

Qualidade de Software (14450)

Structural and Mutation Testing

(adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit,
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

Today's Goals

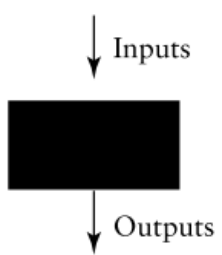
✧ We'll look at:

- white box vs black box
- role and kinds of white box testing
- implementation - source, executable and sampling
- code coverage testing
 - statement coverage
 - basic block coverage
- mutation testing methods

White Box vs. Black Box

Recall: Kinds of Tests

- We divide these tests into:



**Black Box
Testing
Methods**

- **Black box** methods – cannot see the software code (it may not exist yet!) – can only base their tests on the **requirements** or **specifications**



**White Box
Testing
Methods**

- **White box** (aka glass box) methods – can see the software's code – can base their tests on the software's actual **architecture** or **code**

Kinds of White Box Testing

Code Coverage

- Code coverage methods design tests to **cover** (execute) every method, statement or instruction of the program at least once

Logic Path/Decision Point Coverage

- Logic path methods design tests to cover every **path of execution** in the program at least once

Data & Data Flow Coverage

- Data coverage methods explicitly try to cover the **data** aspects of the program code, rather than the control aspects

Fault-Based Testing (e.g. Mutation Testing)

- Mutation testing involves creating many slightly different versions of the code by **mutating** (changing operations) in each version
- Used to check **sufficiency** of test suites in detecting faults

Role of White Box Testing

Completeness for Black Box Methods

- White box code coverage gives a measure of **completeness** for open-ended black box methods
 - **Example:** Black box **shotgun** testing becomes a systematic method if we use code coverage (all statements executed at least once in the set of tests) as the completion criterion

Role of White Box Testing

Finds a Different Kind of Errors

- Black box testing finds errors of **omission** – that is, something that is specified that we have failed to do
- White box testing finds errors of **commission** – that is, something that we have done, but incorrectly

Automation

- Because white box testing involves the program code itself, which has a standard form, we can automate most of it

White Box Testing & Code Injection

Code Injection

- Injection is not itself a test method, but refers to **modifications** of the source or executable code being tested in order to make tests more effective (possible because white box)
 - **Example:** Modify the program to log each statement's line number to a log file as it is executed, in order to check that every line is executed at least once by a test suite (*Produces a file of executed line numbers – check every line there*)
- Injection involves adding **extra** statements or instructions to execute that do not change what the original program does but checks or logs additional information about **execution** of the program (such as which statements have been executed)
- Original code is not changed, instead a **separate copy** with modifications is generated to run the tests on

Applications of Code Injection

✧ Instrumentation Injection

- Involves adding code to instrument the actions of the program at every method, statement or instruction during testing, to keep track of properties such as execution coverage

✧ Performance Instrumentation

- Involves adding code to log the actual time or space used by each method or statement of the program during execution

✧ Assertion Injection

- Involves adding strict run-time assertion code to every method, statement or instruction in the program during testing, to help localize the cause of failures

✧ Fault Injection

- Involves adding code to simulate run-time faults, to test fault handling

Implementation of White Box Code Injection

Three Levels of Implementation

- Although it is not a necessity, white box testing usually involves validation of coverage using [code injection](#)
- This code injection can be implemented in three separate ways:
 1. At the [source](#) level
 2. At the [executable code](#) level
 3. At the [execution sampling](#) level

Implementation of White Box Testing

Three Levels of Implementation

1. At the **source** level
2. At the **executable code** level

1 & 2: A **copy** of the program under test is altered to inject the additional source or executable code to log coverage as the program executes.

3. At the **execution sampling** level

3: The **original** program under test is run but with regular timer interrupts - at each interrupt, the current state and execution location at interrupt time can be **sampled** and logged before continuing execution.

