## Empirical Aspects of Testing

<table>
<thead>
<tr>
<th>Pure white-box (structural)</th>
<th>Gray-box (some structural)</th>
<th>Pure black-box (behavioral)</th>
<th>Live tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development activity</td>
<td></td>
<td>Testing activity</td>
<td>Technical support, beta testing</td>
</tr>
</tbody>
</table>

*Technical support, beta testing*
Black-Box Testing Tools

- Plan & Checklists: tests to perform
  - With each test, pre-condition, post-condition

- Test matrix/spreadsheet: listing tests against use cases and potential error areas

- Logging capability
  - Note pad
  - Keystroke recorder (for playback)
  - Event logger
  - Screen recorder
  - Video camera

- Tracking Database
# Testing Matrix

<table>
<thead>
<tr>
<th>Tested by</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
<th>Test 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case 1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Case 2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Case 3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Test Result Categorization

- Area of defect
- Severity level of defect
- Follow up:
  - Responsibility
  - Time to fix
  - Lines of code affected
**10 Sample Severity Levels, with examples (Boris Beizer)**

- **Mild**: misspellings in output
- **Moderate**: misleading or redundant behavior
- **Annoying**: truncated names, etc.
- **Disturbing**: some transactions not processed
- **Serious**: lost transaction
- **Very serious**: incorrect output
- **Extreme**: frequent “very serious” errors
- **Intolerable**: data corrupted
- **Catastrophic**: shutdown
- **Infectious**: shutdown spreading to others
4 Possible Reaction Levels to failed tests

- **Defer**: fix as time permits
- **Schedule**: fix by some future date
- **Required**: fix before acceptance
- **Immediate**: fix before testing is continued
Testing Forms and Tracking

- Discrepancy report form
- Error investigation form
- Error reporting/tracking system/database
Discrepancy Report Form
(Sherry Pfleeger)

DISCREPANCY REPORT FORM

DRF Number:__________________________________________ Tester name:__________________________
Date: ____________________________ Time: ____________________________
Test Number: ____________________________
Script step executed when failure occurred: __________________________________________________________
Description of failure: ________________________________________________________________________
_______________________________________________________________________________________
_______________________________________________________________________________________
_______________________________________________________________________________________
Activities before occurrence of failure:
_______________________________________________________________________________________
_______________________________________________________________________________________
Expected results:
_______________________________________________________________________________________
_______________________________________________________________________________________
Requirements affected:
_______________________________________________________________________________________
_______________________________________________________________________________________
Effect of failure on test:
_______________________________________________________________________________________
_______________________________________________________________________________________
Effect of failure on system:
_______________________________________________________________________________________
_______________________________________________________________________________________
Severity level:
(LOW) 1 2 3 4 5 (HIGH)
ORIGINATOR: Joe Bloggs

BRIEF TITLE: Exception 1 in dps_c.c line 620 raised by NAS

FULL DESCRIPTION: Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above. (I think the re-enabling is a red herring.)

ASSIGNED FOR EVALUATION TO:

CATEGORISATION: 0 1 2 3 Design Spec Docn
SEND COPIES FOR INFORMATION TO:

EVALUATOR: 8/7/92

CONFIGURATION ID    ASSIGNED TO    PART

dpo_s.c

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

CONFIGURATION ID    IMPLEMENTOR/DATE    REVIEWER/DATE    BUILD/ISSUE NUM    INTEGRATOR/DATE

dpo_s.c v.10    AWJ 8/7/92    MAR 8/7/92    6.120    RA 8-7-92

COMMENTS:

