Software Quality Assurance

CHAPTER 5

UNIT TEST (WHITE BOX TESTING)

DR. JAMIL S. ALAGHA

Software Testing

In software testing will talk about definition of test case and how to generated from project code (white box) and how to execute test cases in code in this chapter.

Next chapter will be about how generate test cases from program (black box).

Outline

Definition

Test Case

Control Flow Testing

Mutation Test

Testing with Junit

Test case

Test case is defined as a sequence of steps to test the correct behavior of a functionality/feature of an application

Test cases Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

Why Test cases

Complete testing is impossible

⇓

Testing cannot guarantee the absence of faults

⇓

How to select subset of test cases from all possible test cases with a high chance of detecting most faults ?

⇓

Test Case Design Strategies

Test case design

During test planning, have to design a set of test cases that will detect defects present

Some criteria needed to guide test case selection

Two approaches to design test cases

- functional or black box
- structural or white box

Both are complimentary; we discuss a few approaches/criteria for both

Test cases for white box

To detect error in following code

```
if(x>y) max = x; else max = x;
```

Test case will be:

{(x=3, y=2); (x=2, y=3)}

Exhaustive Testing is Hard

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

Number of possible test cases (assuming 32 bit integers)

- $2^{32} \times 2^{32} = 2^{64}$

Do bigger test sets help?

- Test set
  {(x=3,y=2), (x=2,y=3)}
  will detect the error
- Test set
  {(x=3,y=2),(x=4,y=3),(x=5,y=1)}
  will not detect the error although it has more test cases

It is not the number of test cases

But, if $T_1 \supseteq T_2$, then T_1 will detect every fault detected by T_2

Test cases for Black box

For example, if the application under test was a payroll system.

The test team manager might say, “at this phase of development, we need to test all the data entry screens.”

There might be five groups of data entry screens:

employee personal information,

employee benefits information,

employee tax information,

state tax information,

and federal tax information.

Test Case

It may take many test cases to determine that a requirement is fully satisfied. In order to fully test that all the requirements of an application are met, there must be at least one test case for each requirement unless a requirement has sub requirements. In that situation, each sub requirement must have at least one test case .

Test Case Documentation

Project Name:	
Test case Id:	Test case name:
Module Name:	Test Priority:
Designed By:	Designed Date:
Executed by:	Execution Date:
Short Description:	

Precondition:

Step	Action	Expected Result	Actual Result	Status (Pass/Fail)	Comments
1					
2					

PostCondition:

Test Case Details

A typical test case is laid out in a table, and includes:

1) Project Name
2) Test case Id
3) Test case name
4) Module Name
5) Test Priority
6) Designed By
6) Designed Date
7) Executed by
8) Execution Date
9) Short Description
10) Precondition
11) Step
12) Action
13) Expected Result
14) Actual Result
15) Status
16) Comments
17) PostCondition

Test Case Details

1) *Project Name: Name of the project you want to test.*

Example: Airline reservation system

2) Test case Id—*String of letters and numbers that uniquely identify* this test case document

For example: c_01.1, c_01.1a, c_01.2,…

Test Case Details

3) Test Case Name: This field is the main way to identify a test case. It is a short, direct description about what the test is intended for.

You use this name to identify the test case and search for it if you have to.

For Examples

Create user

Or Track and process Invoiceable Expenses

Test Case Details

4) **Module Name** – Mention name of main module or sub module you want to write the test cases.

For Examples

Login screen

Bubble sorting method

Test Case Details

5) Test priority:

Out of a large cluster of test cases in our hand, we need to scientifically decide their priorities of execution based upon some rational, non-arbitrary, criteria.

Test Case Details

popular three-level priority categorization scheme is described as under

High: Allocated to all tests that must be executed in any case.

Medium: Allocated to the tests which can be executed, only when time permits.

Low: Allocated to the tests, which even if not executed, will not cause big upsets.

Test Case Details

6) Test Designed By:

Name of tester.

For example: Mohamed Ahmed

7) Test Designed Date:

Date when wrote

For example: 12-1-2015

Test Case Details

8) **Test Executed By:** Name of tester who executed this test. To be filled after test execution.

For example: Mohamed Ahmed

9) **Test Execution Date:** Date when test executed.

For example: 15-3-2015

Test Case Details

10) Short Description: describes the purpose of a specific test.

For example: For Test Case Description: To verify that Login name on login page must be greater than 3 characters

Test Case Details

11) Preconditions which describe the state of the software before the test case. It is the settings/conditions required to execute the test case.

For Example: Precondition :Software should be install on system, login page should be available on desktop.

Test Case Details

12) Step: describes the step sequence

For example: 1

13) Action: *Actions* that describe the specific steps which make up the interaction.

It describes in a detailed sequence what have to be done in order to achieve the test case results.

For example: Enter login name less than 3 chars (i.e. “a”) and password and click Submit button

Test Case Details

14) Expected Results which describe the expected state of the software after the test case is executed. Describe the expected result in detail including message/error that should be displayed on screen.

For example:

an error message “Login not less than 3 characters” must be displayed

Test Case Details

15) Actual Results which describe the actual state of the software after the test case is executed.

The actual result of the test; to be filled after executing the test.

Test Case Details

16) Status : Does the case pass or fail.

17) Comments: Any additional comments about this step.

18) Post-condition: What should be the state of the system after executing this test case?

For example: The user will login inside the system.

Test Case Details

For example:

For the step "enter login name less than 3 chars (say a) and password and click Submit button"

Test Case Details

10) Test priority:

Out of a large cluster of test cases in our hand, we need to scientifically decide their priorities of execution based upon some rational, non-arbitrary, criteria

Test Case Example

Project Name: Math package	
Test case Id: T_1	Test case name: Testing maximum
Module Name: Max	Test Priority: high
Designed By: A.M	Designed Date: 12-2-2015
Executed by: A.M	Execution Date:3-4-2015
Short Description: Testing maximum method	

Precondition: max method code is ready

Step	Action	Expected Result	Actual Result	Status (Pass/Fail)	Comments
1	X=1,Y=3	Y			
2	X=1,Y=-3	X			

PostCondition:

Test Case Example

Test Case #: 2.2	**Test Case Name:** Change PIN	**Page: 1 of 1**
System: ATM	**Subsystem: PIN**	
Designed by: ABC	**Design Date: 28/11/2004**	
Executed by:	**Execution Date:**	
Short Description: Test the ATM Change PIN service		

Pre-conditions
The user has a valid ATM card - The user has accessed the ATM by placing his ATM card in the machine
The current PIN is 1234
The system displays the main menu

Step	Action	Expected System Response	Pass/ Fail	Comment
1	Click the 'Change PIN' button	The system displays a message asking the user to enter the new PIN		
2	Enter '5555'	The system displays a message asking the user to confirm (re-enter) the new PIN		
3	Re-enter '5555'	The system displays a message of successful operation The system asks the user if he wants to perform other operations		
4	Click 'YES' button	The system displays the main menu		
5	**Check post-condition 1**			

Post-conditions
1. The new PIN '5555' is saved in the database

Unit Testing

- Individual component (class or subsystem)
- Carried out by developers
- <u>Goal:</u> Confirm that the component or subsystem is correctly coded and carries out the intended functionality

White Box testing

Software testing approach that uses inner structural and logical properties of the program for verification and deriving test data

Also called: Clear Box Testing, Glass Box Testing and Structural Testing

Outline

Definition

Test Case

Control Flow Testing

Testing with Junit

Mutation Test

Control-Flow Testing

Traditional form of white-box testing

Control-flow testing is a structural testing strategy that uses the program's control flow as a model.

Control-flow testing techniques are based on wisely selecting a set of test paths through the program.

The set of paths chosen is used to achieve a certain measure of testing thoroughness.

- *E.g.,* pick enough test cases assure that every source statement is executed as least once.

Control-flow-based Testing

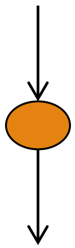
Step 1: From the source code, create a graph describing the flow of control Called the control flow graph. The graph is created (extracted from the source code) manually or automatically

Step 2: Design test cases to cover certain elements of this graph.

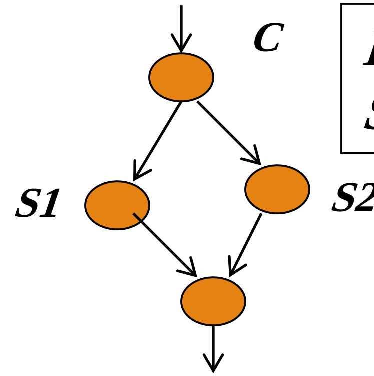
Flowgraphs Consist of Three Primitives

- A **decision** is a program point at which the control can diverge.
  - (*e.g.,* if and *case* statements).
- A **junction** is a program point where the control flow can merge.
  - (*e.g.,* end if, end loop, goto label)
- A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.,* straight-line code).
  - A process has one entry and one exit.
  - A program does not jump into or out of a process.

