

Method Scope Test Patterns



Class Test

- How to test classes, one at a time?
 - Create a test suite
 - Each test case = exercise a method of the class
- If a test case fails, where is the bug?
 - IUT includes the superclasses and server objects of the class under test
 - But, usually, testing focus on one class:
 - Individual classes
 - Small clusters
 - Test the head class only

Why test at class scope ?

- Class scope testing corresponds with the classical definition of unit testing
- Class scope testing can provide significant improvements in quality and productivity
 - Many bugs are likely to go undetected if class scope testing is minimal or skipped
 - System-level testing cannot exercise components to the same degree as class scope testing
 - Decreases cost of bug correction
 - It is reasonable for developers of client classes to expect reliable operation of server class object
- Effective class scope testing reduces schedule risk and improves productivity

Class testing strategy

- How to test a class?
- Testing a class requires exercising all methods
- How to test a method?
 - Consider responsibility of method
 - Implement a set of test cases that exercises the responsibility of the method
 - Each test case focus on a single method
- Testing a class \Rightarrow develop test suite at method scope
- Coverage analysis of method scope test suite can be used for assessing test *completeness*
- Is this enough?

Class testing strategy - 2

- Why method-scope testing is not enough?
- Need to consider semantics of OOPL
 1. Although we test one method at a time, methods cannot exist apart from a class
 - ⇒ Testing the intraclass interactions
 2. A class is a composite of features it defines and inherits
 - ⇒ superclass features must be exercised in the context of the subclass
 - ⇒ Testing the superclass/subclass interactions

Class testing design approach

- **Make a preliminary estimate of class under test (CUT)**
 - Plan budget for testing
- **Design and code a test driver**
 - For non-trivial classes begin with alpha-omega skeleton
 - After the alpha-omega tests pass, add additional tests
- **Select a class scope test pattern**
- **Select test design patterns for each method**
- **Arrange method test cases according to sequence called for by the class scope pattern**
 - Probably need to add or change some method scope tests to satisfy the class scope strategy
- **Build the test package**
 - When all tests pass, evaluate coverage
 - If coverage is insufficient, develop more tests.

Who and When

- Class scope testing is typically the responsibility of the class developer:
 - designs the test suite
 - codes the test suite
 - runs the tests
 - evaluates the results
- Several possibilities for when (or how):
 - No class-level testing
 - All classes are reviewed, but only some classes are tested
 - Test are designed and coded for every class before or during class development
 - Two people work on development and testing of the same class at the same time.
 - Users of this approach report gains in productivity and quality that offset the increased cost of double-teaming

Class Scope Integration

- The purpose of class scope integration is to demonstrate that the CUT is ready to test
- Two approaches for reaching the operability threshold:
 - Small Pop
 - Alpha-Omega Cycle

Small Pop

- Small Pop may be the most expedient route to class scope testing
- It is effective when responsibility-based testing and debugging are unlikely to be impeded by simple coding problems:
 - The class under test (CUT) is small and simple
 - All or most of the servers of the class under test are known to be stable by testing or usage
 - All or most of the inherited features are known to be stable by testing or usage
 - Few intraclass dependencies exist

Small Pop

- Testing strategy is simple:
 - Develop the entire class, without any testing
 - Write a test suite using any appropriate test pattern
 - Run the test suite
 - Debug the class as needed

Alpha-Omega cycle

- Alpha and omega are the two placeholder states that bracket the life cycle of an object.
- The alpha-omega cycle takes the OUT from the alpha state to omega state by sending a message to every method at least once
- Passing this cycle means that the CUT is **ready for extensive testing**
- No attempt is made to achieve any coverage

Alpha-Omega Cycle: Process

- The test driver sends one message to each of the following kinds of methods in the specified order:
 - New or constructor method
 - Accessor (get) method
 - Boolean (predicate) method
 - Modifier (set) method
 - Iterator method
 - Delete or destructor method
 - Order within each step: *private*, *protected* and *public*
- **Useful for incremental development**