CLOSED

FAULT CONTROLLER: 9/7/92
<table>
<thead>
<tr>
<th>Date</th>
<th>Sev</th>
<th>Priority</th>
<th>Risk</th>
<th>Owner</th>
<th>Estimated Start</th>
<th>Subsystem</th>
<th>Configuration</th>
<th>Close Date</th>
<th>Resolution</th>
<th>Root Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>7/21/99</td>
<td>1</td>
<td>1</td>
<td>Muhammad Zam</td>
<td>8/12/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>A.X.0.0.001</td>
<td>8/3/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/21/99</td>
<td>2</td>
<td>2</td>
<td>Chuck Chavall</td>
<td>7/28/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>7/30/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>3</td>
<td>3</td>
<td>Jenny Chung</td>
<td>8/12/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>A.Y.0.0.001</td>
<td>8/1/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/23/99</td>
<td>3</td>
<td>3</td>
<td>Muhammad Zam</td>
<td>7/24/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>A.Z.0.0.001</td>
<td>8/2/99</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>1</td>
<td>4</td>
<td>Chuck Chavall</td>
<td>7/31/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>B.Y.0.0.001</td>
<td>8/1/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>2</td>
<td>5</td>
<td>Jenny Chung</td>
<td>8/17/99</td>
<td>du du du</td>
<td>Tools</td>
<td>C.X.0.0.001</td>
<td>8/7/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/23/99</td>
<td>3</td>
<td>3</td>
<td>Lary Kanemetsu</td>
<td>8/14/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>C.Y.0.0.001</td>
<td>8/4/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/20/99</td>
<td>4</td>
<td>1</td>
<td>Frank Crunt</td>
<td>8/14/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>D.Z.0.0.001</td>
<td>8/5/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/22/99</td>
<td>5</td>
<td>1</td>
<td>Chuck Chavall</td>
<td>8/18/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>C.Z.0.0.001</td>
<td>8/5/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/22/99</td>
<td>3</td>
<td>1</td>
<td>Jenny Chung</td>
<td>7/28/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>8/4/99</td>
<td>Data</td>
<td>Documentation</td>
</tr>
<tr>
<td>7/23/99</td>
<td>2</td>
<td>2</td>
<td>Muhammad Zam</td>
<td>8/17/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>8/7/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/23/99</td>
<td>1</td>
<td>3</td>
<td>Chuck Chavall</td>
<td>7/29/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>8/2/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/23/99</td>
<td>1</td>
<td>2</td>
<td>Jenny Chung</td>
<td>8/13/99</td>
<td>du du du</td>
<td>Other</td>
<td>A.X.0.0.001</td>
<td>8/5/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/23/99</td>
<td>1</td>
<td>2</td>
<td>Lary Kanemetsu</td>
<td>7/27/99</td>
<td>du du du</td>
<td>Unknown</td>
<td>A.X.0.0.001</td>
<td>8/7/99</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>2</td>
<td>3</td>
<td>Frank Crunt</td>
<td>7/25/99</td>
<td>du du du</td>
<td>NA</td>
<td>A.Y.0.0.001</td>
<td>8/7/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>3</td>
<td>4</td>
<td>Chuck Chavall</td>
<td>8/13/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.Z.0.0.001</td>
<td>8/2/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>1</td>
<td>4</td>
<td>Jenny Chung</td>
<td>8/19/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>C.Z.0.0.001</td>
<td>8/1/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/21/99</td>
<td>1</td>
<td>2</td>
<td>Muhammad Zam</td>
<td>7/27/99</td>
<td>du du du</td>
<td>Tools</td>
<td>C.X.0.0.001</td>
<td>7/30/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/19/99</td>
<td>1</td>
<td>3</td>
<td>Chuck Chavall</td>
<td>7/27/99</td>
<td>du du du</td>
<td>File</td>
<td>C.Y.0.0.001</td>
<td>8/1/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/22/99</td>
<td>2</td>
<td>4</td>
<td>Jenny Chung</td>
<td>8/18/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>C.Z.0.0.001</td>
<td>8/1/99</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>7/21/99</td>
<td>3</td>
<td>5</td>
<td>Lary Kanemetsu</td>
<td>7/22/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>D.Z.0.0.001</td>
<td>7/29/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/20/99</td>
<td>4</td>
<td>3</td>
<td>Muhammad Zam</td>
<td>8/13/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>C.Y.0.0.001</td>
<td>8/4/99</td>
<td>Documentation</td>
<td></td>
</tr>
<tr>
<td>7/20/99</td>
<td>1</td>
<td>2</td>
<td>Chuck Chavall</td>
<td>7/20/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>C.Z.0.0.001</td>
<td>8/2/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>4</td>
<td>3</td>
<td>Jenny Chung</td>
<td>8/12/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>D.Z.0.0.001</td>
<td>8/5/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/24/99</td>
<td>5</td>
<td>4</td>
<td>Lary Kanemetsu</td>
<td>8/8/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>8/2/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/28/99</td>
<td>1</td>
<td>5</td>
<td>Frank Crunt</td>
<td>8/22/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>A.X.0.0.001</td>
<td>8/7/99</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>7/29/99</td>
<td>2</td>
<td>4</td>
<td>Chuck Chavall</td>
<td>8/20/99</td>
<td>du du du</td>
<td>Other</td>
<td>A.X.0.0.001</td>
<td>8/11/99</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>7/31/99</td>
<td>2</td>
<td>3</td>
<td>Jenny Chung</td>
<td>8/5/99</td>
<td>du du du</td>
<td>Unknown</td>
<td>A.X.0.0.001</td>
<td>8/9/99</td>
<td>Functional</td>
<td></td>
</tr>
<tr>
<td>7/26/99</td>
<td>2</td>
<td>3</td>
<td>Chuck Chavall</td>
<td>8/17/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>B.Y.0.0.001</td>
<td>8/8/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/27/99</td>
<td>1</td>
<td>1</td>
<td>Muhammad Zam</td>
<td>8/1/99</td>
<td>du du du</td>
<td>User Interface</td>
<td>A.X.0.0.001</td>
<td>8/10/99</td>
<td>Process</td>
<td></td>
</tr>
<tr>
<td>7/30/99</td>
<td>2</td>
<td>2</td>
<td>Chuck Chavall</td>
<td>8/20/99</td>
<td>du du du</td>
<td>Tools</td>
<td>C.X.0.0.001</td>
<td>8/6/99</td>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>7/26/99</td>
<td>4</td>
<td>4</td>
<td>Muhammad Zam</td>
<td>8/14/99</td>
<td>du du du</td>
<td>Install/Config</td>
<td>C.Z.0.0.001</td>
<td>8/2/99</td>
<td>System</td>
<td></td>
</tr>
<tr>
<td>7/27/99</td>
<td>1</td>
<td>5</td>
<td>Chuck Chavall</td>
<td>8/12/99</td>
<td>du du du</td>
<td>Edit Engine</td>
<td>D.Z.0.000.001</td>
<td>8/4/99</td>
<td>Functional</td>
<td></td>
</tr>
</tbody>
</table>
Open/Closed Bug Tracking
General Testing Approach

- Create lists of potential problems and categorize them by area.

