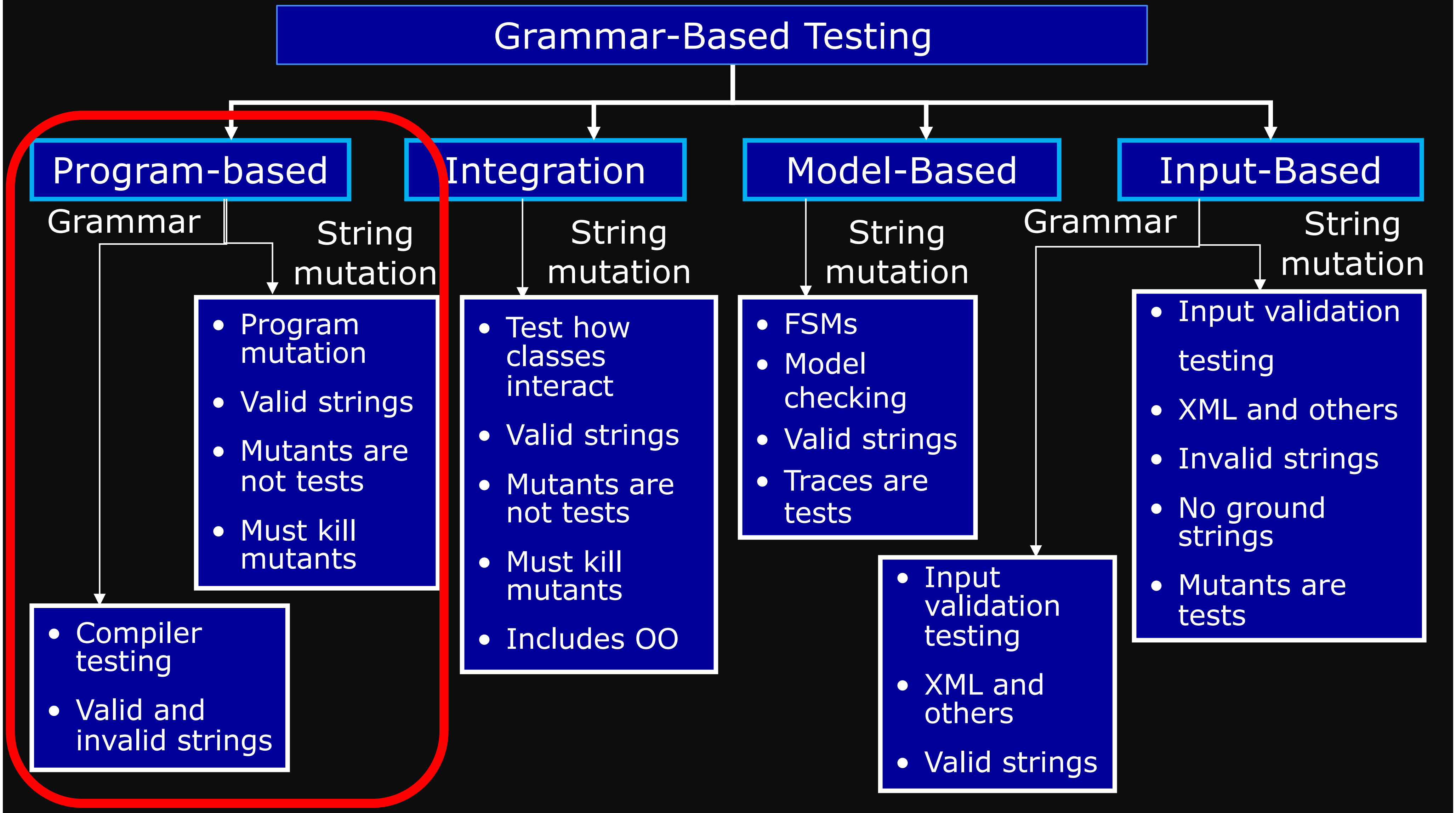
Program-based Mutation Testing

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 9.2]

Instantiating Grammar-Based Testing



Grammar-Based Testing

Program-based

Grammar

- Compiler testing
- Valid and invalid strings

String mutation

- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants

Integration

String mutation

- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

Model-Based

String mutation

- FSMs
- Model checking
- Valid strings
- Traces are tests

Input-Based

Grammar

- Input validation testing
- XML and others
- Valid strings

String mutation

- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests

Syntax-Based Testing

Input space

Grammar
Mutation operators

Ground string(s)
Valid mutants
Invalid mutants

Test case inputs

Program source code

Grammar / original program
Mutation operators

Mutants (compilable & runnable)

Test requirements

Applying Syntax-Based Testing to Programs

- Test requirements are derived from the syntax of software artifacts

- Syntax-based criteria originated with programs and have been used mostly with program source code

- BNF criteria are most commonly used to test compilers
    - Use BNF criteria to generate programs to test all language features that compilers must process

- Mutation testing criteria are most commonly used for unit testing and integration testing revisit

Mutation Testing

- A process of changing the software artifact based on well defined rules

Mutation operators: Rules that specify syntactic variations of strings generated from a grammar

- Rules are defined on syntactic descriptions

Grammars

- We perform mutation analysis when we want to make systematic changes, resulting in variations of a valid string

Mutants: Result of one application of a mutation operator

- We can mutate the syntax or objects developed from the syntax

Grammar

Ground strings
(Strings in the grammar)

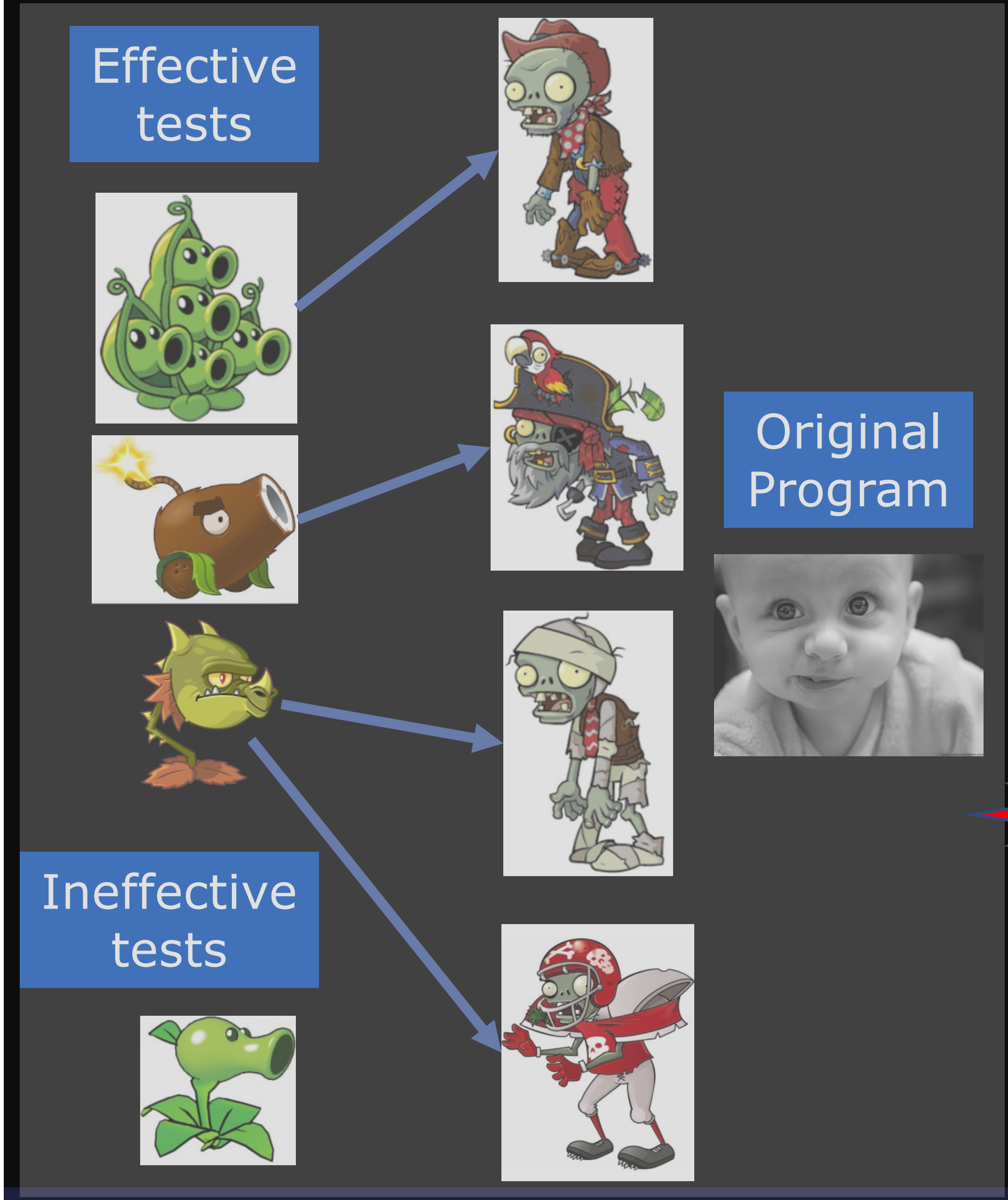
Mutation Testing (Source Code)

