# BCS3323 – Software Testing and Maintenance

## Mutation Testing

**Editors**

**Dr. AbdulRahman A. Alsewari**
**Faculty of Computer Systems & Software Engineering**
**alswari@ump.edu.my**

# Mutation Testing

Aims is to discover

- The purpose of mutation testing
- How to use this technique to design the test cases

Expected Outcomes

- Students be able to show how to design the test cases based on this technique
- Students be able to show when to use this technique .

References

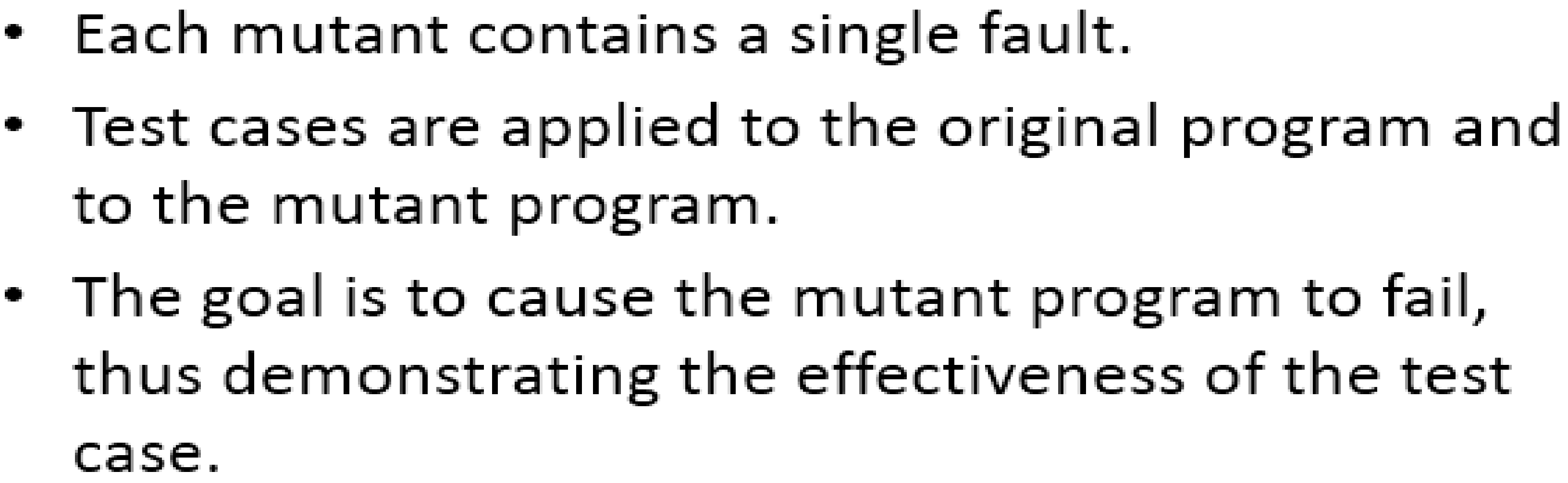
- ISTQB
- MSTB/GTB
- http://www.softwaretestingclass.com/software-testing-tools-list/
- http://www.softwaretestinggenius.com/articalDetails.php?qry=572#commentsList

# What is Mutation Testing?

- Mutation Testing is a testing technique that focuses on measuring the adequacy of test cases.
- Mutation Testing is NOT a testing strategy like path or data-flow testing.  It does not outline test data selection criteria.
- Mutation Testing should be used in conjunction with traditional testing techniques, not instead of them.