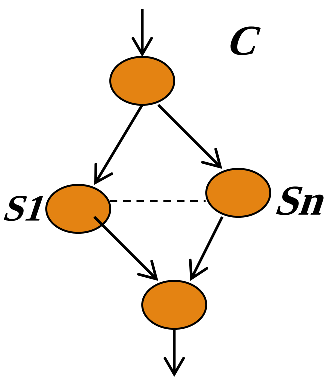
Control Flow Testing



Sequential statement block

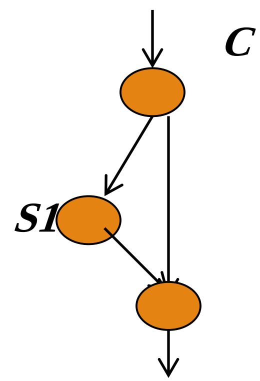


If C Then S1 else S2;



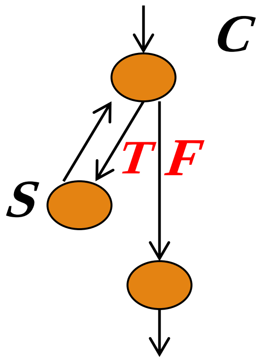
Case C of

L1: S1;
L2: S2;
…
Ln: Sn;
end;

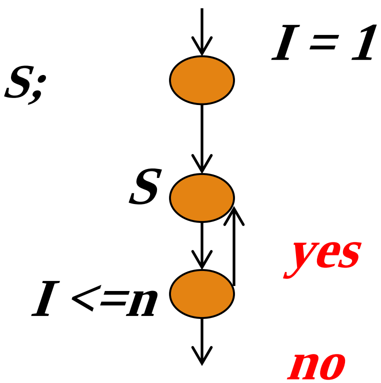


If C Then S1;

Control Flow Testing

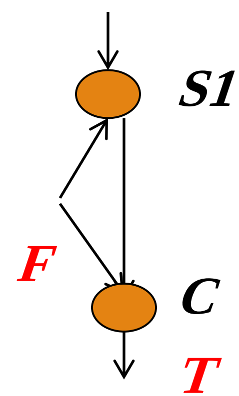


While C do S;



For loop:

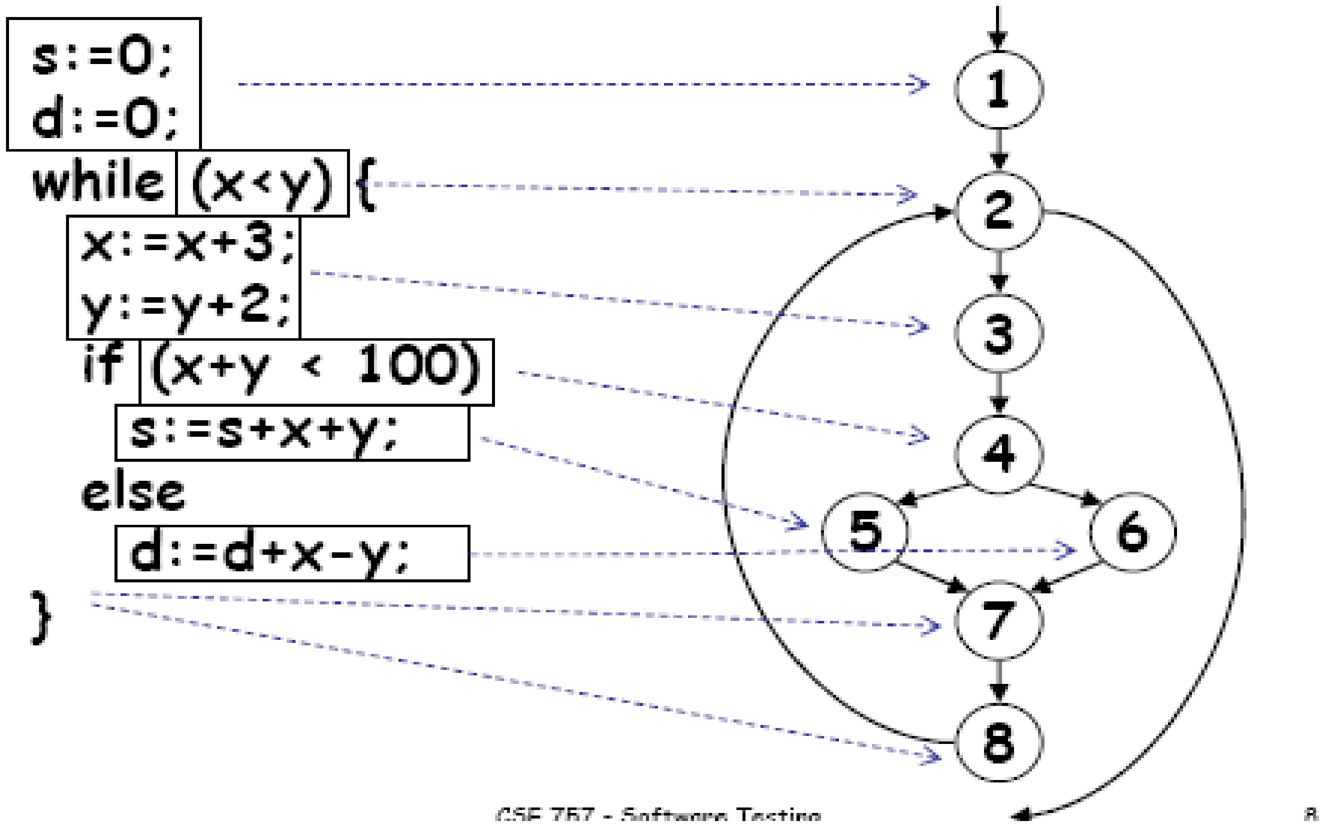
for I = 1 to n do S;



Do loop:

do S1 until C;

