Software Testing Techniques
Chapter 17
Software Testing Strategies
Chapter 18
Software Testing Techniques

• Provide system guidance for designing tests that:
  – Exercise the internal logic of a program
    • “White Box” test cases design techniques
  – Exercise the input and output “Requirements” of a program
    • “Black Box”

To Uncover ERRORS / BUGS / MISUNDERSTANDING OF REQUIREMENTS ETC.
Software Testing Techniques

• Execute the program before the customer.
• To reduce the number of errors detected by customers.
• In order to find the highest possible number of errors software testing techniques must be used.
Software Testing Techniques

Constructive Phases

Analysis → Design → Implementation

Requirement Spic.
Design Document
Code

Destructive Phase

Test

Test Cases
What is testing and why do we do it?

• Testing is the filter to catch defects before they are “discovered” by the customer
  – Every time the program is run by a customer, it generates a “test-case”.
  – We want our test cases to find the defects first.
• Software development is a human activity with huge potential for errors
• Testing before release helps assure quality and saves money
Testing Steps

• Start testing each individual new component and work your way out
  – Unit test
  – Integration test
  – High Order Test
  – Customer Acceptance testing

• Different testing techniques are used at different times in the process

• A test specification is often used as a testing guide for the team.
Testing Objectives

• Purpose of testing is ...
  – To find errors

• A good test ...
  – Tries to discover undetected errors
  – Is successful when errors are found

• Zero Defects is not possible.
  – There is always a condition or usage that can lead to
    an incorrect behavior.
  – You have completed testing when continued testing,
    is no longer economical.
# Tester VS Developer

<table>
<thead>
<tr>
<th>Developer</th>
<th>Tester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructive Process</td>
<td>Destructive Process</td>
</tr>
<tr>
<td>Paid to get code in production</td>
<td>Paid to find errors</td>
</tr>
<tr>
<td>Often focused on their</td>
<td>Often focused on the</td>
</tr>
<tr>
<td>development piece</td>
<td>overall sub-system/system</td>
</tr>
<tr>
<td>Personal involvement in</td>
<td>Viewpoint is customer or</td>
</tr>
<tr>
<td>development can bias viewpoint</td>
<td>overall system health</td>
</tr>
</tbody>
</table>

It is critical that developers and testers work together as a team.
Some Testing Goals - Reality

- To Have “confidence” in the software system.
- To find all major defects with given resources
  - Number of testers
  - Amount of time.
- Design a set of test cases that have a high probability of finding defects.
Testing Case Design

• Test cases should be designed to have the highest likelihood of finding problems

• Can test by either:
  – *Black-box* - using the specifications of what the software should do
    • Tests are derived from the I/O specification.
    • Used in most functional tests.
    • Other names: data-driven, input/output-driven.
  
  – *White-Box* - testing internal paths and working of the software
    • Examine internal program structure and derive tests from an examination of the program’s logic.
    • Used to develop test cases for unit and integration testing
    • Other names: Glass-box, Logic-driven, Structural.
White-Box Testing

• Uses the control structure of the program/design to derive test cases
• We can derive test cases that:
  – Guarantee that all independent paths within a module have been visited at least once.
  – Exercise all logical decisions on their TRUE or FALSE sides
  – Execute all loops at their boundaries
A few White-Box Strategies

• Statement
  – Requires each statement of the program to be executed at least once.

• Branch
  – Requires each branch to be traversed at least once.

• Multi-condition
  – Requires each condition in a branch be evaluated.
More White-Box Strategies

• Basis Path
  – Execute all control flow paths through the code. Based on Graph Theory.
    • Thomas McCabe’s Cyclomatic Complexity:
      • $V(g) : \#edges - \#nodes + 2$
      • Cyclomatic complexity is a SW metric that measures the complexity of a program.
      • The larger $V(g)$ the more complex.

• Data Flow
  – Selects test data based on the locations of definition and the use of variables.
Statement Coverage

- The criterion is to require every statement in the program to be executed at least once
- Weakest of the white-box tests.
- Specified by the F.D.A. as the minimum level of testing.
Statement Coverage

• Example:

```c
void example(int a, int b, float *x)
{
1   if ((a>1) && (b==0))
2     x /= a;
3   if ((a==2) || (x > 1))
4     x++; 
}
```

• Test case(s)
1. a=2, b=0, x=3
• Test Case
  a = 2, b = 0 & x = 3

• Coverage
  – acbed

• What happens with data that takes:
  – abed
  – abd
Branch Coverage

• This criterion requires enough test cases such that each decision has a TRUE and FALSE outcome at least once.

• Another name: Decision coverage

• More comprehensive than statement coverage.
Branch Coverage

• Example:

```c
void example(int a, int b, float *x)
{
1  if ((a>1) && (b==0))
2    x /= a;
3  if ((a==2) || (x > 1))
4    x++; 
}
```

• Test case(s)
  1. a=2, b=0, x=3
  2. a=3, b=1, x=1
Branch Coverage

• Test Case
  1. a=2, b=0 & x=3
  2. a=3, b=1 & x=1

• Coverage
  1. ace
  2. abd

• What happens with data that takes:
  – abe, or
  – acd
Multiple-condition Coverage

- This criterion requires enough test cases such that
  - all possible combinations of condition outcomes in each decision, and
  - all points of entry, are invoked at least once.

