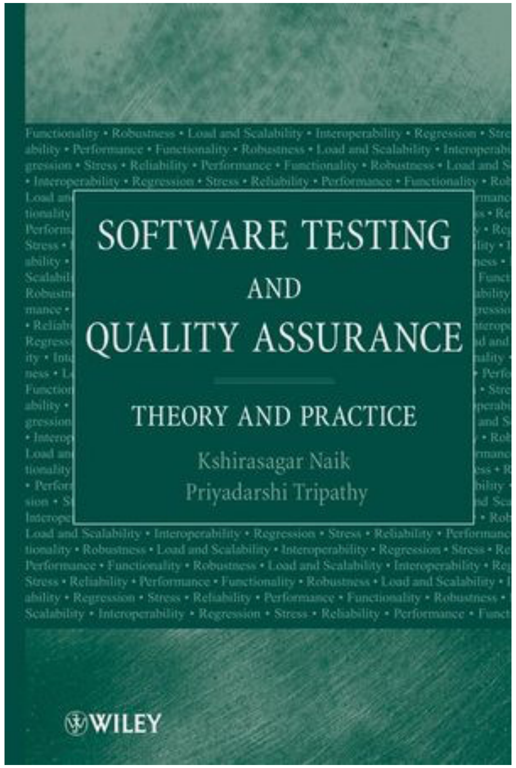
# Software Testing

## ETS 200

## http://cs.lth.se/ets200




Chapter 4, 5, 3.5

Prof. Per Runeson

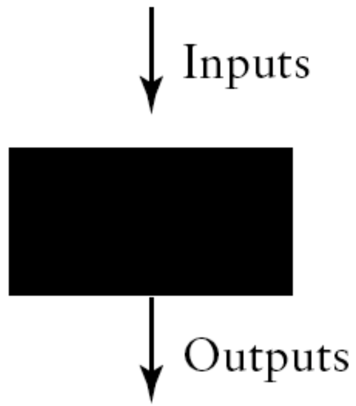
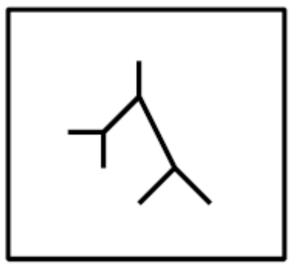
# Lecture

- White-box testing techniques (Lab 1)
  – Control flow (Chapter 4)
  – Data flow (Chapter 5)
- Mutation testing (Section 3.5)

# Why Test Case Design Techniques?

- Exhaustive testing (use of all possible inputs and conditions) is impractical
  - Must use a subset of all possible test cases
  - Must have high probability of detecting faults
- Need processes that help us selecting test cases
  - Different people – equal probability to detect faults
- Effective testing – detect more faults
  - Focus attention on specific types of faults
  - Know you’re testing the right thing
- Efficient testing – detect faults with less effort
  - Avoid duplication
  - Systematic techniques are measurable and repeatable

# Basic Testing Strategies

| Test Strategy | Tester's View | Knowledge Sources | Methods |
|---|---|---|---|
| Black box |  | Requirements document<br>Specifications<br>Domain knowledge<br>Defect analysis data | Equivalence class partitioning<br>Boundary value analysis<br>State transition testing<br>Cause and effect graphing<br>Error guessing |
| White box |  | High-level design<br>Detailed design<br>Control flow graphs<br>Cyclomatic complexity | Statement testing<br>Branch testing<br>Path testing<br>Data flow testing<br>Mutation testing<br>Loop testing |

# Black-Box vs. White-Box

- External/user view:
  - Check conformance with specification
- Abstraction from details:
  - Source code not needed
- Scales up:
  - Different techniques at different levels of granularity

USE

- Internal/developer view:
  - Allows tester to be confident about test coverage
- Based on control or data flow:
  - Easier debugging
- Does not scale up:
  - Mostly applicable at unit and integration testing levels

