CS 4387/5387
SOFTWARE V&V

LECTURE 3
TESTING, DEBUGGING, VERIFICATION

Many of the Slides are adopted from Aditya P. Mathur
Purdue University

Outline

- Inspection Reports
- Testing Phases
- Unit Testing
  - White Box Testing
Software Testing is a critical element of producing quality software.

It represents the ultimate review of the requirements specification, the design, and the code.

It is the most widely used method to insure software quality.

Many organizations spend 40-50% of development time in testing.

Testing is the process of determining if a program has any errors.

Debugging is concerned with finding where defects occur (in code, design or requirements) and removing them. (fault identification and removal)

Even if the best review methods are used (throughout the entire development of software), testing is necessary.
Testing Fundamentals - 3

- Testing is the one step in software engineering process that could be viewed as destructive rather than constructive.

- A successful test is one that uncovers an as yet undiscovered defect.

- For most software systems, exhaustive testing is impossible.
  - We discussed this through an example in lecture 1
Program Behavior

- Relative term: Program P is correct with respect to specification S
- Do specification and program meet customer expectations?
- Impossible to demonstrate
Unit Testing

- **Unit Testing** checks that an individual program unit (subprogram, object, package, module) behaves correctly.

- **Static Testing**
  - testing a unit without executing the unit code

- **Dynamic Testing**
  - testing a unit by executing a program unit using test data
  - Black-box vs. White-box
Black-Box Testing

Function is understood only in terms of its inputs and outputs, with no knowledge of its implementation.
Test case/data

A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A test set is a collection of zero or more test cases.

Sample test case # 1 for Triangle:
Test data: <2, 3, 4>
Expected output: Scalene

Sample test case # 2 for Triangle:
Test data: <5, 15, 25>
Expected output: Not a Triangle

White-Box Testing

Function is understood only in terms of its implementation.
Unite Testing

- Black-box: a strategy in which testing is based on requirements and specifications.
- White-box: a strategy in which testing is based on internal paths, structure, and implementation.
- Gray-box: peek into the “box” just enough to understand implementation.

Testing and verification

Program verification aims at proving the correctness of programs by showing that it contains no errors. This is very different from testing that aims at uncovering errors in a program.

Program verification and testing are best considered as complementary techniques. In practice, one can shed program verification, but not testing.
Testing and verification (contd.)

Testing is not a perfect technique in that a program might contain errors despite the success of a set of tests.

Verification might appear to be perfect technique as it promises to verify that a program is free from errors. However, the person who verified a program might have made mistake in the verification process; there might be an incorrect assumption on the input conditions; incorrect assumptions might be made regarding the components that interface with the program, and so on.

*Verified and published programs have been shown to be incorrect.*

Program representation: Control flow graphs
Program representation: Basic blocks

- A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point.
  - Block has unique entry and exit points.
- Control always enters a basic block at its entry point and exits from its exit point.
  - There is no possibility of exit or a halt at any point inside the basic block except at its exit point.
  - The entry and exit points of a basic block coincide when the block contains only one statement.

Basic blocks: Example

Reverse Engineering: What does this code do?

Example: Computing x raised to y

```plaintext
begin
int x, y, power;
float z;
input (x, y);
if (y<0)
  power=y;
else
  power=y;
z=1;
10  while (power!=0){
  11    z=z*x;
  12    power=power-1;
  13  }
14  if (y<0)
15    z=1/z;
16  output(z);
17  end
```
Basic blocks: Example (contd.)

Basic blocks

<table>
<thead>
<tr>
<th>Block</th>
<th>Lines</th>
<th>Entry point</th>
<th>Exit point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4, 5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>11, 12</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Control Flow Graph (CFG)

- A control flow graph (CFG) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N. We often write G = (N, E) to denote a flow graph G with nodes given by N and edges by E.
Control Flow Graph (CFG)

- In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

- Blocks and nodes are labeled such that block $b_i$ corresponds to node $n_i$. An edge $(i, j)$ connecting basic blocks $b_i$ and $b_j$ implies that control can go from block $b_i$ to block $b_j$.

- We also assume that there is a node labeled Start in $N$ that has no incoming edge, and another node labeled End, also in $N$, that has no outgoing edge.

Control Flow Graphs

- Programs are made of three kinds of statements:
  - Sequence (i.e., series of statements)
  - Condition (i.e., if statement)
  - Iteration (i.e., while, for, repeat statements)

- Control flow graph: visual representation of flow of control.
  - Node represents a basic block (sequence of statements with single entry and last statement is transfer of control)
  - Edge represents transfer of control from one node to another
Example 1: CFG

1. read (result);
2. read (x,k)
3. while result < 0 {
4.   ptr ← false
5.   if x > k then
6.     ptr ← true
7.     x ← x + 1
8.     result ← result + 1 }
9. print result
Quiz 1: Write a CFG (5 minutes)

Example 2
1. if \( a < b \) then
2.     while \( a < n \)
3.          \( a \leftarrow a + 1 \);
4. else
5.     while \( b < n \)
6.          \( b \leftarrow b + 1 \);