- Design repeatable tests, for proof of problems and problem resolutions.
Broad Categories for Errors

- Errors in interpreting requirements
- Errors in translating requirements into design, i.e. in programming
- Errors in implementing design
- Errors in the testing process itself
Program Behavior Views
(sets of behaviors taken over all inputs)

Specified (desired) behaviors

Observable behaviors (produced by program)
The Ideal

Specified (desired) behaviors

Observable behaviors

(coinciding)
Some aspects of a specification may be left arbitrary, unspecified, meaning either:

- The specification is to be regarded as incomplete, or
- Any behavior consistent with the specification will be accepted.
Testing asks questions: Can a behavior occur?

Specified (desired) behaviors

Observable behaviors

Tested Behaviors
(i.e. behaviors tested for)
Testing

Specified (desired) behaviors

Specified, but not tested

Not specified, but tested

Tested Behaviors

Observable behaviors
Which regions of the diagram indicate detected errors?
Black- vs. White Box
(recap)

- **Black-box** focuses on the **specification**: What is in the **spec** that the program doesn’t do?

- **White-box** focuses on the **program**: What does the **program** do that is not in the **spec**?

- Normally don’t rely on one or the other exclusively.
Black-Box Testing
Good Black-Box Test Plan

Specified (desired) behaviors

Observable behaviors

Tested, Specified Behaviors

(as large as is feasible)
Testing Example (3)

- A third reputable programmer has produced a self-contained program “triangle” that reads triples of numbers at a time and classifies them as to whether they are the lengths of the sides of some triangle, and if so, what kind.

- Negative side-length counts as a side with the length as absolute value of the specified length.

- The sides are in a specified range between 1E-150 and 1E150.
Black-Box Techniques

- Recall that “black box” means we do not get to see the code; we only have access to an installation of the product.

- Also called “functional testing” (vs. “structural testing”, which would be “white box”)

- Driven by requirements, use cases
Black-Box Techniques

- **Comparison Testing:**

  Test product side-by-side with a “gold standard”, a program believed to be correctly operating (such as an earlier version having most, if not all, of the features)
Black-Box Techniques

- **Garbage-In Test**: See if unusual input characters, click sequences, etc. can force the system into inconsistent states.
  - **Open-Book Test**

- **Test Monkey**: Provide large sequence of random inputs, with hopes of eliciting a crash of the software.

- **Data-Quantity Stress Test**: See if unusually large amounts of data cause nominal values to be exceeded, revealing untested overflow conditions, etc.

- **Soak Test**: Test the stability and “up-time” of integrated applications by running the program for a long time under varying conditions.
Quantitative Black-Box Testing
Equivalence Partitioning

- Partition the input space into classes, such that one may reasonably assume that:
  - The program behaves analogously for inputs in the same class.
  - A single test with a representative value from a class is sufficient to test for the entire class.
  - If representative detects fault then other class members will detect the same fault.
Black-Box Techniques

- **Equivalence Partitioning:**
  - Use a small number of test equivalence classes, rather than a large number of individual test data points.
  
  - The actual tests are representatives of the equivalence classes.
  
  - Partitioning based on their relative likelihood of exposing logic errors in the code.
  
  - Example (application-dependent): Partition a number space into:
    - less than 0
    - equal to 0
    - greater than 0, less than 100
    - greater than or equal to 100
Black-Box Techniques

- Equivalence Partitioning Examples (cont’d):
  - Partition a two-dimensional number space into \((x, y)\) where:
    - \(x < y\)
    - \(x == y\)
    - \(x > y\)
  - Why these?

- Partition a String space into
  - length = 0
  - length = 1
  - length = 2
  - length = 3
  - length between 4 and 1023
  - length greater than 1023
Black-Box Techniques

- How would you equivalence-partition the triangle program input space?

- What is the rationale?
### Decision Table

A “declarative” means to categorize input

- Partitions input space (triples of numbers) into equivalence classes.
- All inputs in a cell should have the same anticipated equivalence class.

<table>
<thead>
<tr>
<th>Input Categories</th>
<th>Output Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c a triangle?</td>
<td>N</td>
</tr>
<tr>
<td>a = b?</td>
<td>-</td>
</tr>
<tr>
<td>a = c?</td>
<td>-</td>
</tr>
<tr>
<td>b = c?</td>
<td>-</td>
</tr>
<tr>
<td>not a triangle</td>
<td>x</td>
</tr>
<tr>
<td>scalene</td>
<td>x</td>
</tr>
<tr>
<td>isosceles</td>
<td>x</td>
</tr>
<tr>
<td>equilateral</td>
<td>x</td>
</tr>
<tr>
<td>should not occur</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Categories</th>
<th>Output Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, b, c a triangle?</td>
<td>N</td>
</tr>
<tr>
<td>a = b?</td>
<td>Y</td>
</tr>
<tr>
<td>a = c?</td>
<td>N</td>
</tr>
<tr>
<td>b = c?</td>
<td>Y</td>
</tr>
<tr>
<td>not a triangle</td>
<td>Y</td>
</tr>
<tr>
<td>scalene</td>
<td>N</td>
</tr>
<tr>
<td>isosceles</td>
<td>N</td>
</tr>
<tr>
<td>equilateral</td>
<td>N</td>
</tr>
<tr>
<td>should not occur</td>
<td>N</td>
</tr>
</tbody>
</table>
### Decision Table Variant

