fit@hcmus

Software Testing

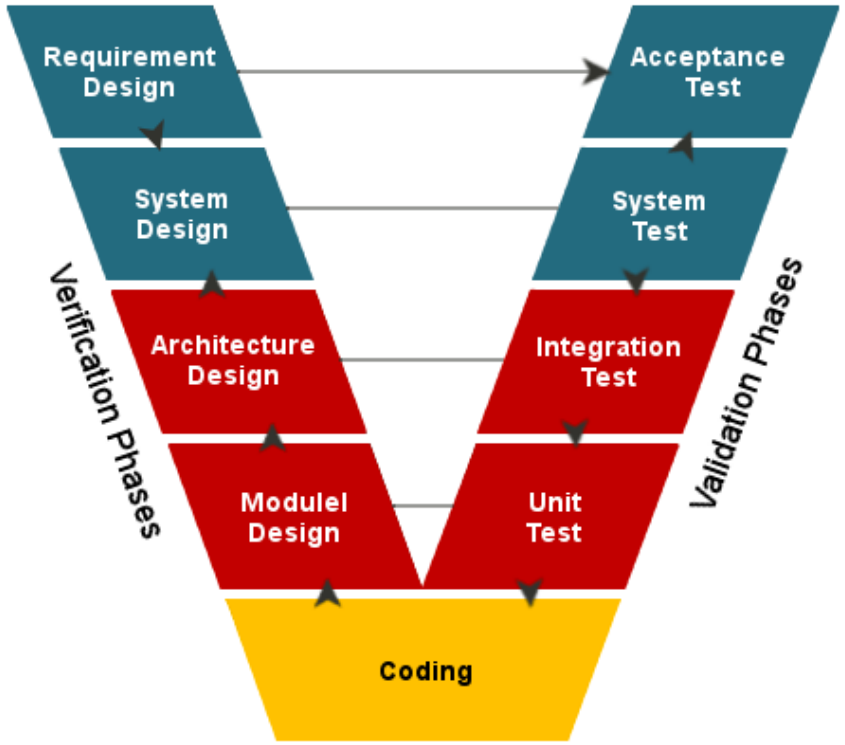
CSC13003

Unit Testing – White Box Testing

Content

- **Unit testing**
- White box testing

V model


Requirement Design
Acceptance Test
System Design
System Test
Architecture Design
Integration Test
Modulel Design
Unit Test
Coding
Verification Phases
Validation Phases

Requirement Analysis

- To gather requirements.
- Business requirement analysis is to understand an aspect from a customer's perspective by stepping into their shoes to completely analyse the functionality of an application from a user's point of view.
- An **acceptance criteria** layout is prepared to correlate the tasks done during the development process with the final outcome of the overall effort.

Acceptance criteria

- Requirement:
  - I should be able to search the name of a book along with its details with the help of a universal search option on the front page of my library management system.
- Acceptance criteria:
  - Search by the name of a book only.
  - Search by the publisher name, that shows all books belonging to the same category.
  - Restrict the search with the name of a website.
  - The user should not be able to execute the search if all the mandatory fields are not entered.
  - The user must see the sequence number of the book.

System design

- It comprises of creating a layout of the system/application design that is to be developed. System design is aimed at writing a detailed hardware and software specification.
- Architectural Design: high-level design
- Module Design: low-level design

Architectural design

- High-level design

Module design

- Low-level design

Unit testing

- To verify singles modules and remove bug, if it exists.
- A unit test is simply executing a piece of code to verify whether it delivers the desired functionality.

Integration testing

- To collaborate pieces of code together to verify that they perform as a single entity.

System testing

- System testing is performed when the complete system is ready, the application is then run on the target environment in which it must operate,

User acceptance testing

- The user acceptance test plan is prepared during the requirement analysis phase because when the software is ready to be delivered, it is tested against a set of tests that must be met in order to certify that the product has achieved the target it was intended to.

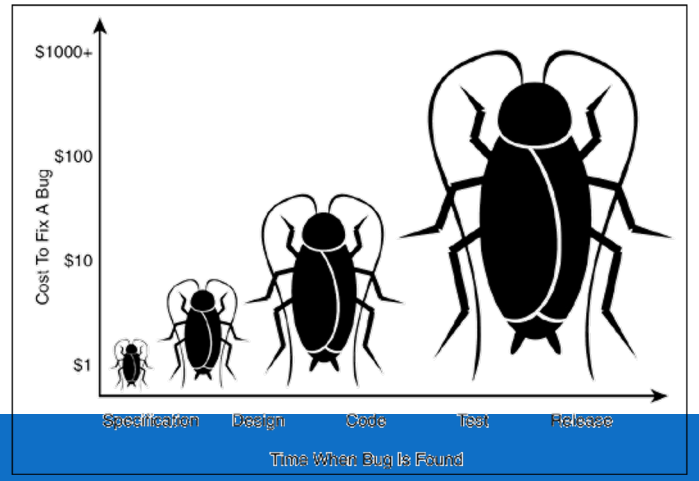
What is unit testing?

- Unit testing is used to verify a small chunk of code by creating a path, **function** or a **method**.

- Unit testing is used to identify defects early in software development cycle.
- Unit testing will compel to read our own code. i.e. a developer starts spending more time in reading than writing.
- Defects in the design of code affect the development system. A successful code breeds the confidence of developer.

Why unit testing?

- It finds bugs early in the code, which makes our code more reliable.
- Unit testing forces a developer to read code more than writing.
- You develop more readable, reliable and bug-free code which builds confidence during development.



TDD & Unit testing

- Test Driven Development
- Tests are written before the code
- Rely heavily on testing frameworks
- All classes in the applications are tested
- Quick and easy integration is made possible

Process

- Write test cases (test methods) to test a method under test.
- Use the unit testing framework to execute test methods.
- Result: Passed / Failed.
- If a test method failed,
  - the method under test has a bug,
  - developers must modify source code of the method under test to fix the detected bug.

Ex: Apply unit testing to the method Triangle::getPerimeter()

- Test method 1: test the method getPerimeter() with a right triangle
  - Init a triangle with 3 points: A(0,0), B(0,3) và C(4,0)
  - Expected perimeter = 3 + 4 + 5 = 12
  - Actual perimeter = invoke the method Triangle::getPerimeter()
  - Assert to compare the value of expected and actual ones.
- Test method 2: test the method getPerimeter() with an isosceles right triangle
  - Init a triangle with 3 points: A(0,0), B(0,1) và C(1,0)
  - Expected perimeter = 1 + 1 + 1.4 = 3.4
  - Actual perimeter = invoke the method Triangle::getPerimeter()
  - Assert to compare the value of expected and actual ones.
- Test method 3: ...

Unit testing vs debugging

- Debugging: manual.
- Unit testing: automatic.

JUnit

- Kent Beck, Erich Gamma, David Saff, Kris Vasudevan
- Latest stable version: 5.4.2 (2019-04-07)

JUnit installation

- Maven repository
- “junit”, “hamcrest-core”
- pom / jar

Hello World JUnit testing

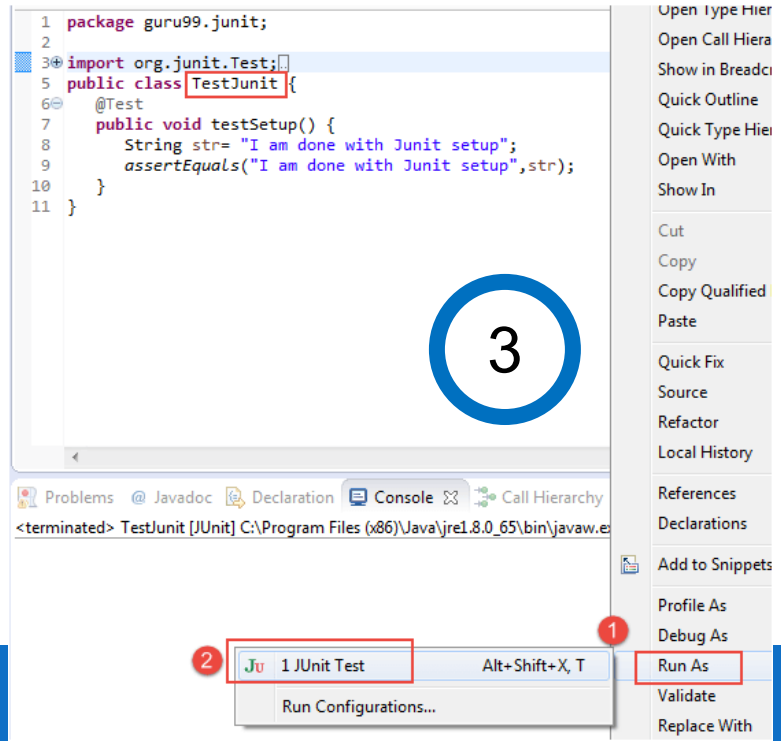
1

```
1  package guru99.junit;
2
3  import org.junit.Test;
5  public class TestJunit {
6     @Test
7     public void testSetup() {
8        String str= "I am done with Junit setup";
9        assertEquals("I am done with Junit setup",str);
10    }
11 }
```

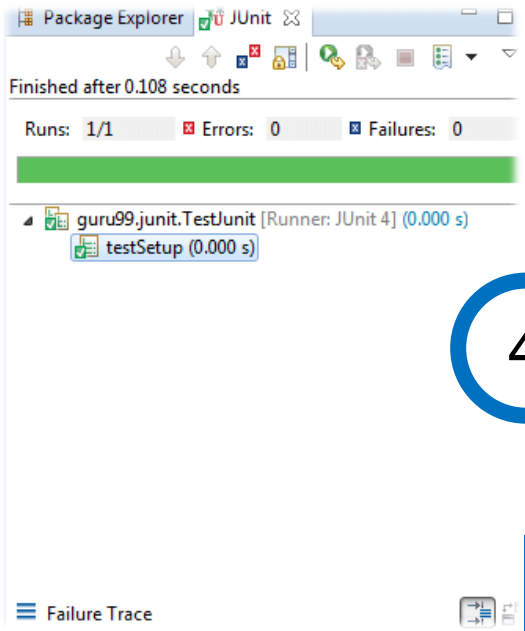
2

```
1  package guru99.junit;
2
3  import org.junit.runner.JUnitCore;
6
7  public class TestRunner {
8     public static void main(String[] args) {
9        Result result = JUnitCore.runClasses(TestJunit.class);
10       for (Failure failure : result.getFailures()) {
11          System.out.println(failure.toString());
12       }
13       System.out.println("Result=="+result.wasSuccessful());
14    }
15 }
```