Source Level Implementation

Implementing Code Injection by Source Modification

- Create a copy of the program with new statements inserted to log coverage

Example: JTest

```
13     final int mid = (lo + hi) / 2;
14     if (list[mid] == key)
15         result = mid;
16     else if (list[mid] > key)
17         hi = mid - 1;
18     else
19         lo = mid + 1;
```

Executable Code Level Implementation

```
log.println (13);  
13  final int mid = (lo + hi) / 2;  
log.println (14);  
14  if (list[mid] == key)  
    {  
15      log.println (15);  
      result = mid;  
    }  
    else  
    {  
16      log.println (16);  
      if (list[mid] > key)  
      {  
17          log.println (17);  
          hi = mid - 1;  
      }  
18      else  
      {  
          log.println (18);  
          log.println (19);  
19          lo = mid + 1;  
      }  
    }  
}
```

Executable Code Level Implementation

```
13      execount[13] += 1;
      final int mid = (lo + hi) / 2;
      execount[14] += 1;
14      if (list[mid] == key)
      {
15          execount[15] += 1;
          result = mid;
      }
      else
      {
16          execount[16] += 1;
          if (list[mid] > key)
          {
17              execount[17] += 1;
              hi = mid - 1;
          }
18          else
          {
              execount[18] += 1;
              execount[19] += 1;
19              lo = mid + 1;
          }
      }
  }
```

White Box Tools

Testing Tools

- Obviously implementing these strategies by hand programming would be tedious and time consuming
- White box coverage testing is almost always supported by tools to implement the necessary code injections
- Often test analysis and selection of test cases for white box testing can also be done automatically by modern tools

Code Coverage Testing

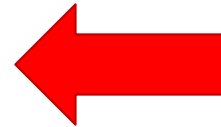
Code Coverage Methods

- Two kinds: statement analysis (**flow independent**), decision analysis (**flow dependent**)
- Statement analysis methods
 - **statement** coverage
 - **basic block** coverage
- Decision analysis methods
 - **decision** coverage
 - **condition** coverage
 - **loop** coverage
 - **path** coverage

Code Coverage Testing

Code Coverage Methods

- Two kinds: statement analysis (flow independent), decision analysis (flow dependent)
- Statement analysis methods
 - statement coverage
 - basic block coverage
- Decision analysis methods
 - decision coverage
 - condition coverage
 - loop coverage
 - path coverage



Statement Coverage

Statement Coverage Method

- Causes every statement in the program to be executed at least once, giving us confidence that every statement is at least *capable* of executing correctly
- System: Make a test case for each statement in the program, independent of the others
 - Test must simply cause the statement to be run, ignoring its actions and sub-statements (but still must check that result of test is correct)
- Completion criterion: A test case for every statement
 - Can be checked by *instrumentation injection* to track statement execution coverage

Example: Statement Coverage

```
// calculate numbers less than x  
// which are divisible by y  
1  int x, y;  
2  x = c.readInt ();  
3  y = c.readInt ();  
4  if (y == 0)  
5      c.println ("y is zero");  
6  else if (x == 0)  
7      c.println ("x is zero");  
8  else  
9      {  
10         for (int i = 1; i <= x; i++)  
11             {  
12                 if (i % y == 0)  
13                     c.println (i);  
14             }  
15     }
```

Example: Statement Coverage

Statement Coverage Tests

- We blindly make one test for each statement, analyzing which **inputs** are needed to cause the statement to be executed
- Create test case for each unique set of inputs

Statement	x input	y input
1	?	?
...		

Example: Statement Coverage

Statement Coverage Tests

<u>Statement</u>	<u>x input</u>	<u>y input</u>	<u>Test</u>	<u>x</u>	<u>y</u>
1	0	0	T1	0	0
2	0	0			
3	0	0			
4	0	0			
5	0	0			
6	0	1	T2	0	1
7	0	1			
8	1	1	T3	1	1
9	1	1			
10	1	1			
11	1	1			

Basic Block Coverage

Basic Block Analysis Method

- Causes every **basic block** (indivisible sequence of statements) to be executed at least once - (usually) generates fewer tests
- System: Identify basic blocks by sequence analysis, design test case for each basic block
 - Sequence of statements in a row, ignoring sub-statements, such that if first is executed then following are all executed
- Completion criterion: A test case for every basic block
 - Can be checked by **instrumentation injection** to track statement execution coverage