- Inject changes into programs
- Strongest testing criterion
- Effective criterion for designing and evaluating tests
- Applied to C, C++, Java, JavaScript, Java EE, PHP, Angular, SQL, Android, spreadsheet, policy, …

Premise:
If the software has a fault, there usually are some mutants that can only be **killed** by a **test** that also detects that fault.

Kill:
The test makes the output of the mutant **different** from the output of the original program

Mutation Testing



Find last index of zero

```
int lastZero (int[] x) {
    for (int i = x.length-1; i >= 0; i--) {
        if (x[i] == 0)
            return i;
    }
    return -1;
}
```

i > 0

~~Input: x = {1, 1, 2};~~
~~Output original: -1~~
Output mutant: -1

Input: x = {0, 1, 2};
Output original: 0
Output mutant: -1

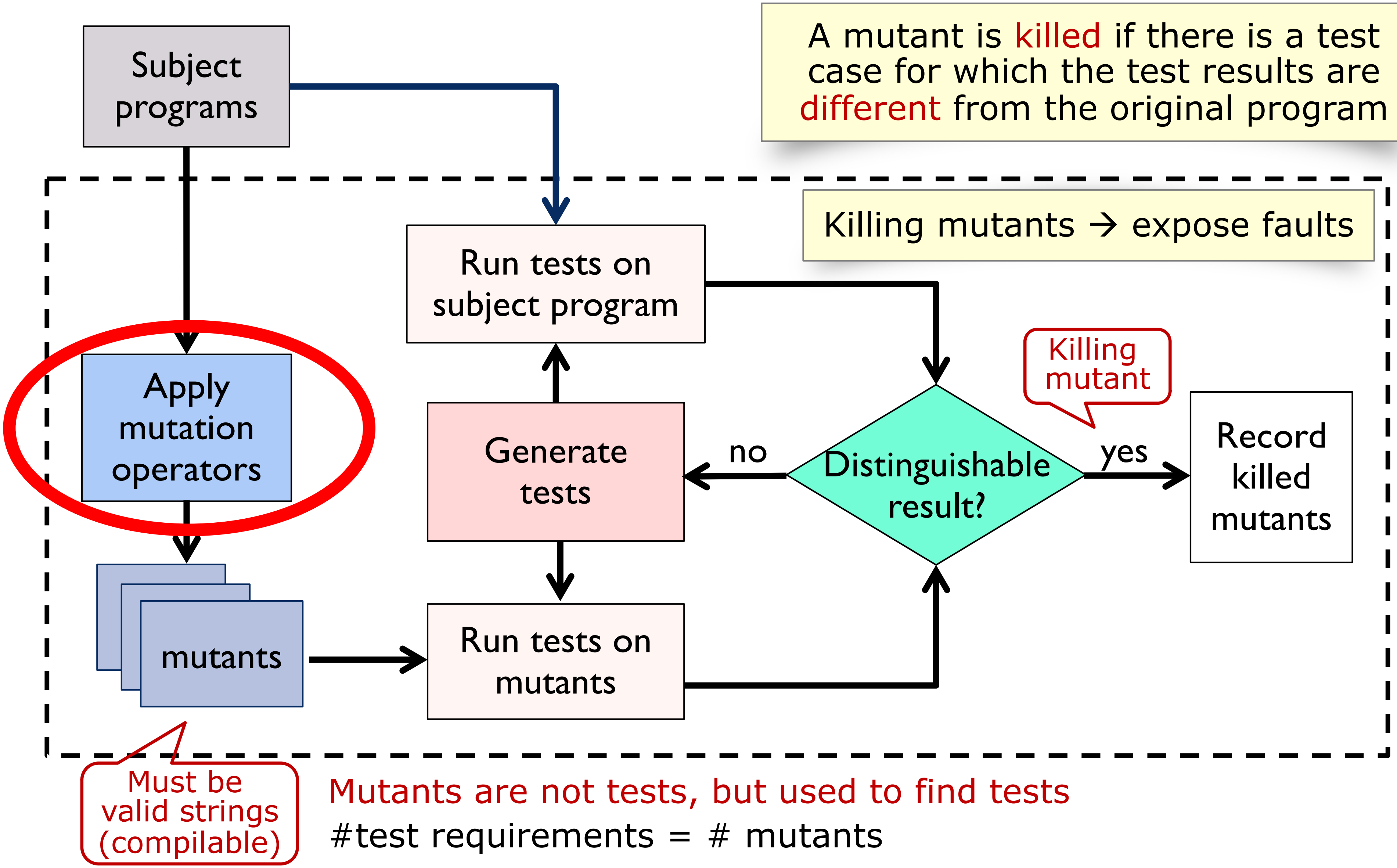
Ineffective

Killed!

Test x = {0, 1, 2};
Very effective at exploring the boundary case

[Thanks to Professor Lin Deng, Towson University]

Mutation Testing



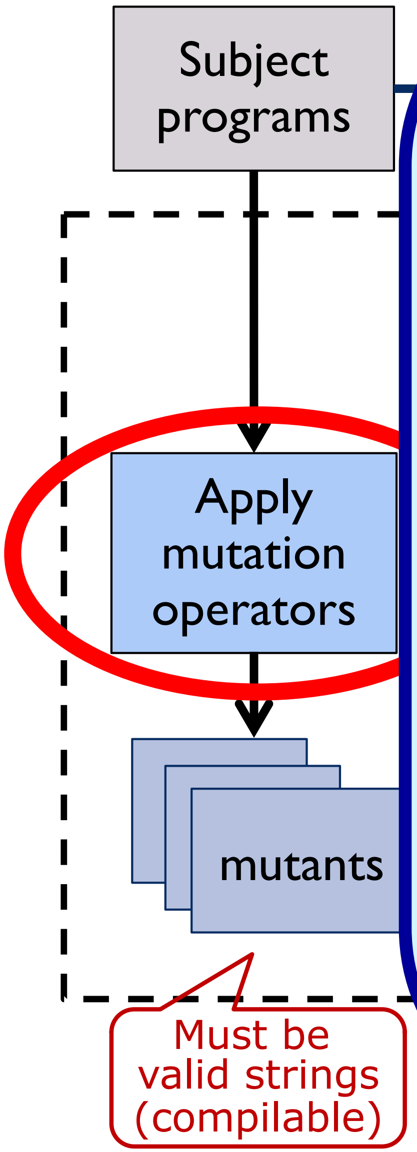
A mutant is killed if there is a test case for which the test results are different from the original program

Killing mutants → expose faults

Mutants are not tests, but used to find tests

#test requirements = # mutants

Mutation Testing



Mutation operators

- Rules that specify how to modify the code (mutate)
- Well designed operators result in powerful tests

Mutation operators do one of two tasks

- Mimic typical programmer mistakes
- Encourage common test heuristics

We use mutation testing to

- Help testers design high quality tests
- Evaluate the quality of existing tests

$$\text{Mutation scores} = \frac{\#\ mutants\ killed}{\#\ non{-}equivalent\ mutants}$$

Killing Mutants

Given a mutant $m \in M$ for a ground string program P and a test t, t is said to kill m if and only if the output of t on P is different from the output of t on m.

