Subsystems

Chapter 12
Incremental Testing

Method scope testing:
1. Integrate parts(statements) and integration testing
2. Responsibility-based testing

Class scope testing:
1. Integrate parts(methods, super classes) and integration testing
2. Responsibility-based testing

Subsystem scope testing:
1. Integrate parts(classes) and integration testing
2. Responsibility-based testing

Complexity of Responsibility increases

Code coverage increases
What is a subsystem?

- A subsystem is composed of several classes and is a logical or physical part of an application system.
- Any testable collection of classes, objects, components, and modules.
- Main characteristics:
  - It is executable and testable as a whole.
  - It implements a cohesive set of responsibilities.
  - It does not provide all of the functionality of the application under test.
Subsystem Implementations

- Small cluster
  - It is composed of several tightly coupled classes
  - Testing its constituent classes in isolation is impractical
  - The cluster head is a single class that uses the others as instance variables or as message parameters
  - We can treat a cluster as a single class
Subsystem Implementations

● Pattern participants
  ● The implementation of a software design pattern can often be tested as a subsystem

● Large Cluster
  ● A large cluster is a group of related classes or related clusters

● Build Group
  ● A collection of software components processed by a linkage editor to generate an executable

● Task Group
  ● A collection of build groups that is linked as an executable process or a collection of build groups that generate processes that do something interesting
Why Test at Subsystem Scope?

- It is a precondition for testing at system scope
- It is a practical necessity to keep scope and complexity at a manageable level
- It supports internal releases during incremental, iterative development
- Use cases can model requirements for any subsystem with a public interface
Why Test at Subsystem Scope?

- Some subsystems must be shown to work before additional funding will be committed to a project
- The testing budget or schedule is limited or reduced
- The dependability requirements of some subsystems are much higher than those of other subsystems, and more extensive testing is indicated
- The target environment of a subsystem is unique or different or must be simulated apart from the rest of the application system
- ...
Subsystem Testing

- The test plan of a subsystem must answer two questions:
  - What features and capabilities should be tested?
  - How should the test plan and test suites be organized?
- The key problem of subsystem testing lies in developing a testable model of aggregate behaviour.
Subsystem Test Design Patterns

- Class Association Test shows how to design a test suite that will exercise the associations defined in a class or object model
- Controlled Exception Test shows how to design a test suite that will exercise exception handling
- Round-trip Scenario Test shows how to design a test suite that will cover all event-response paths in a sequence diagram
- Mode Machine Test shows how to model the aggregate state behaviour of a cluster and develop a state-based test suite
Class Association Test

- **Intent**
  - Design a test suite to verify the implementation of required associations among classes

- **Context**
  - Classes are typically responsible for maintaining associations with other classes
  - Provides a systematic approach to transform association multiplicity parameters into test cases
Association Class Test - Example

Class Model for Dogpatch policy

Person

0..1

Owns

0..*

Dog

Has Owner

Class Model for Kaynine policy

Person

1..1

Owns

2..14

Dog

Has Owner
Fault Model

- Association implementation
  - May be difficult
    - Several possible implementations
    - May need persistence

- Kind of faults
  1. Incorrect multiplicity
  2. Update anomaly
  3. Delete anomaly
  4. Missing link
  5. Wrong link
Fault Model

- These faults can be reached by exercising the code that implements the association constraints.
- This code will lie on a path that will be executed when boundary values of the constraints are input.
Test Model - Multiplicity Test Sets

- Multiplicity parameters specify boundary conditions for the acceptable mappings between two objects

**TABLE 12.3** Canonical Boundary Conditions for Legal Multiplicities

<table>
<thead>
<tr>
<th>Multiplicity of A to B</th>
<th>Boundary Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>$A:n(B) \geq 0$</td>
</tr>
<tr>
<td>0</td>
<td>$A:n(B) = 0$</td>
</tr>
<tr>
<td>1</td>
<td>$A:n(B) = 1$</td>
</tr>
<tr>
<td>42</td>
<td>$A:n(B) = 42$</td>
</tr>
<tr>
<td>0..0</td>
<td>$A:n(B) = 0$</td>
</tr>
<tr>
<td>0..1</td>
<td>$A:n(B) \geq 0$</td>
</tr>
<tr>
<td>0..24</td>
<td>$A:n(B) \geq 0$</td>
</tr>
<tr>
<td>0..*</td>
<td>$A:n(B) \geq 0$</td>
</tr>
<tr>
<td>1..1</td>
<td>$A:n(B) = 1$</td>
</tr>
<tr>
<td>1..24</td>
<td>$A:n(B) \geq 1$</td>
</tr>
<tr>
<td>1..*</td>
<td>$A:n(B) \geq 1$</td>
</tr>
<tr>
<td>2..14</td>
<td>$A:n(B) \geq 2$</td>
</tr>
<tr>
<td>42..4096</td>
<td>$A:n(B) \geq 42$</td>
</tr>
</tbody>
</table>