Example: Basic Block Analysis

```
// calculate numbers less than x  
//   which are divisible by y  
1 | int x, y;  
  | x = c.readInt ();  
  | y = c.readInt ();  
  | if (y == 0)  
  | 2 | c.println ("y is zero");  
  | else  
  | 3 | if (x == 0)  
  |   | 4 | c.println ("x is zero");  
  |   | else  
  |   | {  
  |   | 5 | for (int i = 1; i <= x; i++)  
  |   |   | {  
  |   |   | 6 | if (i % y == 0)  
  |   |   |   | 7 | c.println (i);  
  |   |   |   | }  
  |   |   | }  
  |   | }  
  | }
```

Example: Basic Block Analysis

Basic Block Coverage Tests

- We make one test for each block, analyzing which inputs are needed to cause the block to be entered
- Create test case for each unique set of inputs

<u>Block</u>	<u>x input</u>	<u>y input</u>
1	?	?
...		

Example: Basic Block Analysis

Basic Block Coverage Tests

<u>Block</u>	<u>x input</u>	<u>y input</u>	<u>Test</u>	<u>x</u>	<u>y</u>
1	0	0	T1	0	0
2	0	0			
3	0	1	T2	0	1
4	0	1			
5	1	1	T3	1	1
6	1	1			
7	1	1			

Decision (Branch) Coverage Method

- Causes every **decision** (if, switch, while, etc.) in the program to be made both ways (or every possible way for switch)
- System: Design a test case to exercise each decision in the program each way (true / false)
- Completion criterion: A test case for each side of each decision

Example: Decision Coverage

```
int x, y;
x = c.readInt ();
y = c.readInt ();

1  if (y == 0)
    c.println ("y is zero");
else
2  if (x == 0)
    c.println ("x is zero");
    else
    {
        for (int i = 1; i <=x ; i++)
        {
3      if (i % y == 0)
            c.println (i);
        }
    }
}
```

Example: Decision Coverage

- We make one test for each side of each decision

<u>Decision</u>	<u>x input</u>	<u>y input</u>
1 true	0	0
1 false	0	1
2 true	0	1
2 false	1	1
3 true	1	1
3 false	2	3

Condition Coverage

- Like decision coverage, but causes every **condition** expression to be exercised both ways (true / false)
- A condition is any true / false subexpression in a decision

Example:

if ((**x** == 1 || **y** > 2) && **z** < 3)

Requires separate condition coverage tests for each of:

x == 1 true / false

y > 2 true / false

z < 3 true / false

- More effective than simple decision coverage since exercises the different **entry preconditions** for each branch selected

Loop Coverage

- Most programs do their real work in **do**, **while** and **for** loops
- This method makes tests to exercise each **loop** in the program in four different states :
 - execute body **zero** times (do not enter loop)
 - execute body **once** (i.e., do not repeat)
 - execute body **twice** (i.e., repeat once)
 - execute body **many times**
- Usually used as an enhancement of a statement, block, decision or condition coverage method
- System: Devise test cases to exercise each loop with zero, one, two and many repetitions
- Completion criterion: A test for each of these cases for each loop

Example: Loop Coverage

```
int x, y;
X = c.readInt ();
Y = c.readInt ();

if (y == 0)
    c.println ("y is zero");
else if (x == 0)
    c.println ("x is zero");
else
{
    for (int i=1; i<=x ; i++)
    {
        if (i % y == 0)
            c.println(i);
    }
}
```

