Input/Output Combinations
Establish Session	Bank Customer	(1) Wrong PIN entered once, corrected PIN entered. Display menu. (2) PIN ok, customer's bank not online. Display "Try later." (3) PIN ok, customer's accounts are closed. Display "Call your bank." (4) Stolen card inserted, valid PIN entered. Retain card.
Cash Withdrawal	Bank Customer	(1) Requests $50, account open, balance 1,234.56, $50 dispensed. (2) Requests $100, account closed. (3) Requests $155.39, account open. $150 dispensed.
Cash Replenishment	ATM Operator with Armed Guard	(1) ATM opened, Cash dispenser is empty, $15,000 is added. (2) ATM opened, Cash dispenser is full.

Figure 1. Some Use Cases and Scenarios for an Automatic Teller Machine

2. Define the Domains of the Operational Variables. The domains are developed by defining the set valid and invalid values for each variable. For example, the domain of the PIN is four digits with the range 0000 to 9999. Test cases require that valid (and invalid) values are specified. The domain definitions can be used to generate additional test cases for each variant.

3. Develop the Operational Relation. Relationships among the operational variables that determine distinct classes of system response are modeled. The operational variables are cast into an operational relation. This is a decision table: when all the conditions in a row are true, the expected action (the response component of the use case) is to be produced. Each row (column) in the decision table is a **variant**.[1] Each variant must be mutually exclusive: one and only one variant must be true for any possible set of operational variable values. Each significantly different class of output should have a variant. For example, suppose CustomerName is displayed in some responses but not others. These responses are significantly different and would be modeled with different variants. But we would not model each possible CustomerName, e.g., Jones, Smith, . . . with its own variant.

A partial operational relation for the Establish Session use case is shown in figure 1.2. The first row indicates that when an invalid card is inserted (for example, a department store credit card), the card is immediately ejected and a message is displayed prompting the customer for an ATM card. The second row indicates that "Contact your bank" is displayed and the card is ejected if (1) a valid card is inserted, and (2) the entered PIN matches the card PIN, and (3) the customer's bank acknowledges the account, and (4) the account is reported closed.

[1] A variant can correspond to a scenario, but there are many reasons that it might not. The operational variables can include environmental factors (e.g., time of day), which are not represented in the SUT. The UML defines a scenario as a "useCaseInstance," whose "realization" *may* be modeled with a Sequence Diagram or Collaboration Diagram. Such scenarios may or may not be modeled -- the UML does not require completeness. These diagrams may represent one unconditional message sequence or several conditional sequences. A variant can be represented with an unconditional sequence or as one path possible in a conditional sequence. However, unless the conditional expressions in a sequence or collaboration diagram correspond exactly to those in the operational relation, there is no reason to expect one-to-one correspondence between scenarios and variants.

Variant	Operational Variables				Expected Result	
	Card PIN	Entered PIN	Customer Bank Response	Customer Account Status	Message	Card Action
1	Invalid	DC	DC	DC	Insert an ATM card	Eject
2	Valid	Matches Card PIN	Bank Acknowledges	Closed	Contact your bank.	Eject
3	Valid	Matches Card PIN	Bank Acknowledges	Open	Select a transaction	None
4	Valid	Matches Card PIN	Bank Does Not Acknowledge	DC	Please try later	Eject
5	Valid	Doesn't Match	DC	DC	Reenter PIN	None
6	Revoked	DC	Bank Acknowledges	DC	Card invalid	Retain
7	Revoked	DC	Bank Does Not Acknowledge	DC	Card invalid	Eject

Figure 2. ATM Operational Relation for the *Establish Session* Use Case

4. Develop Test Cases. Every variant is made true once and false once. This requires for each variant (1) a *true* test case: a set of values which satisfies all the conditions in a variant, and (2) a *false* case: changing the input or state values so that at least one condition becomes false. Often, the false case for one variant will qualify as a true case for another variant. Try to choose a *false* case which does not repeat a *true* case for another variant. If an operational variable has a bounded domain, additional tests can be developed with *Invariant Boundaries*. Figure 3 shows some test cases developed for the ATM example.

Oracle

Expected results are typically developed by inspection. That is, a person considers the test case inputs and develops expected results. Ideally, this person has a broad understanding of required capabilities and the ways that the SUT will be used. When an existing system provides a similar capability, it may be possible to use the existing system to generate part or all of the expected results.

Automation

The test suite may be implemented with *Capture / Playback* or *Embedded Harness* [Binder 99] as required by the external interface for the system under test. If assertions have been implemented, they should be enabled to detect otherwise unobservable failures.

Entry Criteria

Development of extended use cases may begin as soon as use cases are developed. However, since initial use cases are often subject to many revisions, it may be better to postpone developing the extended use cases until the basic use cases have settled. Testing can be started when two conditions have been met.

- ✓ Extended use cases have been developed and validated.
- ✓ The SUT has passed an integration test suite which demonstrates that components required to support the use case are minimally operable.

Exit Criteria

All requirements must be exercised to achieve minimally complete system testing. The *TC* coverage metric provides an indication of the completeness of requirements-based testing [IEEE 982.1].

$$TC = \frac{Number\ of\ Implemented\ Capabilities}{Number\ of\ Required\ Capabilities} \times \frac{Total\ Components\ Tested}{Total\ Number\ of\ Components} \times 100$$

The first expression prevents 100 percent coverage from being obtained when some capabilities are missing. This metric can be adapted to extended use cases. At a minimum, every variant of an extended use case should be exercised at least once. This suggests the following minimal coverage metric for an extended use case test suite.