Basic method scope test procedure

1. Set, as needed, the parameters of the test message, class and global variables to the desired pre-test state
2. Set OUT to the desired pre-test state.
3. Send the test message from the test driver to the MUT
4. Code the driver to make as many post-test checks as possible:
 - returned value = expected result
 - state of the OUT = expected state
 - state of class and global variables = expected state
 - If call-by-reference semantic is used compare the state of the arguments with the expected state
 - Catch all exceptions and check whether they were expected or not
5. If one of these conditions fails, diagnose and correct the problem



Method test design patterns

- Category-Partition - A general purpose test design pattern
- Combinational Functional Test – Appropriate for methods that implement complex selection algorithms (case-based logic)
- Recursive Function Test – Appropriate for recursive methods
- Polymorphic Message Test – Used to augment the test suite for a client of a polymorphic server

Concept - Functional Cohesion

- A method implements a fragment of the class contract
- The responsibility of a method is called its *function*
 - A method can implement more than one function
- Functional cohesion: the unity of purpose among two or more functions
- Good Design
 - The functions of a method should have strong functional cohesion

Category-Partition

- Intent
 - Design method scope test suites based on I/O analysis
- Context
 - Appropriate for any method that implements one or more functions
 - Systematic technique to identify functions and exercise each I/O relationship
 - If lack cohesion, the constituents responsibilities should be identified and modelled as separate methods
 - If method selects one of many possible responses or has many constraints on input parameters, consider using *Combinational Function Test*

Fault Model

- Reveal faults such that combinations of message parameters and instance variables result in missing or incorrect method output
- Faults not covered:
 - Depend on certain message sequences
 - That result in the corruption of object state hidden by the MUT's interface

Strategy: Test model

- Black box testing
- Example
 - **getNextElement()** of class **List**
 - **Specification**
 - Returns successive elements of the list
 - Position is established by other methods, otherwise a **NoPosition** exception is thrown
 - An **EmptyList** exception is thrown if the list is empty

Strategy: Test Procedure

1. Identify all functions of the MUT
2. Identify all input and output parameters of each function
3. Identify categories for each input parameter
4. Partition each category into choices
5. Identify constraints on choices
6. Generate test cases by enumerating all choice combinations
7. Develop expect results for each test case using an appropriate oracle

1st step - Identify all functions of the MUT

- Well-designed method implements few cohesive functions
 - Some functions may be side-effects of the primary function
- Functions of *getNextElement()*:
 - Primary function:
 - return next element in the list
 - Secondary functions:
 - Keep track of the last position and wrap it from last to first
 - Throw the NoPosition and EmptyList exceptions if appropriate

2nd step: Identify input and output parameters of MUT

- Identify input parameters for each function
 - Parameters of the function and instance variables
 - May also include class attributes and global variables
 - The output of each function should also be identified
 - Example *getNextElement()*:
 - Input:
 - position of the last referenced object
 - the list itself
 - Output:
 - the returned element
 - the incremented position cursor
 - Can also consider the exception thrown as an output
- Functions for getNextElement():
 - return next element in the list
 - **Inputs:** list and position
 - **Output:** element
 - Keep track of the last position and wrap it from last to first
 - **Inputs:** list and position
 - **Output:** incremented position
 - Throw the NoPosition and EmptyList exceptions if appropriate
 - **Inputs:** position and list
 - **Output :** exception

3rd step: Identify categories for each input parameter



- A category is a subset of parameter values such that:
 - Determines a particular behavior or output
 - Whose values are not included in another category
- Categories for *getNextElement()*

Parameter	Category
Position of the last referenced element	nth element
	Special cases
State of the list	m-elements
	Special cases

4th step: Partition each category into choices

- A choice is a specific test value for a given category
- Choices are made by intuition, experience or by applying domain testing
- Choices should include both legal and illegal values

Parameter	Category	Choices
Position of the last referenced elemento (cursor position)	nth element n in $[2, \text{Max}-1]$	$n = 2$, $n = \text{Max} - 1$ $n = \text{some } x \text{ in }]2, \text{Max} - 1[$
	Special cases	Undefined, First, Last
State of the list (contente)	m-elements	$m = \text{some } x \text{ in } [2, \text{Max}[$
	Special cases	Empty, Singleton, Full ($m = \text{Max}$)