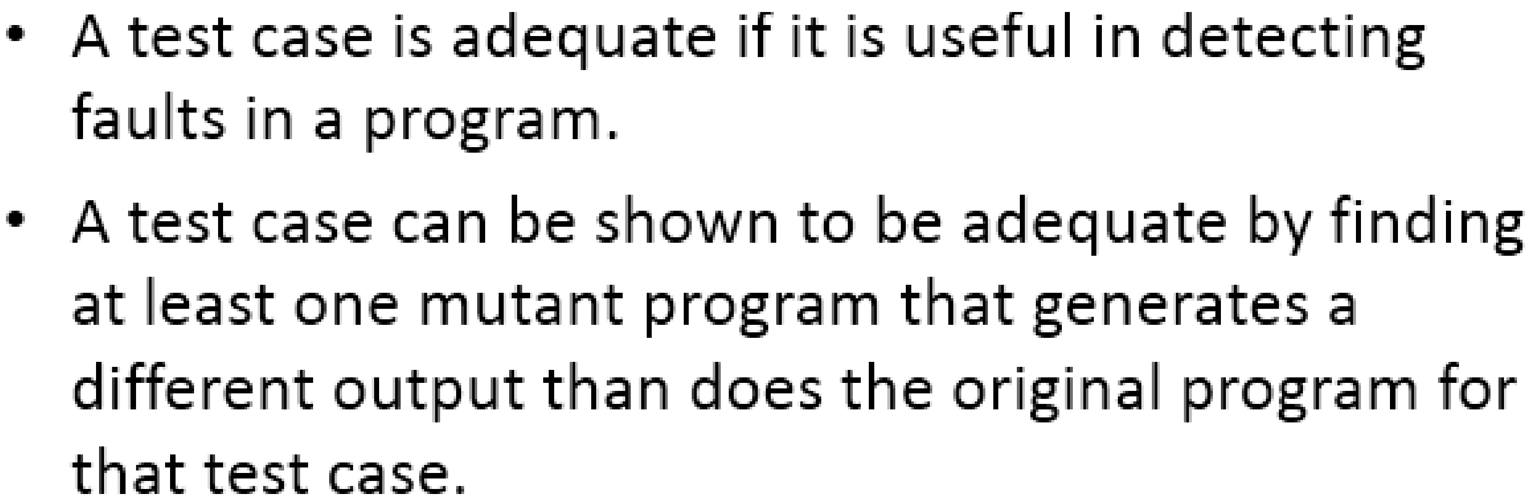
# Mutation Testing

- Faults are introduced into the program by creating many versions of the program called *mutants*.



- Each mutant contains a single fault.
- Test cases are applied to the original program and to the mutant program.