3



4



Method under test

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum -= Integer.valueOf(summand);
    return sum;
  }
}
```

A test method

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

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

assertXXX()

```java
@Test
public void testAssertEquals() {
  assertEquals("failure - strings are not equal", "text", "text");
}
```

```java
@Test
public void testAssertFalse() {
  assertFalse("failure - should be false", false);
}
```

```java
@Test
public void testAssertArrayEquals() {
  byte[] expected = "trial".getBytes();
  byte[] actual = "trial".getBytes();
  assertArrayEquals("failure - byte arrays not same", expected, actual);
}
```

assertXXX

```
@Test
public void testAssertNotSame() {
  assertNotSame("should not be same Object", new Object(), new Object());
}
```

```
@Test
public void testAssertNotNull() {
  assertNotNull("should not be null", new Object());
}
```

```
@Test
public void testAssertSame() {
  Integer aNumber = Integer.valueOf(768);
  assertSame("should be same", aNumber, aNumber);
}
```

```
@Test
public void testAssertNull() {
  assertNull("should be null", null);
}
```

assertThat(value,matcher)

```
@Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertThat(sum, is(equalTo(6));
  }
```

```
assertThat(num, is(12));
assertThat(num, is(equalTo(12));
```

```
assertThat(someString, is(equalTo("blah"));
assertThat(someString, equalTo("blah");

assertThat(someObject, is(nullValue()));
assertThat(someObject, nullValue());
```

assertThat(value,matcher)

```
assertThat(foo, is(true));
```

```
assertThat(str, containsString(substr));
```

```
@Test
public void testAssertThatHasItems() {
  assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
}
```

```
@Test
public void testAssertThatBothContainsString() {
  assertThat("albumen", both(containsString("a")).and(containsString("b")));
}
```

```
@Test
  public void testAssertThatEveryItemContainsString() {
    assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(containsString("n")));
  }
```

assertThat(value,matcher)

```
@Test
public void testAssertThatHamcrestCoreMatchers() {
  assertThat("good", allOf(equalTo("good"), startsWith("good")));
  assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
  assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
  assertThat(7, not(CombinableMatcher.<Integer> either(equalTo(3)).or(equalTo(4))));
  assertThat(new Object(), not(sameInstance(new Object())));
}
```

```
@Test
  public void testReverse() {
    assertThat("oof", is(equalTo(StringUtils.reverseString("foo"))));
    assertThat("rab", is(equalTo(StringUtils.reverseString("bar"))));
 }
```

assertThat(value,matcher)

```
@Test
public void testPalindromes() {
  String[] matches =
    { "a", "aba", "Aba", "abba", "AbBa",
      "abcdeffedcba", "abcdEffedcba" };
  String[] misMatches =
    { "ax", "axba", "Axba", "abbax", "xAbBa",
      "abcdeffedcdax", "axbcdEffedcda" };
  for(String s: matches) {
    assertThat(StringUtils.isPalindrome(s), is(true));
  }
  for(String s: misMatches) {
    assertThat(StringUtils.isPalindrome(s), is(false));
  }
}
```

`@Test` annotation

- Place the annotation at a public, void method.
- JUnit recognized it as a test method (test case).

Expect

- Expect a test case should throw an exception

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
     new ArrayList<Object>().get(0);
}
```

```
@Test
public void example1() {
    try {
        find("something");
        fail();
    } catch (NotFoundException e) {
        assertThat(e.getMessage(), containsString("could not find something"));
    }
    // ... could have more assertions here
}
```

Timeout

- Expect the test case after a period of time.

```
@Test(timeout=1000)
public void testWithTimeout() {
   ...
}
```

```
public class HasGlobalTimeout {
    public static String log;
    private final CountDownLatch latch = new CountDownLatch(1);

    @Rule
    public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds max per method tested

    @Test
    public void testSleepForTooLong() throws Exception {
        log += "ran1";
        TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds
    }

    @Test
    public void testBlockForever() throws Exception {
        log += "ran2";
        latch.await(); // will block
    }
}
```

@Before, @After

- Place the annotation at a public, void method.
- The method helps “prepare” things before executing a test method and to clean things after running a test method.
  - @Before: run before each test method.
  - @After: run after each test method

@BeforeClass, @AfterClass

- @BeforeClass is executed before running any of the test methods.
- @AfterClass is executed after running any of the test methods.

Example

```
private Collection collection;

@BeforeClass
public static void oneTimeSetUp() {
    // one-time initialization code
    System.out.println("@BeforeClass - oneTimeSetUp");
}

@AfterClass
public static void oneTimeTearDown() {
    // one-time cleanup code
    System.out.println("@AfterClass - oneTimeTearDown");
}
```

```
@Before
public void setUp() {
    collection = new ArrayList();
    System.out.println("@Before - setUp");
}

@After
public void tearDown() {
    collection.clear();
    System.out.println("@After - tearDown");
}
```

```
@Test
public void testEmptyCollection() {
    assertTrue(collection.isEmpty());
    System.out.println("@Test - testEmptyCollection");
}

@Test
public void testOneItemCollection() {
    collection.add("itemA");
    assertEquals(1, collection.size());
    System.out.println("@Test - testOneItemCollection");
}
```

```
@BeforeClass - oneTimeSetUp
@Before - setUp
@Test - testEmptyCollection
@After - tearDown
@Before - setUp
@Test - testOneItemCollection
@After - tearDown
@AfterClass - oneTimeTearDown
```

@Ignore

- Use this annotation to ignore a specific test method.

```
@Ignore("Not Ready to Run")
@Test
public void divisionWithException() {
  System.out.println("Method is not ready yet");
}
```

@Parameterized

```
public class Fibonacci {
    public static int compute(int n) {
        int result = 0;

        if (n <= 1) {
            result = n;
        } else {
            result = compute(n - 1) + compute(n - 2);
        }

        return result;
    }
}
```

@Parameterized

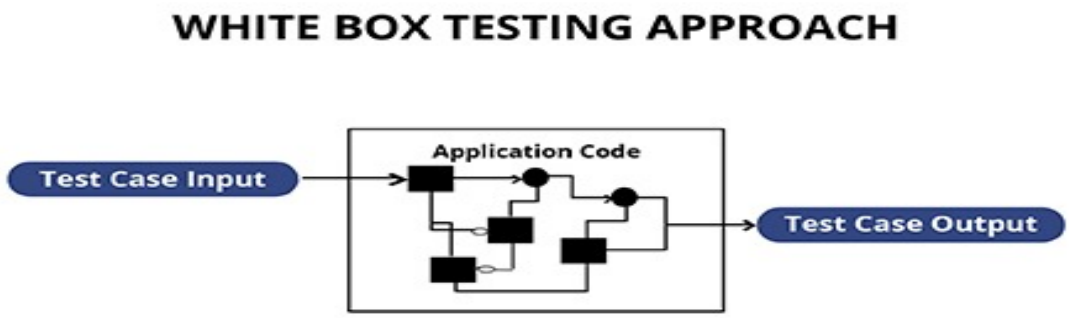
```
@RunWith(Parameterized.class)
public class FibonacciTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
           });
    }

    private int fInput;

    private int fExpected;

    public FibonacciTest(int input, int expected) {
        fInput= input;
        fExpected= expected;
    }

    @Test
    public void test() {
        assertEquals(fExpected, Fibonacci.compute(fInput));
    }
}
```

Content

- Unit testing
- **White box testing**

White Box Testing

- A strategy testing based on the internal paths, structure, and implementation of the System Under Test
- Can be applied at all levels of system development - unit, integration, and system



White Box Testing Techniques

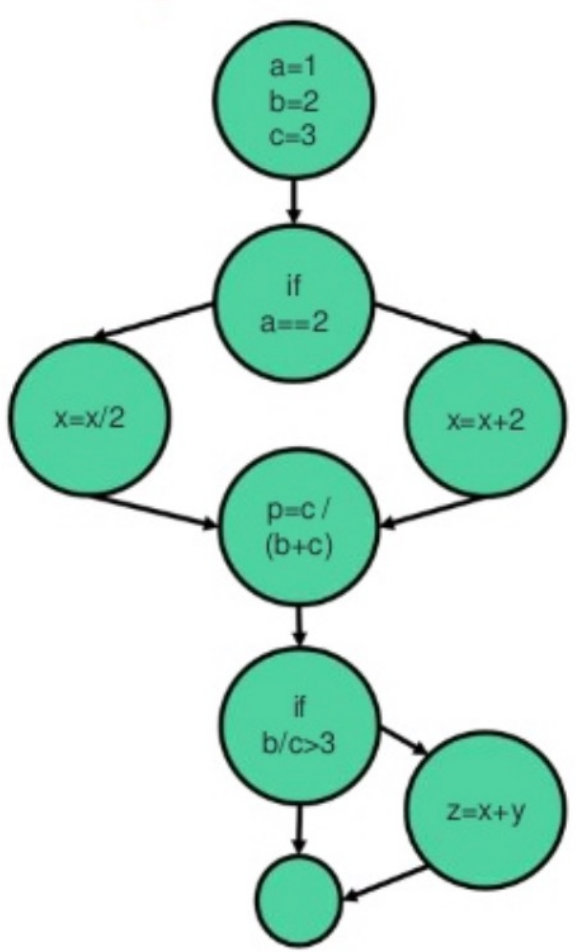
- Control Flow Testing
  - Identify the **execution path**s through a module of **program code**
- Data Flow Testing
  - Identify paths in the program that go from the **assignment** to the **use** of a **variable** in a module of program code