5th step – Identify constraints on choices

- Some choices may be mutually exclusive or inclusive
- When more than one combination of choices result on the same action, we can reduce number of test cases
- Choices or combinations of choices can be excluded for practical reasons
- Example *getNextElement()*:
 - An undefined position precludes a response for all states of a list
 - If list is singleton, last and first refer the same position

6th step: Generate test cases by enumerating all choices



- Test cases are generated by the cross product of all choices
- Mutually exclusive or inclusive choices can reduce the number of generated test cases

Parameter	Choices
Cursor position	$n = 2$
	$n = \text{some } x \text{ in }]2, \text{Max}[$
	$n = \text{Max} - 1$
	Undefined
	First
	Last
content	$m = \text{some } x \text{ in }]2, \text{Max}[$
	Empty
	Singleton
	Full ($m = \text{Max}$)

Test cases for getNextElement()

Input		
Test Case	Cursor position	content
1	Undefined	m = rand(x)
2	first	empty
3	first	singleton
4	first	m = rand(x)
5	first	full
6	n = 2	empty
9	n = 2	singleton
8	n = 2	m = rand(x)
9	n = 2	full
10	n = rand(x)	empty
11	n = rand(x)	singleton
12	n = rand(x)	m = rand(x)
13	n = rand(x)	full
14	n = Max - 1	empty
15	n = Max - 1	singleton
16	n = Max - 1	m = rand(x)
17	n = Max - 1	full
18	last	empty
19	last	m = rand(x)
20	last	full



7th step: Develop expected values for each test case

- Expected values can be determined by an Oracle, or
- Determined by analysis and manually simulating the MUT

Test cases for getNextElement()



Test Case	Input		Expected Output		
	Cursor position	content	returned	Cursor position	Exception
1	Undefined	m = rand(x)	-	Undefined	NoPosition
2	first	empty	-	Undefined	EmptyList
3	first	singleton	First element	first	-
4	first	m = rand(x)	Second element	second	-
5	first	full	Second element	second	-
6	n = 2	empty	-	Undefined	EmptyList
7	n = 2	singleton	-	Undefined	NoPosition
8	n = 2	m = rand(x)	Third element	third	-
9	n = 2	full	Third element	third	-
10	n = rand(x)	empty	-	Undefined	EmptyList
11	n = rand(x)	singleton	-	Undefined	NoPosition
12	n = rand(x)	m = rand(x)	Element n + 1	n + 1	-
13	n = rand(x)	full	Element n + 1	n + 1	-
14	n = Max -1	empty	-	Undefined	EmptyList
15	n = Max -1	singleton	-	Undefined	NoPosition
16	n = Max -1	m = rand(x)			
17	n = Max -1	full			
18	last	empty	-	Undefined	EmptyList
19	last	m = rand(x)	first	first	-
20	last	full	first	first	-

Entry and Exit Criteria

- **Entry Criteria**

- Small Pop or Alpha-Omega cycle

- **Exit Criteria**

- Every combination of choices is tested once
- Each exception must be thrown at least once
- Branch coverage must be satisfied

Consequences & Known uses

- Consequences

- General purpose, straightforward technique
- Identification of categories and choices is a subjective process
- The size of the test suite may become quite large
- The generated test suite can also be used for subclasses

- Known Uses

- Elements of the Category-Partition approach appear in nearly all black-box test design strategies

5th step – Identify constraints on choices: **Error constraint**



- Error/invalid categories
 - No need to test all possible combinations of values belonging to invalid categories
 - One test case per error value is enough
 - Mark error choices with **[error]**
 - **Just one input with error value per test case**
- Decreases number of generated test cases

Combinational Function test

- Intent
 - Design test suite for behaviors selected according to combinations of state and/or message parameters
- Context
 - Behavior of method: **select** one of many distinct actions based on many combinations of input parameters or has complex logic on input parameters
 - The *assignCarrier* of class *Shipment* determines which carrier to use for a shipment based on the shipment's weight, size, material type and maximum transit time
 - A boolean accessor method returns true or false based on object state, and this response is determined by checking value combinations of 3 or more instance variable
 - A method implements a business rule modeled by a decision table or decision tree
 - How to choose input combinations to exercise selection logic adequately?