- More condensed, based on logical equivalences
- Eliminate or reduce “should not occur” entries

<table>
<thead>
<tr>
<th>Input Categories</th>
<th>a, b, c a triangle?</th>
<th>a = b?</th>
<th>a = c?</th>
<th>b = c?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Categories</td>
<td>not a triangle</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>scalene</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>isosceles</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>equilateral</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Y</th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>Y</th>
<th>Y</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Black-Box Techniques

- **Boundary-value testing**: Pick test cases on, and near to, “natural” boundaries in data space, so as to test whether the program performs the correct classification of borderline cases.
Black-Box Techniques

- **Logarithmic testing**: Repeatedly split the data space into two, testing sample points in the upper and lower halves of the split, then repeat on the lower half only.
Black-Box Techniques

- **Cause-Effect Graphing**: Examine requirements specification for logical **chains of conditions**; develop test cases that check whether these chains are actually observed.

  - Example: “If the clipboard is empty, then the paste menu option should *not* be selectable.”

  - Therefore: Develop a test in which the clipboard should be empty, and check that the menu option is not selectable.
Black-Box Techniques

- **Cause-Effect Graphing**, possible relationships:
  - A condition *implies* an action
  - A condition *precludes* an action
  - Two actions are *mutually exclusive*
  - A *combination* (conjunction) of two conditions implies an action
  - etc.
Example Cause-Effect Graph

Causes

- C1
- C2
- C3
- C4
- C5

Effects

- E1
- E2
- E3

Conjunctions, negations, etc.

(disjunction implied)
White-Box Testing
Good White-Box Test Plan

Specified (desired) behaviors

Observable behaviors

Tested Behaviors
(as large as possible)
Approaches

- Qualitative/Empirical
- Structural: Things to make code more testable
- Quantitative: Mathematical analysis
Code Inspection and Walkthrough

- Used for:
  - Identifying errors
  - Identifying test cases
  - Helping yourself and others to understand how the program works.
Categories of Programming Errors

- Logic errors
- Pointer errors
- Numeric accuracy/precision/roundoff errors
- Input/output representation errors
- Data structure usage errors
- Memory errors
- User-interface errors (windows, etc.)
- Environment errors, such as misuse of file-system, devices, etc.
- Timing errors, such as in a real-time system
Exercise

- Each team take one category of error.
- List as many specific sub-categories of this error as you can.
- Are there testing approaches specific to this sub-category of error?

- Example: Logic Errors:
Logic Error Categories (1)

- Numeric and character boundary
  - Off-by-1
- Array-out-of-bound
  - Insufficient space allocated
  - Input or output buffer overflow
- Inequality comparison (used > instead of >= or <)
- Used ++ instead of --.
- Used pre-incrementation rather than post.
Logic Error Categories (2)

- Negation error
- Loop continuation criterion
  - Infinite loops
- Flag not cleared when used
- Flag, count, or sum not initialized
- Routine not reinitialized before subsequent use
- Dynamic type-casting exception
White-Box in Conjunction with Verification Techniques

- Verification is based on techniques for reasoning about programs.

- These techniques can also be used to simplify the number of white-box test cases.
Using Live Assertions

- Example from Triangles program:

  order(a, b);
  order(b, c);

  assert( a <= b && b <= c );

- The program will now tell us when the assumption is wrong.

- It will provide an indication of what has to be rethought.
Verification + W.B. Testing

- Could use verification to help fix the problem.

- Having *verified* the assertion

  ```
  assert( a <= b && b <= c );
  ```

  will testing be simplified?

- How?
Some Ideas for
More Effective Testability
Design for Test (DFT)

- **Instrument** your code as you build it; this could help with unit tests
  - Code-in traces, explanations, indications, ...
  - Being able to turn instrumentation on or off can help understand whether sub-systems are working correctly.
Example: C++ Instrumentation

class Trace
{
    static int currentLevel;

    static void trace(string message, int level)
    {
        if( level >= currentLevel )
            cerr << message << endl;
    }

    static void setLevel(int level)
    {
        currentLevel = level;
    }
};
Design for Test
(used extensively in hardware domain)

- Build **testing interfaces** into the code. These are interfaces that can be seen by the developer but not the user.

<table>
<thead>
<tr>
<th>User Interface</th>
<th>Testing Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Model” aspect of the product</td>
<td></td>
</tr>
</tbody>
</table>

- These interfaces allow greater automation in the testing process, since they can be driven by a testing program more readily than one requiring a user interface (such as a GUI or CLI).
Testing Interface

<table>
<thead>
<tr>
<th>User Interface</th>
<th>Testing Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Model” aspect of the product</td>
<td></td>
</tr>
</tbody>
</table>
Build Self-Test Routines into the Code

- **Self-tests** can employ data generators that will stress test the code.
Quantitative Aspects of Whitebox Testing Coverage Analysis
White-Box Coverage

- "Coverage" notion:
  - View the program as a directed graph (flowchart or data-flow diagram)
  - Develop (external or internal) tests that “cover”, i.e. exercise program to various degrees.

Specified (desired) behaviors \( \bigcap \) Observable behaviors

Tested Behaviors (= amount of “coverage”)
Pairwise Testing
(aka “All Pairs” Testing)
[Applies to BB and WB.]
If there are $N$ different attributes, with value set sizes $S_1$, $S_2$, ..., $S_N$, then the number of combinations is the product:

$$S_1 \times S_2 \times ... \times S_N$$

Assuming $S_i \geq 2$ for each $i$, we have a number of combinations exponential in $N$. 