Control Flow Testing

- Create and executes test cases to **cover the execution paths** through a module or program code
- **Path**: *a sequence of statement execution that begins at an entry and ends at an exi*t

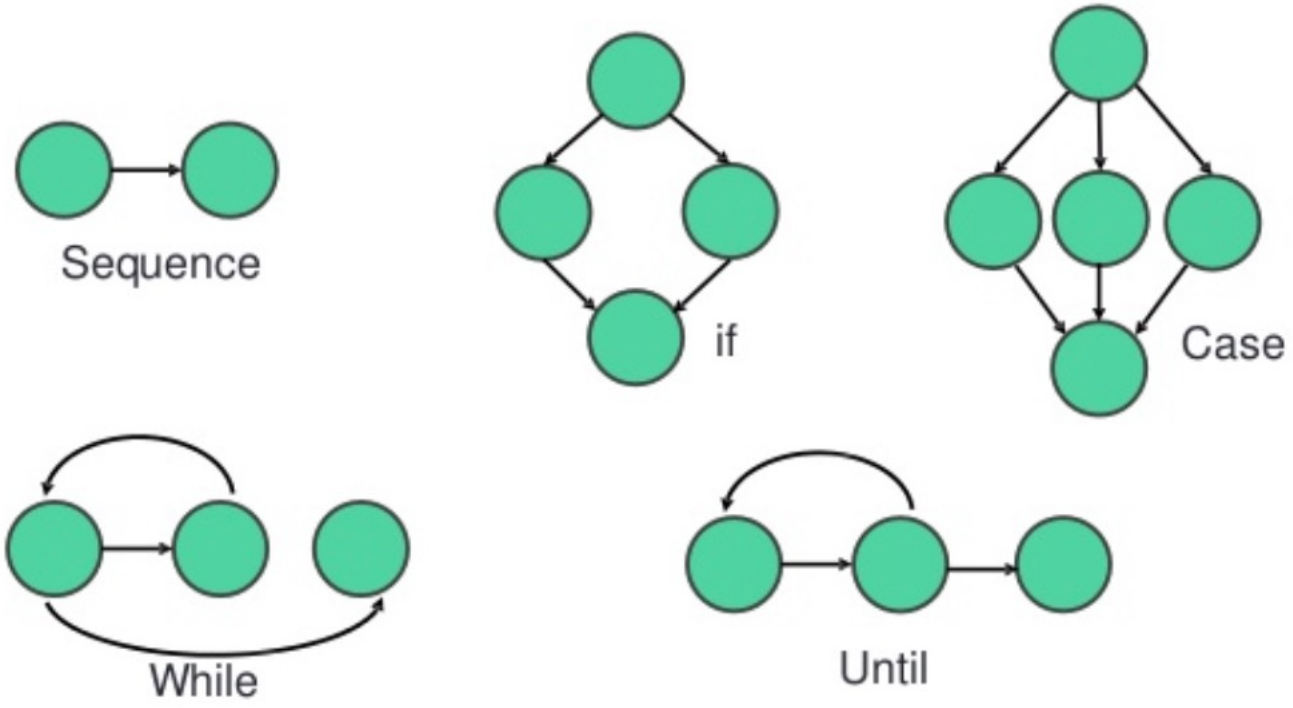
Technique: Control Flow Graph

```
a = 1;
b = 2;
c = 3;
if(a == 2){
      x = x + 2;
}
else {
      x = x / 2;
}
p = c /(b + c);
if(b/c > 3){
      z = x + y;
}
```



How many test cases?

Technique: Control Flow Graph



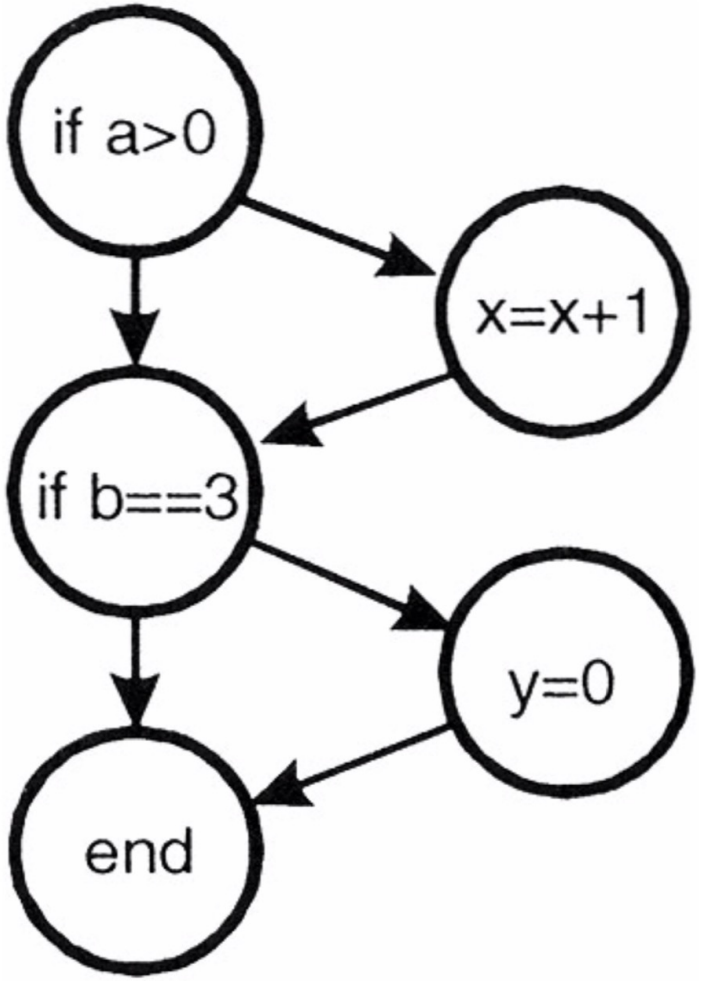
Code Coverage

The percentage of the code that has been tested

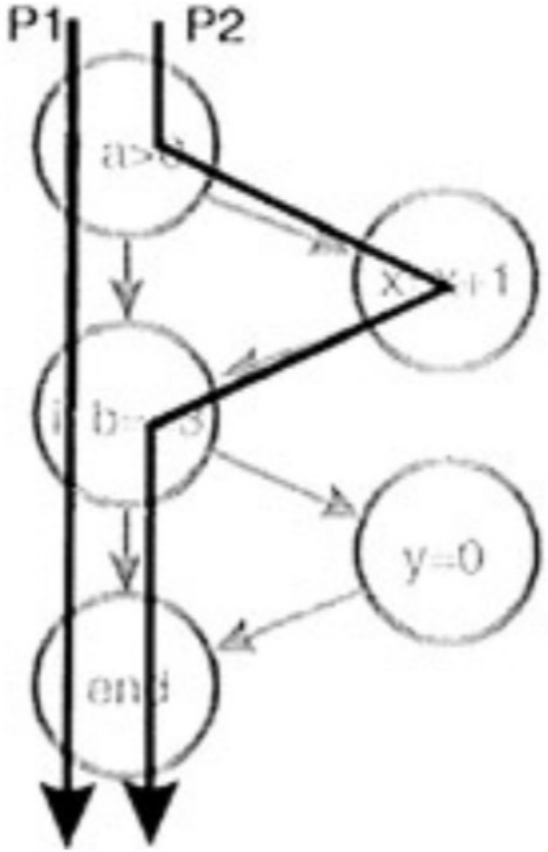
1. Statement Coverage
2. Branch/Decision Coverage
3. Condition Coverage
4. Multiple Condition Coverage
5. Path Coverage

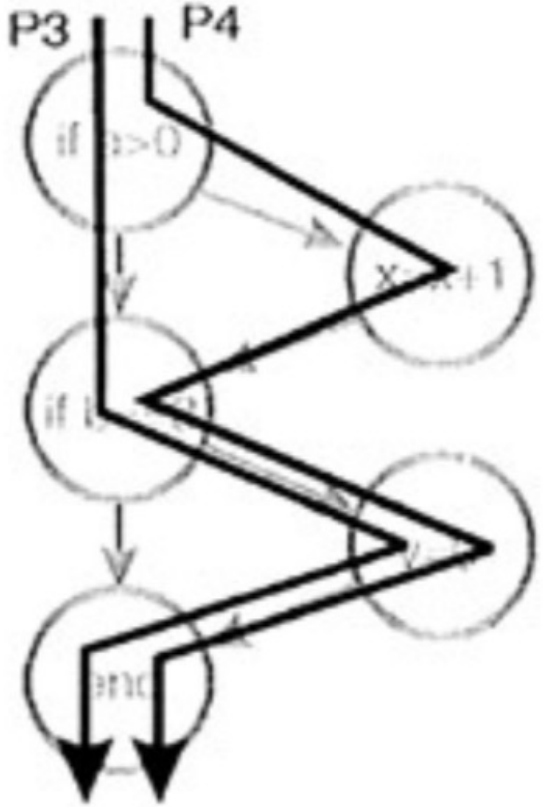
Example

```
1    if (a > 0) {
2        x = x + 1;
3    }
4    if (b == 3) {
5        y = 0;
6    }
```