Fault Model

- Incorrect value assigned to a decision variable
- Incorrect or missing operator/variable in a predicate
- Incorrect structure in a predicate (dangling else, a misplaced semicolon, ...)
- Incorrect or missing default case
- Incorrect action
- Extra action(s)
- Structural errors in a decision table implementation: falling through (missing break in switch) or computing an incorrect value for an action selector
- ...

Strategy: Test Model - 1

- Combinational methods can be modeled with a decision table
 - The condition section lists the input combinations that result in different actions
 - The action section represents the expected response of the MUT
 - The condition cells may contain only Boolean-types expressions
 - Each unique combination of input variable values is a variant
- Can also use a decision tree

Strategy: Test Model - 2

TABLE 10.11 Decision Table Test Model for Reporter Methods in Class Triangle

	Variant										
	1	2	3	4	5	6	7	8	9	10	11
Condition											
a > 0	T	T	T	T	T	T	T	T	F	T	T
b > 0	T	T	T	T	T	T	T	T	T	F	T
c > 0	T	T	T	T	T	T	T	T	T	T	F
a + b > c	T	T	T	T	T	F	T	T	DC	DC	DC
a + c > b	T	T	T	T	T	T	F	T	DC	DC	DC
b + c > a	T	T	T	T	T	T	T	F	DC	DC	DC
a = b	T	T	F	F	F	DC	DC	DC	DC	DC	DC
b = c	T	F	T	F	F	DC	DC	DC	DC	DC	DC
a = c	T	F	F	T	F	DC	DC	DC	DC	DC	DC
Action											
is_isocles()	F	T	T	T	F	F	F	F	F	F	F
is_scalene()	F	F	F	F	T	F	F	F	F	F	F
is_equilateral()	T	F	F	F	F	F	F	F	F	F	F

Key: DC = don't care, T = true, F = false

Strategy: Test Procedure - 1

- Build decision table with Conditions/Actions
- At least one test for each variant
- **Non-boolean decision variables**
 - Exercise boundaries conditions
 - Generate several test cases
 - How to be systematic?

TABLE 10.12 Test Cases for Triangle Reporter Methods in Class Triangle

Variant	Test Case			Expected Result		
	a	b	c	isEquilateral()	isIsosceles()	isScalene()
Equilateral	1	1	1	T	F	F
	1	33	33	T	F	F
	1	100	100	T	F	F
Isosceles	2	3	2	F	T	T
	3	40	5	F	T	T
	4	100	100	F	T	T
Scalene	5	2	3	F	F	T
	5	6	3	F	F	T
	5	99	98	F	F	T
Nct a triangle	6	24	1	F	F	F
	7	99	42	F	F	F
	8	2	1	F	F	F
	9	0	20	F	F	F
	10	16	0	F	F	F
	11	8	8	F	F	F
		0	0	F	F	F
	null	3	3	F	F	F

Strategy: Test Procedure - 2

- Problem when variant contains boundary conditions
 - Values chosen for each variable in each variant
 - Not systematic
 - condition $(a == b) \rightarrow (a_n, a_n + 1), (a_n, a_n)$ and $(a_n, a_n - 1)$
 - Similar for $(a == c)$ and $(b == c)$
- Solution
 - Apply domain analysis
 - For each variant build a domain matrix
 - Do not repeat test cases
 - An ON point in one variant may be an OFF point in another

Entry and Exit Criteria

- **Entry criteria**

- Small Pop

- **Exit criteria**

- Produce every action at least once
- Force each exception due to incorrect input (if any) at least once
- Branch coverage for MUT
- If polymorphic binding is used instead of a case statement to select an action, be sure that each possible binding is executed at least once

Consequences

- Design of the decision table often reveals design errors and omissions
- Detects faults that are incorrect response actions to test messages
- Faults resulting from the order of messages to other methods or faults corrupting object variables hidden by the MUT interface may not be shown

Known Uses

- Combinational logic has been applied for many years in hardware and software testing
- This technique is embedded in several commercial testing tools

Recursive Function Test

- Intent
 - Test recursive methods
- Context
 - Recursion bugs can easily hide from low-coverage testing
 - How can we have high confidence that a responsibility implemented through a recursive method is correct?
 - How to test the recursion part of the code?
 - Specify the model
 - A recursive method sends a message to itself
 - Descendent phase
 - Ascendant phase