- The goal is to cause the mutant program to fail, thus demonstrating the effectiveness of the test case.

# Test Case Adequacy



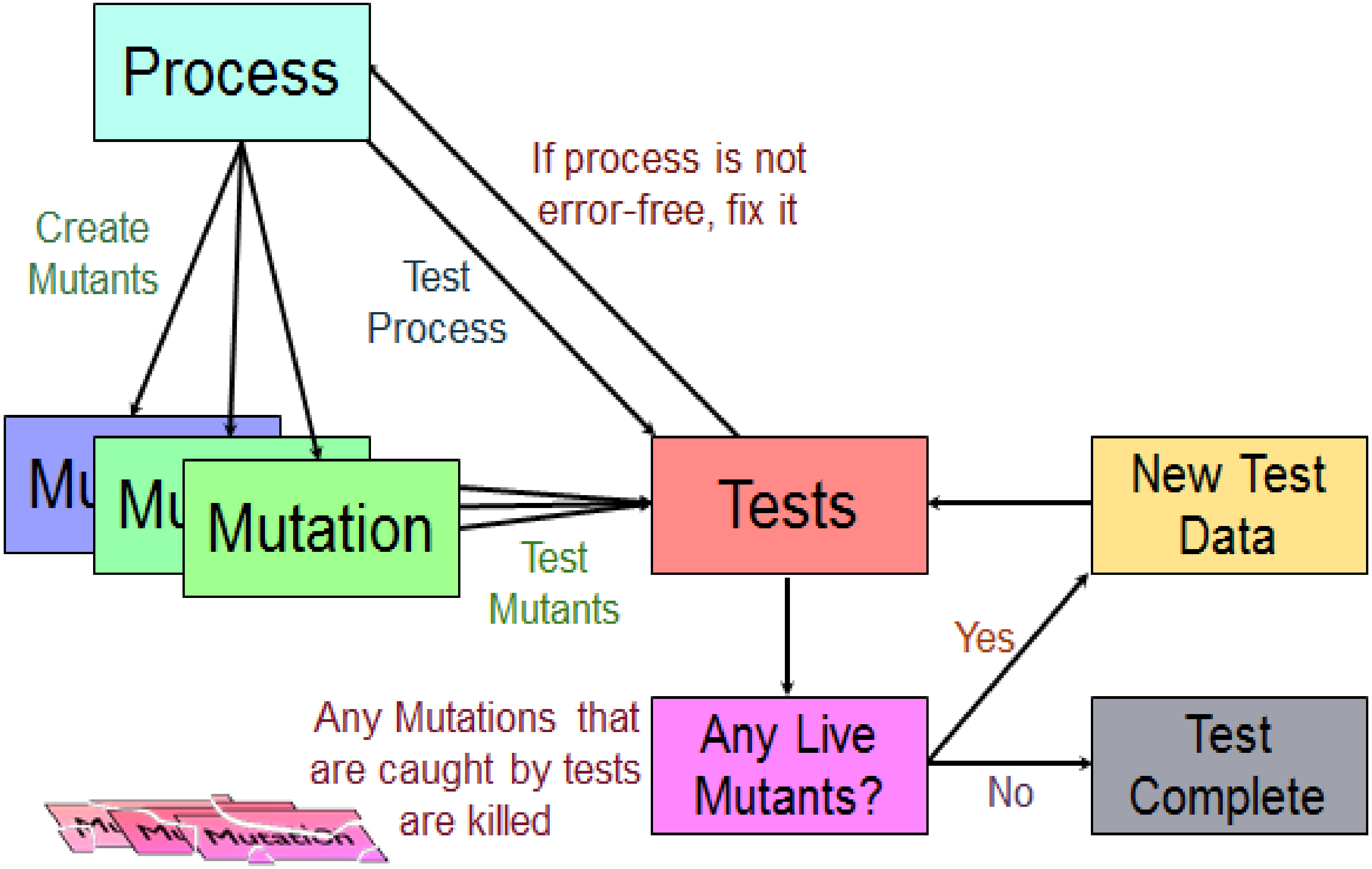
- A test case is adequate if it is useful in detecting faults in a program.
- A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case.