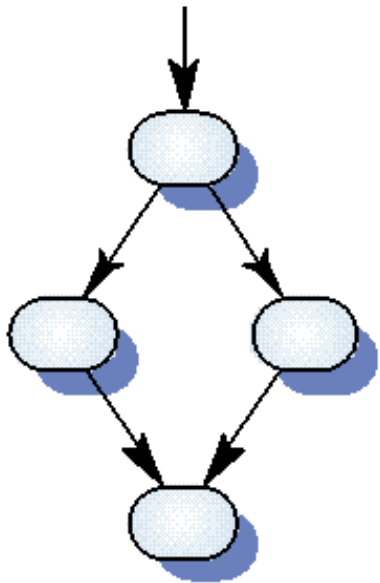
<u>Loop Body</u>	<u>x</u>	<u>y</u>
zero times	N/A	
Once	1	1
Twice	2	1
many times	10	1

Execution Paths

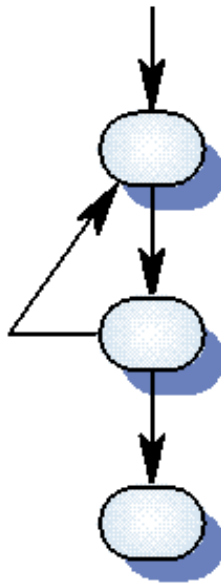
- An **execution path** is a sequence of executed statements starting at the **entry** to the unit (usually the first statement) and ending at the **exit** from the unit (usually the last statement)
- Two paths are **independent** if there is at least one statement on one path which is not executed on the other
- **Path analysis** (also know as **cyclomatic complexity** analysis) identifies all the **independent paths** through a unit

Execution Path Analysis

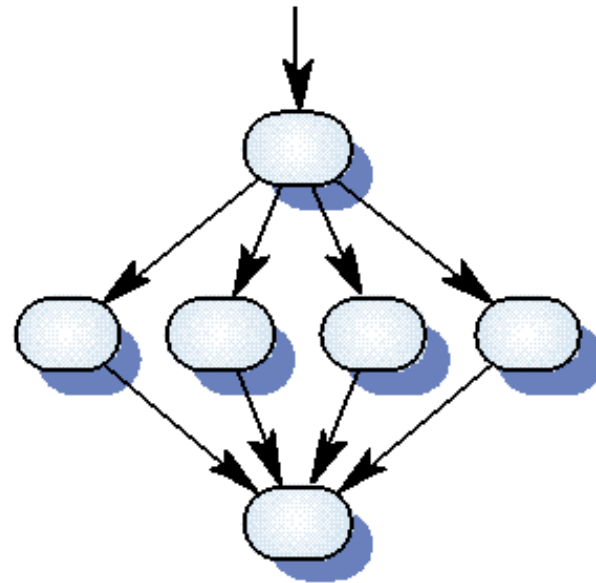
- It is easiest to do path analysis if we look at the execution **flow graph** of the program or unit
- The flow graph simply shows program **control flow** between **basic blocks**



if-then-else



do-while



switch

Path Coverage Testing

- **Advantages**

- Covers all **basic blocks** (does all of basic block testing)
- Covers all **conditions** (does all of decision/condition testing)
- Does all of both, but with **fewer tests!**
- Automatable (actually, in practice requires automation)

- **Disadvantages**

- Does not take **data** complexity into account at all

Path Coverage Testing – Disadvantages

- Example: These fragments should be tested the same way, since they actually implement the same solution - but the one on the left gets **five** tests, whereas the one on the right gets only **one**

```
//
switch (n) {
    case 1:
        s = "One";
        break;
    case 2:
        s = "Two";
        break;
    case 3:
        s = "Three";
        break;
    case 4:
        s = "Four";
        break;
    case 5:
        s = "Five";
        break;
}
```

```
//
String numbers[] =
    {"One", "Two",
     "Three", "Four", "Five"};

s = numbers[n];
```

White Box Data Coverage

- Data flow coverage is concerned with variable **definitions** and **uses** along execution paths
- A variable is **defined** if it is assigned a new value during a statement execution
- A variable definition in one statement is **alive** in another if there is a path between the two statements that does not redefine the variable
- There are two types of variable **uses**
 - A **P-use** of a variable is a predicate use (e.g. if statement)
 - A **C-use** of a variable is a computation use or any other use (e.g. I/O statements)