Recursive Method Model

```
int factorial(int n) {  
    assert(n >= 0);           // pre-condition  
    if (n == 0)  
        return 1;           // base case  
    else {  
        int fact = n * factorial(n-1); // recursive case  
        assert (fact > 0);         // post-condition  
        return fact;  
    }  
}
```

Fault Model - 1

- Execution continues when the precondition is violated for some illegal starting point or during the descendent phase
- Base case is omitted or is miscoded
 - \geq instead of $==$
 - Value returned by base case is incorrect
- Recursive case expression implements an incorrect algorithm
- Incorrect arguments appear in the recursive case message
- Execution continues when the postcondition is violated during the ascendant phase
- Fails to handle exceptions

Fault Model - 2

- Recursion to traverse data structures
 - Incorrectly initializes or corrupts the data structure
 - Does not correctly traverse the data structure when it is instantiated with a boundary case
- May also have space and time faults
 - Allows a recursion depth that exceeds available runtime memory
 - Has a non-linear runtime, causing real-time deadline to be missed

Strategy: Test Model - 1

- The test model should define:
 - Base case
 - Recursive case
 - Pre-conditions for the initial call
 - Post-condition
 - All descent-phase calls
 - All ascent-phase bindings

Strategy: Test procedure

- Test suite should contain:
 - Attempt to violate the precondition in the initial call and at least once in the descent phase
 - Attempt to violate the postcondition at least once in the ascent phase
 - Test boundary cases on depth: zero, one and maximum
 - Attempt to force all exceptions in the server objects or the environment on which the method depends
 - Use domain analysis to choose arguments for methods with multiple arguments (can also use *Category Partition*)
 - Use data structure states for recursive methods that traverse complex data structures
 - Apply Category-partition to determine the interesting states

Example – Java binarySearch

Where is the bug?

```
public static int binarySearch(int low, int high, int a[], int key) {
```

```
    if (low == high)
        return low;
```

```
    int m = (low+high)/2;
```

```
    int midval = a[m];
```

```
    if (midval < key)
```

```
        return binarySearch(m + 1, high, a, key);
```

```
    else if (midval > key)
```

```
        return binarySearch(low, m - 1, a, key);
```

```
    return low;
```

```
}
```

```
int m = low / 2 + high / 2;
```

```
int m = low + ((high - low) / 2);
```

```
int m = (low + high) >>> 1;
```

Bug can be caught applying Recursive function test

- Case test of maximum recursion

Entry and exit criteria

- Entry criteria
 - Small Pop
- Exit criteria
 - Null, Singleton and Maximal cases
 - Attempted violation of pre-condition in initial call and during the descent-phase
 - Attempted violation of post-condition during the ascendent-phase
 - Invariant Boundaries defined for values of multiple arguments and/or the states of data structures traversed
 - Worst-case runtime given system load and maximum depth

Consequences and Known uses

- Consequences
 - Requires no more than two dozen test cases
 - Reveals faults
 - that result in incorrect method evaluation for a given test message and state
 - but not those that occur only under certain sequence of messages to other methods or that corrupt instance variables hidden by the MUT's interface
- Known Uses
 - Some of these strategies are part of the oral tradition of LISP programming

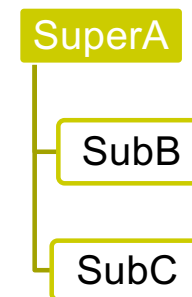
Polymorphic Message Test

- Intent

- Design a test suite for a client of a polymorphic server that exercise all client bindings to the server

- Context

- A polymorphic method can be binded to several implementations
- Would you have confidence in code for which only a fraction of statements or branches were executed?