- If  the original program and all mutant programs generate the same output, the test case is inadequate.

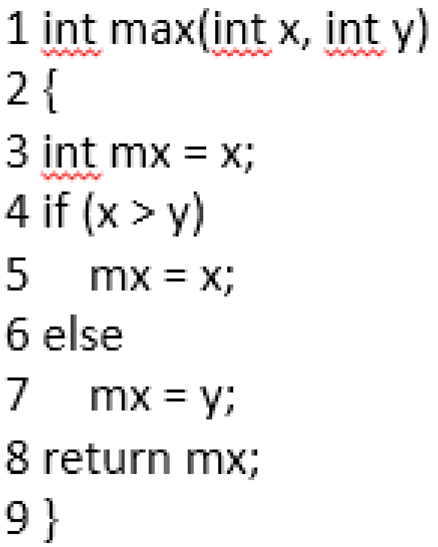
# Mutant Programs

- Mutation testing involves the creation of a set of mutant programs of the program being tested.
- Each mutant differs from the original program by one *mutation*.
- A *mutation* is a single syntactic/operator change that is made to a program statement.

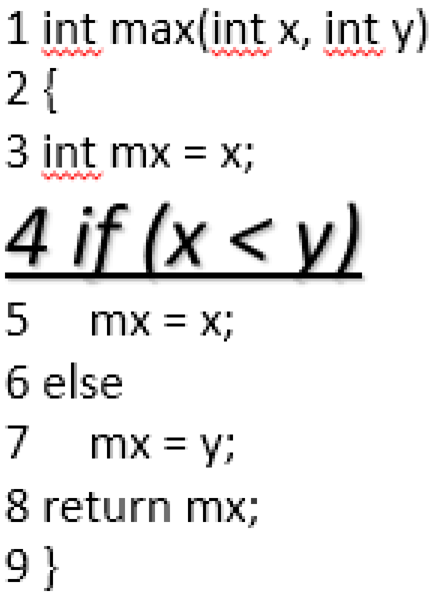
# The Mutation Process



# Example of a Program Mutation

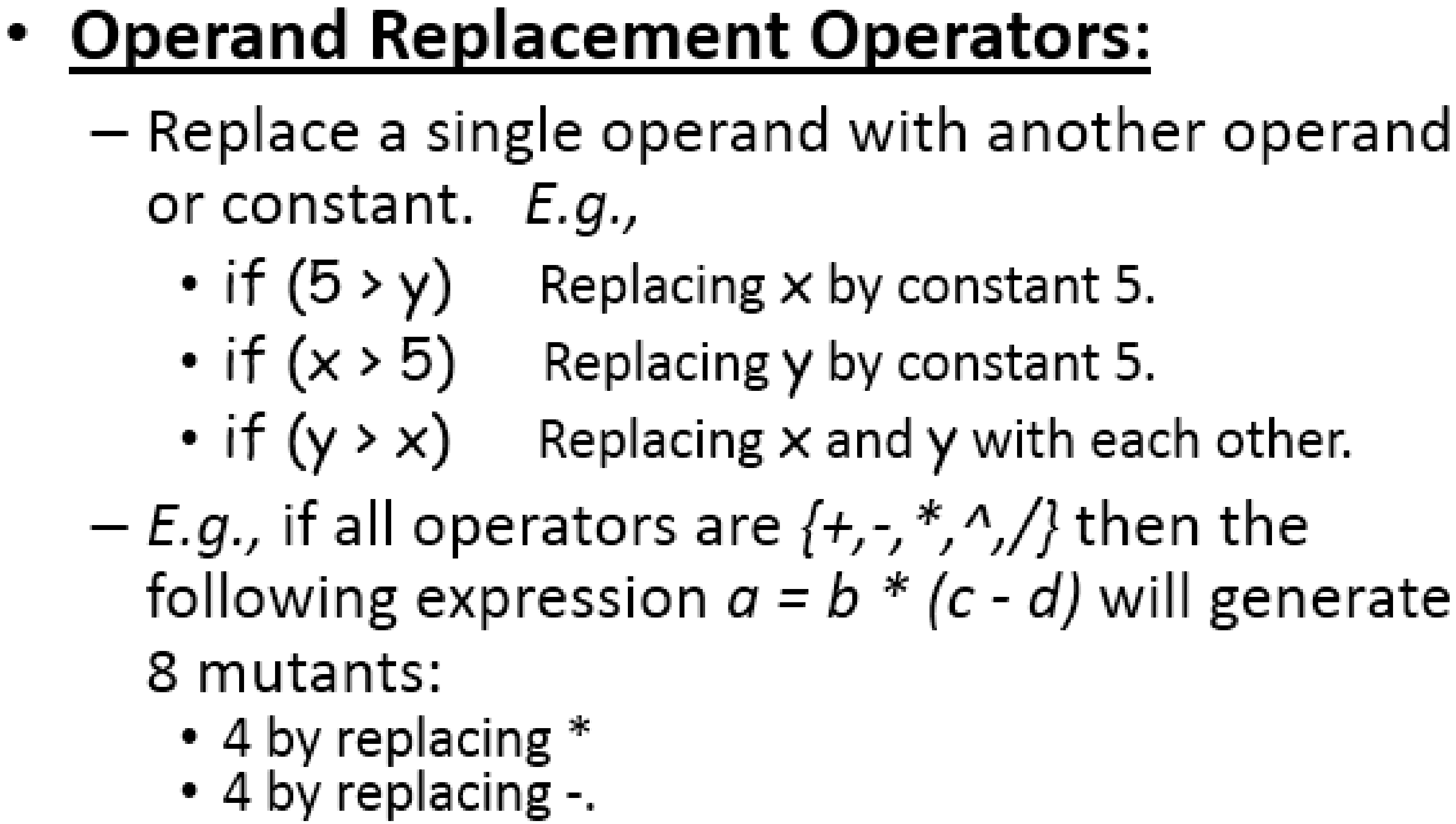


```
int max(int x, int y)
{
int mx = x;
if (x > y)
    mx = x;
else
    mx = y;
return mx;
}
```



```
int max(int x, int y)
{
int mx = x;
if (x < y)
    mx = x;
else
    mx = y;
return mx;
}
```