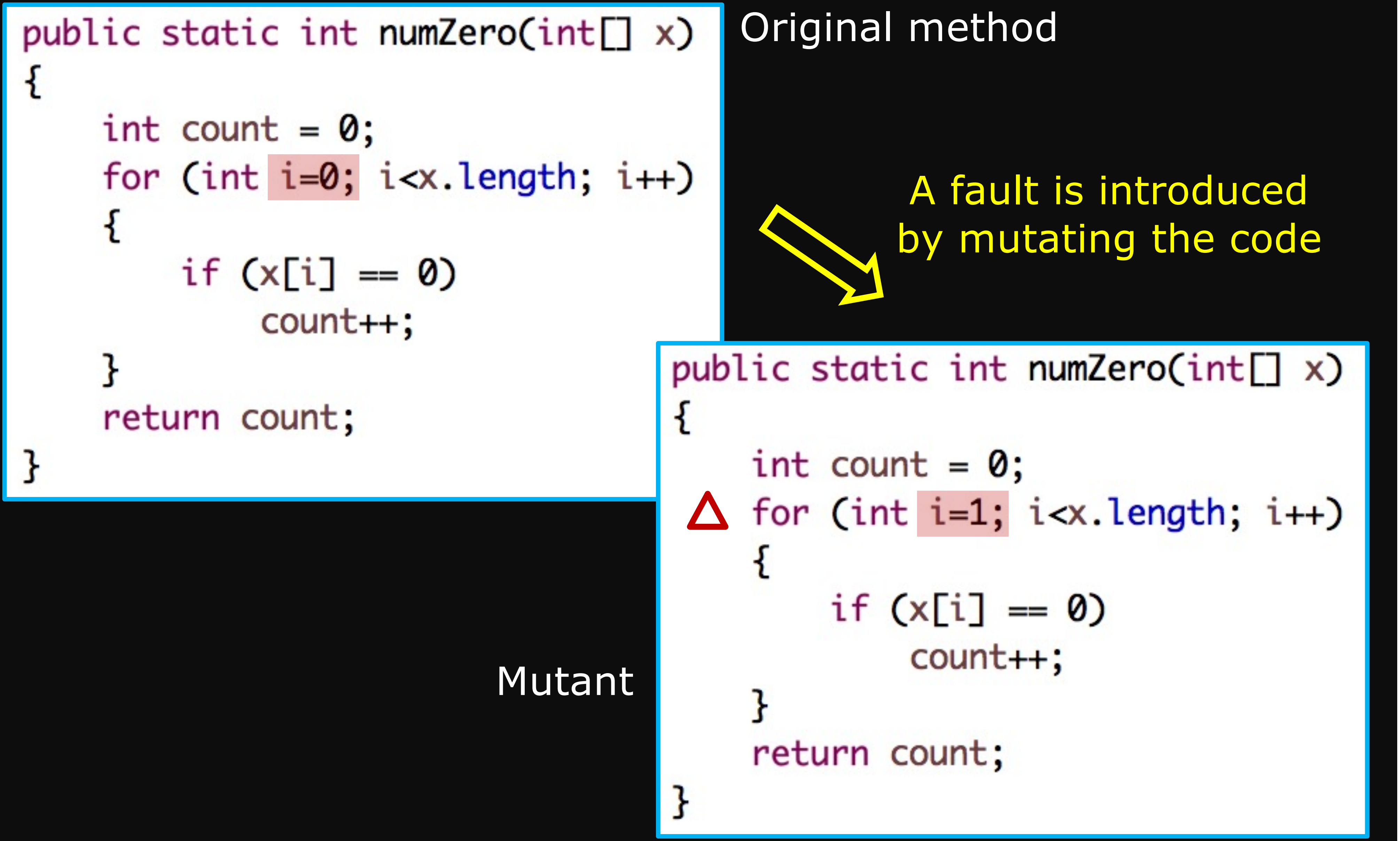
- The quality of tests depends on mutation operators
- Different operators must be defined for different goals (and possibly for different programming languages)
- Testers add tests until all mutants have been killed
  - A mutant is killed if there is a test case for which the test results are different from the original program

Killing mutants ≈ exposing faults

Categories of Mutants

- Dead mutant
  - A test case has killed it
  - The fault that a dead mutant represents will be detected by the same test that killed it
- Uncompilable mutant
  - Syntactically illegal
  - Should not be generated or should be immediately discarded
- Trivial mutant
  - Almost every test can kill it
- (Functionally) equivalent mutant
  - No test can kill it (same behavior or output as original, for all inputs)
  - Infeasible test requirements

Example: Program Mutation



Original method

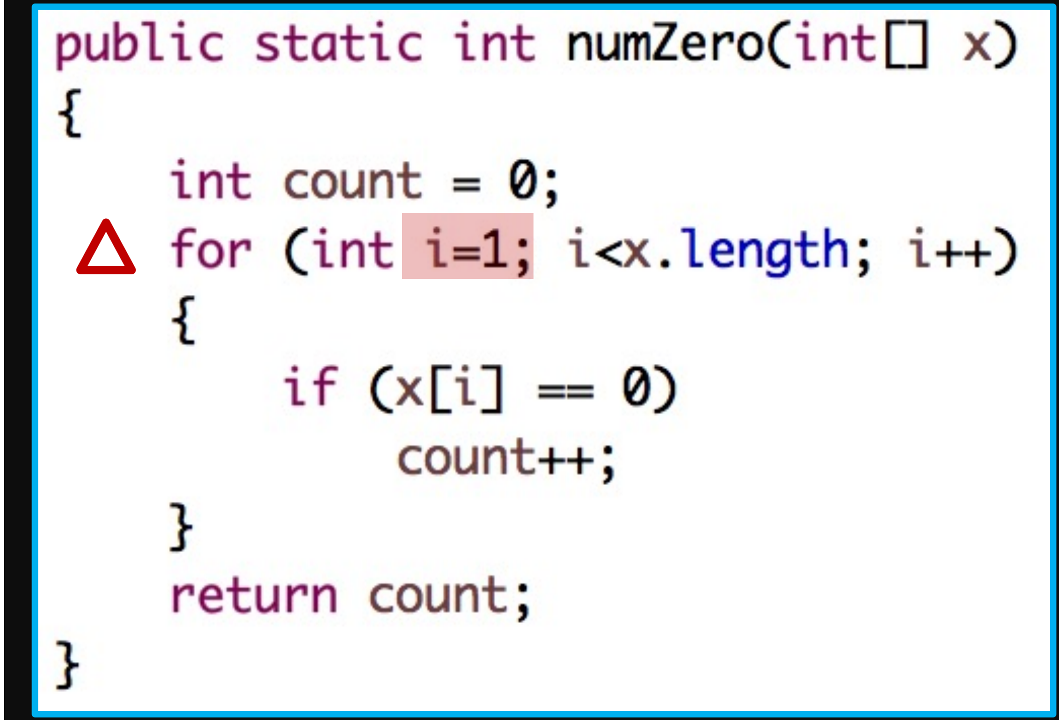
```
public static int numZero(int[] x)
{
    int count = 0;
    for (int i=0; i<x.length; i++)
    {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```

A fault is introduced by mutating the code

Mutant

```
public static int numZero(int[] x)
{
    int count = 0;
△   for (int i=1; i<x.length; i++)
    {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```

Example: Program Mutation



```
public static int numZero(int[] x)
{
    int count = 0;
△   for (int i=1; i<x.length; i++)
    {
        if (x[i] == 0)
            count++;
    }
    return count;
}
```

- i=1 is a mutation of i=0
- The code obtained by changing i=0 to i=1 is called a mutant of numZero
- A test kills the mutant if the mutant yields different outputs from the original code

- Consider t1 = {1, 0, 0}
    - Original returns 2, mutant returns 2, the mutant is not killed
- Consider t2 = {0, 1, 0}
    - Original returns 2, mutant returns 1, the mutant is killed

Example 2

Original method

```
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}
```

mutant1

```
public static int min(int x, int y)
{
    int v;
△   if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```

Each mutated statement represents a separate program

mutant2