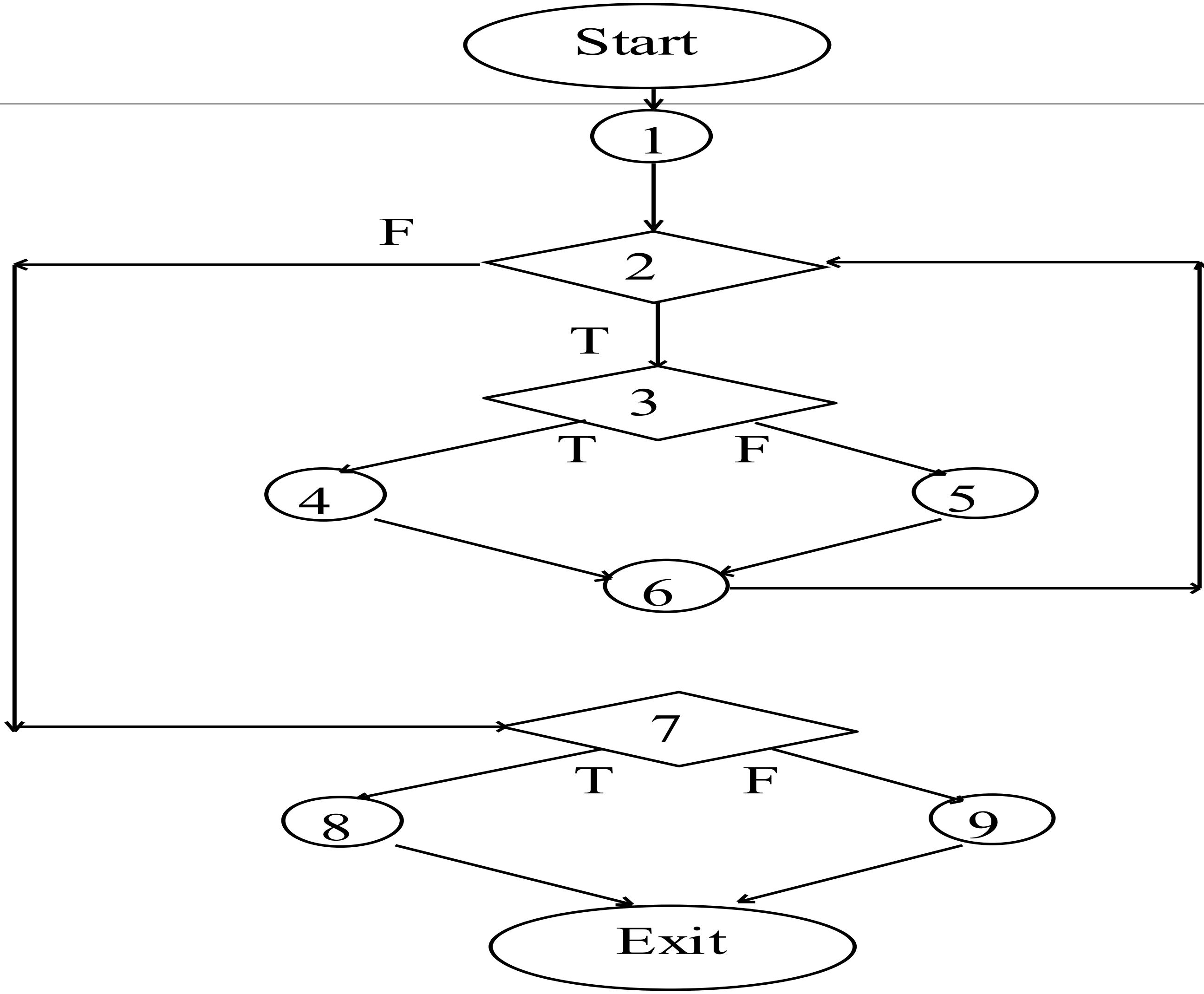
Example of a Control Flow Graph



White-box Testing: Determining the Basic Blocks

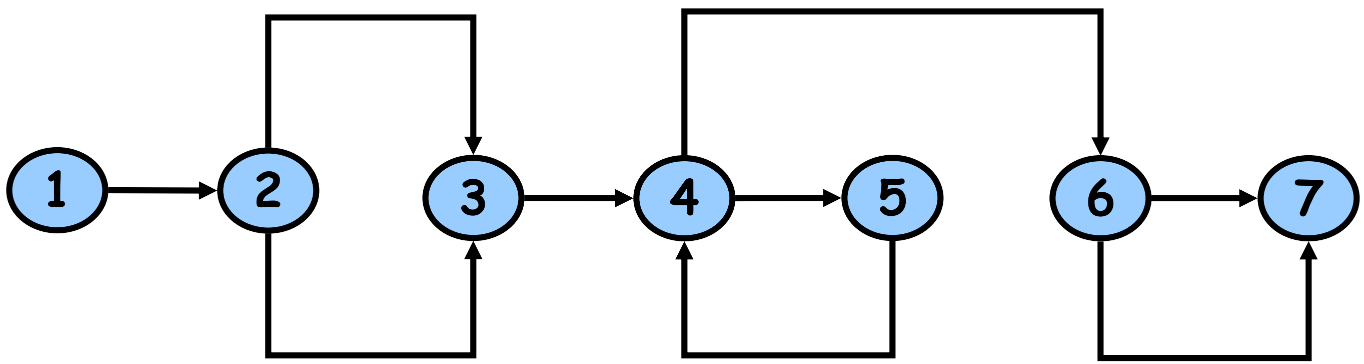
```
FindMean (FILE ScoreFile)
{  float SumOfScores = 0.0;
   int NumberOfScores = 0;
   float Mean=0.0; float Score;                          1
   Read(ScoreFile, Score);
2  while (! EOF(ScoreFile) {
3     if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;           4
            NumberOfScores++;
5           }
      Read(ScoreFile, Score);                            6
   }
   /* Compute the mean and print the result */
7  if (NumberOfScores > 0) {
         Mean = SumOfScores / NumberOfScores;            8
         printf(" The mean score is %f\n", Mean);
   } else
         printf ("No scores found in file\n");           9
}
```

Constructing the Logic Flow Diagram



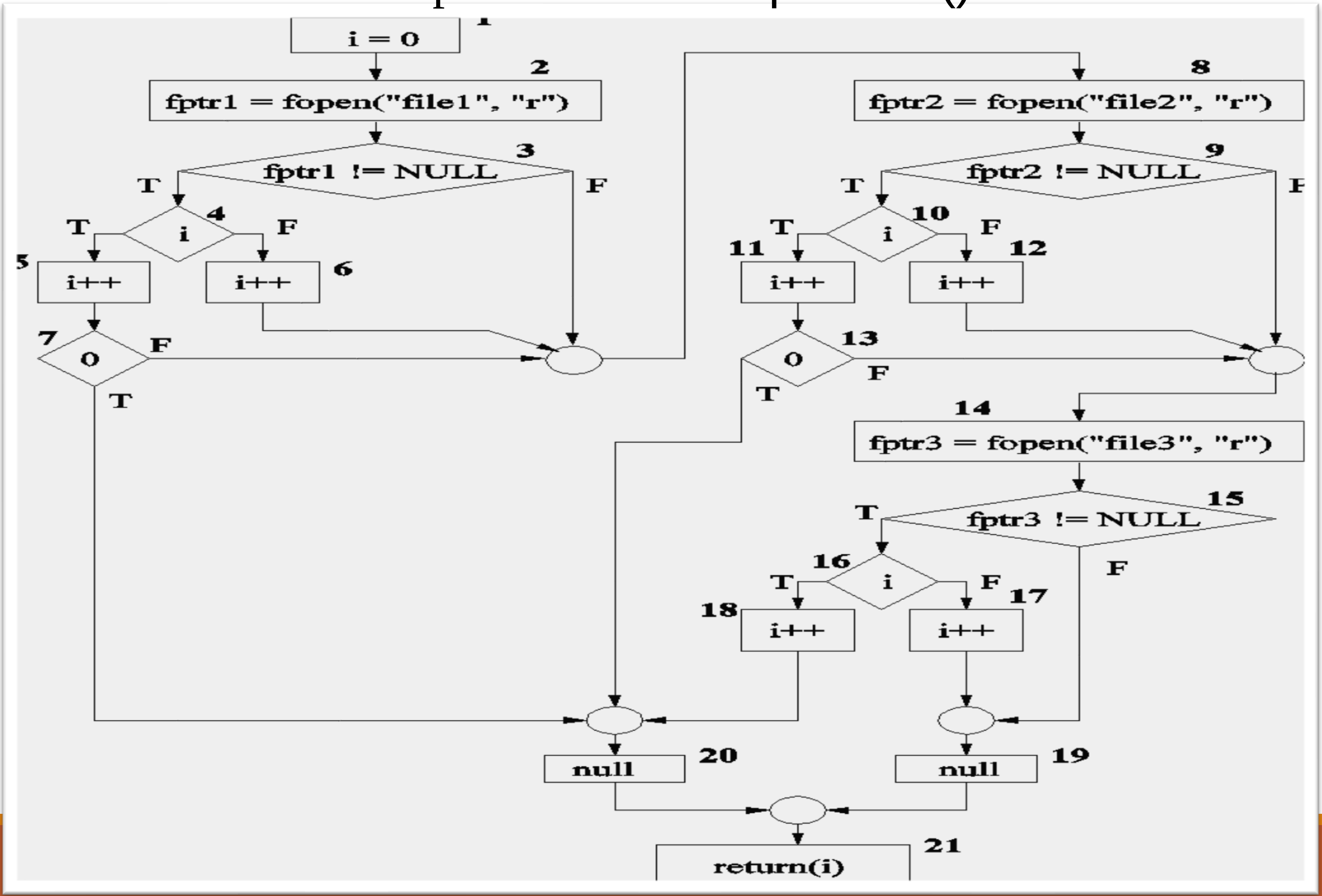
Exponentiation Algorithm

```
1   scanf("%d %d",&x, &y);
2   if (y < 0)
       pow = -y;
     else
       pow = y;
3   z = 1.0;
4   while (pow != 0) {
        z = z * x;
        pow = pow - 1;
5         }
6   if (y < 0)
        z = 1.0 / z;
7   printf ("%f",z);
```

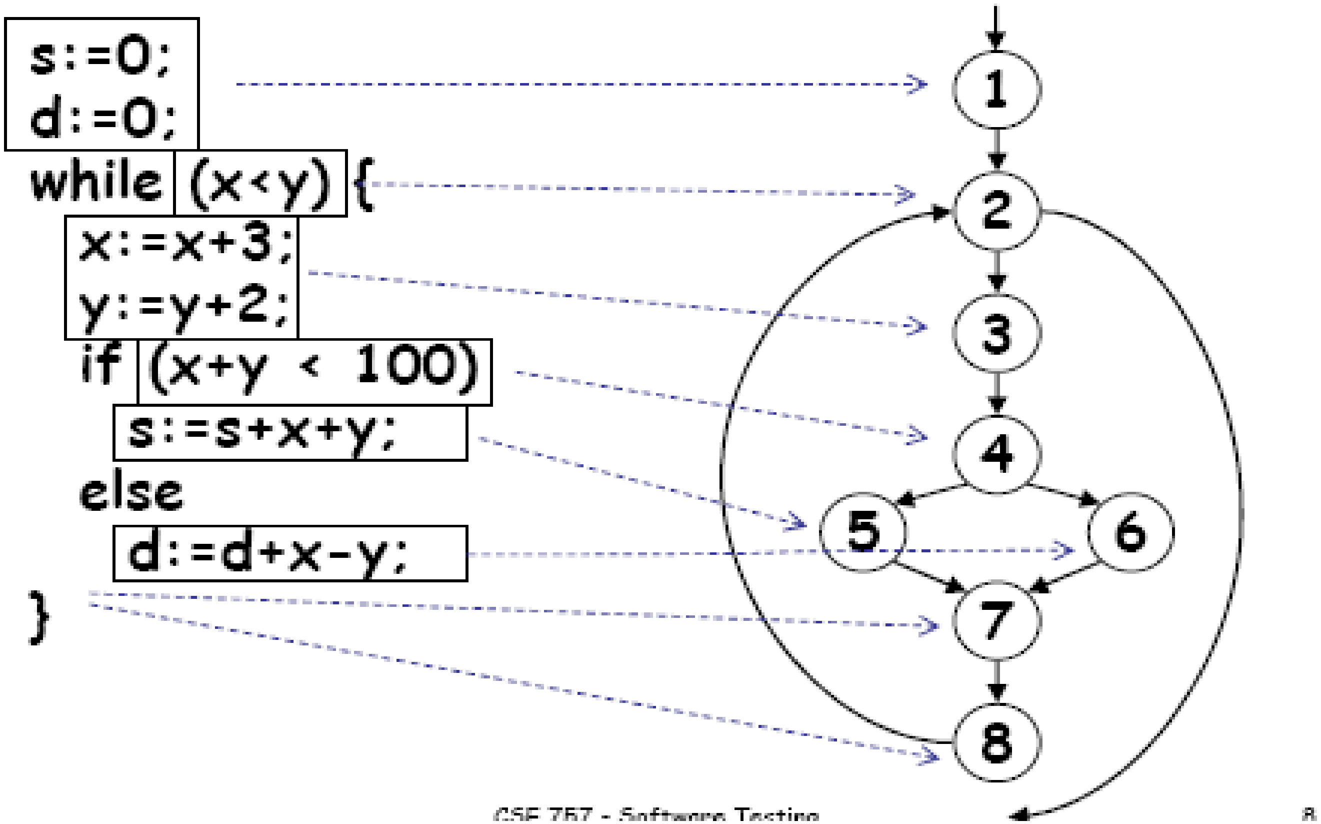


Control Flow Graph

A detailed CFG representation of openfiles().