Quiz 2: Write a CFG (3 Minutes)

Example 3
1. read \((a,b)\);
2. if \((a \neq 0 \&\& b \neq 0)\) then {
3.       \( c \leftarrow a + b \);
4.     if \( c > 10 \) then
5.         \( c \leftarrow \text{max} \)
6.     else \( c \leftarrow \text{min} \) 
7. else print 'Done'
Example 4

1. read \((a,b)\);  
2. if \((a \neq 0 \; \text{or} \; b \neq 0)\) then \{ 
3. \hspace{1em} c \leftarrow a + b 
4. \hspace{1em} \textbf{while}(c < 100) \} 
5. \hspace{1em} c \leftarrow a + b; \} 
6. \hspace{1em} c \leftarrow a \times b 

Statement (Line) Coverage:

- Develop test cases so that every statement is executed at least once

Example 4

1. read \((a,b)\);  
2. if \((a \neq 0 \; \text{or} \; b \neq 0)\) then \{ 
3. \hspace{1em} c \leftarrow a + b 
4. \hspace{1em} \textbf{while}(c < 100) \} 
5. \hspace{1em} c \leftarrow a + b; \} 
6. \hspace{1em} c \leftarrow a \times b
Quiz 4: Develop Test Cases Using Statement Coverage (5 Minutes)

- Develop test cases so that every statement is executed at least once

**Example 4**
1. read (a,b);
2. if (a ≠ 0 || b ≠ 0) then {
   3.   c ← a + b
   4.   while( c < 100)
   5.     c ← a + b;
   6.   c ← a * b

Test Cases:
Test1: a = 50, b = 49 (1,2,3,4,5,4,5,4,5,4,5,...)

Quiz 4: Develop Test Cases Using Statement Coverage (5 Minutes)

- Develop test cases so that every statement is executed at least once

**Example 4**
1. read (a,b);
2. if (a ≠ 0 || b ≠ 0) then {
   3.   c ← a + b
   4.   while( c < 100)
   5.     c ← a + b;
   6.   c ← a * b
Quiz 4: Develop Test Cases Using 
Statement Coverage (5 Minutes)

- Develop test cases so that every statement is executed at least once

Example 4
1. `read (a,b);`
2. `if (a ≠ 0 || b ≠ 0) then {
3. \(c \leftarrow a + b\)
4. \(\text{while} (c < 100)\)
5. \(c \leftarrow a + b;\} \)
6. \(c \leftarrow a \times b\)

Test Cases:
Test1: \(a = 50, b = 49\) (1,2,3,4,5,4,5,4,5…)
Test Cases:
Test2: \(a = 0, b = 0\) (1,2,1,6)

Statement Coverage (Problems)

- Software developers and testers commonly use line coverage because of its simplicity and availability in object code instrumentation technology.
- Of all the structural coverage criteria, line coverage is the weakest, indicating the fewest number of test cases.
- Bugs can easily occur in the cases that line coverage cannot see.
- The most significant shortcoming of line coverage is that it fails to measure whether you test simple if statements with a false decision outcome. Experts generally recommend to only use line coverage if nothing else is available. Any other measure is better.
Branch Coverage

- Each edge of a program’s CFG is traversed at least once in some test.
- Independent paths:
  - 1, 2, 3, 9
  - 1, 2, 3, 4, 5, 6, 7, 8, 3, ..., 9
  - 1, 2, 3, 4, 5, 7, 8, 3, ..., 9

Example 1

Quiz 5: Develop MINIMAL Set of Test Cases Using Branch Coverage (7 Minutes)

1. read (result);
2. read (x,k)
3. while result < 0 {
4. ptr ← false
5. if x > k then
6. ptr ← true
7. x ← x + 1
8. result ← result + 1 }
9. print result
Quiz 5: Develop MINIMAL Set of Test Cases Using Branch Coverage (7 Minutes)

1. read (result);
2. read (x,k)
3. while result < 0 {
4.   ptr ← false
5.   if x > k then
6.     ptr ← true
7.     x ← x + 1
8.     result ← result + 1 }
9. print result

Possible test set: dc stands for don't care
{<result = 5>, x = dc , k = dc >}

Quiz 5: Develop MINIMAL Set of Test Cases Using Branch Coverage (7 Minutes)

1. read (result);
2. read (x,k)
3. while result < 0 {
4.   ptr ← false
5.   if x > k then
6.     ptr ← true
7.     x ← x + 1
8.     result ← result + 1 }
9. print result

Possible test set: dc stands for don't care
{<result = -1, x = 5, k = 3>,
Quiz 5: Develop MINIMAL Set of Test Cases Using Branch Coverage (7 Minutes)

1. read (result);
2. read (x,k)
3. while result < 0 {
4. ptr ← false
5. if x > k then
6. ptr ← true
7. x ← x + 1
8. result ← result + 1 }
9. print result