# Categories of Mutation Operators

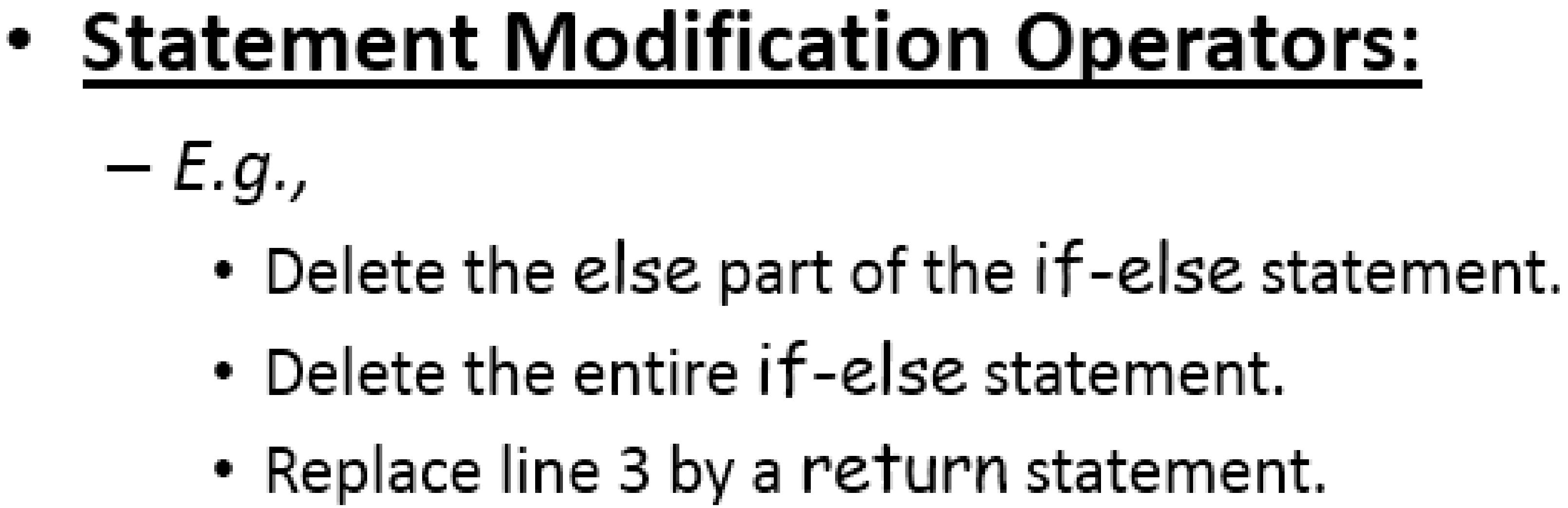


- **Operand Replacement Operators:**
  - Replace a single operand with another operand or constant. *E.g.,*
    - if (5 > y)    Replacing x by constant 5.
    - if (x > 5)    Replacing y by constant 5.
    - if (y > x)    Replacing x and y with each other.
  - *E.g.,* if all operators are *{+,-,*,^,/}* then the following expression *a = b * (c - d)* will generate 8 mutants:
    - 4 by replacing *
    - 4 by replacing -.

# Categories of Mutation Operators

- **Expression Modification Operators:**
  - Replace an operator or insert new operators. *E.g.,*
    - if (x == y)
    - if (x >= y) Replacing == by >=.
    - if (x == ++y) Inserting ++.

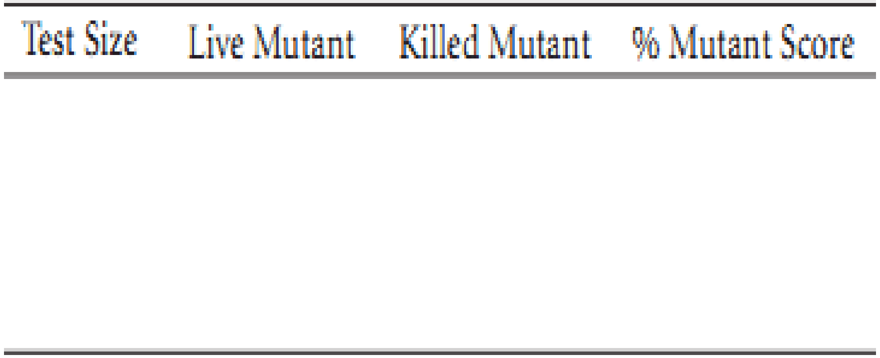
# Categories of Mutation Operators



- **<u>Statement Modification Operators:</u>**
  - *E.g.,*
    - Delete the *else* part of the *if-else* statement.
    - Delete the entire *if-else* statement.
    - Replace line 3 by a *return* statement.

# Exercise

- Consider f(x) = 1/(1-x)
  - Consider mutating f(x) into:
    - f(x) = 1/(1+x)
    - f(x) = 1/(1*x)
    - f(x) = 1/(1^x)
    - f(x) = 1/(1/x)
  - If the values of x = { -1,0,1,10,100}. Generate the test oracle for each of the f(x) mutant. Compare the test oracle against that of original f(x) in order to show which values of x are unsuitable as test values. Hint: Use the given table.



| Test Size | Live Mutant | Killed Mutant | % Mutant Score |
|---|---|---|---|