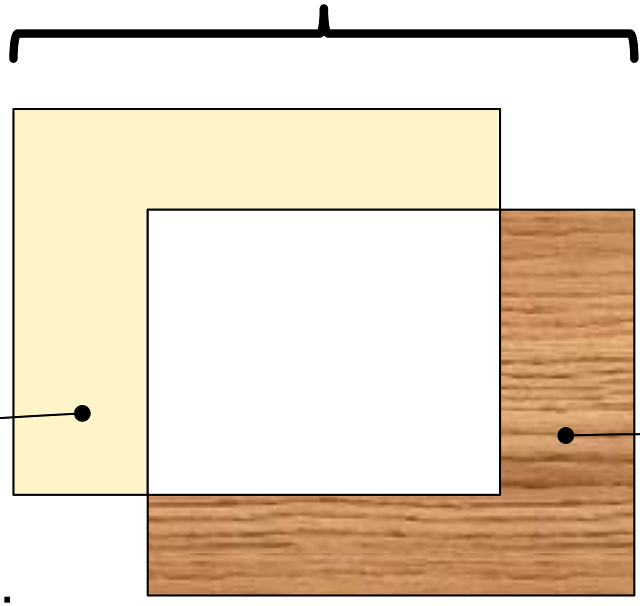
BOTH!

# Black-Box vs. White-Box

System

Specification

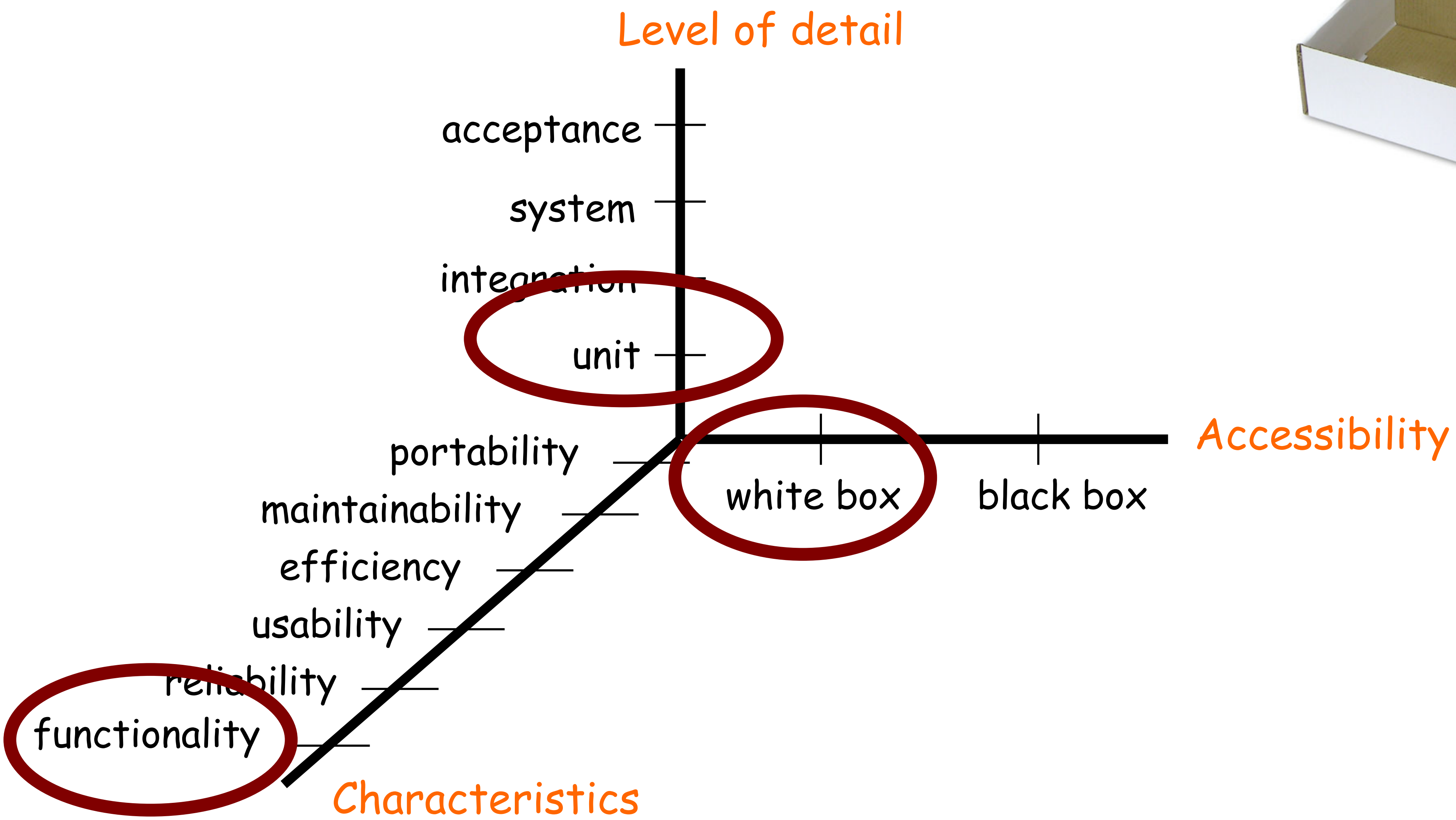
Implementation



Missing functionality:
Cannot be revealed by white-box techniques

Unexpected functionality:
Cannot be revealed by black-box techniques

# Types of Testing




# Control Flow Graph (CFG)

- Structurally, a path is a sequence of statements in a program unit
- Semantically, it is an execution instance of the unit
- For a given set of input data, the program unit executes a certain path.