$A:n(B) \leq M$
### Multiplicity Test Sets – Example 1

<table>
<thead>
<tr>
<th>Associations</th>
<th>Boundary Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person owns dog 0..*</td>
<td>Person:n(Dog) &gt;= 0 Person:n(Dog) &lt;= M</td>
</tr>
<tr>
<td>Dog has owner 0..1</td>
<td>Dog:n(Person) &gt;= 0 Dog:n(Person) &lt;= 1</td>
</tr>
</tbody>
</table>

- Draw the domain matrix concerning the associations
- some Off impossible points and *

<table>
<thead>
<tr>
<th>Association/Condition/Type</th>
<th>Number of Dogs</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Person:n(Dog) &gt;= 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Person:n(Dog) &lt;= M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td></td>
</tr>
<tr>
<td>Typical</td>
<td>In</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5342</td>
</tr>
<tr>
<td></td>
<td></td>
<td>300</td>
</tr>
</tbody>
</table>

|                              | Dog:n(Person) >= 0 |                |
|                              | Off              |                 |
|                              |                 | 0               |
|                              |                 | -1              |
|                              |                 |                 |
| Typical                      | In              | 1               |
|                              |                 | 1               |
|                              |                 | 1               |
|                              |                 | 1               |

- ➪- Accept
- ➪- Reject
- ~- Infeasible
## Multiplicity Test Sets – Example 2

<table>
<thead>
<tr>
<th>Associations</th>
<th>Boundary Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person owns dog</td>
<td>Person:n(Dog) &gt;= 2, Person:n(Dog) &lt;= 14</td>
</tr>
<tr>
<td>Dog has owner</td>
<td>Dog:n(Person) == 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Association/Condition/Type</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Number of Dogs</td>
<td></td>
</tr>
<tr>
<td>Person:n(Dog) &gt;= 2</td>
<td>ON</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td>Person:n(Dog) &lt;= 14</td>
<td>ON</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td>Typical</td>
<td>IN</td>
</tr>
<tr>
<td>Number of Persons</td>
<td></td>
</tr>
<tr>
<td>Dog:n(Person) == 1</td>
<td>ON</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
</tr>
<tr>
<td>Typical</td>
<td>IN</td>
</tr>
</tbody>
</table>

Expected Result:

★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
Testing for Update/Delete Bugs

- A common bug is that only one instance is changed but all instances should have been modified
- Setup a owner with 5 dogs (for example)
- Design tests to attempt:
  - Update bug
  - Delete dependency
    - Application specific behaviour
  - Delete side-effect
Automation

- Anomaly and multiplicity testing may be combined
- The subsystem under test may be exercised via an API driver interface
- If the associations are embedded within a larger system, exercising all of them may prove difficult
- If the instances of collections are stored in a DBMS setting up test cases with a database loader utility is usually easier than entering them through the user interface
Entry and Exit Criteria

● Entry Criteria
  ● Each class in the scope of the model has passed a alpha-omega cycle
  ● Keyed collections are typically used to implement associations. These classes should have been individually tested with Quasi-modal Class Test

● Exit Criteria
  ● Each multiplicity on point test passes
  ● Each multiplicity off point test passes
  ● A deletion anomaly test passes
  ● An update anomaly test passes
Consequences

● With n associations in the system under test, there will be at most 8n basic multiplicity tests
  ● 2 test cases for each minimum and maximum boundary on the A:B association
  ● 2 test cases for each minimum and maximum boundary on the B:A association
● Requires a model of the class association in the subsystem under test
● Achieving sufficient control of the system under test may prove difficult
Controlled Exception Test

- **Intent**
  - Develop a test suite that will exercise exception handlers and provide a systematic means to generate all exceptions

- **Context**
  - How can we control application servers and the target environment to achieve adequate coverage of a client’s exception-handling code?
Context