```
public static int min(int x, int y)
{
    int v;
△   if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

mutant3

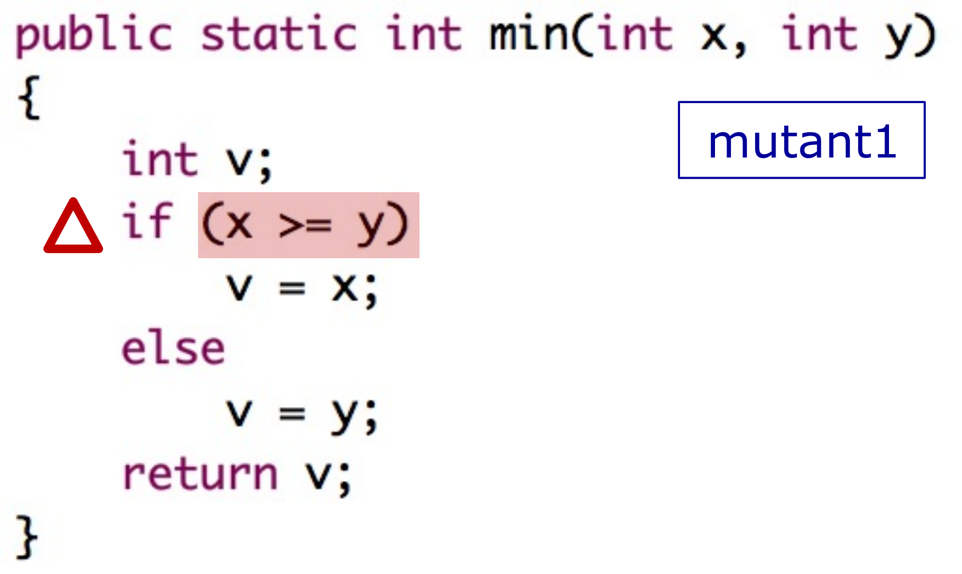
```
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
△       v = -y;
    return v;
}
```

Example 2

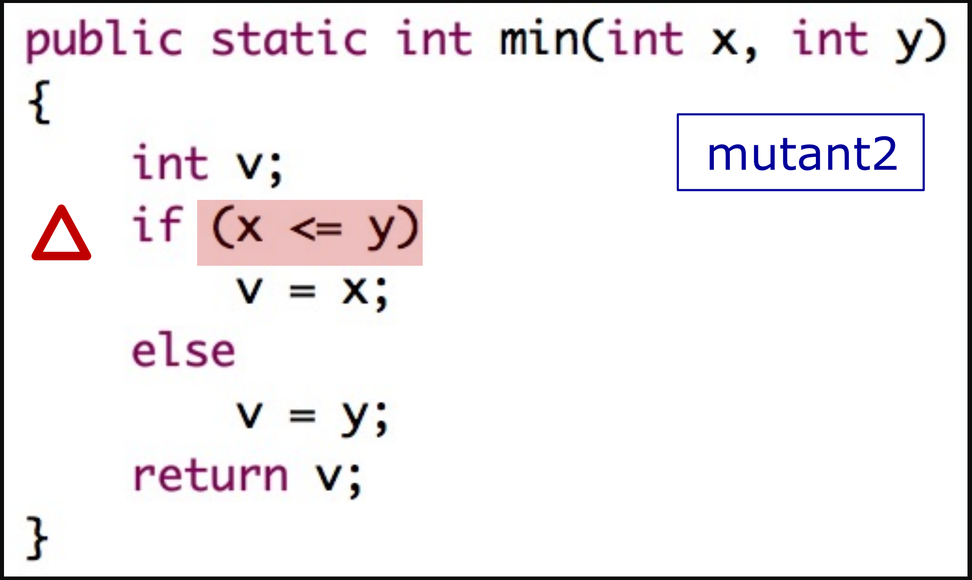
Consider the following tests

- t1 = min(0, 0)
- t2 = min(0, 1)
- t3 = min(1, 0)

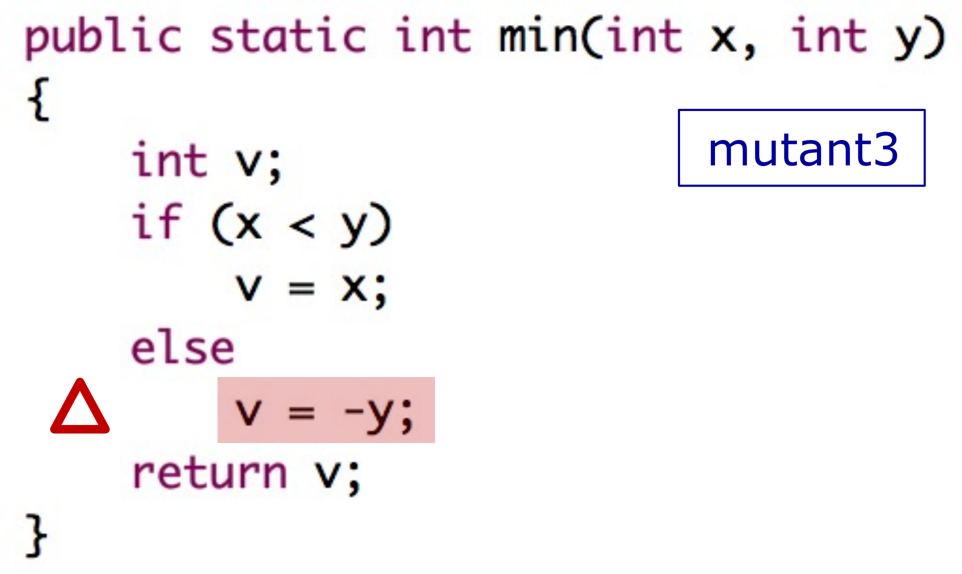
Which mutants will be killed by which tests?


mutant1

```
public static int min(int x, int y)
{
    int v;
△   if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```


mutant2

```
public static int min(int x, int y)
{
    int v;
△   if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

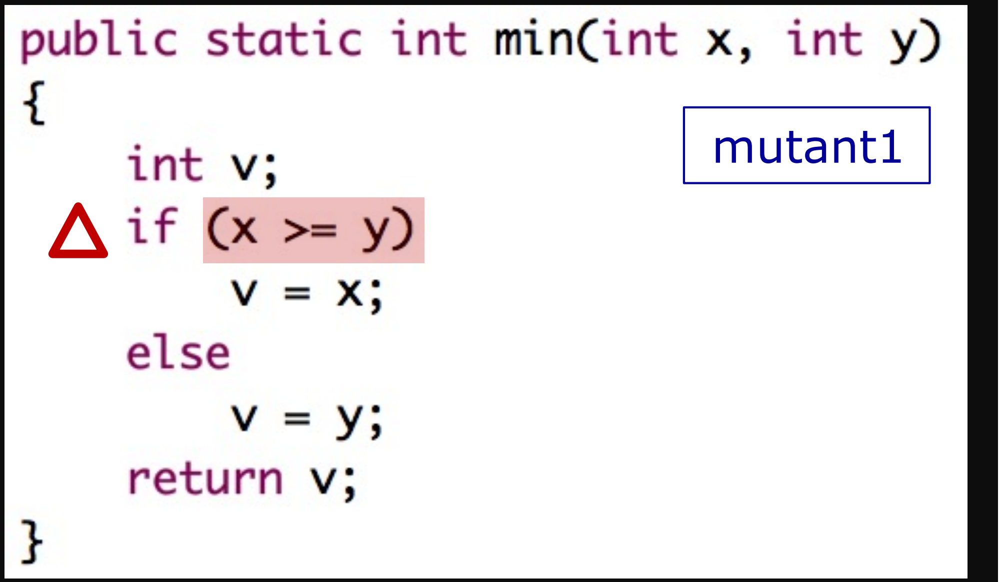

mutant3

```
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
△       v = -y;
    return v;
}
```