- More comprehensive than branch coverage

- First step is to identify the test conditions
  - ifs, whiles, fors
  - reduce to simple predicates
Multiple-condition Coverage

- Example:
  ```c
  void example(int a, int b, float *x)
  {
      if ((a>1) && (b==0)) x /= a;
      if ((a==2) || (x > 1)) x++;
  }
  ```

- Test Conditions
  - `a>1, b=0; a>1, b!=0; a<=1, b=0; a<=1, b!=0;`  
  - `a==2, x > 1; a!=2, x>1; a==2, x<=1; a!=2, x<=1.`
Multiple-condition Coverage

- **Test Conditions**
  1. $a > 1$, $b = 0$
  2. $a > 1$, $b \neq 0$
  3. $a \leq 1$, $b = 0$
  4. $a \leq 1$, $b \neq 0$
  5. $a = 2$, $x > 1$
  6. $a = 2$, $x \leq 1$
  7. $a \neq 2$, $x > 1$
  8. $a \neq 2$, $x \leq 1$

- **Test Cases**
  1. $a = 2$, $b = 0$ & $x = 4$ (1, 5)
  2. $a = 2$, $b = 1$ & $x = 1$ (2, 6)
  3. $a = 1$, $b = 0$ & $x = 2$ (3, 7)
  4. $a = 1$, $b = 1$ & $x = 1$ (4, 8)

- **Coverage**
  - all
Basis Path

• Execute all *independent* flow paths through the code. Based on a flow graph.
  – An *independent* flow path is one that introduces at least 1 new set of statements or conditions
  – Must move along at least 1 new edge on *flow graph*
  – *Flow graph* shows the logical control flow using following notation:

Sequence

If

while

until
i=1;
total.input =
total.valid = 0;
sum = 0;

value[i] <> -999

total.input < 100

value[i] >= min && value[i] <= max

sum=sum+ value[i];

i++;

Enddo

total.valid > 0

aver = sum/ total.valid;

aver=-999

No -

Yes

Value[i] >= min & & value[i] >=

Yes

No -

Done

1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 12. 13.
Number of Paths

\[ V(g) = E - N + 2 \]

\[ 17 - 13 + 2 = 6 \]

\[ R = 6 \]
• $V(g)$ gives us the upper bound to the number of paths to guarantee coverage of all program statements.

• This is the number of test cases we expect to have to create to achieve statement coverage.
Black-Box Testing

• Focuses on functional requirements of the software without regard to the internal structure.

• A.K.A. data-driven, input/output-driven or behavior testing

• Used in most system level testing
  – Functional,
  – Performance
  – Recovery
  – Security & stress

• Tests set up to exercise full functional requirements of the system
Black Box Testing Find Errors in ... 

- Incorrect or missing functions (compare to white box)
- Interface errors
- Errors in External Data structures
- Behavior performance problems (Combinations of input make it behave poorly).
- Initialization and Termination errors (Sensitive to certain inputs (e.g., performance))
- Blackbox done much later in process than white box.
A few Black-box Strategies

• **Exhaustive input testing**
  – A test strategy that uses every possible input condition as a test case.
  – Ideal
  – Not possible!

• **Random**
  – Test cases are created from a pseudo random generator.
  – Broad spectrum. Not focused.
  – Hard to determine the result of the test.
Black-box Strategies

• **Equivalence Partitioning**
  – A black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.

• **Boundary Value Analysis**
  – A test case design technique that complements equivalence partitioning, by selecting test cases at the “edges” of the class.
Equivalence Partitioning

• Divides the input domain of a program into classes of data from which test cases can be derived.
  – 1 test case uncovers classes of errors
    • Incorrect processing of all integer number
• Helps reduce the number of inputs
• What are the properties of a well-selected test cases:
  – It reduces, by more than one, the number of test cases that must be developed since one test case might uncover a class of errors
  – It covers a large set of other possible test cases.
Identifying Equivalence Classes

• Take each input condition (a sentence or phrase in the specification) partition or divide it into 2 or more classes.

• Class
  – Valid equivalence classes
  – Invalid equivalence classes
Rules for Equivalence Classes

• **Range** - If an input condition specifies a range (i.e. n is an integer from 1 to 1000).
  - 1 valid \(1 \leq n \leq 1000\)
  - 2 invalid \(n < 1 \text{ and } n > 1000\)

• **Specified Value** - an input condition specifies a specific value (i.e. 6 character string) identify:
  - 1 valid \(6\) character string\)
  - 2 invalid \(5\) character string, \(7\) char string\)
Rules for Equivalence Classes

• *Value Set* - If the input specifies a set of valid values, define:
  – 1 valid condition within the set.
  – 1 invalid condition outside the set.
Rules for Equivalence Classes

• **Boolean** - If an input condition specifies a “must be” situation (e.g. “first character alpha”) then identify:
  – 1 valid (first character alpha).
  – 1 invalid (first character not alpha).