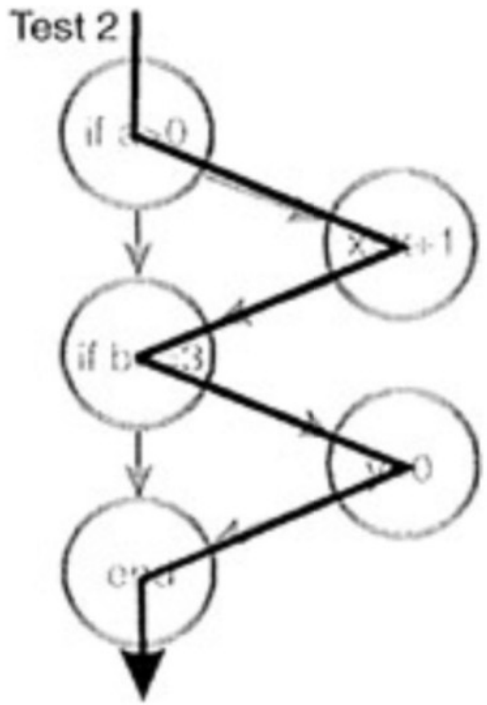
Execution paths





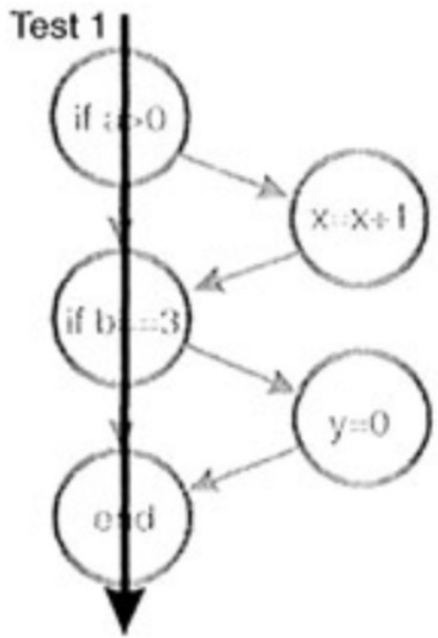
Statement Coverage

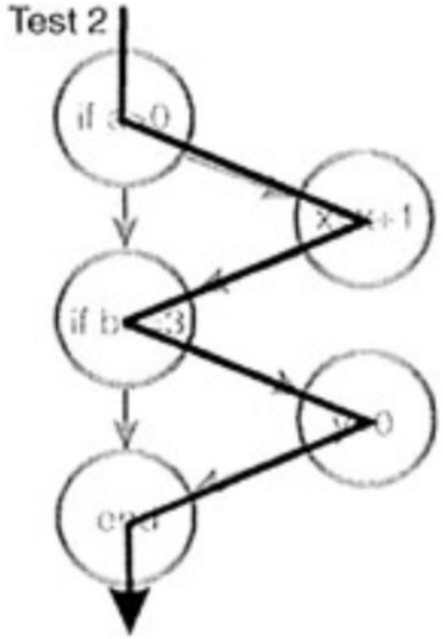
- Every statement is tested at least one
- 1 test case: 100% Statement Coverage
  - a=6, b=3



Branch/Decision Coverage

- Each decision that has a TRUE and FALSE outcome is evaluated at least one
- 2 test cases: 100% Decision Coverage
  - a=0, b=2
  - a=4, b=3





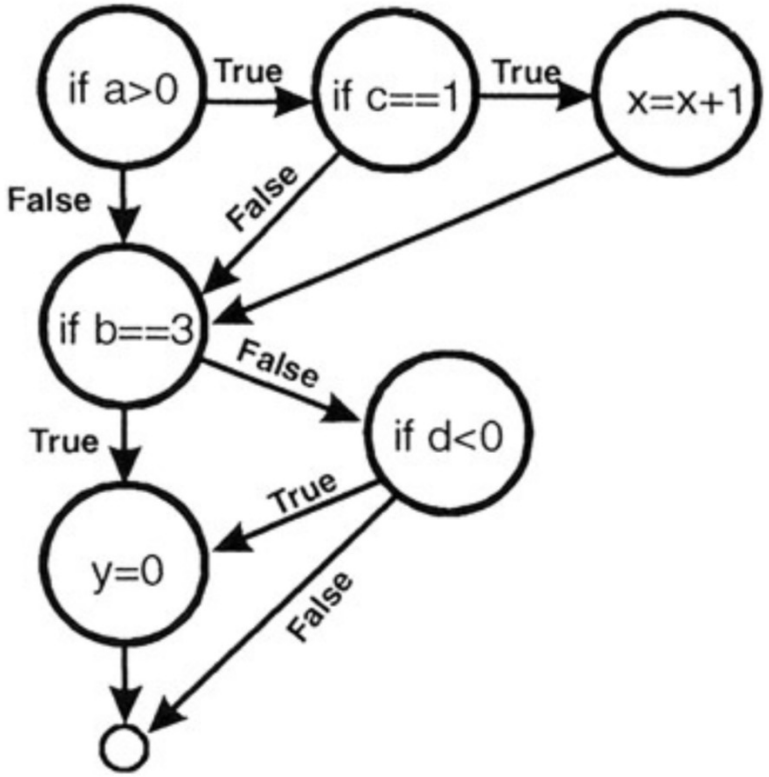
Condition Coverage

- Each condition that has a TRUE and FALSE outcome that makes up a decision is evaluated at least one
- 2 test cases:
  - a=1, c=1, b=3, d=-1
  - a=0, c=2, b=2, d=0

```
1    if (a > 0 && c == 1) {
2        x = x + 1;
3    }
4    if (b == 3 || d < 0) {
5        y = 0;
6    }
```

Multiple Condition Coverage

```
1    if (a > 0 && c == 1) {
2        x = x + 1;
3    }
4    if (b == 3 || d < 0) {
5        y = 0;
6    }
```



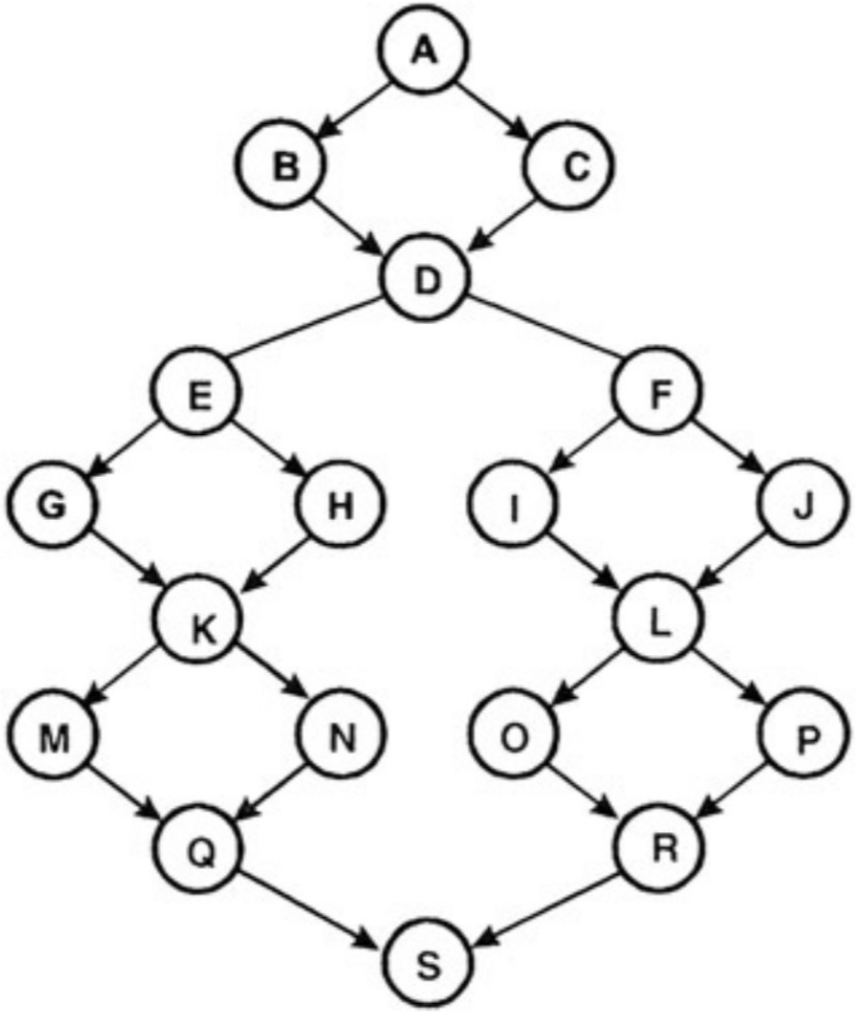
- 4 test cases:
  - a=1, c=1, b=3, d=-1
  - a=0, c=1, b=3, d=0
  - a=1, c=2, b=2, d=-1
  - a=0, c=2, b=2, d=0

Path Coverage

- Structure Testing / Basic Path Testing
  1. Derive the control flow graph
  2. Compute the graph's Cyclomatic Complexity
     - C = edges – nodes + 2
  3. Select a set of C basis paths
  4. Create a test case for each basis path
  5. Execute these tests