Example 2

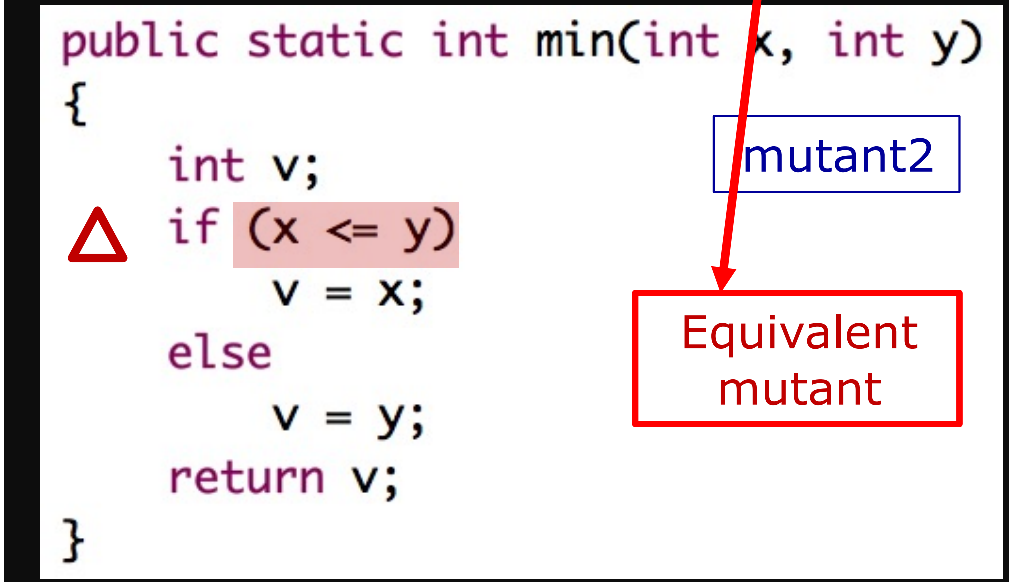
	x	y	min	m1	m2	m3
t1	0	0	0	0	0	0
t2	0	1	0	1	0	0
t3	1	0	0	1	0	0

- t1 kills none of the mutants
- t2 kills m1
- t3 kills m1



mutant1

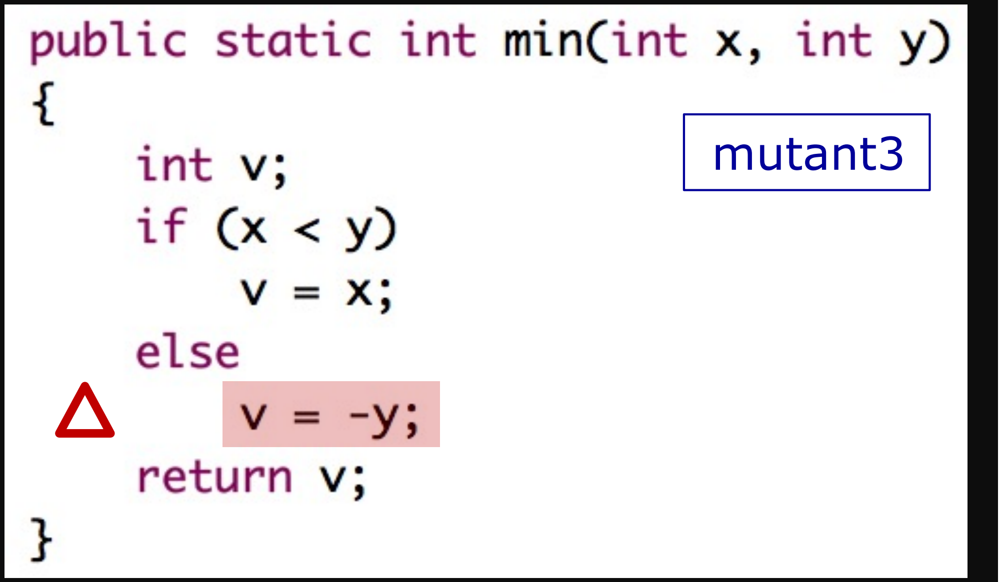
```
public static int min(int x, int y)
{
    int v;
△   if (x >= y)
        v = x;
    else
        v = y;
    return v;
}
```



mutant2

```
public static int min(int x, int y)
{
    int v;
△   if (x <= y)
        v = x;
    else
        v = y;
    return v;
}
```

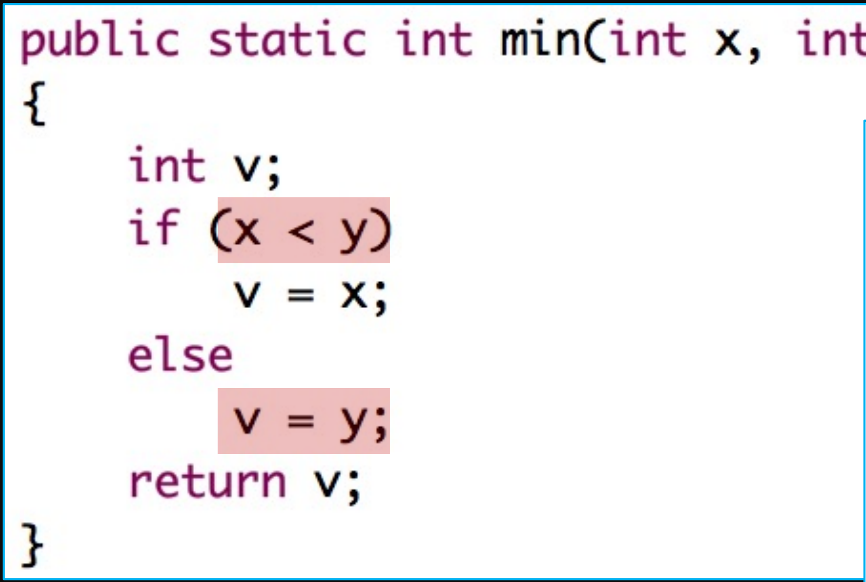
Equivalent mutant



mutant3

```
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
△       v = -y;
    return v;
}
```

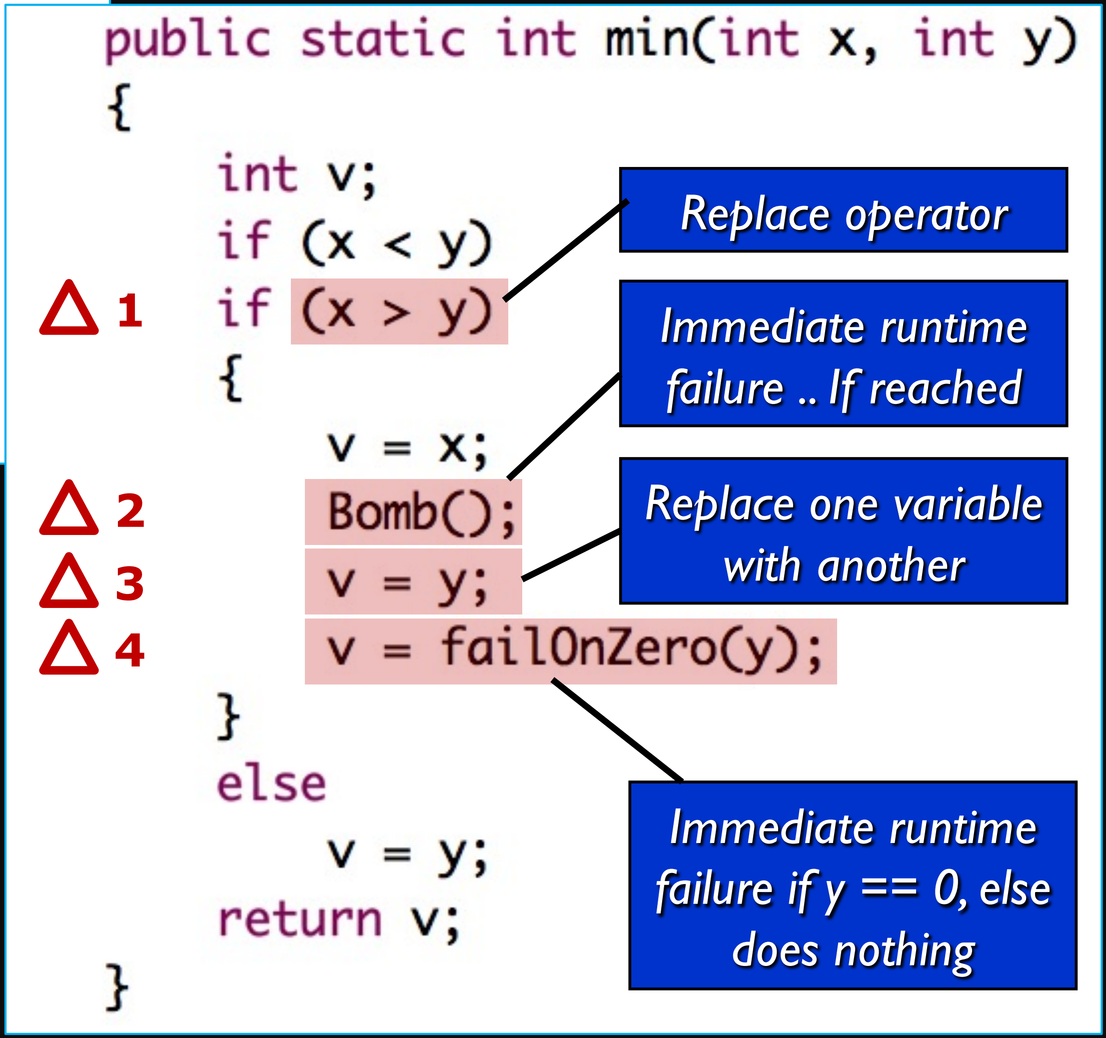
Example 3



```
public static int min(int x, int y)
{
    int v;
    if (x < y)
        v = x;
    else
        v = y;
    return v;
}
```

