Testing Techniques

- Based on Pressman, Chapters 18-19 (VT – CS5704)
Testability

- **Operability**—it operates cleanly
- **Observability**—the results of each test case are readily observed
- **Controllability**—the degree to which testing can be automated and optimized
- **Decomposability**—testing can be targeted
- **Simplicity**—reduce complex architecture and logic to simplify tests
- **Stability**—few changes are requested during testing
- **Understandability**—of the design
What is a “Good” Test?

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be “best of breed”.
- A good test should be neither too simple nor too complex.
Internal and External Views

- Any engineered product (and most other things) can be tested in one of two ways:
  - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.
Software Testing

Methods

white-box methods

black-box methods

Strategies
White-Box Testing

... our goal is to ensure that all statements and conditions have been executed at least once ...
Why Cover?

- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability.

- We often believe that a path is not likely to be executed; in fact, reality is often counterintuitive.

- Typographical errors are random; it's likely that untested paths will contain some.
Exhaustive Testing

There are $10^{14}$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing

Selected path

loop < 20 X
Basic Path Testing

First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

number of enclosed areas + 1

In this case, \( V(G) = 4 \)
Cyclomatic Complexity

A number of industry studies have indicated that the higher $V(G)$, the higher the probability or errors.

modules

modules in this range are more error prone

$V(G)$
Basic Path Testing

Next, we derive the independent paths:

Since $V(G) = 4$, there are four paths

- Path 1: 1, 2, 3, 6, 7, 8
- Path 2: 1, 2, 3, 5, 7, 8
- Path 3: 1, 2, 4, 7, 8
- Path 4: 1, 2, 4, 7, 2, 4, ... 7, 8

Finally, we derive test cases to exercise these paths.
Basic Path Testing Notes

- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basic path testing should be applied to critical modules
Deriving Test Cases

- **Summarizing:**
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the set.
Example

i=1

total.input = total.valid = 0

sum = 0;

DO WHILE (v[i]<>-999 AND total.input < 100)

  total.input++
  IF v[i] >= min and v[i]<= max
    THEN total.valid++
    sum = sum + v[i]
  ENDIF
  i++

ENDDO

IF total.valid > 0
  THEN average = sum/total.valid
  ELSE average = -999
ENDIF

Paths:
1-2-10-11-13
1-2-10-12-13
1-2-3-10-11-13
1-2-3-4-5-8-9-2-...
1-2-3-4-5-6-8-9-2-...
1-2-3-4-5-6-7-8-9-2-...
Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
# Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>.8</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](attachment:tree.png)
Loop Testing

Simple loop

Nested Loops

Concatenated Loops

Unstructured Loops

These slides are designed to accompany Software Engineering: A Practitioner’s Approach, 7/e (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.
Black-Box Testing
Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.
Equivalence Partitioning

- **Equivalence partitioning** is an approach to black box testing that divides the input domain of a program into classes of data from which test cases can be derived.

- **Example**: Suppose a program computes the value of the function $\sqrt{(X-1)(X+2)}$. This function defines the following valid and invalid equivalence classes:
  
  - $X \leq -2$ valid
  - $-2 < X < 1$ invalid
  - $X \geq 1$ valid

- Test cases would be selected from each equivalence class.
Boundary Value Analysis is a black box testing technique that where test cases are designed to test the boundary of an input domain. Studies have shown that more errors occur on the "boundary" of an input domain rather than on the "center".

Boundary value analysis complements and can be used in conjunction with equivalence partitioning.
In previous example, after using equivalence partitioning to generate equivalence classes, boundary value analysis would dictate that the boundary values of the three ranges be included in the test data. That is, we might choose the following test cases (for a 32 bit system):

- $X \leq -2$:
  - $-2^{31}$, -100, -2.1, -2

- $-2 < X < 1$:
  - -1.9, -1, 0, 0.9

- $X \geq 1$:
  - 1, 1.1, 100, $2^{31} - 1$
Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.
**OO Testing**

- To adequately test OO systems, three things must be done:
  - the definition of testing must be broadened to include classes of errors introduced with object-oriented concepts
  - the strategy for unit and integration testing must change significantly, and
  - the design of test cases must account for the unique characteristics of OO software.
‘Testing’ OO Models

- Capitalize on model-based approaches
  - Use cases can form the basis of test cases
  - Execute sequence diagram scenarios
  - State machines provide formal definitions of inputs, modes, and desired behavior
- Review OO models with domain experts
  - Ensure models are valid for the real-world scenarios and system requirements
- Ensure consistency across the OO models
  - Requirements, static, and dynamic models
- Many modern OO modeling tools help this process
OO Testing Strategies

- **Unit testing**
  - the concept of the unit changes
  - the smallest testable unit is the encapsulated class
  - each method in a class should be tested as part of the unit

- **Integration Testing**
  - *Thread-based testing* integrates the set of classes required to respond to one input or event for the system
  - *Scenario-based testing* expands on thread-based testing to consider one or more input events captured in a use case scenario
    - During integration testing, we would consider a “white box” approach in which we know the specific objects participating in the scenario and how they should communicate

- **Validation Testing**
  - details of class connections disappear
  - test use cases from a black-box perspective
Handling OO Features

- Interface definitions are particularly important in OO
  - Verify desired parameters and return values and types
  - Differences in units are a common problem
- Reusable classes / components
  - Must verify reuse within proper context
- Dynamic binding
  - Probably the most difficult feature of OO to test
  - OO allows subclasses to be substituted for parent classes dynamically
    - Don’t know until run-time which class instance (and specific method) will be executed
    - Test cases must be designed to test all possible subclasses where parent classes are referenced
Some tools to aid testing

- **JUnit**
  - Assists unit testing with Java programs

- **IBM Rational Rhapsody**
  - Allows development of executable UML models

- **Promela / SPIN**
  - Model-checker used on many critical software applications

- And many others – just Google
  - Software applications are becoming much too complex to rely on testing by hand