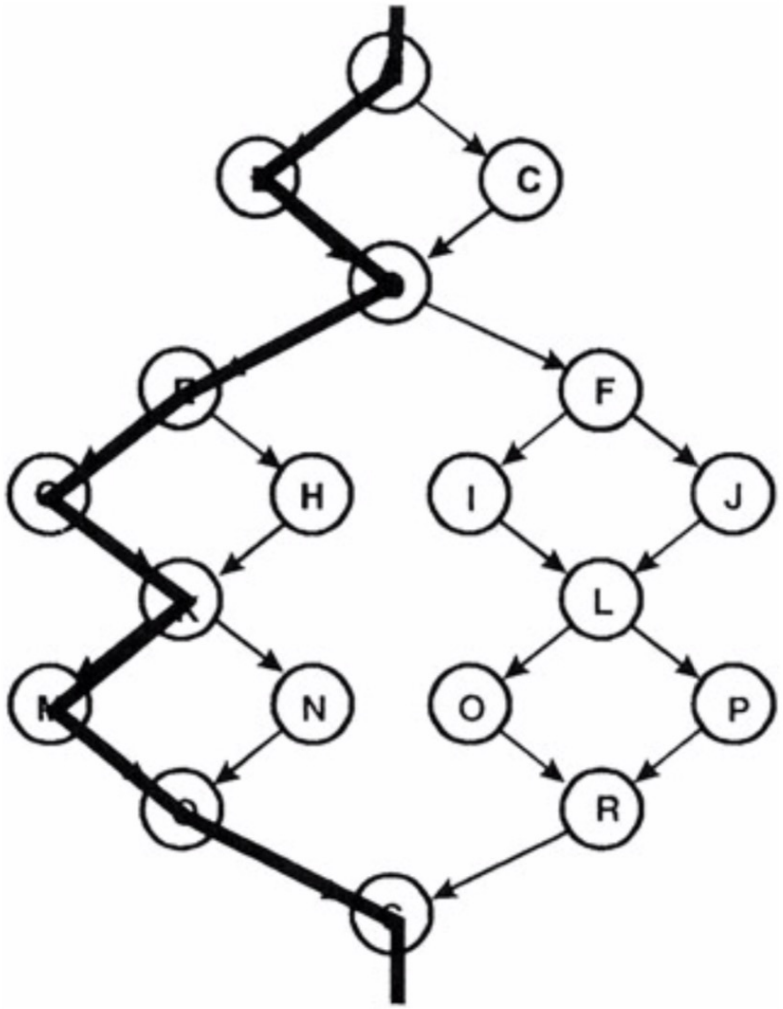
Basic Path Testing

- 24 edges, 19 nodes
- Cyclomatic Complexity
  - 24 – 19 + 2 = 7



Create a set of basis paths

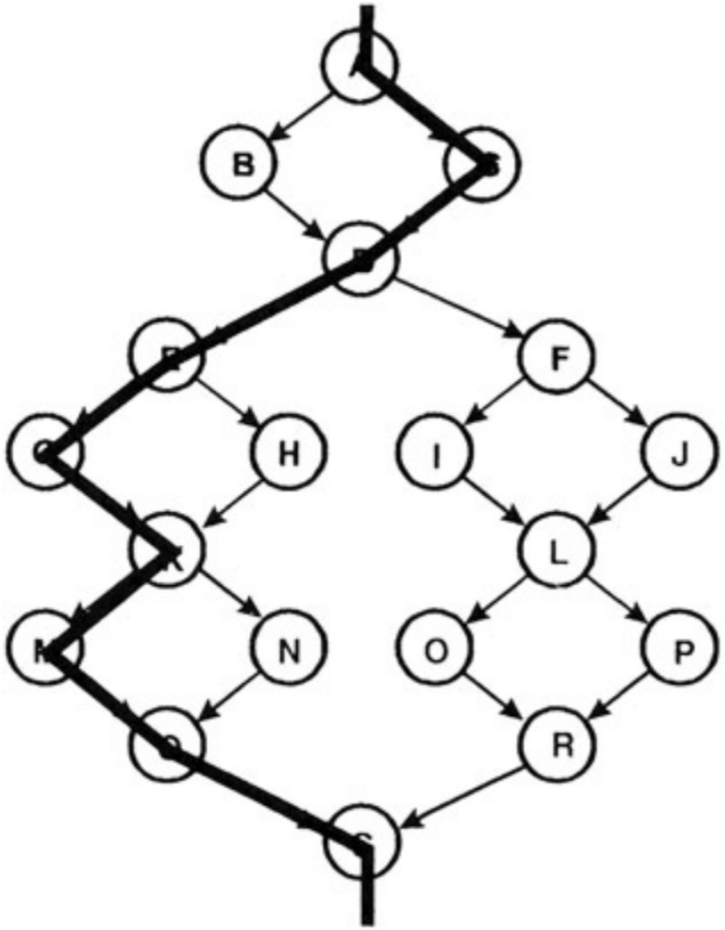
1. Pick a “baseline” path
   - ABDEGKMQS



Choose the next path

2. Change the outcome of the first decision along the baseline path while keeping the maximum number of other decisions the same

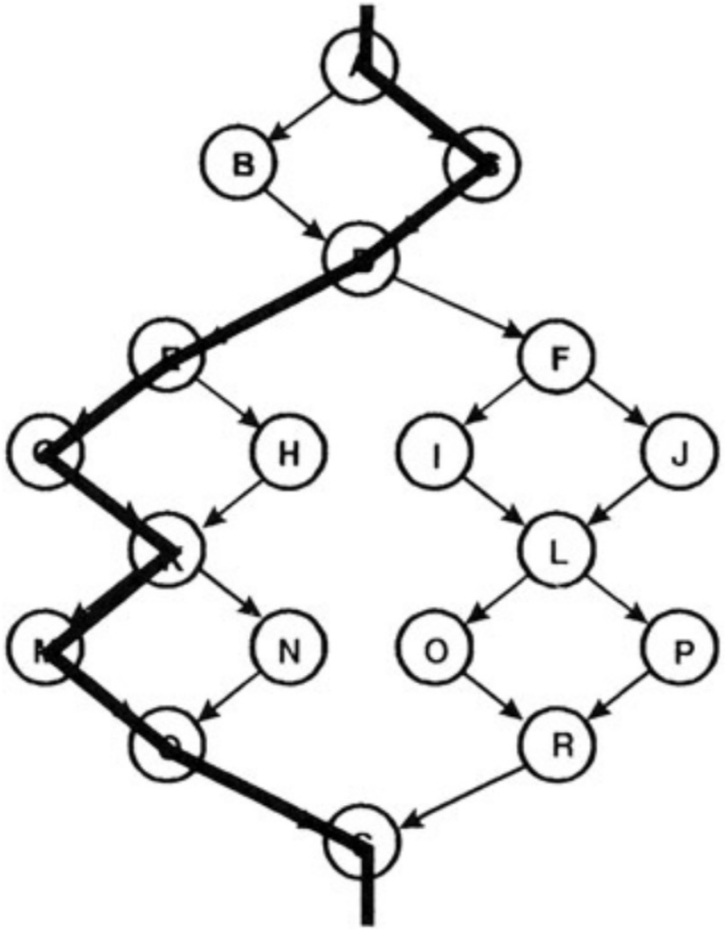
- ACDEGKMQS



Generate the third path

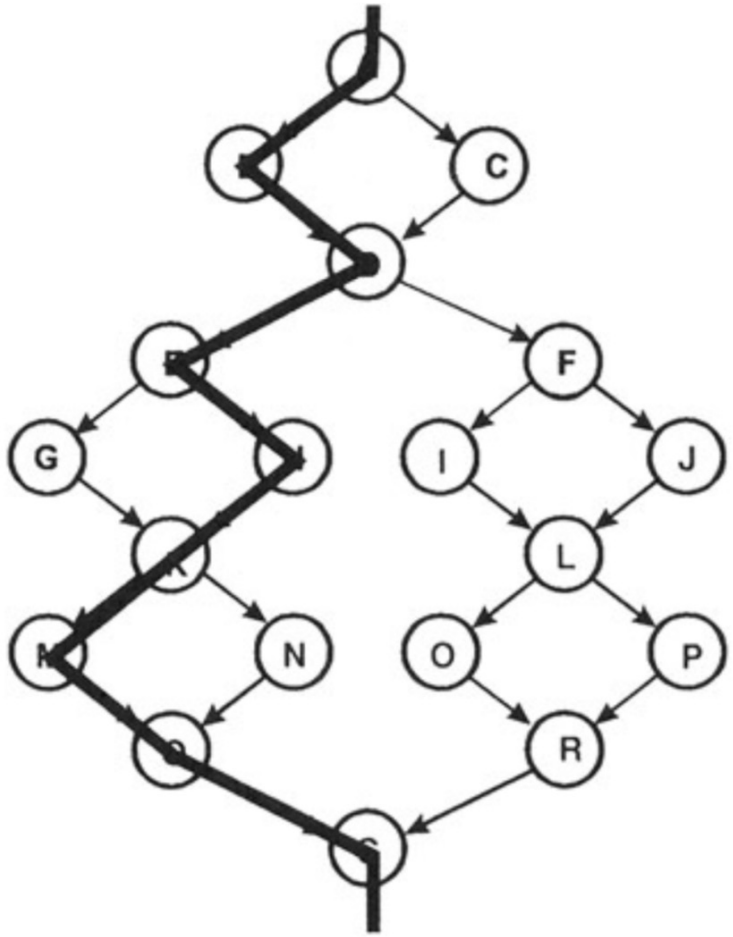
3. Begin again with the baseline but vary the second decision rather than the first