• If there is any reason to believe that elements in an equivalence class are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.
### Equivalence Partition Example

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Area Code</strong></td>
<td>Blank or 3 Digit Number</td>
</tr>
<tr>
<td><strong>Prefix</strong></td>
<td>3 Digit not beginning with 0 or 1</td>
</tr>
<tr>
<td><strong>Suffix</strong></td>
<td>4 Digit #</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>6 digit alphanumeric string (not required)</td>
</tr>
<tr>
<td><strong>Command</strong></td>
<td>Things like <em>check</em>, <em>deposit</em>, <em>pay</em>, …</td>
</tr>
</tbody>
</table>
# Equivalence Partition Example

| Area Code | **Input Condition 1.** Area (Boolean- there or not) | **Input Condition 2.** Range Values between 200-999 | 1. 1 valid, 1 not  
| | 2. 1 valid, 2 not |  
| Prefix | Input Condition: Range Specified > 200 | Input Condition < 999 | 1. 1 Valid 2 not |  
| Password | 1. Boolean (There or not) | 2. Value (6 characters) | 1. 1 valid 1 not  
| | 2. 1 valid 1 not |  
| Command | Set of valid commands | 1. 1 Valid 1 not |
Boundary Value Analysis

• Experience shows that test cases *exploring boundary conditions* have a high payoff.
  – E.g., Most program errors occur in loop control.

• Different from equivalence partitioning:
  – Rather than *any* element in class, BVA selects tests at the *edge* of the class.
  – In addition to input condition, test cases can be derived for output conditions.

• Similar to Equivalence partitioning. First identify Equivalence classes, then look at the boundaries.
Guideline for Boundary-Value Analysis

• If an *input condition* specifies a range of values,
  – write test cases for the ends of the range, and
  – invalid-input test cases for situations just beyond the ends.

• If a domain of an input is -1.0 to 1.0
  – write test cases for the situation -1.01 to 1.01.
  – Or in general, if bounded by *a* and *b* write test cases just above and below
Guideline for Boundary-Value Analysis

• If an input condition specifies a number of values,
  – write test cases for the minimum and maximum number of values and
  – one beneath and beyond these values.
  – For example an input file can contain 1-255 records, write test cases for 0, 1, 255 and 256
Guideline for Boundary-Value Analysis

• Apply the preceding rules to the output.
  – For example, if output is an output report, then create an output report with maximum and minimum allowable table entries.

• Apply rules to internal data structures ...
  – If use an array that has 1-100 elements max then set up test cases for 0, 1, 100, 101 elements.
Test Case Grid

• A method to help organize test cases. Helps to track what condition each test case tests.
• If a condition changes, you can easily track to the test cases that must be changed. (Convenient for regression testing.)
• Equivalence Class case and Boundary-Value analysis cases can be shown on the same table.
  – Separate sections for Equivalence Class cases and Boundary-Value analysis.
  – Equivalence Class cases first
<table>
<thead>
<tr>
<th>ID</th>
<th>Condition</th>
<th>TC 1</th>
<th>TC 2</th>
<th>TC 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1&lt; item count &lt; 999</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>item count &lt; 1</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>item count &gt; 999</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>.5 &lt; item count &lt; 1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test Case Documentation

• Minimum information for a test case
  – Identifier
  – Input data
  – Expected output data
• Recommended to add the condition being tested (hypothesis).
• Format of test case document changes depending on what is being tested.
• Always include design worksheets.
Simple Test Case Format

<table>
<thead>
<tr>
<th>Id</th>
<th>Condition</th>
<th>Input Data</th>
<th>Expected</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test Case Formats

• Testing worksheet
  – Test Case
    • Identifier (serial number)
    • Condition (narrative or predicate)
    • Input (Stimuli data or action)
    • Expected Output (Results)
  – Test Results
    • Actual Output (Results)
    • Status (Pass/Fail)
## PSP Test Case Format

<table>
<thead>
<tr>
<th>Test Name/Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Objective</td>
</tr>
<tr>
<td>Test Description</td>
</tr>
<tr>
<td>Test Conditions</td>
</tr>
<tr>
<td>Expected Results</td>
</tr>
<tr>
<td>Actual Results</td>
</tr>
</tbody>
</table>

ANSI/IEEE Test Case Outline

- Test-case-specification Identifier
  - A unique identifier

- Test Items
  - Identify and briefly describe the items and features to be exercised by this case

- Input Specifications
  - Specify each input required to execute the test case.

- Output Specifications
  - Specify all of the outputs and features required of the test items.
ANSI/IEEE Test Case Outline

• Environmental needs
  – Hardware
  – Software
  – Other

• Special procedural requirements
  – Describe any special constraints on the test procedures which execute this test case.

• Interfaces dependencies
  – List the id’s of test cases which must be executed prior to this test case