Example of a Control Flow Graph



Example

Suppose that we write and execute two test cases

Test case #1: (x=3 and y=2) follows path 1-2-exit

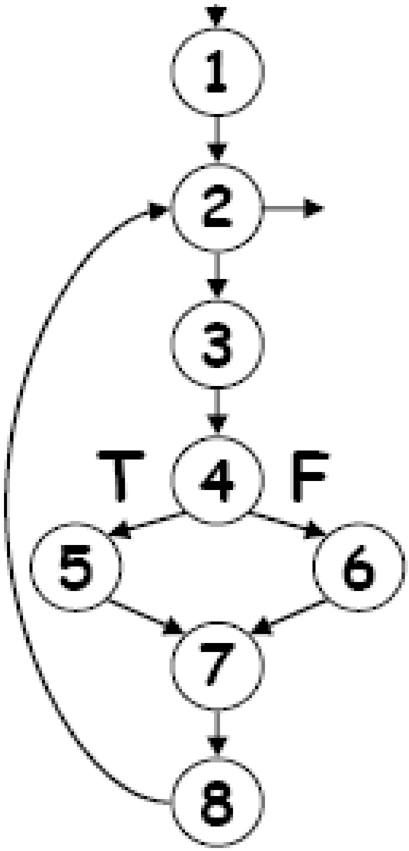
Test case #2: (x=2 and y=4) 1-2-3-4-5-

7-8-2-3-4-5-7-8-2-exit

(loop twice, and both times

take the true branch)

Problem: node 6 or branch f of 4 is never executed, so we don't have 100% branch coverage or statement coverage



Branch Coverage

Target: write test cases that cover all branches of predicate nodes

- True and false branches of each IF
- The two branches corresponding to the condition of a loop
- All alternatives in a SWITCH

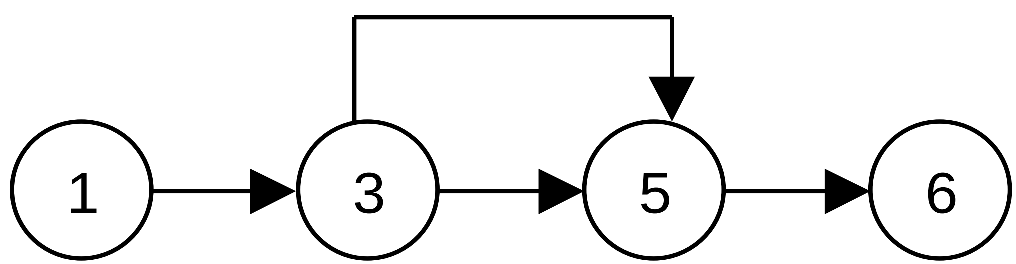
Using Control-flow Testing to Test Function ABS

- Consider the following function:

```
/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
int ABS(int x)
{
if (x < 0)
    x = -x;
return x;
}
```

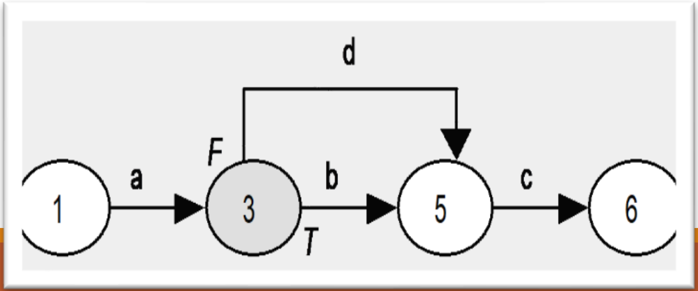
The Flowgraph for ABS

```
/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
            int ABS(int x)
            {
            if (x < 0)
                x = -x;
            return x;
            }
```



Test Cases to Satisfy **Branch Testing** Coverage for ABS

PATHS	DECISIONS Node 3	TEST CASES	
		INPUT	**OUTPUT**
abc	T	A Negative Integer, -2	2
adc	F	A Positive Integer, 2	2



Branch Coverage

Question: Does every decision have a T (true) and a F (false) in its column?

Answer: Yes implies branch coverage.

Which Paths?

You must pick enough paths to achieve statement and branch coverage.

Question: What is the fewest number of paths to achieve statement and branch coverage?

Answer:

- It is better to take many simple paths than a few complicated ones.
- There is no harm in taking paths that will exercise the same code more than once.

Statement Coverage

Given the control flow graph, we can define a “coverage target” and write test cases to achieve it

Traditional target: statement coverage

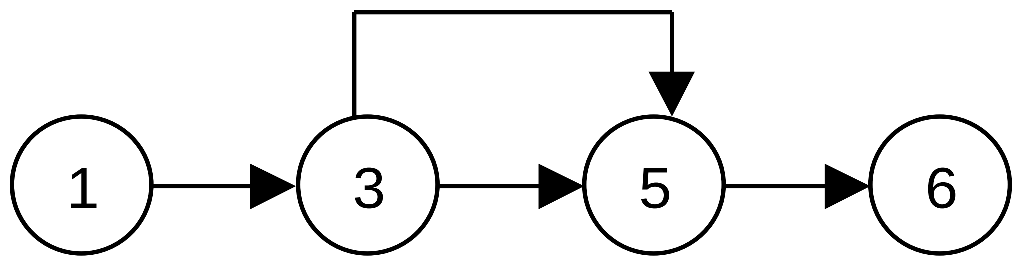
- Test cases that cover all nodes

Code that has not been executed during testing is more likely to contain errors

- Often this is the “low-probability” code

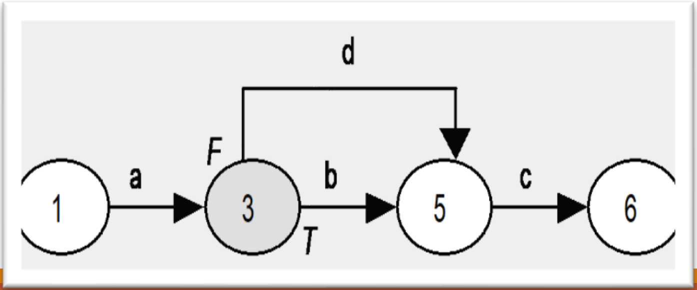
The Flowgraph for ABS

```
/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
1           int ABS(int x)
2           {
3           if (x < 0)
4               x = -x;
5           return x;
6           }
```



Test Cases to Satisfy **Statement Testing** Coverage for ABS

PATHS	PROCESS LINKS				TEST CASES	
	a	**b**	**c**	**d**	**INPUT**	**OUTPUT**
abc	√	√	√		A Negative Integer, -2	-2
adc	√		√	√	A Positive Integer, 2	2

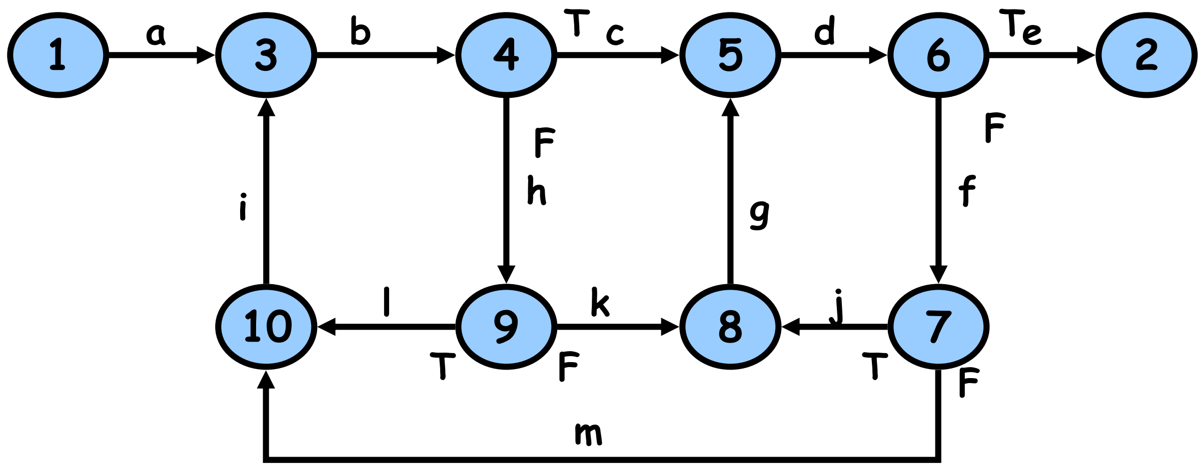


Statement Coverage

Question: Is every link covered at least once?

Answer: Yes implies statement coverage.