- Applies to implementations expected to handle application-specific exceptions or exceptions raised by the target environment
  - Computational exceptions (arithmetic exceptions, underflow/overflow)
  - I/O errors (device not ready, parity check)
  - File errors (empty, overflow, missing)
  - Main storage management (allocate, deallocate, garbage collection)
  - Task management (process start, suspend, resume, terminate, inter process communication)
Fault model

- An exception is improperly passed from a server to a client
- An exception propagates out of scope
- An application-specific exception handler is missing or incorrect
- An exception is ignored and results in an immediate process/task abend
- An exception is ignored and results in a mysterious failure in unrelated code.
- An exception handler creates an adverse side effect
Fault model - 2

- The implementation under test does not meet target environment requirements for recognizing and handling exceptions.
- The recovery attempted by an exception is incorrect or incomplete.
- The exception structure loops; that is, the exception handler retries an operation without limitation.
- The wrong exception is thrown owing to a control flow bug or incorrect name.
- The wrong catch block is activated, possibly because of an incorrect name resolution/scoping error or incorrect build/configuration error.
Test Model

- The behaviour of an application-specific exception handler can be quite complex if it attempts restart or recovery
  - We need an appropriate test model: state machine or combinational logic or Category-Partition.

- Exception testing can be achieved in three ways:
  - Activation
  - Simulation
  - Inducement
Exception Activation/Simulation

- Exceptions are activated through input and state values supplied by test cases
  - If the server is not part of the application, system exception activation may be difficult or impossible
- Exceptions are simulated by using controllable wrappers for application servers or virtual machine services allowing
  - Check the call order or/and the parameters
  - Call the original function
  - Set the return value
  - Throw an exception
  - Change the output parameters
  - Call the original function with modified parameters
  - …
Exception Inducement

- Exceptions are induced by directly manipulating the IUT or the target environment
  - Mistune (cut down the available storage, disk space, etc)
  - Cripple (remove, rename, disable, delete, or unplug necessary resources)
  - Pollute (selectively corrupt input data, files, etc)
  - Mutate (run the SUT under a debugger and alter the object code or data, etc)
    - Note: Make copy of the source code under test. Alter copy to force exceptions.
- Requires time-consuming manual setup, is nonrepeatable
Automation

- Stubs are commonly used to simulate exceptions
- Scripts can be developed to generate anomalous situations to trigger exceptions
Automation – Wrapper case
Entry and Exit Criteria

- **Entry Criteria**
  - It usually makes sense to do exception handling testing after responsibility testing of the server and client has been completed

- **Exit Criteria**
  - Each server exception should be raised at least once, and each handler in the IUT should be activated at least once
  - If the handler is nontrivial, coverage for the responsibilities of the exception handler is achieved
Consequences

- Design of an exception-handling test suite requires a technical understanding of which exceptions can be raised by the target environment and which application-specific exception strategy applies.
- Target environment exceptions are usually well documented but application-specific exception handling typically goes undocumented and remains embedded in the code.
- In some situations, testing exception handling in an isolated test environment may not be possible.
Round-trip Scenario Test

- **Intent**
  - Extract a control flow model from a UML Sequence Diagram and develop a path set that provides minimal branch and loop coverage

- **Context**
  - A round-trip scenario is a complete stimulus-response path on a Sequence Diagram
    - A Sequence Diagram includes many such scenarios
    - How can we ensure that all scenarios are tested?
Sequence Diagram- ATM
Fault Model

- Incorrect or missing output
- Correct message passed to the wrong object
- Incorrect message passed to the right object
- Correct exception raised, but caught by the wrong object
- Acceptance of incorrect selection or object
- Action missing on external interface
- Missing function/feature in a participant
- …

The test model calls out all round-trip paths to ensure that these bugs are reached
Test Model

- Sequence diagrams often strike a good balance between too little or too much detail.
- They can model interfaces to large clusters or other subsystems, maintaining focus on SUT.
- Sequence Diagrams are not ideal test models:
  - They do not provide a good representation of repetitive, recursive, or conditional sequences.
  - Lack of precision.