Original method

With embedded mutants



```
      public static int min(int x, int y)
      {
          int v;
          if (x < y)
△ 1       if (x > y)
          {
              v = x;
△ 2           Bomb();
△ 3           v = y;
△ 4           v = failOnZero(y);
          }
          else
              v = y;
          return v;
      }
```

Replace operator

Immediate runtime failure .. If reached

Replace one variable with another

Immediate runtime failure if y == 0, else does nothing

Mutant 4: force the tester to create tests that cause every variable and expression to have the value of zero

Mutation Coverage

Mutation Coverage (MC): For each $m \in M$, TR contains exactly one requirement, to kill m.

- The RIPR model
  - Reachability: the test causes the faulty (mutated) statement to be reached
  - Infection: the test causes the faulty statement to result in an incorrect state
  - Propagation: the incorrect state propagates to incorrect output
  - Revealability: the tester must observe part of the incorrect output
- The RIPR model leads to two variants of mutation coverage: Strong mutation and Weak mutation

1. Strong Mutation Coverage

Strong Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement, to strongly kill *m*.

- Require RIPR

Output of running a test set on the original program

$\neq$

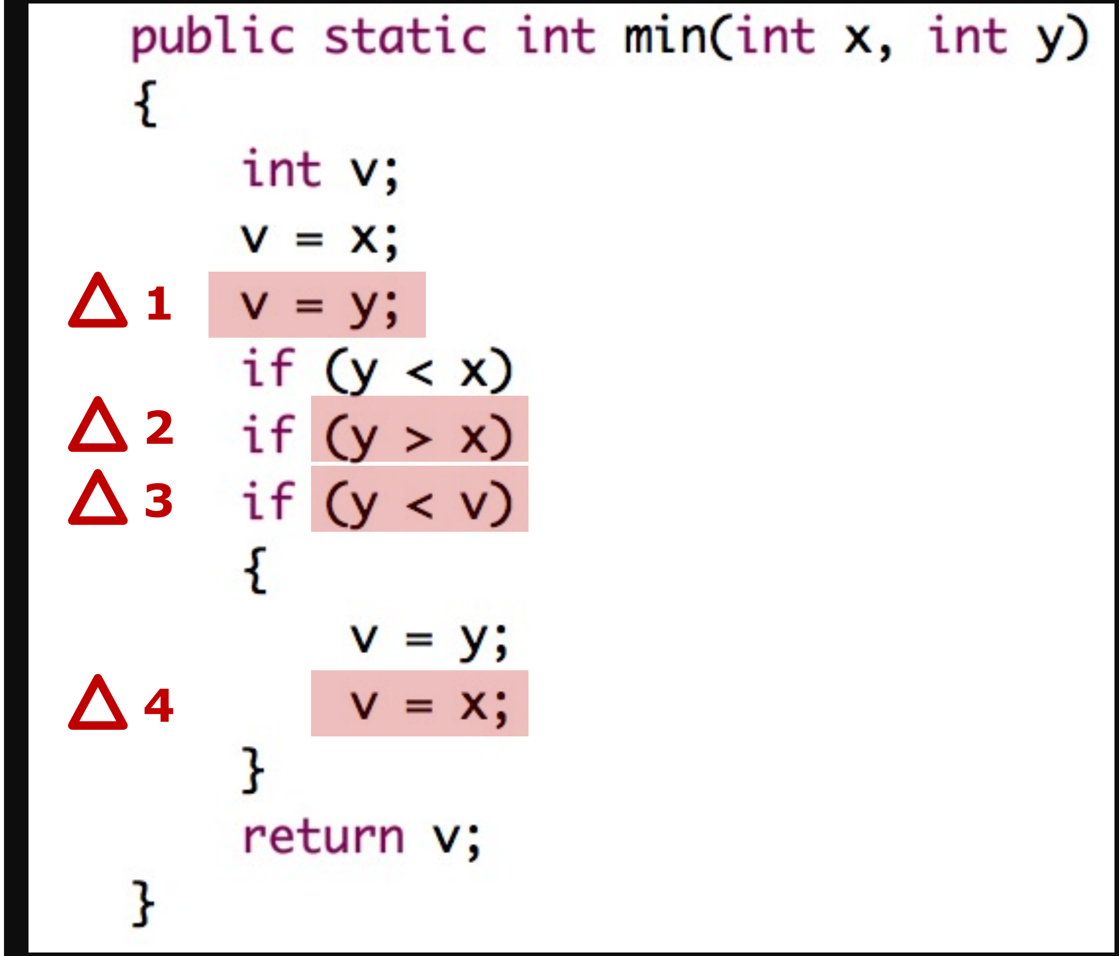
Output of running a test set on a mutant

2. Weak Mutation Coverage

Weak Mutation Coverage (WMC): For each $m \in M$, TR contains exactly one requirement, to weakly kill *m*.

- Require RI-R
  - Check internal state immediately after execution of the mutated statement
  - If the state is incorrect, the mutant is killed
- A few mutants can be killed under weak mutation but not under strong mutation (no propagation)
  - Incorrect state does not always propagate to the output
- Test sets that weakly kill all mutants also strongly kill most mutants

Example (Mutant 1)



```
       public static int min(int x, int y)
       {
          int v;
          v = x;
△ 1       v = y;
          if (y < x)
△ 2       if (y > x)
△ 3       if (y < v)
          {
             v = y;
△ 4          v = x;
          }
          return v;
       }
```