Example: Definition, P-Use, C-Use of Variables

```
static int find (int list[], int n, int key)
{
    int lo = 0;
    int hi = n - 1;
    int result = -1;    <- Definition of result
    while (hi >= lo)
    {
        if (result != -1)    <- P-Use of result
            break;
        else
        {
            final int mid = (lo + hi) / 2;
            if (list[mid] == key)
                result = mid;    <- Definition of result
            else if (list[mid] > key)
                hi = mid - 1;
            else // list[mid] < key
                lo = mid + 1;
        }
    }
    return result;
}
```

Example: Definition, P-Use, C-Use of Variables

```
static int find (int list[], int n, int key)
{
    int lo = 0;
    int hi = n - 1;      <- Definition of hi
    int result = -1;
    while (hi >= lo)     <- P-Use of hi
    {
        if (result != -1)
            break;
        else
        {
            final int mid = (lo + hi) / 2;  <- C-Use of hi
            if (list[mid] == key)
                result = mid;
            else if (list[mid] > key)
                hi = mid - 1;                <- Definition of hi
            else // list[mid] < key
                lo = mid + 1;
        }
    }

    return result;
}
```

White Box Data Coverage

- There are a variety of different **testing strategies** related to data flow:
 - **All-Uses coverage**: test all uses of each definition
 - **All-Defs coverage**: test each definition at least once
 - **All C-Uses/Some P-Uses coverage**: test all computation uses. If no computation uses for a given definition then test at least one predicate use
 - **All P-Uses/Some C-Uses coverage**: test all predicate uses. If no predicate uses for a given definition then test at least one computation use
 - **All P-Uses coverage**: Test each predicate use

White Box Data Coverage

- We have covered definitions of data, uses of data, and testing strategies for data flow coverage.
- System: Identify definitions (and uses) of variables and testing strategy. Design a set of test cases that cover the testing strategy.
- Completion criterion: Depends on the test strategy. For example, in All-Defs we are done when we have a test case for each variable definition.

Mutation Testing

What is it for?

- Mutation testing is a white box method for checking the **adequacy** of test suites
- As you have already discovered, creating a test suite can be an expensive and time consuming effort
- No matter what test method is used, discovering if test suites are adequate to uncover faults is **itself** an even more difficult task
- Mutation testing offers an almost completely **automated** way to check the adequacy of a set of tests in uncovering faults in the software

Mutation Testing

How does it work?

- In order to test the adequacy of a **test suite**, we first run the software on the suite and fix any problems until we are satisfied that the software passes the tests
- We then **save the results** of the tests in a file or set of files to serve as the correct output to compare to

Mutation Testing

How does it work?

- We then use **mutation** of the source code to create a set of **mutants**, each of which is a program slightly diferente from the original
- For each mutant, we run the test suite on the mutante and **compare the results** to the saved results from the original
 - If the results differ, then the mutant has been **“killed”** (detected) by the test suite
 - If the results do not differ, then the test suite is **inadequate** to detect the mutant, and a new test must be added to the suite to “kill” that mutant

Systematic Mutation

Systematic Approach

- In order for mutation testing to be systematic, there must be a **system** and a **completion criterion** for creating mutants
- The system for mutation normally specifies simple **syntactic changes** to the program source representing errors in the code
- Each mutant has exactly **one** change in it
- We are done when every possible single change **mutant** of the system has been generated and tested

Systematic Mutation

Systematic Approach

- Example systematic mutations are:
 - **value** mutations (changing constants, subscripts or parameters by adding or subtracting one, etc.)
 - **decision** mutations (inverting or otherwise modifying the sense of each decision condition in the program)
 - **statement** mutations (deleting or exchanging individual statements in the program)

Example #1: Value Mutation

Value Mutation Example

- System: Mutate the value of each **constant** in the program to be off by one (or more generally, each integer expression)
- Completion criterion: One mutant for each constant in the program
- Note that there are many other possible **value mutations**:
 - **constants** modified in some other way, e.g. off by -1
 - all **integer expressions** modified (not just constants), e.g., x changed to $x+1$, etc.