Example of Branch and Statement Coverage



PATHS	DECISIONS 4	6	7	9	PROCESS LINKS a	b	c	d	e	f	g	h	i	j	k	l	n
abcde	T	T			*	*	*	*	*								
abhkgde	F	T		F	*	*		*	*		*	*			*		
abhlibcde	TF	T		T	*	*	*	*	*			*	*			*	
abcdfjgde	T	TF	T		*	*	*	*	*	*	*			*			
abcdfmibcde	T	TF	F		*	*	*	*	*	*			*				*

Example: Using Control-flow Testing to Test Program COUNT

- Consider the following program:

```
/* COUNT
   This program counts the number of characters and lines in a text file
    INPUT: Text File
    OUTPUT: Number of characters and number of lines.
*/
        main(int argc, char *argv[])
        {
        int numChars = 0;
        int numLines = 0;
        char chr;
        FILE *fp = NULL;
```

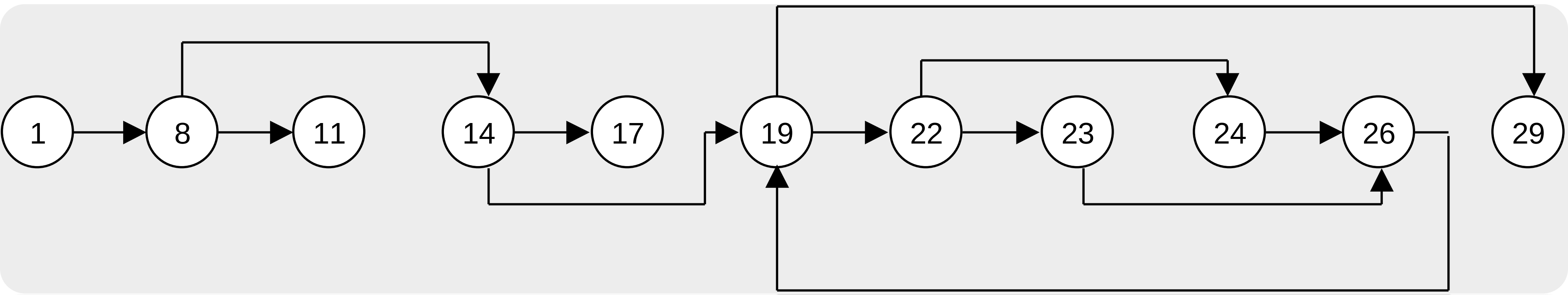
Program COUNT (Cont'd)

```
 if (argc < 2)
    {
    printf("\nUsage: %s <filename>", argv[0]);
    return (-1);
    }
fp = fopen(argv[1], "r");
if (fp == NULL)
   {
    perror(argv[1]);      /* display error message */
    return (-2);
    }
```

Program COUNT (Cont’d)

```
    while (!feof(fp))
      {
      chr = getc(fp);         /* read character */
      if (chr == ‘\n’)       /* if carriage return */
           ++numLines;
      else
           ++numChars;
      }
  printf(“\nNumber of characters = %d”, numChars);
  printf(“\nNumber of lines = %d”, numLines);
  }
```

The Flowgraph for COUNT

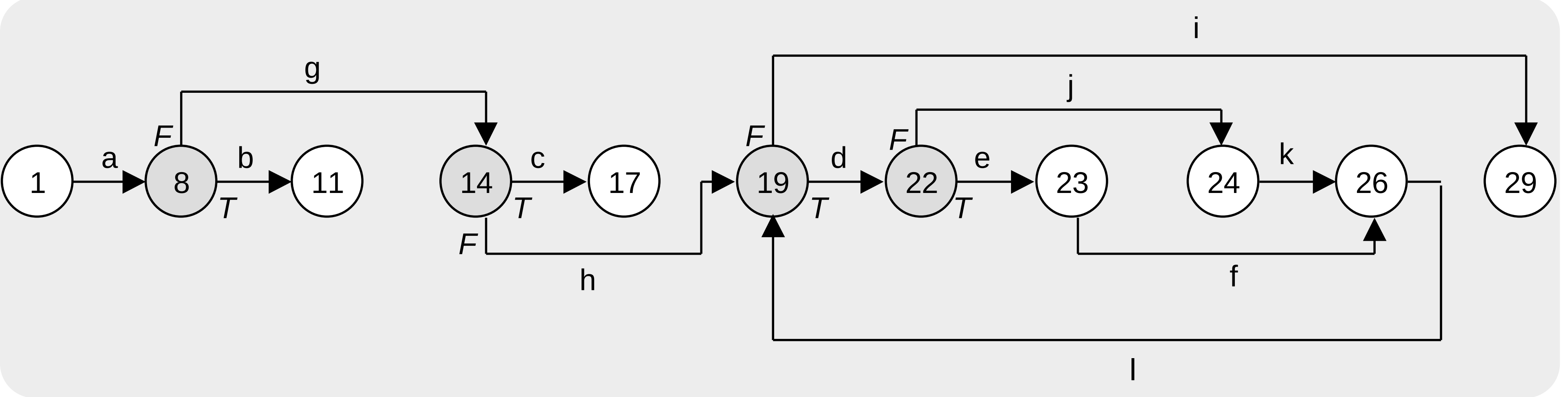


- The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.

Test Cases to Satisfy **Path** Coverage for COUNT

Complete path testing of *COUNT* is impossible because there are an infinite number of distinct text files that may be used as inputs to *COUNT*.

Test Cases to Satisfy **Statement Testing** Coverage for COUNT



Test Cases to Satisfy **Statement Testing** Coverage for COUNT

PATHS	PROCESS LINKS												TEST CASES	
	a	b	c	d	e	f	g	h	i	j	k	l	INPUT	OUTPUT
ab	√	√											None	“Usage: COUNT <filename>”
agc	√		√				√						Invalid Input Filename	Error Message
aghdjkli	√			√			√	√	√	√	√	√	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghdefli	√			√	√	√	√	√	√			√	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Test Cases to Satisfy **Branch Testing** Coverage for COUNT

PATHS	DECISIONS				TEST CASES	
	8	14	19	22	INPUT	OUTPUT
ab	T				None	“Usage: COUNT <filename>”
agc	F	T			Invalid Input Filename	Error Message
aghdjkli	F	F	T,F	F	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghdefli	F	F	T,F	T	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Limitations of Control-flow Testing

Control-flow testing as a sole testing technique is limited:

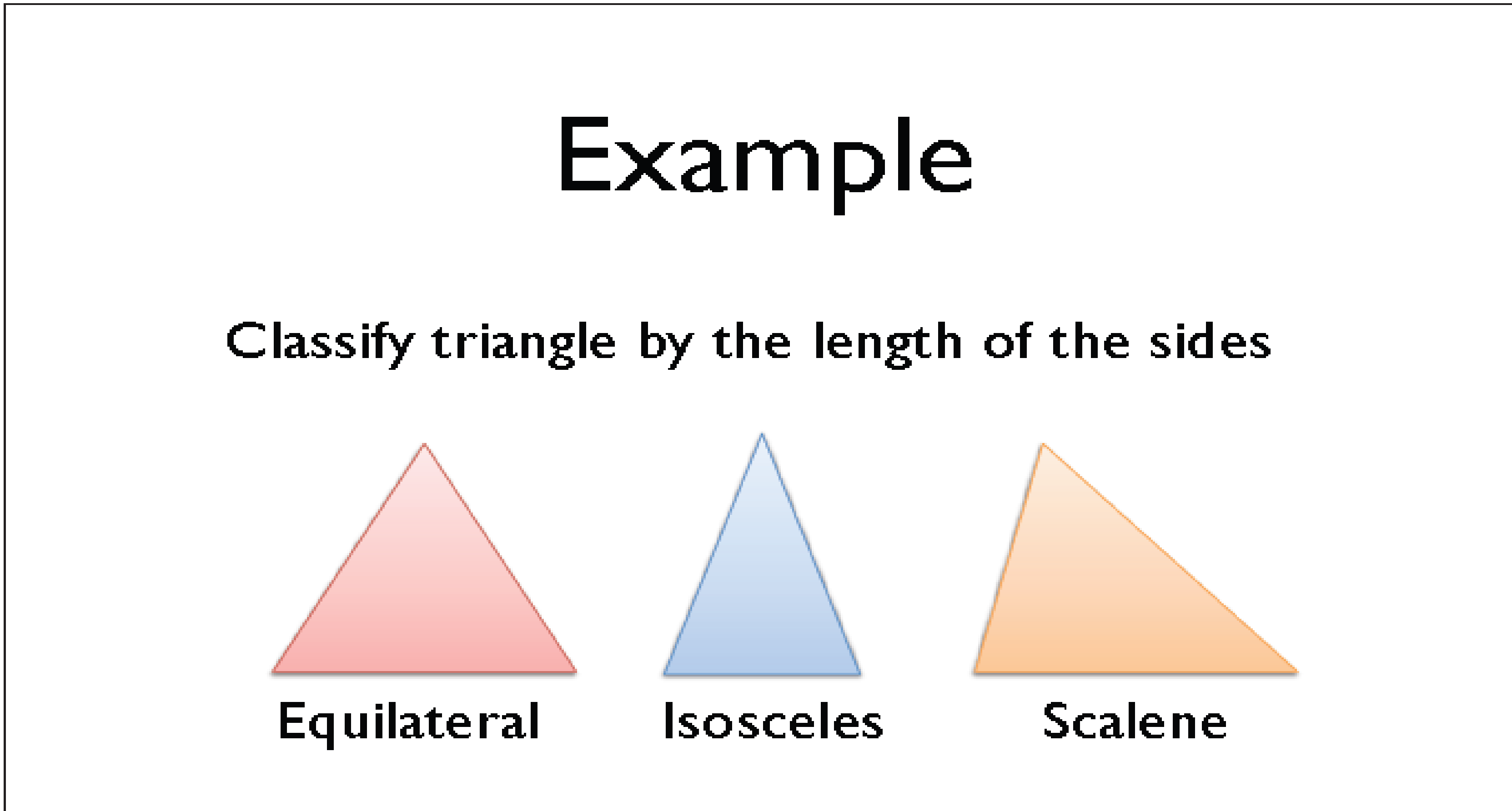
- Interface mismatches and mistakes are not caught.
- Not all initialization mistakes are caught by control-flow testing.
- Specification mistakes are not caught.

White Box testing

Is statement and path coverage is enough to select test cases?

As an example , see the following example which is classify at the type of triangle given its size of the sides.