- ✓ Variant Coverage: at least one true-false test pair is passed for each variant in the extended use case decision table. This a minimally adequate test suite, in the same sense that statement coverage is minimally adequate. This formula may be reinterpreted for use-case based testing as follows.

$$XUCV = \frac{Number\ of\ Implemented\ UseCases}{Number\ of\ Required\ UseCases} \times \frac{Total\ Variants\ Tested}{Total\ Number\ of\ Variants} \times 100$$

- ✓ Use cases are traceable to test cases. Every use case has a test and this test passes. Figure 4 suggests how this use case traceability may be achieved.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	. . .	Test 9999
Use Case 1				✔					
Use Case 2		✔			✔				
Use Case 3	✔					✔			✔
Use Case 4					✔		✔		
Use Case 5									
. . .									
Use Case 999		✔							✔

Figure 4. Use case test case traceability matrix.

Consequences

Disadvantages

- There is no agreement among methodologists about the appropriate abstraction and specificity for a use case. However, test design must resolve the ambiguities. Too much detail or not enough will create extra work for the tester.
- Use cases are not typically used to specify performance, fault-tolerance, etc. These capabilities must be defined and tested.
- Some methodologies support the *extends* and *includes* constructs for use cases. A composite use case must be flattened to produce testable model of the inter-use case dependencies. The alternative is to hope that your choices for tests at the high level exercise all the low level dependencies.

Advantages

- Use cases are incorporated in nearly all OOA/D methodologies and widely accepted.
- Use-cases may be developed by analysts and testers who do not have object-oriented programming experience.
- Use cases are often the only available requirements documentation.
- Development of testable use-cases leverages (and thereby encourages) investment in good OOA/D.
- Use cases reflect the user/customer point of view. They focus on capabilities which will determine the success or failure of the SUT. In contrast, developers often tend to focus on a particular technical strategy. The customer-oriented focus is often more effective revealing omissions or inconsistent capabilities.
- Extended use cases provide a systematic approach to developing the information necessary for test design. This information will have to be developed to test any use case. Subjective, non-systematic test design can miss subtle relationships
- The extended use case format supports test design based on combinational logic and software reliability engineering. These approaches have been the subject of extensive formal research and wide-spread application. Extended use cases provide a simple interface to these powerful techniques.
- An extended use case makes explicit all relationships which a use case must implement. The arbitrary narrative which often suffices for use cases results in ambiguous, incomplete, or inconsistent models. Often, development of the extended use case is sufficient to find errors and omissions.
- If the SUT was developed from ambiguous, inconsistent, or incomplete use cases, it is likely to be buggy. A test suite developed from such information is likely to be inadequate. Developing extended use case from the outset is preferable.
- If the implementation has been developed from non-testable use cases, a test suite designed from extended use cases can be expected to reveal design bugs resulting from an ambiguous, inconsistent, or incomplete use cases.

Known Uses

Jacobson suggests four general kinds tests be derived from use cases: (1) basic courses, or "the expected flow of events," (2) odd courses, i.e., "all other flows of events," (3) tests of any line-item requirements traceable to each use case, and (4) tests of features described in user documentation traceable to each use case." [Jacobson+92]. The Use Case Testing pattern [Firesmith 96] suggests that an acceptance test suite may be prepared from use cases, but does not discuss any details of test design. Siegel recommends developing a system test suite from use cases and an operational profile but does not provide details [Siegel 96].

Boisvert describes the results of a system test suite developed from the use cases of a telecommunications system [Boisvert 97]. I have worked with several clients to apply extended use cases to system testing of high-volume financial applications. The Appendix [Binder 99] discusses some of the test automation considerations for Extended Use Case Test. The integration of use cases and the operational profile is discussed in [Runeson+98].

Related Patterns

If relative frequencies of each use case are developed, system testing may be conducted under the *Allocate Tests by Profile* pattern. This allows systematic generation of test cases in manner that will maximize field reliability for a given test budget. Test suites developed with *Round Trip Scenarios* can be added to Extended Use Case test suites.

Sources

[Binder 99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[Boisvert 97] Jean Boisvert. OO testing in the Ericsson pilot project. *Object Magazine* 7(5):27-33, July 1997.

[Firesmith 96] Donald G. Firesmith. Pattern language for testing object-oriented software. *Object Magazine* 5(9):32-28, January 1996.

[IEEE 982.1] *ANSI/IEEE Standard 982.1-1988: IEEE Standard Dictionary of Measures to Produce Reliable Software.* New York: The Institute of Electrical and Electronic Engineers, 1989.

[Jacobson+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-oriented software engineering*. Reading, Mass.: Addison-Wesley, 1992.

[Runeson+98] Per Runeson and Björn Regnell. Derivation of an integrated operational profile and use case model. In *Proceedings, 9th International Symposium on Software Reliability Engineering*. Los Alamitos, Calif: IEEE Computer Society Press. November 1998. 70-79.

[Siegel 96] Shel Siegel. *Object-oriented software testing: a hierarchical approach.* New York: John Wiley and Sons, 1996.