- ABDFILORS

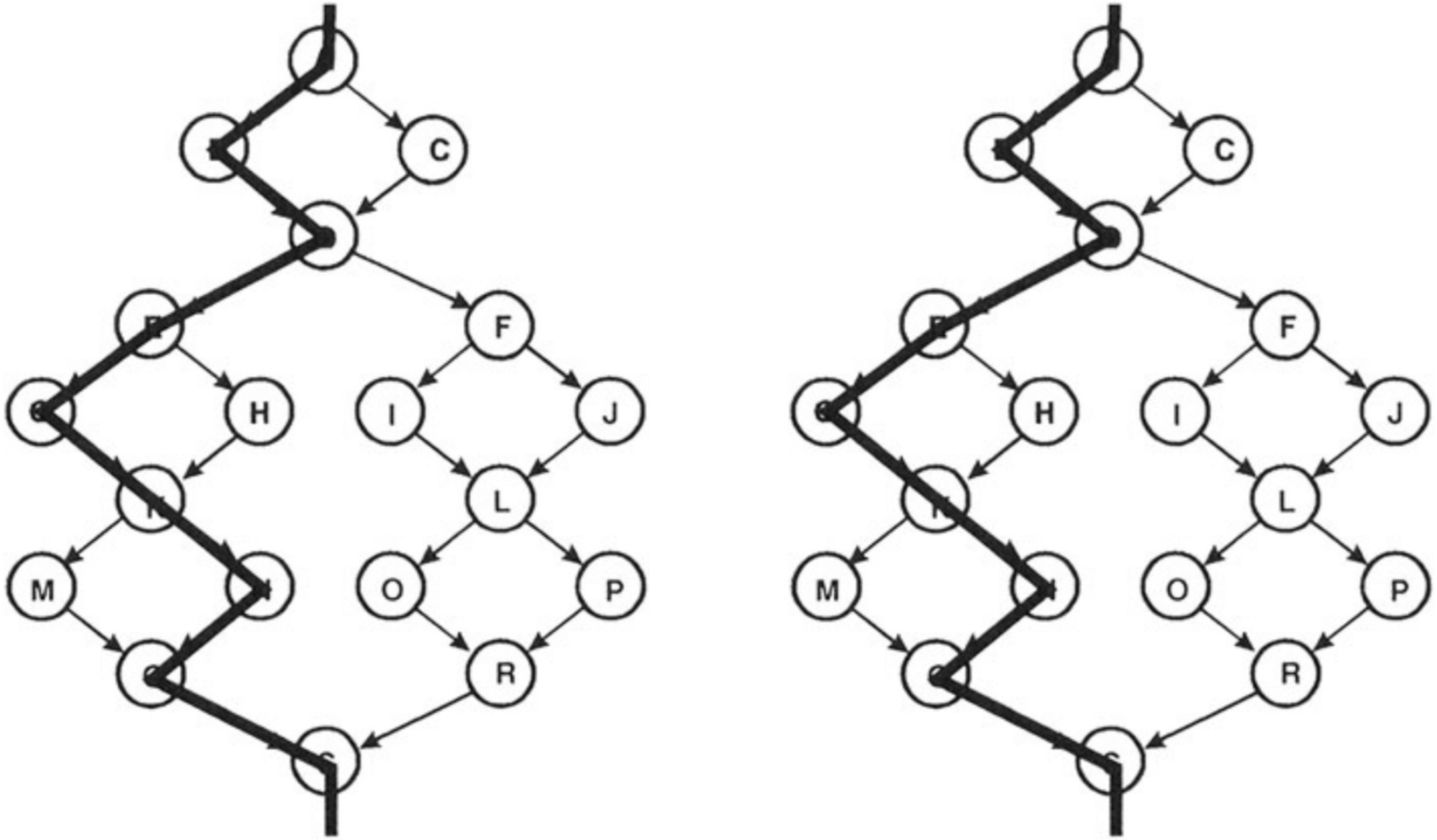


Generate the next paths

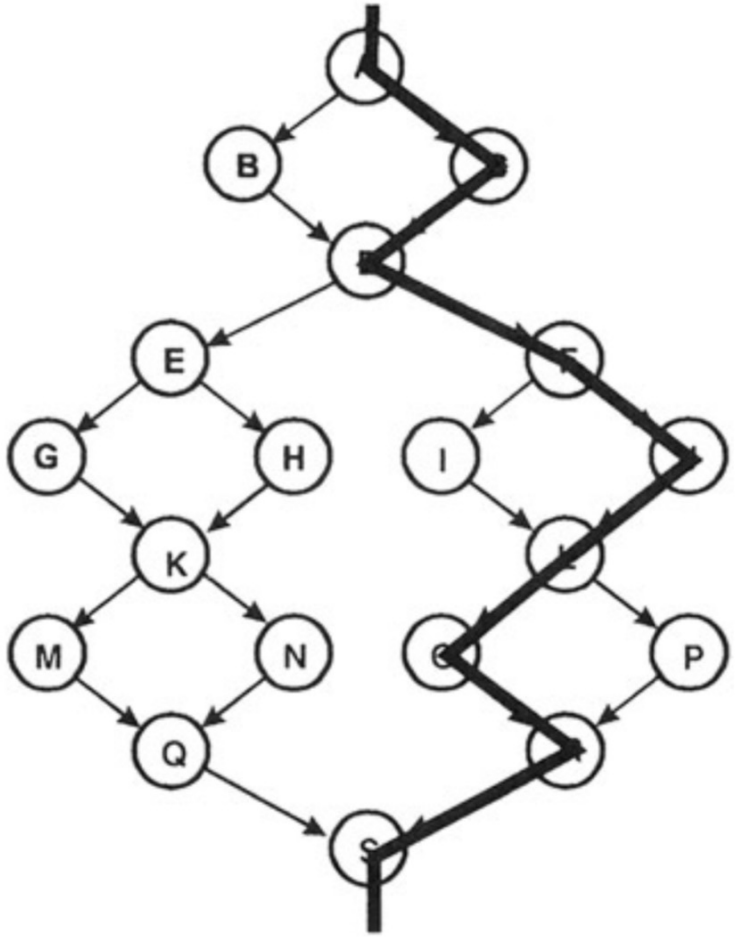
- Continue varying each decision, one by one, until the bottom of the graph is reached
- ABDEHKMQS



Generate the next paths



Generate the next paths



ABDEGKMQS

ACDEGKMQS

ABDFILORS

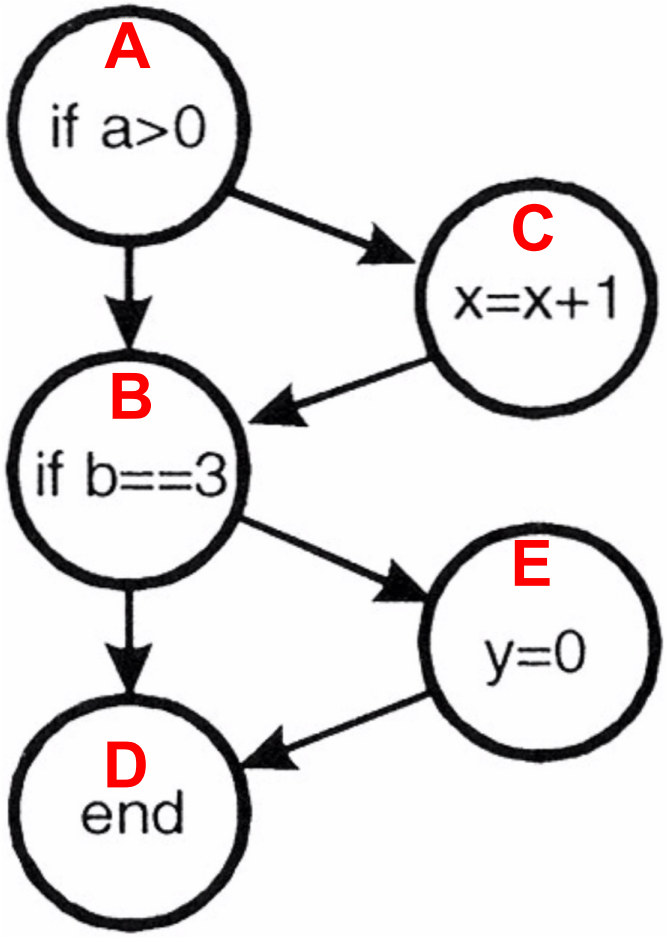
ABDEHKMQS

ABDEGKNQS

ACDFJLORS

ACDFILPRS

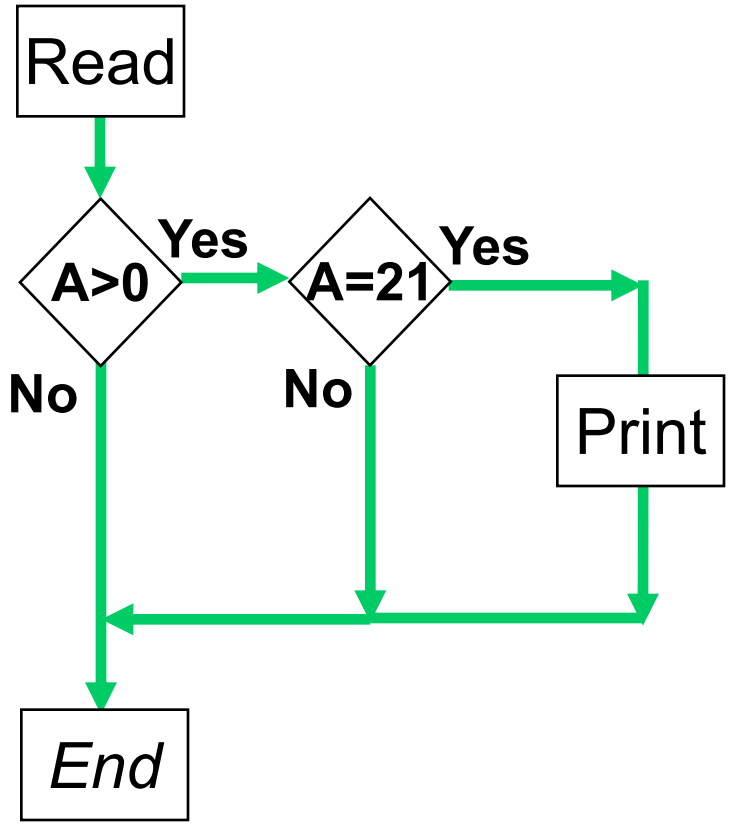
Example



- 6 edges, 5 nodes
- Cyclomatic Complexity
  - 6 – 5 + 2 = 3
- 3 basis paths
  - ABD
  - ACBD
  - ABED
- 3 Test cases
  - ABD: a=0, b=2
  - ACBD: a=1, b=2
  - ABED: a=0, b=3

