

Category-Partition Test Design Pattern

Intent

Design method scope test suites based on input/output analysis.

Context

How can we develop a test suite to exercise the functions implemented by a single method? A method may implement one or several functions, which each may present varying levels of cohesion. Even cohesive functions can have complex input/output relationships. A systematic technique to identify functions and exercise each input/output relationship is needed.

The Category-Partition pattern is appropriate for any method that implements one or more independent functions. For methods or functions that lack cohesion, the constituent responsibilities should be identified and modeled as separate functions. This approach is qualitative and may be worked by hand. It provides systematic implementation-independent coverage of method-scope responsibilities.

If the method selects one of many possible responses or has many constraints on parameter values, consider using the *Com binational Function Test* pattern [Binder 99].

Fault Model

This pattern assumes that faults are related to value combinations of message parameters and instance variables and these faults will result in missing or incorrect method output. Offutt and Irvine found that a Category-Partition approach was able to reveal 23 common C++ coding blunders [Offutt+95]. However, faults that are only manifested under certain sequences or which corrupt instance variables hidden by the MUT's interface may not be revealed by category-partition tests.

Strategy

Test Model

Category-partition was first introduced as a general purpose black-box test design technique for software modules with parameterized functions [Ostrand+88]. Development of a method scope category-partition test suite is illustrated with C++ member function `List::getNextElement()`. This member function returns successive elements of a `List` object. A position in a list of m elements is established at the n th element by other member functions. A call to `getNextElement()` returns element $n + 1$. Successive calls to `getNextElement()` return element $n + 2$, $n + 3$, etc. If no position has been established by a previous operation or the previous position no longer exists due to an intervening change or delete, a `NoPosition` exception is thrown. An `EmptyList` exception is thrown if the list is empty.

Test Procedure

The test suite is produced in seven steps.

1. *Identify the testable functions of the MUT.* A well-designed method implements only a few cohesive functions, which are typically suggested by its name and parameters. A method may implement several functions. Other functions may be side-effects of the primary function. For example, the current position of a `List` object may be incremented as a side-effect of returning the next element. A poorly designed or poorly named method may support several unrelated functions, or may have obscure side-effects. In any case, the tester must tease out all the independently testable functions in the MUT. A function is independently testable if it meets three criteria.

- It can be invoked by setting message parameters to particular values and sending a message to the method.
- Its output can be observed in the values returned by the method, indirectly by using other class methods, or by an invasive inspector.
- Its output can be differentiated from the output of another function or an incorrect function.

Clearly, testability is improved when methods are explicitly designed and coded to be testable functions.

The primary function of `getNextElement()` is to return the next list element. Its secondary functions are to (1) keep track of the last position and wrap it from last to first, (2) throw the `NoPosition` and `EmptyList` exceptions if appropriate. Each of these functions may be controlled and observed independently.

2. *Identify the input and output parameters of each testable function.* At method scope, these are method interface arguments and abstract class state(s) which determine the function’s response. Class variables, globals, and system environmental variables should be modeled, if they can influence the output or may be changed by the function call. For example, the inputs for `List::getNextElement()` are the position of the last referenced element, and the list itself. The outputs are the element returned by the member function and the incremented position cursor. None of these test parameters appear in the formal argument list of the member function.

3. *Identify categories for each input parameter.* Categories do not overlap and must result in distinctly different output. A **category** is a subset of parameter values that (1) determines a particular behavior or output, and (2) whose values are not included in any other category. Categories are conceptually similar to Myers’ equivalence classes [Myers 79] and Marick’s C++ test requirements [Marick 95]. Figure 1 shows the categories for the `getNextElement()` parameters.

Parameter	Category
Position of the last referenced element	nth element
	Special Cases
State of the list	—elements
	Special Cases

Figure 1 Categories for getNextElement()

4. *Partition each category into choices.* A **choice** is a specific test value for a category. Any reasonable criteria may be used to choose choices. Choices for primitive types can be selected by suspicions, past bugs, or the *Invariant Boundaries* pattern [Binder 99].

Choices for the `getNextElement()` parameters are shown in figure 3. These choices reflect the basic domain fault model: the implementation of boundary conditions is error-prone and if a function works correctly in one non-boundary situation it usually works correctly for all non-boundary situations. To increase the variation in our test suite, in-range values for *m* and *n* are chosen by a random number generator. Redundant input for choices Full, Maximum Size, and Last positions have been dropped.

Choices should include legal and illegal values as shown for `getNextElement()` to exercise error and exception handling. “A well-defined partition should include some input classes that consist solely of error inputs.” [Balcer+89] 211.

Parameter	Category	Choices
Position of the last referenced element	nth Element	n = 2
		n = some x > 2, x < Max
		n = Max
	Special Cases	Undefined
		First
		Last, n < Max
State of the list	—elements	m = some x > 2, x < Max
	Special Cases	Empty
		Singleton
		Full (m = Max)

Figure 2 Categories and Choices for `getNextElement()`

5. *Identify constraints on choices.* Some choices may be mutually exclusive or inclusive. With `getNextElement()` parameters, an undefined position precludes a response for all states of a `List`. This exclusion is implemented by the `NoPosition` exception. Exclusions can be structural or practical. See *Don't Care, Don't Know, Can't Happen*, [Binder 99], for a discussion of exclusions. Choices or combinations of choices can be excluded for practical reasons: the time, cost, or complexity of generating or evaluating a combination is prohibitive.

6. *Generate test cases by enumerating all choice combinations.* The test cases are identified by generating the cross-product of all the choices. Mutually inclusive or exclusive choices will result in some test cases being dropped from the cross product set. With categories of 6 and 4 choices, there are $24 = 6 \times 4$ test cases. Figure 2.31 shows the test suite formed by the cross product of `getNextElement()` choices.