Consider mutant 1

Reachability: true

Infection: x ≠ y

Propagation: (y < x) = false

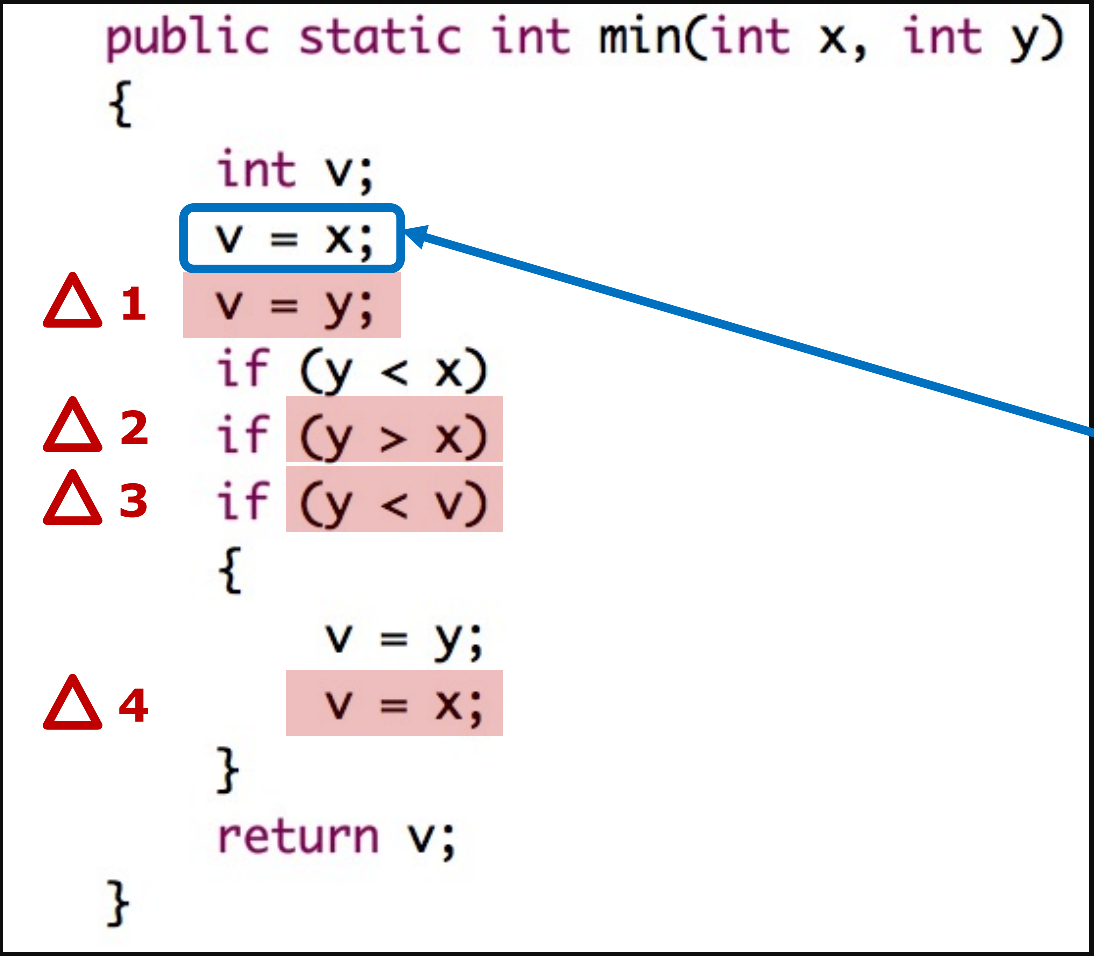
Full test specification:

$$\text{true} \wedge (x \neq y) \wedge ((y < x) = \text{false})$$
$$\equiv (x \neq y) \wedge (y \geq x)$$
$$\equiv (y > x)$$

Test case value:

(x = 3, y = 5) strongly kill, weakly kill mutant 1
(x = 5, y = 3) weakly kill, but not strongly kill

Example (Mutant 3)



```
      public static int min(int x, int y)
      {
          int v;
          v = x;
Δ 1       v = y;
          if (y < x)
Δ 2       if (y > x)
Δ 3       if (y < v)
          {
              v = y;
Δ 4           v = x;
          }
          return v;
      }
```

Consider mutant 3

Reachability: true

Infection: (y < x) != (y < v)

However, the previous statement was v = x

Substitute the infection condition, we get

(y < x) != (y < x)

"Logical contradiction"

No input can kill this mutant … "Equivalent mutant"

Designing Mutation Operators

Mutation Operators do one of two tasks:

- Mimic typical programmer mistakes
- Encourage common test heuristics

What are some of the common mistakes you may have made when writing programs?

Designing Mutation Operators (cont.)

Mutation Operators do one of two tasks:

- Mimic typical programmer mistakes
- Encourage common test heuristics

Researchers design many operators, then experimentally

- Select the most useful operators
- Remove the redundant operators

Effective Mutation Operators

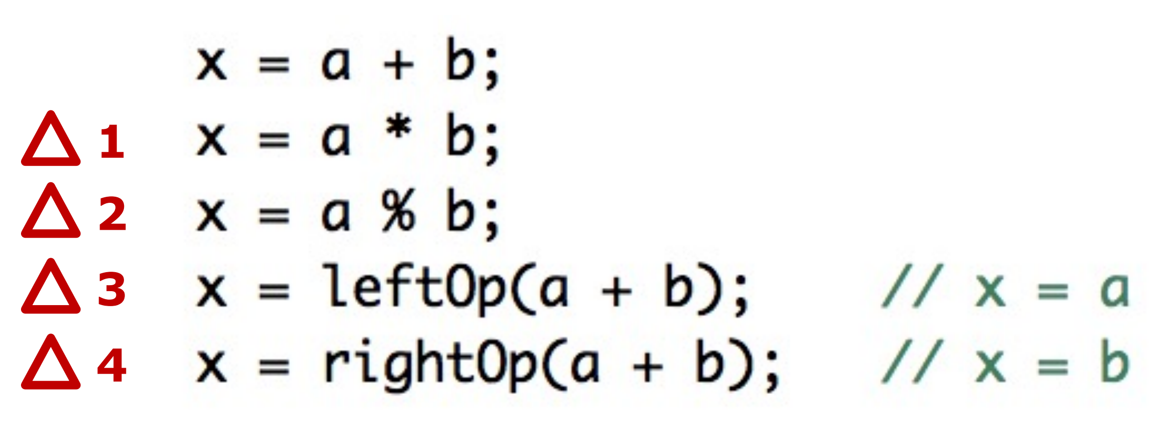
- If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o1, o2, \ldots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an *effective* set of mutation operators

Example: Mutation Ops for Java Programs

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, —, *, ⁄, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators *leftOp*, and *rightOp*.

Example:



```
     x = a + b;
Δ 1  x = a * b;
Δ 2  x = a % b;
Δ 3  x = leftOp(a + b);    // x = a
Δ 4  x = rightOp(a + b);   // x = b
```

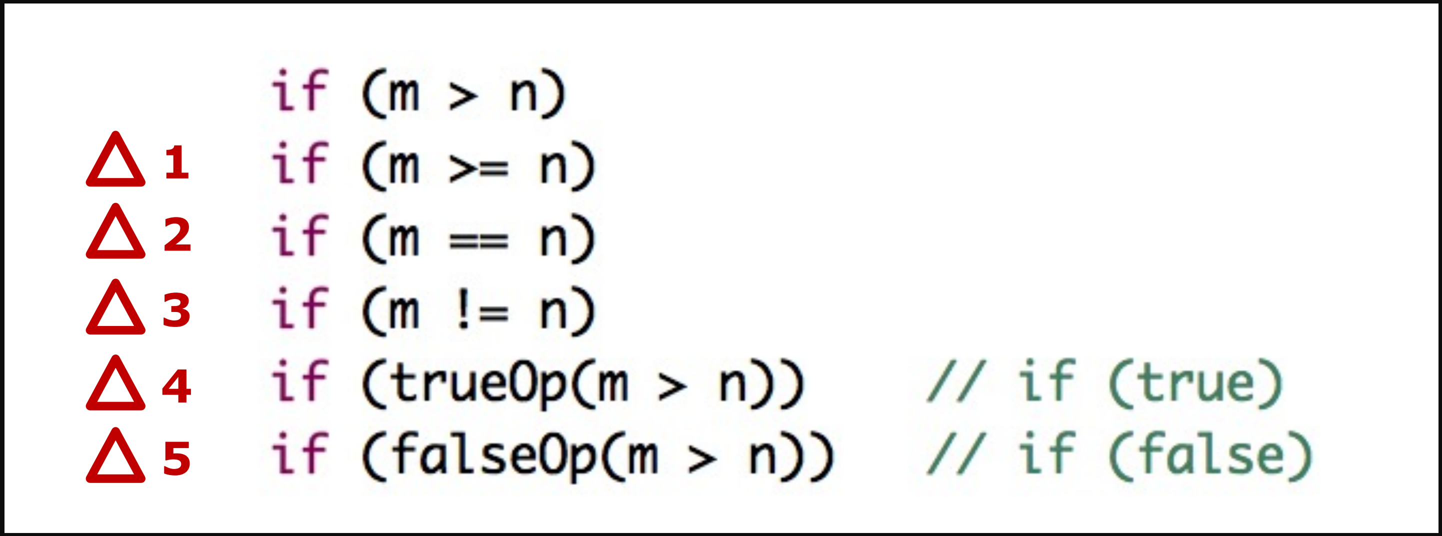
[http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf]

Example:
Mutation Ops for Java Programs

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by falseOp and trueOp.