Example

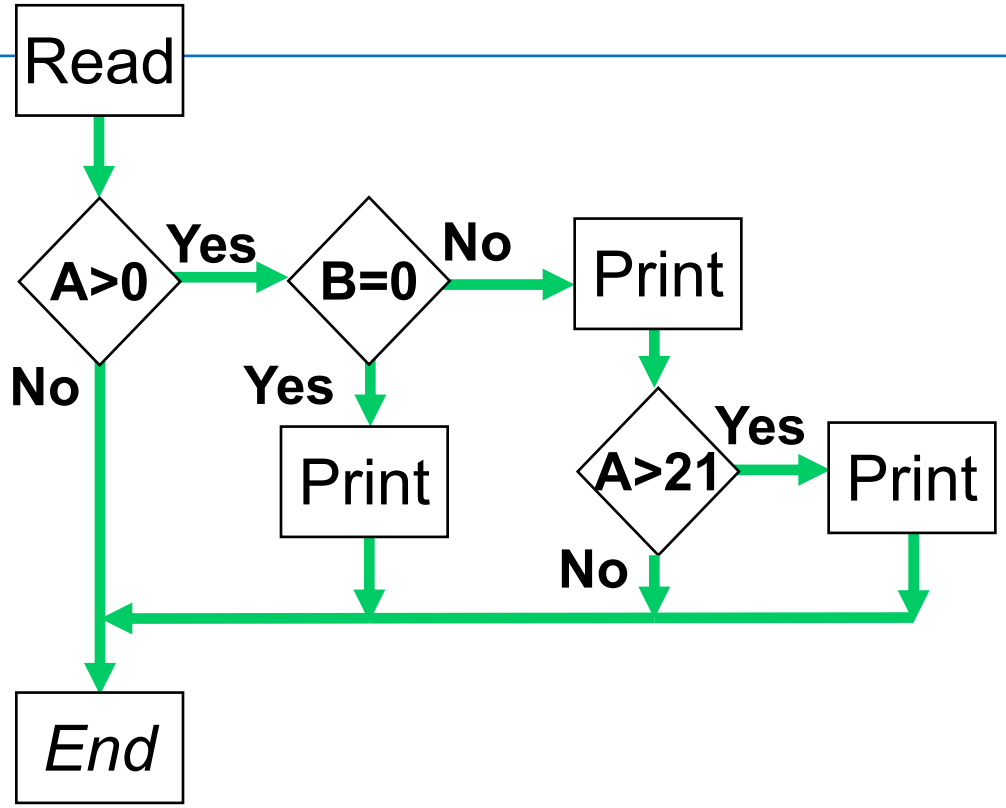
```
Read A
IF A > 0 THEN
  IF A = 21 THEN
    Print “Key”
  ENDIF
ENDIF
```



- Cyclomatic complexity: _____
- Minimum tests to achieve:
  - Statement coverage: _____
  - Branch coverage: _____

Example

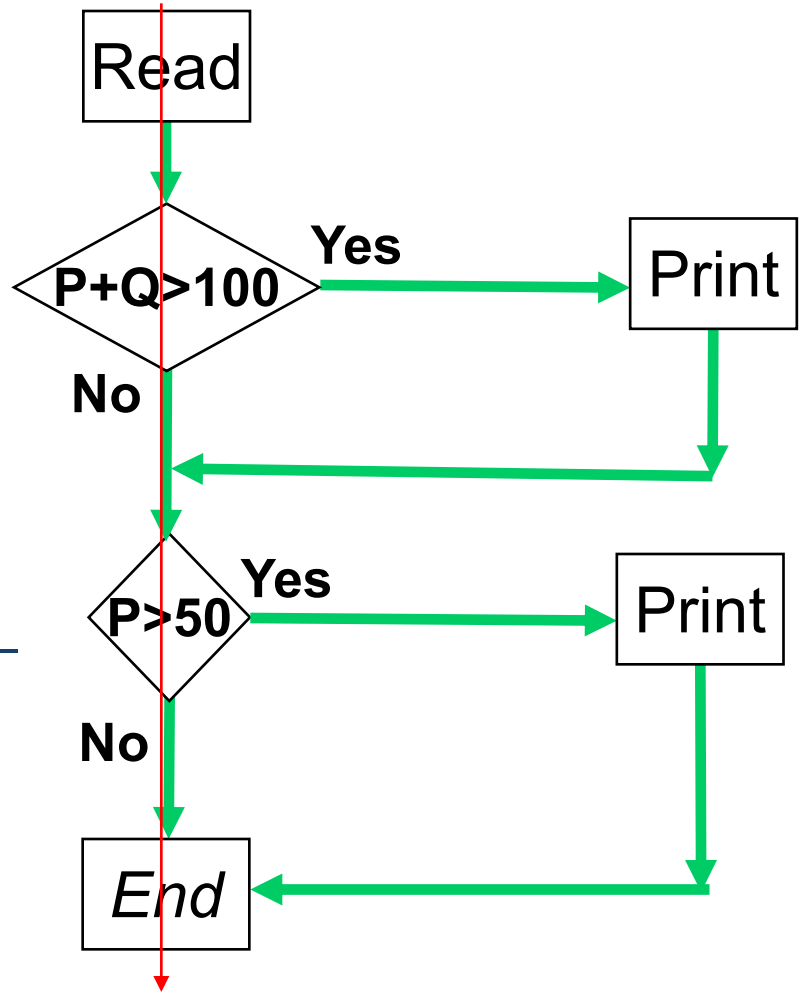
```
Read A
Read B
IF A > 0 THEN
    IF B  = 0 THEN
        Print “No values”
    ELSE
        Print B
        IF A > 21 THEN
            Print A
        ENDIF
    ENDIF
ENDIF
```



- Cyclomatic complexity: _____
- Minimum tests to achieve:
  - Statement coverage: _____
  - Branch coverage: _____

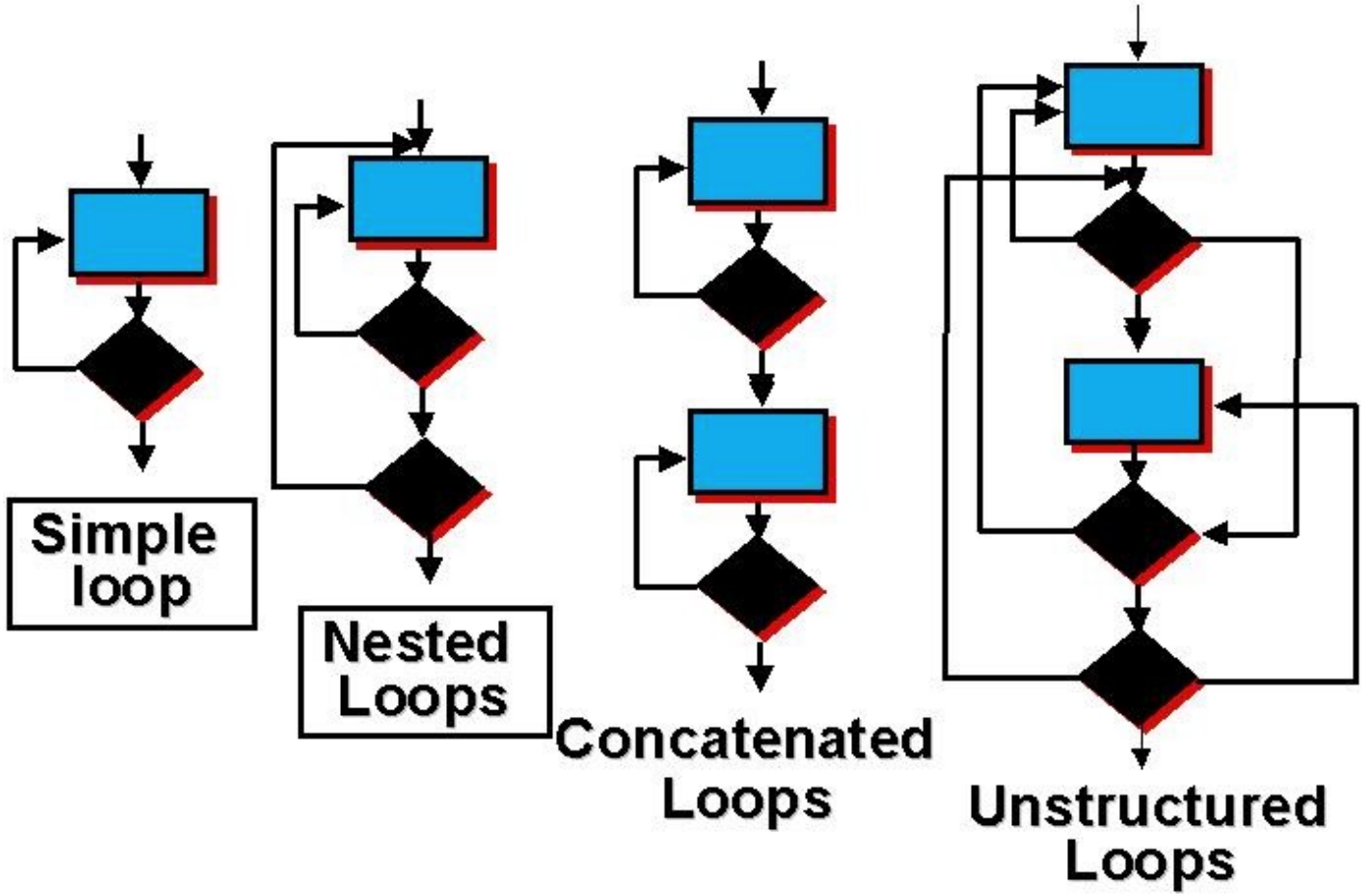
Example

```
Read P
Read Q
IF P+Q > 100 THEN
Print “Large”
ENDIF
If P > 50 THEN
Print “P Large”
ENDIF
```



- Cyclomatic complexity: _____
- Minimum tests to achieve:
  - Statement coverage: _____
  - Branch coverage: _____

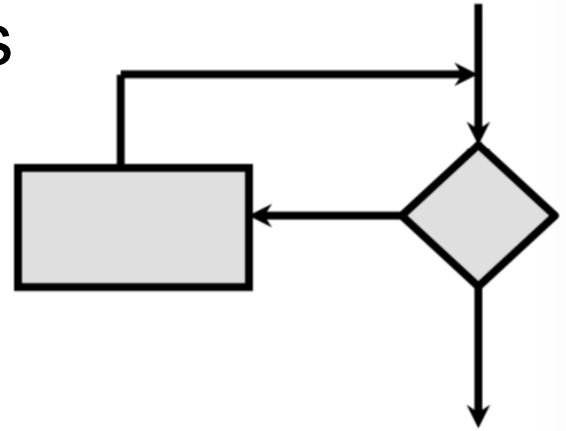
Loop Testing



Loop Testing: Simple Loops

- Minimum conditions – simple loops
  1. **Skip** the loop entirely
  2. Only **one pass** through the loop
  3. **Two passes** through the loop
  4. **m passes** through the loop m < n
  5. **(n-1), n, and (n+1) passes** through the loop