White Box Testing



Original Program

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

White Box Testing

In the example, one test case is selected for each branch, and the all the tests are passed. (100% coverage).

However; if we have the following program which change some code.

Example with mistake

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

White Box Testing

From the above example, we can see that our test cases does not caught the mistake, since still 100% coverage.

The mistake can be caught if we have test case with sides (1,1,1). The results should be equilateral , but the program give me invalid.

These types of error can be caught using mutation test.

Outline

Definition

Control Flow Testing

Test Case

Testing with Junit

Mutation Test

Mutation Testing

A mutation is a small change in a program

Mutation Testing is a method of inserting faults into programs to test whether the tests pick them up, thereby validating or invalidating the tests.

Mutation Testing

Mutation testing: randomly change code, run against test suite

For example, change

```
int min(int a, int b)
{ int result = a;
  if ( b < a ) result = b;
  return result;
}
```

into

```
int min(int a, int b)
{ int result = b;
  if ( b < a ) result = b;
  return result;
}
```

Now: `min(6, 1)` **gives 1, but** `min(1, 6)` **gives 6**

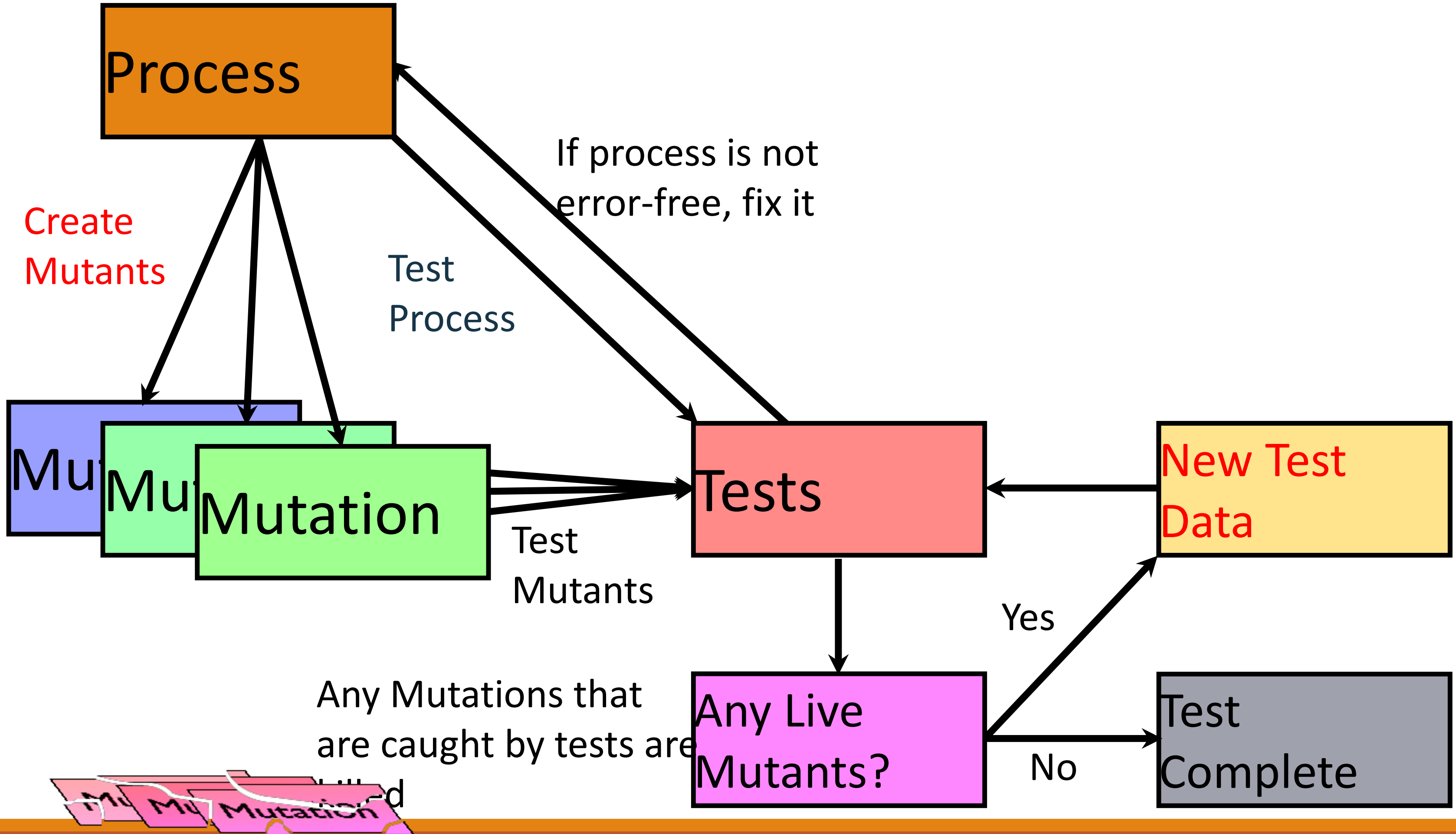
Mutation Testing

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.

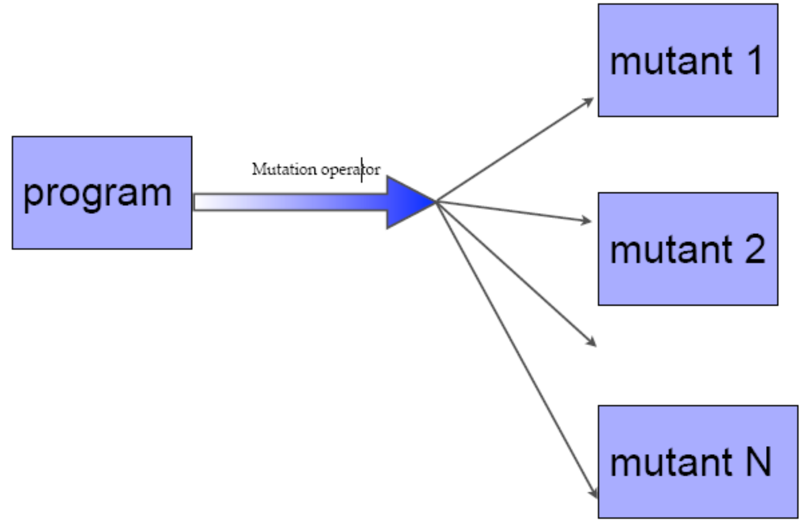
Mutation Testing should be used in conjunction with traditional testing techniques, not instead of them.

The Mutation Process



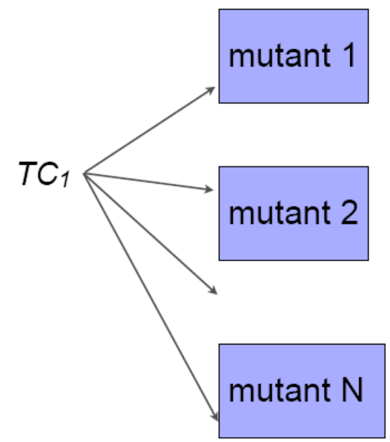
Mutation Testing

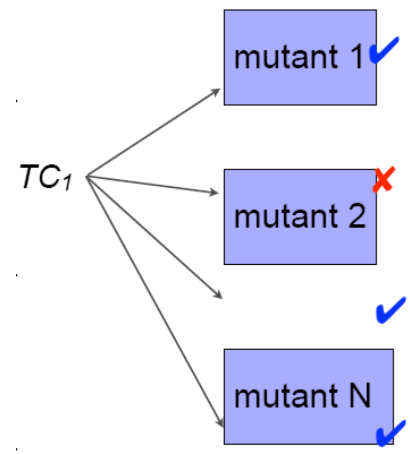
Mutant: a copy of the original program with a small change (seeded fault)



Mutant killed: Any Mutations that are caught by tests are killed

Mutant score: number of killed mutants





Mutation Testing

Execute each test case against each alive mutant.

– If the output of the mutant differs from the output of the original program, the mutant is

considered incorrect and is killed.

•

Mutation Testing

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***.

Mutation Testing

Example of a Program Mutation

- Mutants

```
int getMax(int x, int y) {
    int max;

    if (x > y)
        max = x;
    else
        max = y;
    return max;
}
```

```
int getMax(int x, int y) {
    int max;

    if (x >= y)
        max = x;
    else
        max = y;
    return max;
}
```

```
int getMax(int x, int y) {
    int max;

    if (x > y)
        max = x;
    else
        max = x;
    return max;
}
```

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.

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

Statement Modification Operators:

- *E.g.,*
  - Delete the `else` part of the `if-else` statement.
  - Delete the entire `if-else` statement.
  - Replace line 3 by a `return` statement.

Mutation Testing

Two kinds of mutants survive:

– *Functionally equivalent to the original program: Cannot be killed.*

– *Killable: Test cases are insufficient to kill the* mutant. New test cases must be created.

Mutation Testing

Equivalent mutants
The original code:

```
int index=0;
while (…){
 index++;
 if (index == 10)
 break;
}
```

Replace "==" with ">=" and produce the following mutant:

```
int index=0;
while (…){
 index++;
 if (index >= 10)
 break;
}
```

However, it is not possible to find a test case which could kill this mutant.

Mutation Score

The *mutation score* for a set of test cases is the percentage of non-equivalent mutants killed by the test data.

Mutation Score = 100 * D / (N - E)

- ***D*** = Dead mutants
- ***N*** = Number of mutants
- ***E*** = Number of equivalent mutants

A set of test cases is *mutation adequate* if its mutation score is 100%.