7. *Develop expected results for each test case using an appropriate oracle.* Figure 3 shows the expected results for each `getNextElement()` test case. These values were determined by analysis and manually simulating the method.

CATEGORY-PARTITION TEST DESIGN PATTERN

Test case	Function Parameters/Choices		Expected Result		
	position of the last referenced element	state of the list	Returned	Exception	Position of the last referenced element
1	undefined	empty	null	noPosition	undefined
2	undefined	singleton	null	noPosition	undefined
3	undefined	m= rand(x)	null	noPosition	undefined
4	undefined	full	null	noPosition	undefined
5	first	empty	null	listEmpty	undefined
6	first	singleton	first		first
7	first	m= rand(x)	second element		second element
8	first	full	second element		second element
9	n = 2	empty	null	listEmpty	undefined
10	n = 2	singleton	null	noPosition	undefined
11	n = 2	m= rand(x)	element n + 1		element n + 1
12	n = 2	full	element n + 1		element n + 1
13	n = rand(x)	empty	null	listEmpty	undefined
14	n = rand(x)	singleton	null	noPosition	undefined
15	n = rand(x)	m= rand(x)	element n + 1		element n + 1
16	n = rand(x)	full	element n + 1		element n + 1
17	Max	empty	null	listEmpty	undefined
18	Max	singleton	null	noPosition	undefined
19	Max	m= rand(x)	element n + 1		element n + 1
20	Max	full	element n + 1		element n + 1
21	Last	empty	null	listEmpty	undefined
22	Last	singleton	null	noPosition	undefined
23	Last	m= rand(x)	first element		first element
24	Last	full	first element		first element

Figure 3 Test Suite for getNextElement()

Automation

A Category-Partition test suite may be implemented with any of the API Test Harness patterns presented in chapter 19 [Binder 99]. It is particularly well-suited to *Incremental Testing Framework* since it is a systematic yet simple way to design method scope test cases.

Entry Criteria

- ✓ Small pop.

Exit Criteria

- ✓ Every combination of choices is tested once. Suppose there are 3 choices for variable a , 7 choices for variable b , and 2 choices for variable c . Then a full test suite requires $3 \times 7 \times 2 = 42$ test cases.
- ✓ If the method under test can throw exceptions for incorrect input or other anomalies, the test suite should force each exception at least once. The inclusion of “illegal” values or combinations in the choices should exercise all the exceptions. If they do not, add tests to force each exception.
- ✓ Executing the test suite should exercise at least all branches in the method under test. Verify this by checking for method scope branch coverage.

Consequences

The Category-Partition pattern is general-purpose, non-quantitative, and straightforward. It does not require advanced analysis or automated support. However, identification of categories and choices is subjective. Skilled and novice testers may identify different categories and choices. Individual blind-spots may reduce effectiveness. Bugs that are only manifested under certain sequences of messages to other methods or which corrupt instance variables hidden by the MUT’s interface may not be revealed.

The size of a Category-Partition test suite is the product of the number of choices, less the number of combinations excluded by constraints. This can result in many test cases for even moderately complex methods. For example, suppose a function has seven parameters: three formal arguments, two significant class variables, and two significant encapsulated attributes. If we follow the weak 1×1 domain selection to define both categories and choices (one on-point and one off-point for each parameter) we have $2^7 = 128$ test cases.

If the parameters have more than two interesting choices, we could easily generate thousands of test cases. This may be practically infeasible. The *Invariant Boundaries* pattern may be used to produce a smaller test suite whose size is a sum of the number of choices plus one, not the product (see below.)

With proper attention to interface details, a superclass Category-Partition method test suite may be reused to test an overriding subclass method. For example, the `getNextElement()` test suite could be repeated for `List` subclasses such as `PersonList`, `EmployeeList`, `ProgrammerList`, etc. or to testing a `List` template instantiated with type parameters like `integer`, `string`, `float`, `date`, `money`, etc. The reusability depends on the extent to which the class contract follows the Liskov Substitution Principle for subclasses and parameterized types.

Known Uses

Elements of Category-Partition appear in nearly all black-box test design strategies, for example [Myers 79] [Marick 95] [Beizer 95]. An automated test tool for procedural code that supports category-partition approach is presented in [Balcer+89]. An Z-based approach to automating the category-partition model is presented in [Laycock 92]. A study of manually developed category-partition test suites for a small C++ system found 55 out of 75 inserted faults. The inserted faults were selected to represent common C++ coding errors. Of 20 faults not found by the category-partition test suite, nineteen were memory leaks and one was masked by test tool initiation sequence [Offutt+95].

Sources

- [Balcer+89] Mark Balcer, William Halsing, and Thomas Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*. New York: ACM Press. December 1989.
- [Beizer 95] Boris Beizer. *Black box testing*. New York: John Wiley & Sons, 1995.
- [Binder 99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.
- [Laycock 92] Gilbert Laycock. Formal specification and testing: a case study. *Journal of Software Testing, Verification, and Reliability* 2(1):7-23, May 1992.
- [Marick 95] Brian Marick. *The craft of software testing*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1995.
- [Myers 79] Glenford J. Myers. *The art of software testing*. New York: John Wiley & Sons, 1979.
- [Offutt+95] A. Jefferson Offutt and Alisa Irvine. Testing object-oriented software using the category-partition method. In *Proceedings, TOOLS 17*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc. 1995.
- [Ostrand+88] Thomas J. Ostrand and Marc J. Blacer. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 31(6):676-686, June 1988.