Combinatorial Explosion
Example from Impro-Visor

- Note sequence display:
  - Note value (1, 2, 4, 8, 16, 32)
  - Dotted or not
  - #, b, natural, no accidental
  - Singlets, Triplets, 5-Tuplets
  - Time Signature (4 vs. 3)
  - Coloration or not
  - Clef (bass, treble, grand)
  - Ledger lines or not
  - Beamed or not
  - Tied or not

- Combinations = $6 \times 2 \times 4 \times 3 \times 2 \times 2 \times 3 \times 2 \times 2 \times 2 = \quad$
Example: Web media serving

- 5 different browsers {IE, Firefox, Netscape, Opera, Safari} (but some have multiple versions)
- 2 different media players {MediaPlayer, Real}
- 3 different webservers {IIS, Apache, Weblogic}
- 3 different operating systems {Widows, Apple, Linux}
- At least $5 \times 2 \times 3 \times 3 = 80$ combinations
  (not considering multiple release versions)
Pairwise Testing

- Pairwise testing is a **heuristic** for achieving good test coverage *without* trying all combinations:
  - Select a set of combinations such that:
    - For any pair of attributes, there is a combination that includes that pair.
Example

- Combinations = \{1, 2\} \times \{3, 4\}
- Every combination is needed

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Example

- Combinations = \{1, 2\} \times \{3, 4\} \times \{5, 6\}
- 4 of 8 possible suffice for pairwise testing
Does this alternate work?

- Combinations = \{1, 2\} \times \{3, 4\} \times \{5, 6\}

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>
Is this heuristic effective?

- Empirical examples from: http://www.pairwise.org/results.asp

[... ] a set of 29 pair-wise AETG tests gave 90% block coverage for the UNIX sort command. We also compared pair-wise testing with random input testing and found that pair-wise testing gave better coverage.

[D. M. Cohen et al., 1997]

Our initial trial of this was on a subset Nortel’s internal e-mail system where we able cover 97% of branches with less than 100 valid and invalid testcases, as opposed to 27 trillion exhaustive testcases.

[K. Burr and W. Young, 1998]
Checking for all-pairs coverage

- It is easy to check whether a proposed set of combinations covers all pairwise combinations.

- Generating such a set, while keeping the number of combinations to a minimum, is another matter.

- General area is “Orthogonal Arrays” (distinct from orthogonal matrices).

See also: http://www.research.att.com/~njas/oadir/
TConfig tool: Alan Williams
(http://www.site.uottawa.ca/~awilliam/)
More than 70% of bugs were detected with two or fewer conditions (75% for browser and 70% for server) and approximately 90% of the bugs reported were detected with three or fewer conditions (95% for browser and 89% for server). [...] It is interesting that a small number of conditions (n<=6) are sufficient to detect all reported errors for the browser and server software.

[R. Kuhn and M. J. Reilly, 2002]
Minimal Coverage

- Tests should exercise, at least once, every:
  - variable
  - assignment box
  - decision box
  - simple path through flow chart
Testing Based on Paths

Example of a single path to be covered by test (in red).

Insufficient to simply identify path; need to have data set that will exercise it.
Coverage “Basis”

- Infeasible to test *all* paths, etc.
- Instead, identify a “basis” from which all paths can be constructed.
- Make sure every element of basis is covered by *some* test
- Example: Basis could be set of all edges; Compute a set of tests that covers all.
Estimate the number of elements in a basis set of paths that will cover all edges.
D-D Path Nomenclature
(Ed Miller 1977)

- D-D = “Decision to Decision”
- Path between two decisions, that contains no decision itself
- Dependence among D-D paths, e.g. a variable defined in one path is referenced in another.
Classification of Structural Test Coverage

- $C_0$: Every statement tested
- $C_1$: Every D-D path tested
- $C_{ip}$: Every predicate & outcome tested
- $C_d$: $C_1$ + every inter-dependent pair of D-D paths tested
- $C_i^k$: Every path that contains up to $k$ repetitions of a loop (e.g. $k = 2$) tested
- $C_\infty$: Every path tested
“Baseline”-based method for constructing a basis

- Pick a single linear path through the program, the “baseline”.
- Pick the next path by taking decisions alternate to the baseline.
- Repeat picking other alternates to the alternates, etc.
- Eventually a basis number of tests will be reached.
Baseline Testing Method
(in McCabe’s Cyclomatic Tool)

Test Path 1 (baseline): 0 1 2 3 4 5 2 3 6 7 10 11 14 16 17
11( 1): string[index]=='A' ==> TRUE
13( 3): string[index]=='B' ==> TRUE
13( 3): string[index]=='B' ==> FALSE
18( 7): string[index]=='C' ==> FALSE
25( 11): string[index]!=='0' ==> FALSE

Test Path 2: 0 1 15 16 17
11( 1): string[index]=='A' ==> FALSE

Test Path 3: 0 1 2 3 6 7 10 11 14 16 17
11( 1): string[index]=='A' ==> TRUE
13( 3): string[index]=='B' ==> FALSE
18( 7): string[index]=='C' ==> FALSE
25( 11): string[index]!=='0' ==> FALSE