# CFG symbols

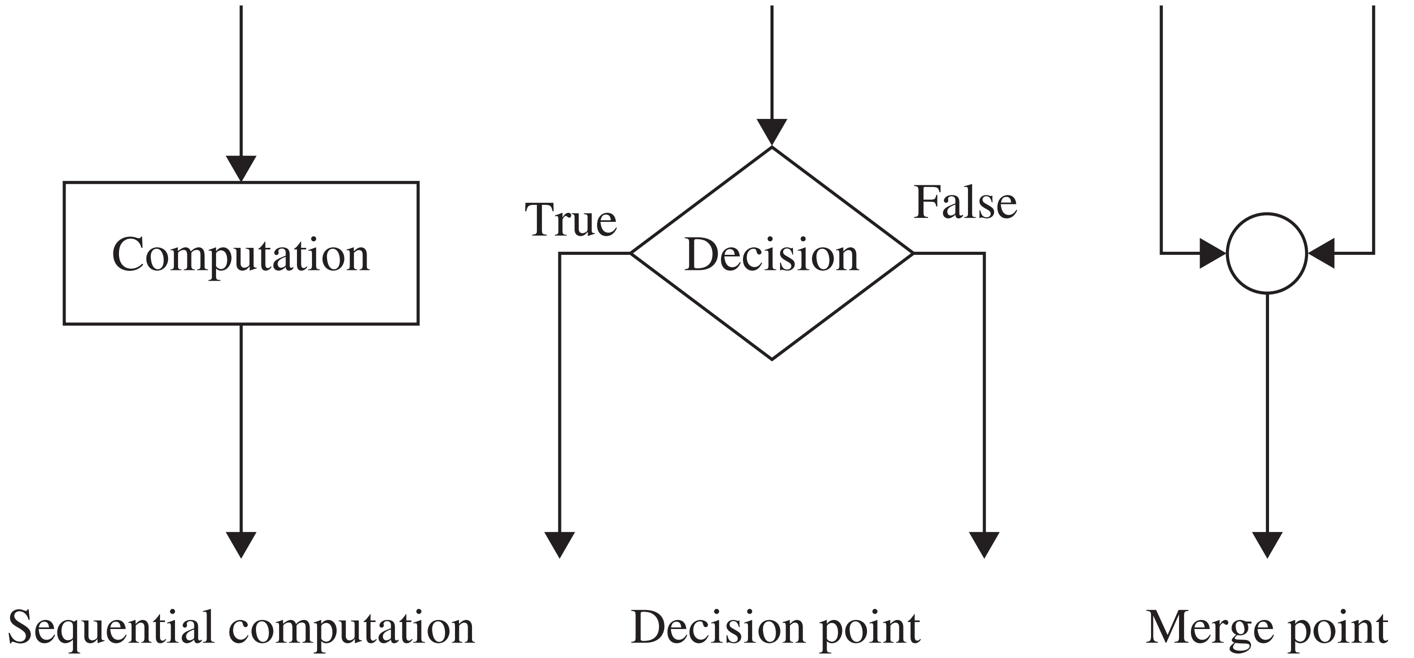


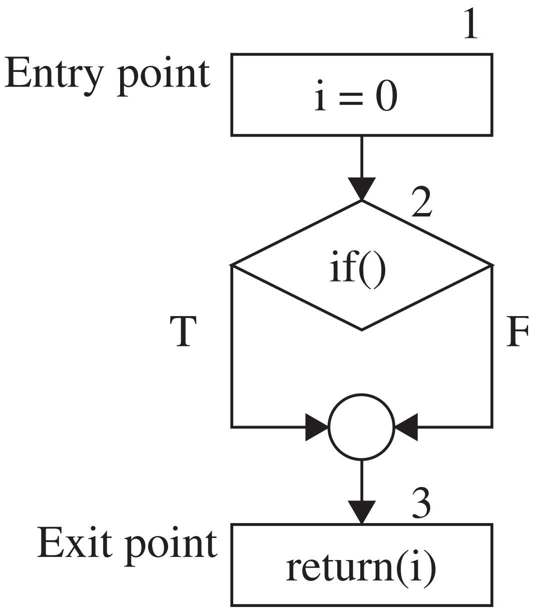

Figure 4.2 Symbols in a CFG.