  Where n is the maximum number of allowable passes



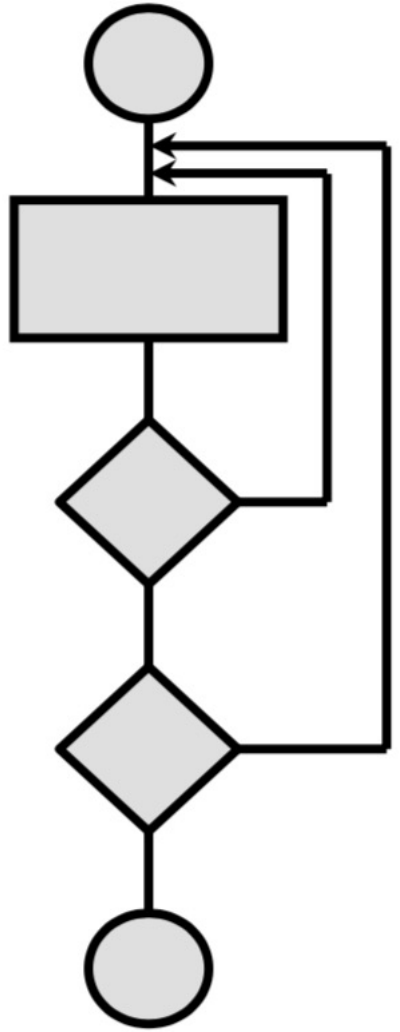
Loop Testing

```
public class loopdemo
{
    private int[] numbers = {5,-3,8,-12,4,1,-20,6,2,10};

    /** Compute total of numItems positive numbers in the array
     *  @param numItems how many items to total, maximum of 10.
     */
    public int findTotal(int numItems)
    {
        int total = 0;
        if (numItems <= 10)
        {
            for (int count=0; count < numItems; count = count + 1)
            {
                if (numbers[count] > 0)
                {
                    total = total + numbers[count];
                }
            }
        }
        return total;
    }
}
```

numItems
0
1
2
5
9
10
11

Nested Loops

- Extend simple loop testing
- Reduce the number of tests
  - start at the innermost loop; set all other loops to minimum values
  - conduct simple loop test; add out of range or excluded values
  - work outwards while keeping inner nested loops to typical values
  - continue until all loops have been tested



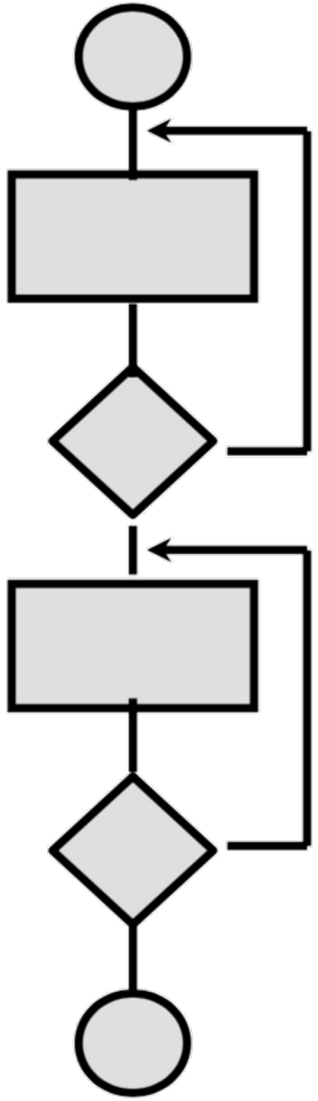
Concatenated Loops

- Independent Loops

=> Test as simple loops

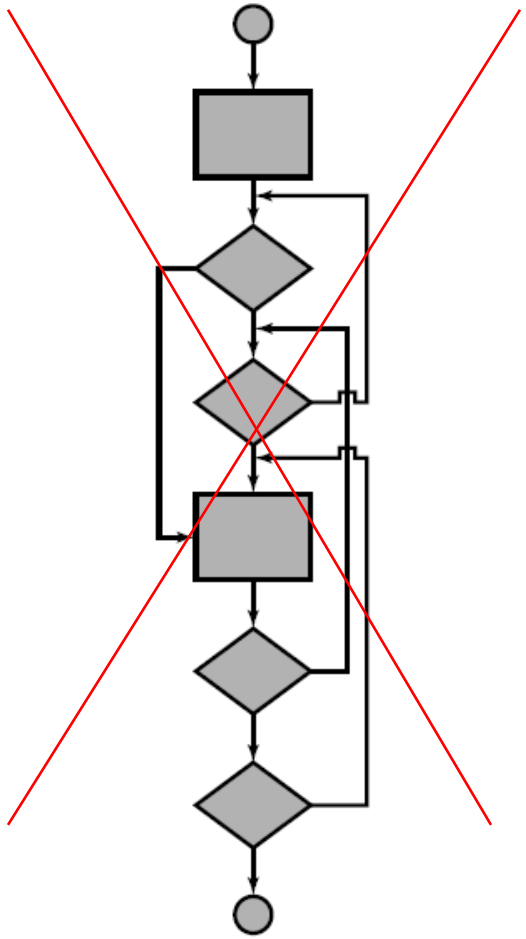
- Dependent Loops

=> Test as nested loops



Unstructured Loops

- DON’T test
- Re-design



Data Flow Testing: Definition

- Def – assigned or changed
- Uses – utilized (not changed)
  - C-use (Computation): right-hand side of an assignment, an index of an array, parameter of a function
  - P-use (Predicate): branching the execution flow (if, while, for statement)

Data Flow Testing

- Def-Use testing
  - All navigation paths from every definition of a variable to every use of it is exercised
- All-Use testing
  - At least one navigation path from every definition of a variable to every use of it is exercised

Data Flow Testing

1	sum = 0	*sum, def*
2	read (n),	*n, def*
3	i = 1	*i, def*
4	while (i <= n)	*i, n p-sue*
5	read (number)	*number, def*
6.	sum = sum + number	*sum, def, sum, number, c-use*
7	i = i + 1	*i, def, c-use*
8	end while	
9	print (sum)	*sum, c-use*

Def-Use Testing

- Ví dụ

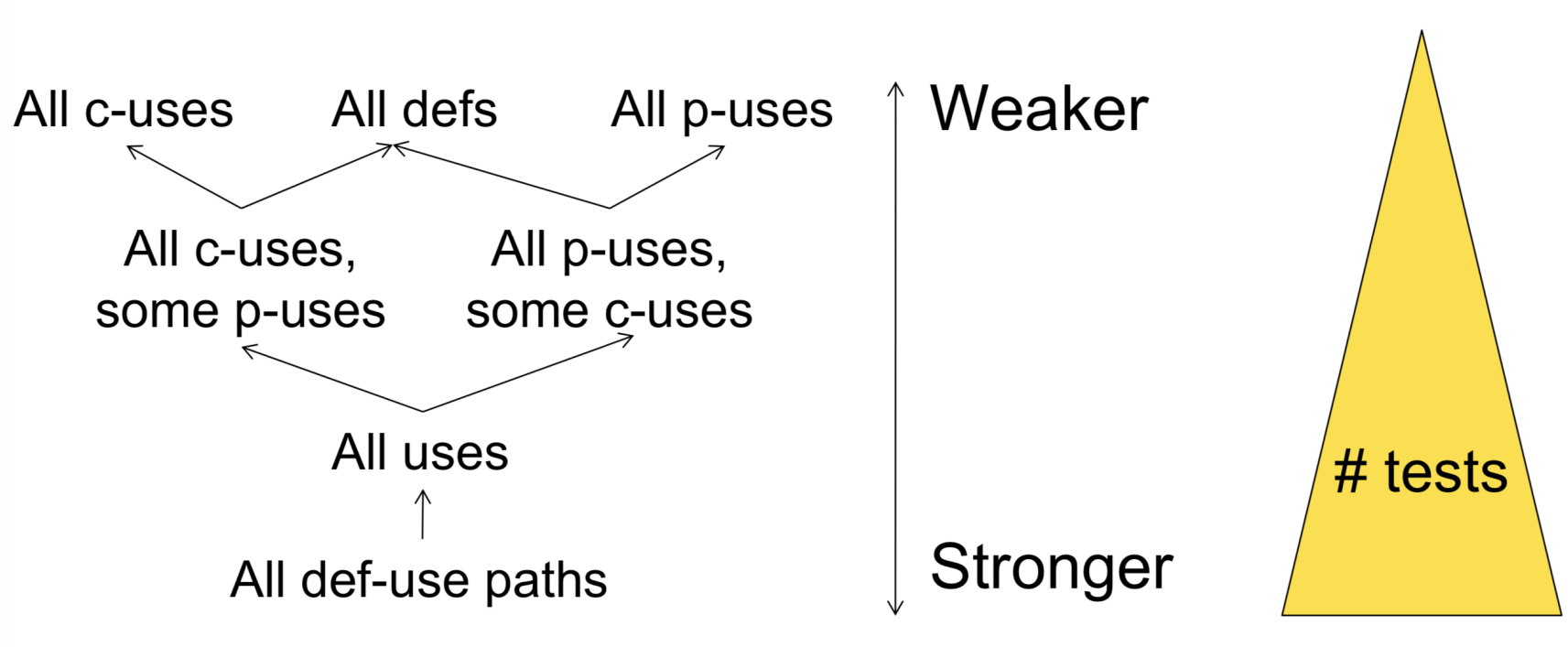
Table for sum

pair id	def	use
1	1	6
2	1	9
3	6	6
4	6	9

Table for i

pair id	def	use
1	3	4
2	3	7
3	7	7
4	7	4

Data Flow Criteria



White box Testing Disadvantages

1. The number of execution paths may be so large that they cannot all be tested
2. The chosen test cases may not detect data sensitivity errors
   - p = q / r
   - May execute correctly except when r=0
3. The tests are based on the existing paths, nonexistent paths cannot be discovered
4. Testers must have the programming skills


Q
A