Method m()

- Defined in SuperA
- Overridden in SubB, SubC

```

class D {
    void f(SuperA a) {
        ...
        a.m();
        ...
    }
    ...
}
  
```

Fault Model - 1

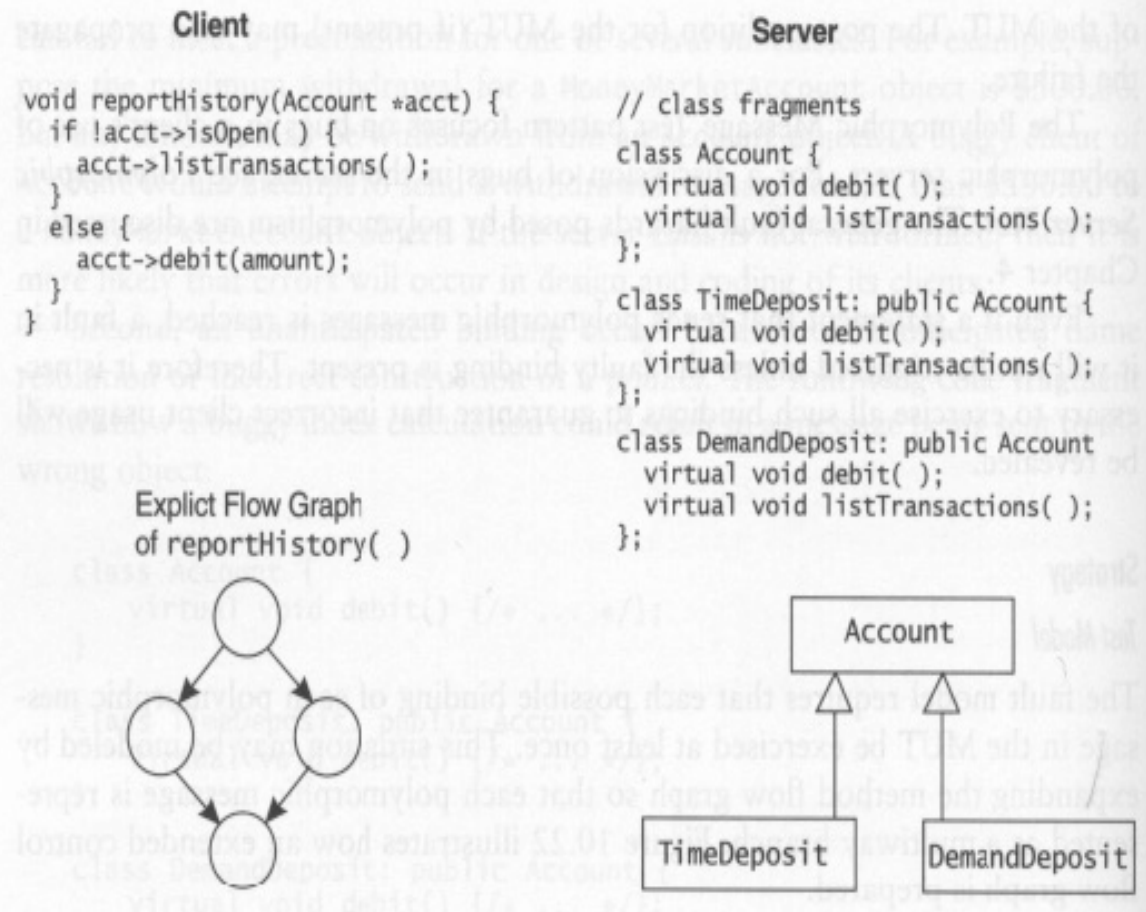
- Client fails to meet all preconditions for all possible bindings
 - Class MoneyMarketAccount imposes a minimum withdrawal of \$500
 - However, superclass Account allows any positive value
 - A buggy client of Account would attempt to send a withdrawal message for less than \$500 to a MoneyMarketAccount object

Fault Model - 2

- The implementation of a server class is changed or extended
 - Although clients have not been changed, the client code is rendered inconsistent because of the server revision
- To find faults must exercise all possible bindings
- Focus on bugs from the point of view of the client
 - See Polymorphic Server Test