Test Path 4: 0 1 2 3 4 5 2 3 6 7 8 9 2 3 6 7 10 11 14 16 17
Complexity-Based Testing

- Construct the flowchart for the program.
- Calculate the graph complexity $C$.
- Find $C$ independent paths and corresponding test data for each.
- Execute program on test data.
- Check results with what is expected.
Classifying Loop Complexity (Informal)

Concatenated

Nested

Intersecting
Program Graph
Complexity Measures

- Used to determine *how much* testing required for a given sub-system

- Examples
  - McCabe Cyclomatic complexity
  - Halstead software metric
McCabe Cyclomatic complexity for a program graph

- Popular, but theoretically questionable
- $\nu(G) = e - n + p$
  - $e$ is the number of edges (arcs), which represent data processing boxes
  - $n$ is the number of nodes, which represent points where edges are connected
  - $p$ is the number of separate parts (connected sub-graphs)
Example

Flowchart
Example

Flowchart

Corresponding Graph
Edge Correspondence

\[ \nu(G) = e - n + p = 6 - 5 + 1 = 2 \]
McCabe’s measure drawn from Cyclomatic Number of a Graph

- Defined by Berge (*Theory of graphs and its applications*, 1962), motivated by Euler
- Originally defined for *undirected* graphs
- \( \nu(G) = \text{edges} - \text{nodes} + \text{separate parts} \)
- **Property**: *Connecting* two nodes increases \( \nu \) value by 1 if there was a path between the two nodes; otherwise it leaves \( \nu \) value the same.
Illustration of Property

Connecting two nodes increases $\nu$ value by 1 if there was a path between the two nodes; otherwise leaves $\nu$ value the same.

$\nu = 0 - 2 + 2 = 0$

$\nu = 1 - 2 + 1 = 0$

$\nu = 2 - 2 + 1 = 1$

$\nu = 3 - 2 + 1 = 2$

$\nu = 2 - 3 + 1 = 0$

$\nu = 3 - 3 + 1 = 1$

$\nu = 5 - 3 + 1 = 3$
Illustration of Property

Connecting two nodes increases $\nu$ value by 1 if there was a path between the two nodes; otherwise leaves $\nu$ value the same.

$\nu = 6 - 4 + 2 = 4$  
$\nu = 7 - 4 + 1 = 4$
Further Properties of the Cyclomatic Number of a Graph

- $\nu(G) = 0$ iff $G$ contains no undirected cycles (i.e. is “straight-line”).

- If $G$ is strongly connected (has only one component), then $\nu(G)$ is the maximum number of linearly-independent undirected cycles.

[This can be used to compute size of a basis.]
Linear Independent Circuits?

- Think of the edges of a graph as components of a vector.
- A circuit is abstracted by the number of times each edge is traversed in the (undirected) circuit.
- One circuit can be constructed from others by adding (mod 2) the corresponding vectors.
- (But not all vectors so-constructed correspond to executable paths, or even circuits themselves.)
Adding Circuits

\[ v(G) = e - n + p \]
\[ = 11 - 7 + 1 \]
\[ = 5 \]

\[ c1 = 1-4-9-11 = (1 0 0 1 0 0 0 0 1 0 1) \]
\[ c2 = 2-7-10-11 = (0 1 0 0 0 0 1 0 0 1 1) \]
\[ c3 = 1-4-9-10-7-2 = (1 1 0 1 0 0 1 0 1 1 0) \]

\[ c3 = c1 + c2 \]
\[ c1 \text{ and } c2 \text{ are linearly independent, etc.} \]

5 circuits form a basis
McCabe Complexities of Typical Program Graphs

- McCabe: If a separate part of a graph is not strongly connected, add a **phantom arc** from finish to start.

- Might think of this phantom arc as representing the **repeated use** of the program graph.

- **Alternatively**, use the *modified* formula on graph without adding arc:
  \[ \mu(G) = e - n + 2p \]

- Above, 2p accounts for one **phantom arc** in each of p separate parts.
Adding one phantom edge per separate part

\[ \mu(G) = e - n + p \] if phantom edges \textit{counted} in \( e \)

\[ \mu(G) = e - n + 2p \] if phantom edges \textit{not counted} in \( e \)
Complexities of Typical Program Graphs

\[ \mu(G) = e - n + 2p \]

Straight-line programs have \( \mu(G) = 1 \).

Two-way branch programs have \( \mu(G) = 2 \).
Complexities of Typical Program Graphs

\[ \mu(G) = e - n + 2p \]
\[ = 6 - 5 + 2 \]
\[ = 3 \]

Adding a branch increases \( e - n \) by 1, so by induction:

Simple k-way branch programs have \( \mu(G) = k \).

Three-way branch programs have \( \mu(G) = 3 \).
Complexities of Typical Program Graphs

Complexity formula: \( \mu(G) = e - n + 2p \)

- Simple 'while' programs have \( \mu(G) = 2 \).
- Nested programs have the same complexity.
- Intersecting programs have the same complexity. This may be misleading.

For a simple 'while' program:
\[
\mu(G) = 4 - 4 + 2 = 2
\]

For a nested program:
\[
\mu(G) = 8 - 7 + 2 = 3
\]

For an intersecting program:
\[
\mu(G) = 8 - 7 + 2 = 3
\]
Complexities of Typical Program Graphs

\[ \mu(G) = e - n + 2p \]

\[ = 9 - 7 + 2 \]

\[ = 4 \]

Simple k-tuply nested programs have \( \mu(G) = k + 1 \).
Complexities of Typical Program Graphs

\[ \mu(G) = e - n + 2p \]
\[ = 12 - 10 + 2 \]
\[ = 4 \]

k-section “lattices” have \( \mu(G) = k+1 \).