Example #1: Value Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    c.println ("y is 0");  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println (i);  
    }  
}
```

Example test suite

(statement coverage):

<u>Test</u>	<u>x</u>	<u>y</u>	<u>output</u>
T1	0	0	"y is 0"
T2	0	1	"x is 0"
T3	1	1	1

Example #1: Value Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 1)  
    c.println ("y is 0");  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"x is 0"
T2	0	1	"x is 0"	"y is 0"
T3	1	1	1	"y is 0"

Example #1: Value Mutation

```
// calculate numbers less than x  
// which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    c.println ("y is 0");  
else if (x == 1)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	
T3	1	1	1	"x is 0"

Example #1: Value Mutation

```
// calculate numbers less than x
// which are divisible by y
```

```
int x, y;
x = c.readInt();
y = c.readInt();

if (y == 0)
    c.println ("y is 0");
else if (x == 0)
    c.println ("x is 0");
else
{
    for (int i = 2; i <= x; i++)
    {
        if (i % y == 0)
            c.println (i);
    }
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

Example #1: Value Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    c.println ("y is 0");  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 1)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

Example #2: Decision Mutation

Decision Mutation Example

- System: Invert the sense of each **decision condition** in the program
(e.g., change **>** to **<**, **==** to **!=**, and so on)
- Completion criterion: One mutant for each decision condition in the program

Example #2: Decision Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y != 0)  
    c.println ("y is 0");  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"x is 0"
T2	0	1	"x is 0"	"y is 0"
T3	1	1	1	"y is 0"

Example #2: Decision Mutation

```
// calculate numbers less than x  
//   which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    c.println("y is 0");  
else if (x != 0)  
    c.println("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println(i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	
T3	1	1	1	"x is 0"

Example #2: Decision Mutation

```
// calculate numbers less than x  
// which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    c.println ("y is 0");  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y != 0)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original</u> <u>output</u>	<u>mutant</u> <u>output</u>
T1	0	0	"y is 0"	"y is 0"
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	

Example #3: Statement Mutation

Statement Mutation Example

- System: Delete each **statement** in the program
- Completion criterion: One mutant for each statement
- Note that there are many other possible **statement mutations**:
 - **interchanging** adjacent statements
 - **shuffling** sequences of statements
 - **doubling** statements
 - many more

Example #3: Statement Mutation

```
// calculate numbers less than x  
// which are divisible by y
```

```
int x, y;  
x = c.readInt();  
y = c.readInt();  
  
if (y == 0)  
    ;  
else if (x == 0)  
    c.println ("x is 0");  
else  
{  
    for (int i = 1; i <= x; i++)  
    {  
        if (i % y == 0)  
            c.println (i);  
    }  
}
```

<u>Test</u>	<u>x</u>	<u>y</u>	<u>original output</u>	<u>mutant output</u>
T1	0	0	"y is 0"	
T2	0	1	"x is 0"	"x is 0"
T3	1	1	1	1

Mutation Testing in Practice

Some Final Observations

- In practice, simple statement coverage tests are **often sufficient** to “kill” most kinds of mutants. The adequacy of suite can be measured as: $(\# \text{ mutants killed}) / (\text{total mutants})$
- If we do mutation testing on acceptance, functionality coverage, input/output coverage or other **black box** test suites, on the other hand, we are likely to find **many** mutants not “killed” by the tests
- Since most projects use **primarily** black box techniques, automated mutation testing can be a very valuable help in making test suites more effective

Key Points (1 of 2)

- ✧ White box testing includes: code coverage, logic path/decision point coverage, data & data flow coverage, fault-based testing (e.g. mutation testing)
- ✧ White box methods often involve code injection to instrument execution using source modification, executable code modification or run time sampling
- ✧ Today we started to look at one class of code coverage methods: Statement analysis methods (statement, basic block coverage)

Key Points (2 of 2)

- ✧ Mutation Testing is a white box method for automatically checking test suites for completeness
- ✧ Mutations are simple, **syntactic** variants of programs that can be generated automatically
- ✧ Typical mutations are **value** mutations, **decision** mutations, **statement** mutations