Mutation Testing

Lets take a program the detect the rank corresponding to the first time where the max appears in the array.

```
r:= 1;
For i:= 2 to 3 do
 if a[i] > a[r] then r:=i;
```

We call this program P.

Mutation Testing

M1)
```
r:= 1;
For i:= 1 to 3 do
    if a[i] > a[r] then r:=i;
```

M2)
```
r:= 1;
For i:= 2 to 3 do
    if i > a[r] then r:=i;
```

M3)
```
r:= 1;
For i:= 2 to 3 do
    if a[i] >= a[r] then r:=i;
```

M4)
```
r:= 1;
For i:= 2 to 3 do
    if a[r] > a[r] then r:=i;
```

Mutation Testing

Lets consider the following Test Data

	a[1]	a[2]	a[3]
DT1	1	2	3
DT2	1	2	1
DT3	3	1	2

Mutation Testing

We apply these Test Data to mutants M1, M2, M3 and M4

	P	M1	M2	M3	M4	Killed Mutants
DT1	3	3	3	3	1	M4
DT2	2	2	3	2		M2
DT3	1	1		1		none

Mutation Testing

We have to look at the efficiency of the proposed test Data. We notice that M1 and M3 are not “killed” yet.

We deduce that the test is incomplete.

We have to add a new Test Data

DT4 = {2,2,1} , P will have r=1 and M3 will have r=2, then this test data permits to kill M3.

Mutation Testing

Now, we have to analyze M1 in order to derive a “killer” test.

The difference between P and M1 is the starting point. M1 starts with i=1 and P starts with i=2.

This has no impact on the result r.

We conclude that M1 is a not effective mutant

Mutation Testing

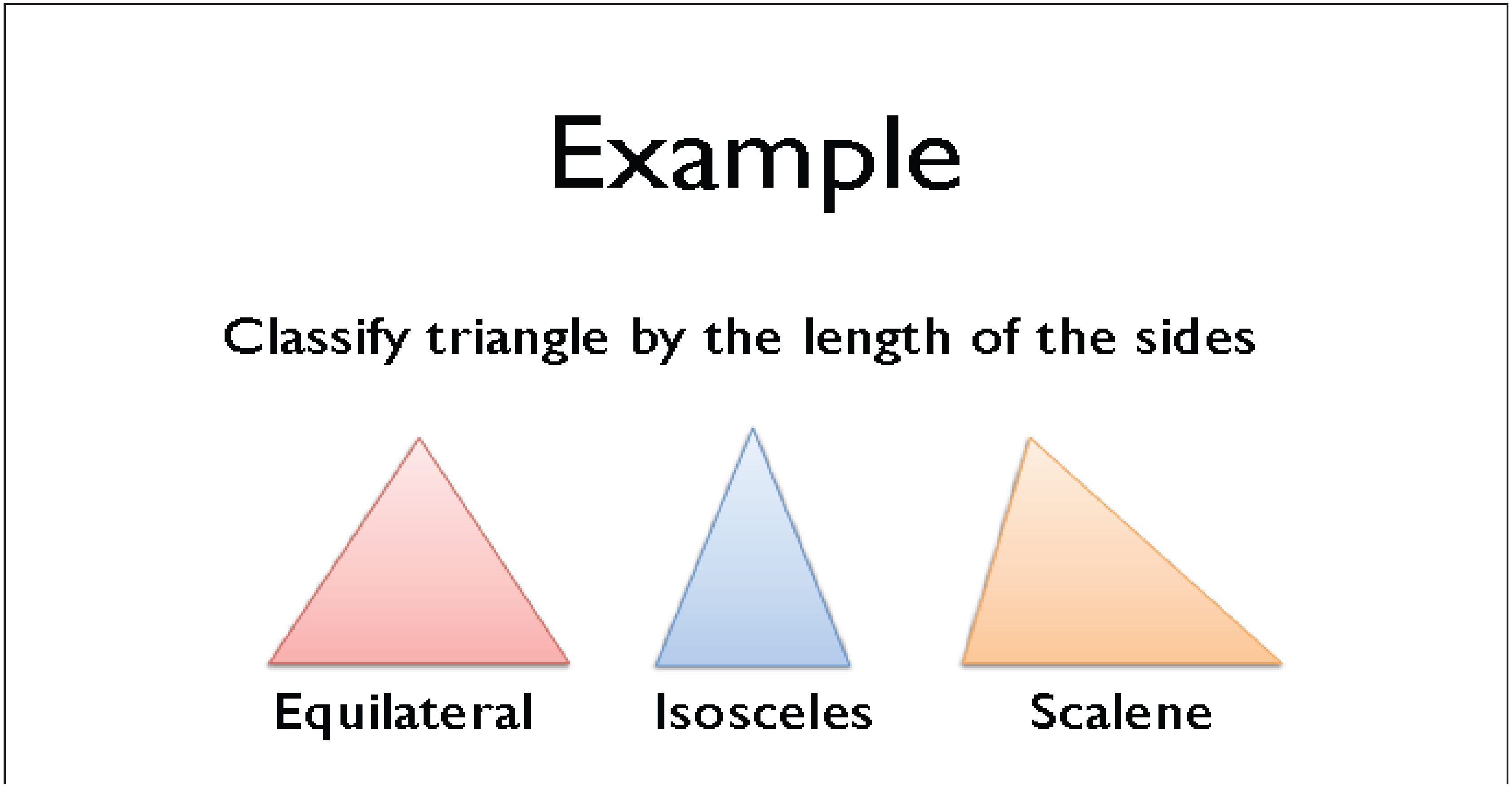
Mutation Score before adding DT4 will be

mutation Score = 100 * D / (N - E)

=100 * 2/(4-1)

=66%

Another Example



Original Program

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

Mutant 1

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a - b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

Mutant 2

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a * b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

Mutant 3

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b || b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

- (0, 0, 0) ✓
- (1, 1, 3) ✓
- (2, 2, 2) ✓
- (2, 2, 3) ✗
- (2, 3, 4) ✓

Mutant 4

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || !(a == c)) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✗

Mutant 5

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a >= c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

Mutant 6

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

Mutant 7

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b++ == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓
(1, 1, 3) ✓
(2, 2, 2) ✓
(2, 2, 3) ✓
(2, 3, 4) ✗

Example

Mutant 1,3,4,7 killed

Mutant 2 add new case (1,1,1)

Mutant 5 add new case (4,3,2)

Mutant 6 is equivalent

mutation Score = 100 * D / (N - E)

=100 * 4/(7-1)

=66%

Outline

Definition

Test Case

Control Flow Testing

Mutation Test

Testing with Junit

Testing with JUnit

Junit is a **<u>unit</u> test environment** for Java programs developed by *Erich Gamma* and *Kent Beck*.

- Writing test cases
- Executing test cases
- Pass/fail? (expected result = obtained result?)

Testing with JUnit

Consists in a **framework** providing all the tools for testing.

- framework: set of classes and conventions to use them.

It is **integrated into eclipse and netBeans** through a graphical plug-in.

How to write JUnit-based testing code (Minimum)

Create a Java class

```java
package Test;

public class MyClass
 {
 public int multiply(int x, int y) {
  return x / y;
  }
 }
```

How to write JUnit-based testing code (Minimum)

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyClassTest
{
 @Test
 public void testMultiply()
   {
 MyClass tester = new MyClass();
 assertEquals("Result", 50, tester.multiply(10, 5)); }
}
```

SimpleMath.java

```
 */
 * @author Kiki
 */
public class SimpleMath {


        public int doAddition(int a, int b ){

            return a + b ;
        }
        public int doSubtraction(int a, int b){

            return a / b;

        }
        public void printAddition(int a, int b){

                System.out.println("var1 = "+a+" , var2 = "+b+" " +
                        "hasilnya adalah = "+doAddition(a, b));