Possible test set: dc stands for don't care
{<
<
\<result = -1, x = 10, k = 15>}

Minimal test set
{<result = -1, x = 5, k = 5>}

Quiz 5: Develop MINIMAL Set of Test Cases Using Branch Coverage (7 Minutes)

1. read (result);
2. read (x,k)
3. while result < 0 {
4. ptr ← false
5. if x > k then
6. ptr ← true
7. x ← x + 1
8. result ← result + 1 }
9. print result

Possible test set: dc stands for don't care
{<
<
\<result = -1, x = 10, k = 15>}

Minimal test set
{<result = -1, x = 5, k = 5>
Condition Coverage

- Each edge is traversed and all possible values of constituents of compound condition are executed at least once
- Test each way a condition can be true or false
- Requires rewriting the CFG to identify new paths
- Basis set yields minimal test set

AND Condition

1. read (a, b);
2. if (a == 0 && b == 0) then {
3.   c ← a + b
4. else c ← a * b}

Basis set:
1. 2A, 2B, 3, J
1. 2A, 2B, 4, J
1. 2A, 4, J
OR Condition

1. read \((a, b)\);
2. if \(a == 0 \lor b == 0\) then {
3. \(c \leftarrow a + b\);
4. while \(c < 100\)
5. \(c \leftarrow a + b\)\

Basis set:
- 1, 2A, 3, 4, 5, 4 ... J
- 1, 2A, 3, 4, J
- 1, 2A, 2B, 3, 4, 5, 4, ... J
- 1, 2A, 2B, J

Quiz 6: Develop Minimal Set of Test Cases Using Condition Coverage (5 Minutes)

1. read \((a, b)\);
2. if \(a == 0 \lor b == 0\) then {
3. \(c \leftarrow a + b\);
4. while \(c < 100\)
5. \(c \leftarrow a + b\)\

Test1: \(a = 0, \ b = 99\)
Test2: \(a = 0, \ b = 100\)
Test3: \(a = 99, \ b = 0\)
Test4: \(a = 1, \ b = 1\)
Cyclomatic Complexity

- Software metric that provides a quantitative measure of the logical complexity of a program.
- Defines the number of independent paths in the basis set of a program
  - Independent path introduces at least one new set of process statements or a new condition.
  - Basis set is set of paths such that if test cases force execution of these set of paths, then every statement in the path and every condition will be executed (both true and false). Question is; which CFG are we talking about?
- Provides the upper bound for the number of tests that must be conducted to ensure that all statements been have executed at least once
  - $V(G) = E - N + 2$
  - $V(G) = B + 1$

Example: What is the Cyclomatic Complexity?

V(G) = 11 - 9 + 2 = 4

Independent paths:
- 1-11
- 1-2-3-4-5-10-1-11
- 1-2-3-6-8-9-10-1...-11
- 1-2-3-6-7-9-10-1...-11
Path Coverage-1

- Every distinct path through code is executed at least once
- Basis set does not yield minimal test set

Example
1. read (x)
2. read (z)
3. if x ≠ 0 then begin
   4. y ← x * z;
   5. x ← z end
6. else print ‘Invalid’
7. if y > 1 then
8. print y
9. else print ‘Invalid’

Test Paths:
1, 2, 3, 4, 5, J1, 7, 8, J2
1, 2, 3, 4, 5, J1, 7, 9, J2
1, 2, 3, 6, J1, 7, 8, J2,
1, 2, 3, 6, J1, 7, 9, J2

Def-Use Coverage

- Def-use coverage: every path from every definition of every variable to every use of that definition is exercised in some test.

Example
1. read (x)
2. read (z)
3. if x ≠ 0 then begin
   4. y ← x * z;
   5. x ← z end
6. else print ‘Invalid’
7. if y > 1 then
8. print y
9. else print ‘Invalid’
Quiz 7: Develop Test Cases using Def-Use Coverage (5 Minutes)

- **Def-use coverage**: every path from every definition of every variable to every use of that definition is exercised in some test.

**Example**

1. read (x)
2. read (z)
3. if x ≠ 0 then {
   4. y ← x * z;
   5. x ← z }
4. else print ‘Invalid’
5. if y > 1 then
6. print y
7. else print ‘Invalid’

Test Path: 1, 2, 3, 4, 5, 7, 8, J

---

Strength of Coverage

Arrows point from weaker to stronger coverage.

Stronger coverage requires more test cases.
Group Exercise # 1 (15 Minutes)

- Develop the CFG for the Triangle Problem Implementation

Group Exercise # 2 (15 Minutes)

- Develop a Test Suite Using Branch Testing for the Triangle Implementation
- HINT: Start by identifying the upper limit on the number of tests you need and what the distinct paths are
Group Exercise # 3 (20 Minutes)

- Develop Def-Use test cases
- Does it matter which variable(s) you pick?

Group Exercise # 4 (25 Minutes)

- Develop test cases using condition coverage
- Let’s just give it a try
- You should be able to reuse some old test cases
Next Week

- Quiz
- HW Assignment Presentation
- Black-Box Testing