Strategy: Test Model - 1

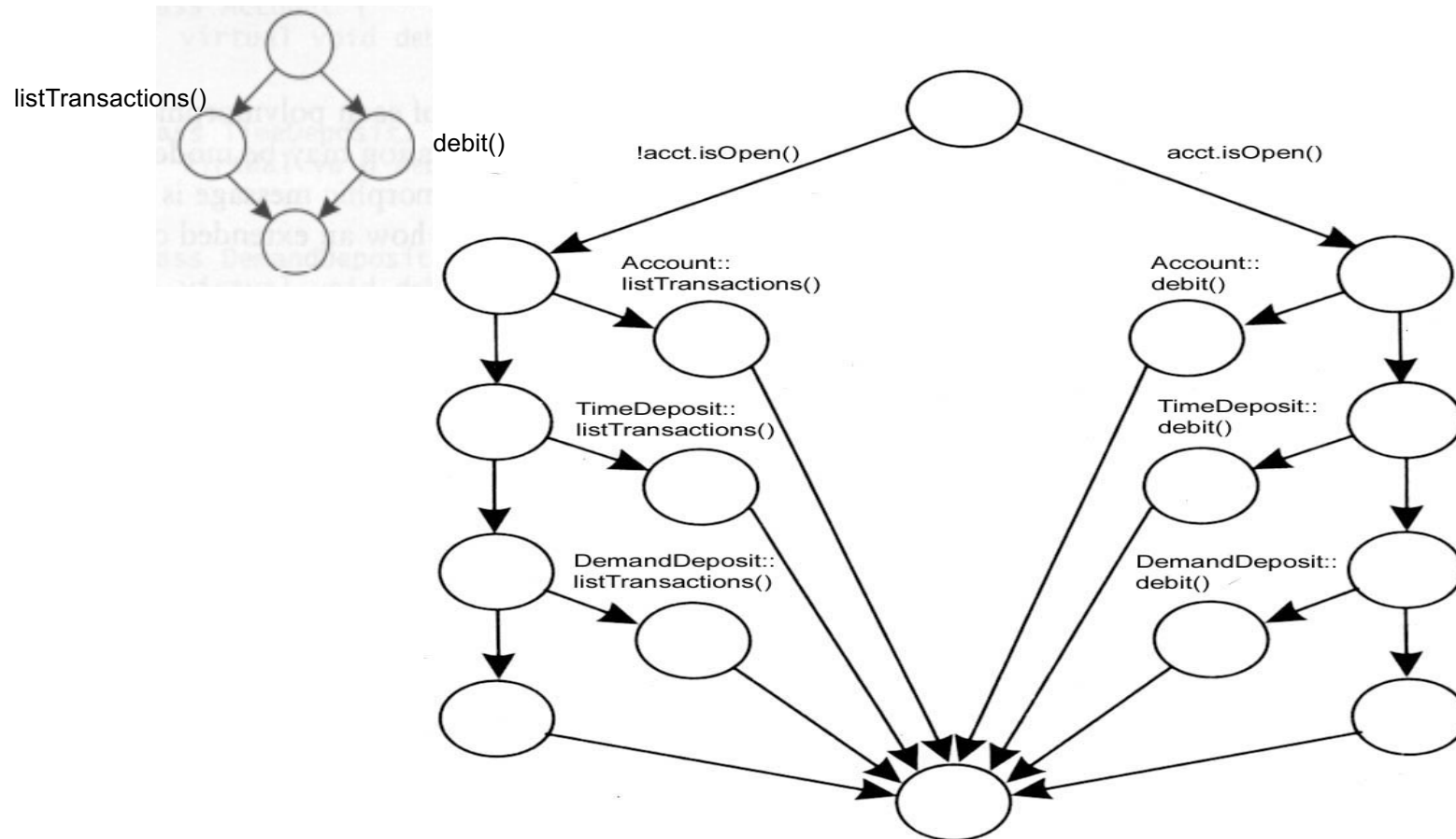
- Develop extended flow graph of MUT
- How?



Strategy: Test Procedure

1. Determine the number of candidate bindings for each message sent to a polymorphic server object
2. Expand segment with multiway branch sub-graph for each segment that has a polymorphic message
 - Add two nodes for each binding: a branch node and sequential node
 - Add a final, catch-all node to represent runtime binding error
3. Draw the test cases based on this model that exercise all branches

Strategy: Model Test - 2



Entry and Exit Criteria

- **Entry criteria**

- Small Pop
- Server class should be stable

- **Exit criteria**

- Achieve branch coverage of extended message flow graph

Consequences

- Breaks the black-box strategy
- Reveals many bugs in the client usage of the server class
- Should be used together with a responsibility-based pattern
- Requires analysis of server class hierarchy to determine all possible bindings