Somehow this seems to hide the possible “combinatorial explosion.”
Uses of complexity $\mu(G)$

- $\mu(G)$ can be used to indicate the **minimum number of test cases required**.

- Keep $\mu(G)$ small (e.g. $< 10$) for “understandable” programs.

- Certain constructs (e.g. switch) may be exempted from the count in some organizations.
Module: euclid
Basis Test Paths: 3 Paths

Test Path B1: 0 1 5 6 7 11 12 13
     8( 1): n>m ==> FALSE
     14( 7): r!=0 ==> FALSE

Test Path B2: 0 1 2 3 4 5 6 7 11 12 13
     8( 1): n>m ==> TRUE
     14( 7): r!=0 ==> FALSE
     14( 7): r!=0 ==> FALSE

Test Path B3: 0 1 5 6 7 8 9 10 7 11 12 13
     8( 1): n>m ==> FALSE
     14( 7): r!=0 ==> TRUE
     14( 7): r!=0 ==> FALSE

\( \mu(G) = 6 \)
Simplified Complexity Measures

- **Counting binary predicate tests only:**
  \[ \eta(G) = \text{predicates} + 1 \]

- **Counting regions** = \( e - n + 2p \)
  (from Euler's formula: \( N-E+F=2 \), \( F = \text{faces or regions} \)
Counting Regions

regions numbered
Use of $\mu(G)$ in Integration Testing

- Composition of modules' complexity: what is $\mu$ of overall system in terms of individual $\mu$'s?

![Diagram showing subsystems 1 and 2 with no connection and $\mu_1$, $\mu_2$, and $\mu$]
Use of $\mu(G)$ in Integration Testing

- Composition of modules’ complexity: what is $\mu$ of overall system in terms of individual $\mu$’s?

\[ \mu_1(G) = e_1 - n_1 + 2p_1 \quad \mu_2(G) = e_2 - n_2 + 2p_2 \]

\[ e = e_1 + e_2 \]
\[ n = n_1 + n_2 \]
\[ p = p_1 + p_2 \]

\[ \mu = \mu_1 + \mu_2 \]
Use of $\mu(G)$ in Integration Testing

- What if connected by one edge?

$$\mu = \mu_1 + \mu_2 + 1 - 1$$

one fewer separate part
one more edge

So $\mu = \mu_1 + \mu_2$, even if the subsystems are singly connected.
“Design-Reduction” Technique: Lumping hierarchically
"Design-Reduction" Technique

Original graph, $v(G) = 5$

Intermediate graph, after rules 3 and 4 have been applied

Reduced graph, $v(G) = 2$

○ = Non-call node  ● = Call node
Testing Based on Data Flow
Data Flow Testing

- As an alternative to basing tests on control paths, base it on data paths.

- In fact, examining data paths can reveal problems that control paths cannot.
“Definitions” and “Uses”

● “Definitions” for a variable:
  ● Creating a variable and initializing it
  ● Assigning a value to a variable
  ● Reading the value as input
  ● Initializing the variable as a parameter to a method
“Definitions” and “Uses”

- “Uses” for a variable:
  - The variable appears in an expression, such as:
    - RHS of an assignment
    - Conditional or while
    - Function call
  - The value of the variable is used in an output statement.
Definitions and Uses

<table>
<thead>
<tr>
<th>Stmt. No.</th>
<th>Statement sequence</th>
<th>Definition</th>
<th>Use</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>u = 5;</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x = u + 9;</td>
<td>x</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y = x * x;</td>
<td>y</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>x = u + 11;</td>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>y = x * y;</td>
<td>y</td>
<td>x, y</td>
<td>y</td>
</tr>
</tbody>
</table>

Note that a re-assignment is considered a new definition. It is said to “kill” the old one.
A DU (Definition-Use) Path for a variable is a path through a contiguous sequence of statements, that:

- Begins with a definition of the variable
- Ends with a use of the variable

A DC (Definition-Clear) path is a DU path that also:

- Has no intervening definitions (but may have intervening uses).
Possible Testing Criteria

- **All-Defs** testing:
  - For every variable definition, test at least one DU path from that definition.

- **All-Uses** testing:
  - For every variable definition, and for every use, test at least one DU path between the definition and use.
Coverage Inclusions

All-Paths

All-DU-Paths

All-Uses

All-C-Uses
Some-P-Uses

All-C-Uses
Some-P-Uses

All-Defs

All-P-Uses

All-Edges

All-Nodes

P-use = Predicate Use
C-use = Computation Use
# Coverage Assessment Tools

<table>
<thead>
<tr>
<th>Supported languages</th>
<th>Tool name</th>
<th>Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Statement/line/block</td>
</tr>
<tr>
<td>Java</td>
<td>Agitar [14]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Clover [16]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Cobertura [17]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>EMMA [20]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>JCover [24]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Koalog [25]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Jtest [26]</td>
<td>X</td>
</tr>
<tr>
<td>C/C++</td>
<td>Bullseye [15]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CodeTEST [18]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Dynamic [19]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Gcov [22]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Intel [23]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>C++test [26]</td>
<td>X</td>
</tr>
<tr>
<td>Java and C/C++</td>
<td>eXVantage [21]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>PurifyPlus [27]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>SD [28]</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>TCAT [29]</td>
<td>X</td>
</tr>
</tbody>
</table>
Example GCOV (Gnu)