Example:



```
     if (m > n)
Δ 1  if (m >= n)
Δ 2  if (m == n)
Δ 3  if (m != n)
Δ 4  if (trueOp(m > n))    // if (true)
Δ 5  if (falseOp(m > n))   // if (false)
```

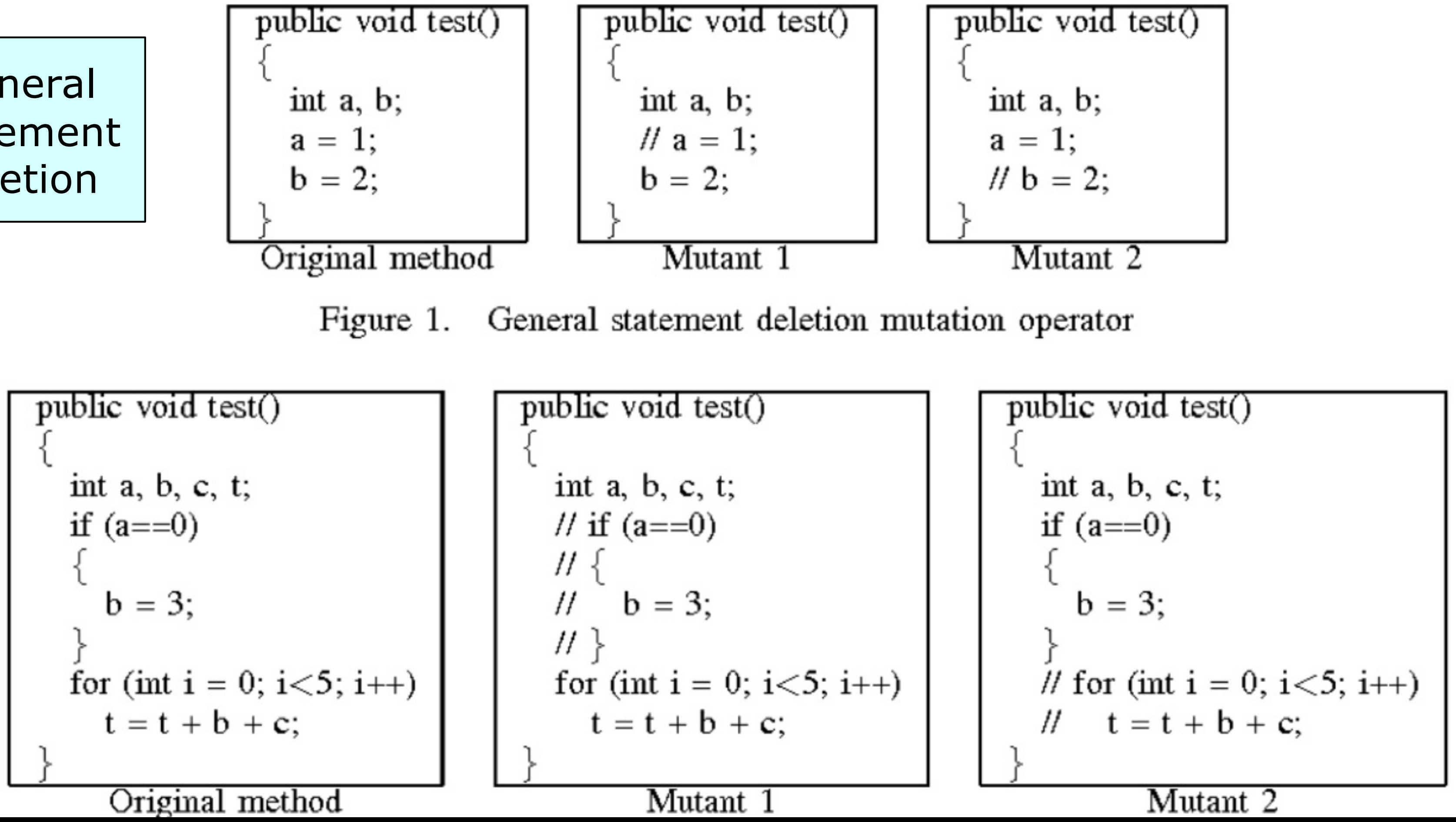
[http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf]

Example: Mutation Ops for Java Program

SDL – Statement Deletion

SDL deletes each executable statement by commenting them out. It does not delete declarations.

General statement deletion



```
public void test()
{
    int a, b;
    a = 1;
    b = 2;
}
```

Original method

```
public void test()
{
    int a, b;
    // a = 1;
    b = 2;
}
```

Mutant 1

```
public void test()
{
    int a, b;
    a = 1;
    // b = 2;
}
```

Mutant 2

Figure 1. General statement deletion mutation operator

```
public void test()
{
    int a, b, c, t;
    if (a==0)
    {
        b = 3;
    }
    for (int i = 0; i<5; i++)
        t = t + b + c;
}
```

Original method

```
public void test()
{
    int a, b, c, t;
    // if (a==0)
    // {
    //     b = 3;
    // }
    for (int i = 0; i<5; i++)
        t = t + b + c;
}
```

Mutant 1

```
public void test()
{
    int a, b, c, t;
    if (a==0)
    {
        b = 3;
    }
    // for (int i = 0; i<5; i++)
    //     t = t + b + c;
}
```

Mutant 2

[http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf]

Example:
Mutation Ops for Web Apps

FOB – FailOnBack

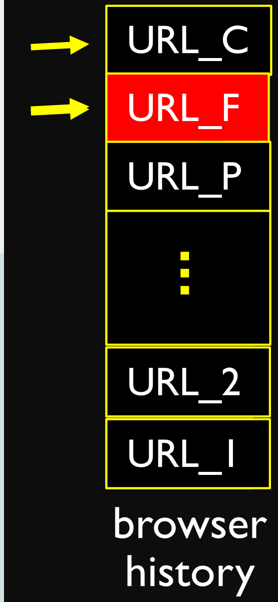
```
<html>
   …
     <body>
△    <body onload="manipulatehistory()">
          <script src="failOnBack.js"></script>
          …
</html>
```

failOnBack.js

```
function manipulatehistory()
{
   var currentpage = window.document.toString();
   var currenturl = window.location.href;
   var pageData = window.document.toString();

   // add a dummy url right before the current url
   history.replaceState( pageData, "dummyurl", "failonback.html" );
   history.pushState( currentpage, "currenturl", currenturl);
}

// update the page content
window.addEventListener( 'popstate', function(event) {
   window.location.reload();     });
```



[https://cs.gmu.edu/~offutt/documents/theses/UpsornPraphamontripong-Dissertation.pdf]

Example: Mutation Ops for Web Apps

WSCR – Scope replacement

```
<html>
    …
    <jsp:useBean id = id_l  scope = “page”  class = class_l />
△   <jsp:useBean id = id_l  scope = “session”  class = class_l />
    …
</html>
```

WSIR – Session initialization replacement

```
Public class logout extends HttpServlet
{
     public void doGet( … )
     {
          session = request.getSession(true);
△         session = request.getSession(false);
           …
     } }
```

WSAD – Session setAttribute deletion

```
Public class logout extends HttpServlet
{
     public void doGet( … )
     {
          session.setAttribute(attr_l , value_l );
△       //  session.setAttribute(attr_l , value_l );
            …
     } }
```

[https://cs.gmu.edu/~offutt/documents/theses/UpsornPraphamontripong-Dissertation.pdf]

Example: Mutation Ops for Android Apps

- OnClick Event Replacement (ECR)
  - Replaces event handlers with other compatible handler

```
mPrepUp.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        decrementPrepTime (); }
});
mPrepDown.setOnClickListener (new OnClickListener() {
    public void onClick (View v) {
        decrementPrepTime (); }
});
```

- OnTouch Event Replacement (ETR)
  - Replaces OnTouch events, similar to ECR

[https://cs.gmu.edu/~offutt/documents/theses/LinDeng-Dissertation.pdf]

Mutation Testing in Practice

- Strongest test criterion but very difficult + expensive to apply
- Subsumes other criteria by including specific mutation operators
- First-order mutation due to Competent programmers and coupling effect

Do fewer

- Selective mutation operators
- Removing redundancy

Do smarter

- Weak mutation
- Distributed execution

Do faster

- Schemata

+

Automation

- Mutant generation
- Mutant execution

Summary

- Mutation is very effective – the “gold standard” of testing
- Used to evaluate other criteria
- Applied to various software artifacts, languages, frameworks with different implementation and specific definition of mutation operators
- Most expensive … # test requirements = # mutants
- Very difficult to apply by hand – need automation
- To improve the test process, use selective mutation operators