- Solution:
  - Transform Sequence Diagram into a flow graph.
Test Procedure

1. Translate the sequence diagram into a flow graph
2. Identify paths to test from the flow graph
3. Identify special cases
4. Identify inputs and states needed to cause a particular path to be taken
1. Translate the sequence diagram
2. Identify paths to test

- Loops and branches result in many possible entry-exit paths
  - Number of paths can easily reach astronomical numbers

- Approach
  - Achieve decision/iteration coverage
    - Every branch is taken at least once
  - Each loop is used at least in three ways:
    - Bypass, do one iteration and do the maximum number of iterations
  - Each path becomes a test case
2. Identify paths to test (cont)

**FIGURE 12.7** Nonlooping paths for ATM Sequence Diagram.
2. Identify paths to test (cont)
## 2. Identify paths to test (cont)

<table>
<thead>
<tr>
<th>Path</th>
<th>Comment</th>
<th>Path Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Longest path, all false. Bypass loop</td>
<td>!notATMCard, !isStolen, accountClosed, validPIN, !declineFee</td>
</tr>
<tr>
<td>2</td>
<td>Branch 1 true. Bypass loop</td>
<td>notATMCard</td>
</tr>
<tr>
<td>3</td>
<td>Branch 2 true. Bypass loop</td>
<td>!notATMCard, isStolen</td>
</tr>
<tr>
<td>4</td>
<td>Branch 3 true. Bypass loop</td>
<td>!notATMCard, !isStolen, accountClosed</td>
</tr>
<tr>
<td>5</td>
<td>Branch 4 true Bypass loop</td>
<td>!notATMCard, !isStolen ! accountClosed, validPIN, declineFee</td>
</tr>
</tbody>
</table>
2. Identify paths to test (cont)
2. Identify paths to test (cont)

Round-trip scenario test cases path conditions

<table>
<thead>
<tr>
<th>Path</th>
<th>Comment</th>
<th>Path Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Longest path, all false. Bypass loop</td>
<td>!notATMCard, !isStolen !accountClosed, validPIN, !declineFee</td>
</tr>
<tr>
<td>2</td>
<td>Branch 1 true. Bypass loop</td>
<td>notATMCard</td>
</tr>
<tr>
<td>3</td>
<td>Branch 2 true. Bypass loop</td>
<td>!notATMCard, isStolen</td>
</tr>
<tr>
<td>4</td>
<td>Branch 3 true. Bypass loop</td>
<td>!notATMCard, !isStolen, accountClosed</td>
</tr>
<tr>
<td>5</td>
<td>Branch 4 true Bypass loop</td>
<td>!notATMCard, !isStolen !accountClosed, validPIN, declineFee</td>
</tr>
<tr>
<td>6</td>
<td>Same as path 1 up to the loop. Loop once</td>
<td>!notATMCard, !isStolen !accountClosed, !validPIN &amp;&amp; n== 1, validPin &amp;&amp; n== 2, !declineFee</td>
</tr>
<tr>
<td>7</td>
<td>Same as path 1 up to the loop. Loop max times</td>
<td>!notATMCard, !isStolen !accountClosed, !validPIN &amp;&amp; n== 1, !validPin &amp;&amp; n== 2, !validPIN &amp;&amp; n == 3, !validPIN</td>
</tr>
</tbody>
</table>
3. Identify special cases

- Exception handling paths
  - Controlled Exception Test
- Polymorphic servers
  - The yo-yo problem – it is practically impossible to identify all these paths
4. Identify inputs and states

- A path is followed when all the path conditions are met
- Test cases define the specific input and state needed to satisfy the path conditions
- How to identify test case values?
  - look at the predicates on each path
- Finally, determine expected result
Automation

- GUI scripting language
- API test driver
Entry and Exit Criteria

- **Entry Criteria**
  - The objects, components, or subsystems participating in the scenarios should have passed at least a minimal test suite or have been sufficiently reliable in actual use

- **Exit Criteria**
  - Each branch in a derived flowchart is taken at least once
  - Each loop is bypassed or iterated for minimum number of iterations
  - Each loop is iterated once and then exited
  - Each loop is iterated for the maximum number of iterations
Consequences

- Requires an **accurate** Sequence Diagram
- The minimum number of test cases depends on the structure of the implementation
- The round-trip heuristic will suffice for most practical problems
- The test model does not address the expansion of paths resulting from polymorphic bindings
Mode Machine Test

● Intent
  ● Develop a test suite for subsystems scope behaviour by analysing component behaviour

● Context
  ● Subsystems whose behaviour is sequentially constrained: the same sequence of external event does not always produce the same response
  ● Presence of a dominant control object