        }
    }
```

Create Unit Test

Choose this menu in netbeans

- Tools > Create Junit Test

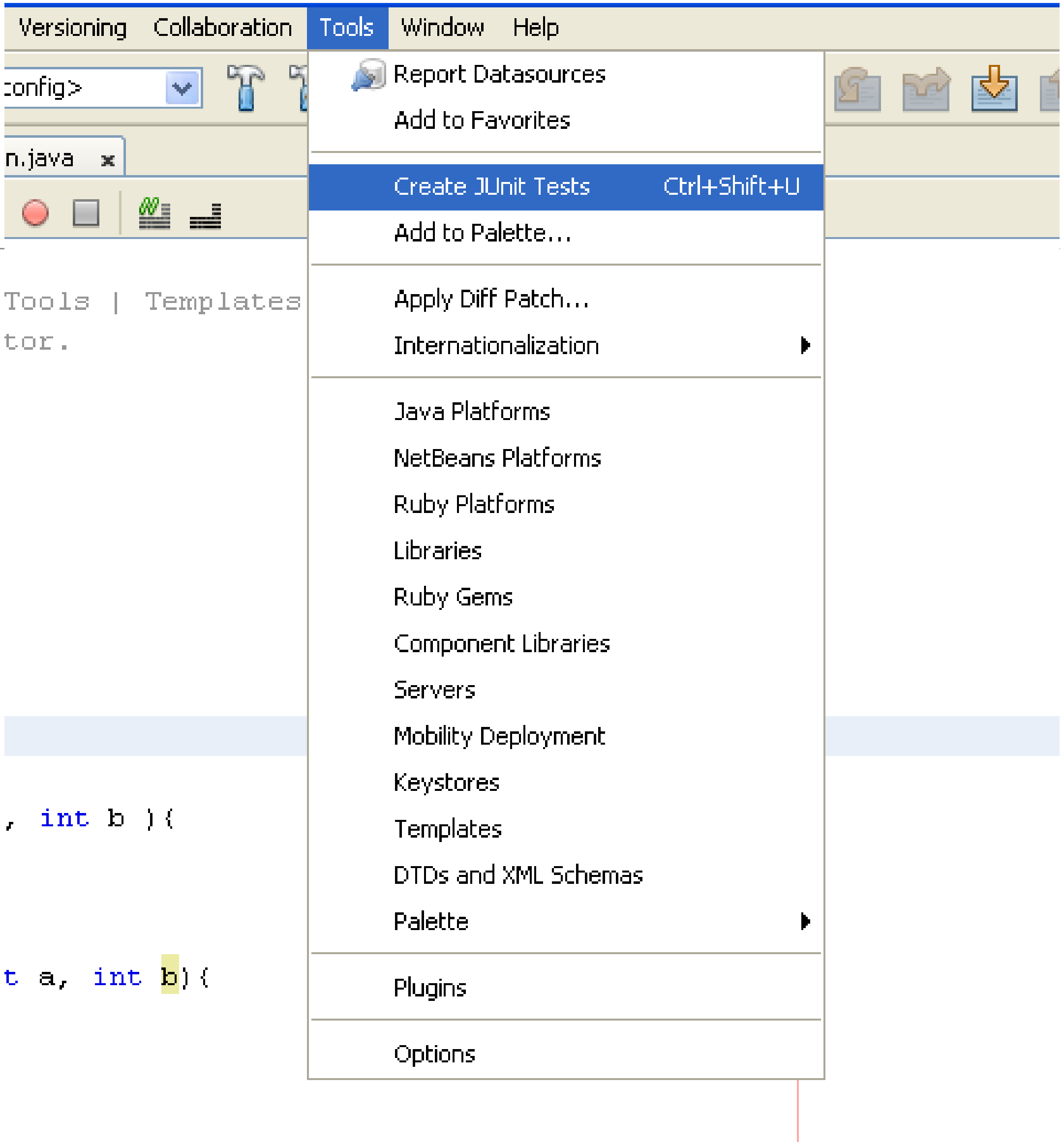
Or just simply press Ctrl + Shift + U.

A window dialogue will appear, choose suitable options.

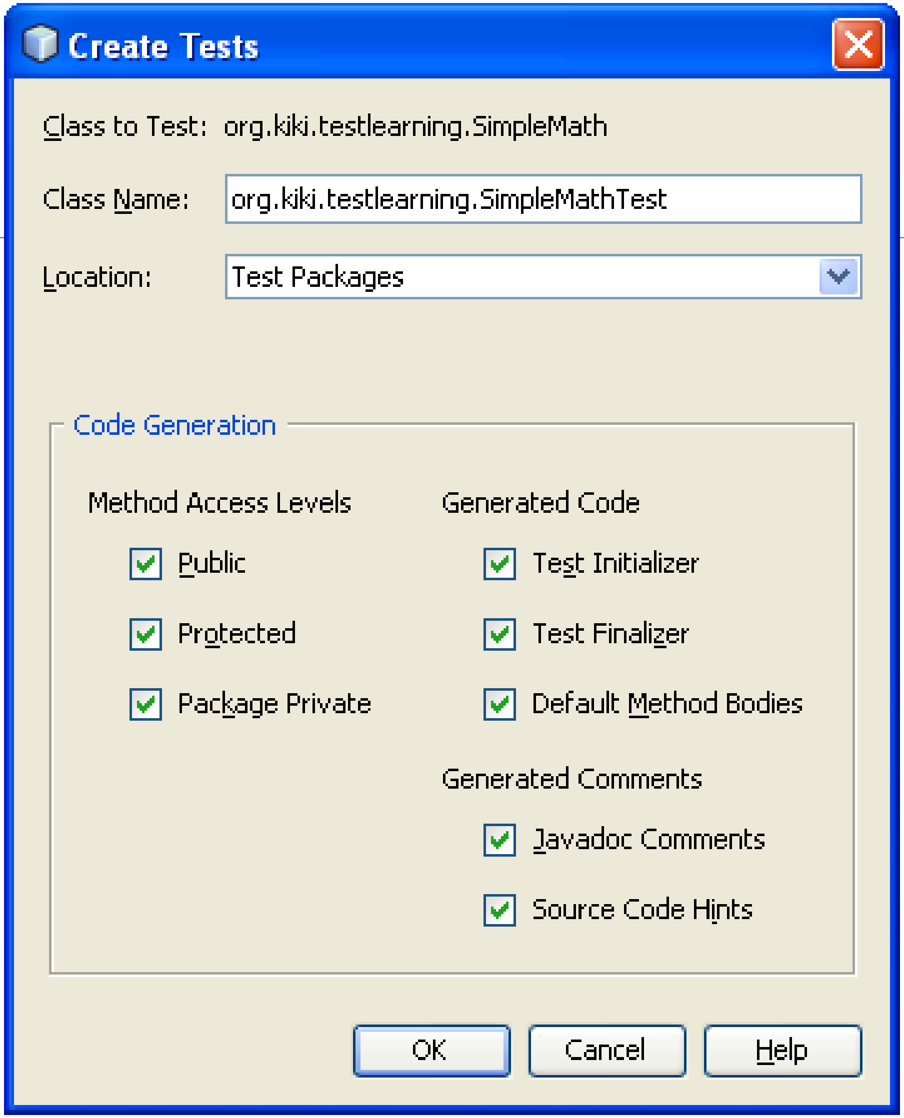
Or you can leave it as is.

Test case will automatically build inside the test package folder.

Unit Test Menu



Unit Test Window



SimpleMathTest.java

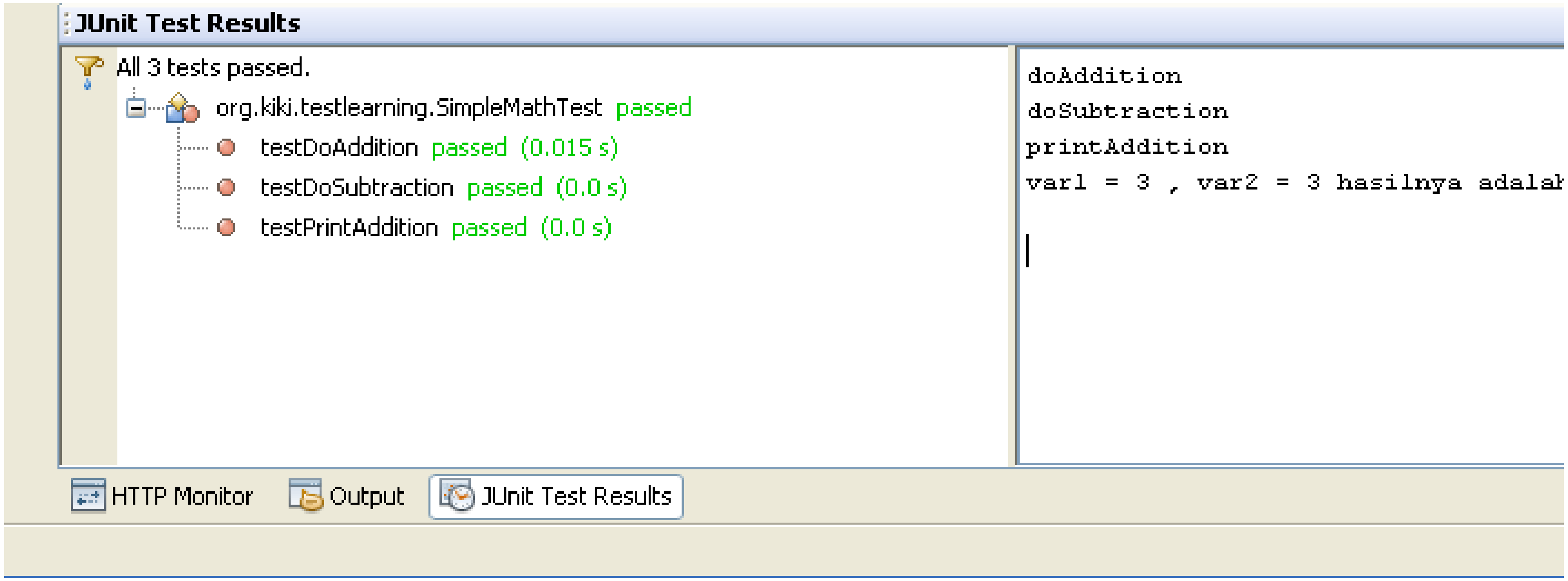
```java
     * Test of doAddition method, of class SimpleMath.
     */
    @Test
    public void testDoAddition() {
        System.out.println("doAddition");
        int a = 2;
        int b = 2;
        SimpleMath instance = new SimpleMath();
        int expResult = 4;
        int result = instance.doAddition(a, b);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.

    }

    /**
     * Test of doSubtraction method, of class SimpleMath.
     */
    @Test
    public void testDoSubtraction() {
        System.out.println("doSubtraction");
        int a = 3;
        int b = 1;
        SimpleMath instance = new SimpleMath();
        int expResult = 3;
        int result = instance.doSubtraction(a, b);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.

    }
```

Test Result



JUnit Test Results
All 3 tests passed.
org.kiki.testlearning.SimpleMathTest passed
testDoAddition passed (0.015 s)
testDoSubtraction passed (0.0 s)
testPrintAddition passed (0.0 s)
doAddition
doSubtraction
printAddition
var1 = 3 , var2 = 3 hasilnya adalah
HTTP Monitor
Output
JUnit Test Results