- Works with gcc, cousin of gprof (timing profile)
- Annotates a copy of source files.
- Shows times executed for each statement
  - Other stats
- Accumulates times over several runs:
  - Typically different test sets
- Try 'man gcov'
- See also:
Testing - Code Coverage

- Used GNU’s gcov tool to determine code coverage of the current set of test cases.
- Statement coverage after running all test cases was 83%.
- Some surprising hotspots involved getting mnemonic names (~2 billion times) and string comparison on mnemonic names (~1.5 billion times).
### Example LCOV

**html graphical coverage from GCOV**

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Coverage</th>
<th>Instrumented lines</th>
<th>Executed lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/survey/winefromwinehgvws/include</td>
<td>65.9%</td>
<td>27 / 41 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/include/wine</td>
<td>49.4%</td>
<td>44 / 89 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/misc</td>
<td>28.4%</td>
<td>278 / 979 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/windows</td>
<td>45.7%</td>
<td>5753 / 12586 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/advapi32</td>
<td>26.3%</td>
<td>820 / 3118 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/advapi32/tests</td>
<td>89.9%</td>
<td>453 / 504 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/cabinet</td>
<td>0.1%</td>
<td>2 / 2821 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/comctl32</td>
<td>5.7%</td>
<td>1559 / 27195 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/comctl32/tests</td>
<td>97.4%</td>
<td>374 / 384 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/ddehelp</td>
<td>17.7%</td>
<td>702 / 3667 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/dkcmd</td>
<td>13.1%</td>
<td>712 / 5439 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/dkcmd/tests</td>
<td>25.8%</td>
<td>603 / 2240 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/dxerr8</td>
<td>16.4%</td>
<td>9 / 55 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/gdi</td>
<td>31.3%</td>
<td>3777 / 12073 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/gdi/enhafdrv/enhafdrv</td>
<td>22.0%</td>
<td>376 / 1264 lines</td>
<td></td>
</tr>
<tr>
<td>/home/survey/winefromwinehgvws/wine_dirty2/dlls/gdi/enhafdrv/enhafdrv/enhafdrv</td>
<td>22.0%</td>
<td>376 / 1264 lines</td>
<td></td>
</tr>
</tbody>
</table>
Differential Coverage Project
(HMC Clinic project sponsored by Google)

- What does a user run of the application cover that the test suite doesn’t?
- Use the answers to extend the test suite.
- Display graphically by augmenting LCOV.
Example: Testing Notepad

Test Suite

Notepad

Differential
## Summary Page

**Test Suite**

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Coverage</th>
<th>Instrumented lines: 305705</th>
<th>Executed lines: 101981</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/home/wine/dlls/user</code></td>
<td>0.0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/wine/include</code></td>
<td>65.9 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/wine/include/wine</code></td>
<td>49.4 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notepad**

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Coverage</th>
<th>Instrumented lines: 190562</th>
<th>Executed lines: 31949</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/advapi32</code></td>
<td>50.0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/comctl32</code></td>
<td>50.0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/commdlg</code></td>
<td>16.7 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compare**

<table>
<thead>
<tr>
<th>Directory name</th>
<th>Coverage</th>
<th>Instrumented lines: 190562</th>
<th>Executed lines: 5981</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/advapi32</code></td>
<td>0.0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/comctl32</code></td>
<td>0.0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/home/cpierog/build testsuite/dlls/commdlg</code></td>
<td>16.7 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When to Stop Testing?

Errors found

Effort Expended
There is no single, valid, rational criterion for stopping.

Furthermore, given any set of applicable criteria, how each is weighed depends very much upon the product, the environment, the culture and the attitude toward risk.
# Software Test Standards Reference

(Order from [http://www.12207.com/test1.htm](http://www.12207.com/test1.htm))

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS-7738-1</td>
<td>Specification for Information Systems Products Using SSADM--(Structured Systems Analysis and Design Method)</td>
</tr>
<tr>
<td>BS-7925-1</td>
<td>Software Testing - Vocabulary</td>
</tr>
<tr>
<td>BS-7925-2</td>
<td>Standard for Software Component Testing</td>
</tr>
<tr>
<td>FDA Guidance Document 2</td>
<td>Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices. (<a href="http://www.12207.com/test1.htm">Checklist available</a>)</td>
</tr>
<tr>
<td>IEC 60880</td>
<td>Software for Computers in the Safety Systems of Nuclear Power Stations</td>
</tr>
<tr>
<td>IEC 62304</td>
<td>Medical device software - Software life cycle processes (<a href="http://www.12207.com/test1.htm">Checklist available</a>)</td>
</tr>
<tr>
<td>IEEE 829</td>
<td>Software Test Documentation</td>
</tr>
<tr>
<td>IEEE 1008</td>
<td>Software Unit Testing</td>
</tr>
<tr>
<td>IEEE 1044</td>
<td>Classification for Software Anomalies</td>
</tr>
<tr>
<td>IEEE 1044.1</td>
<td>Guide to Classification for Software Anomalies</td>
</tr>
</tbody>
</table>