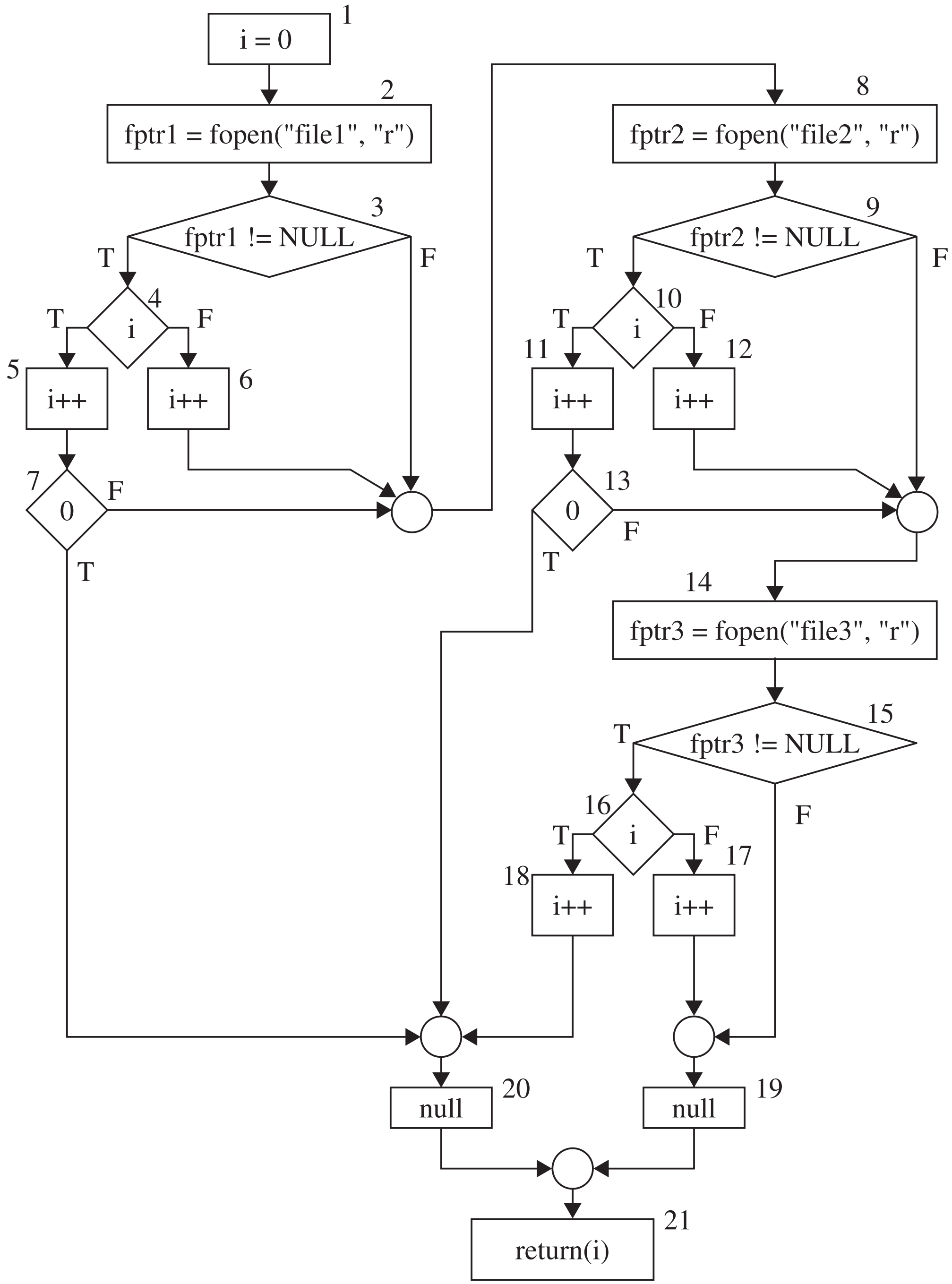
```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
   /*
     This function tries to open files "file1", "file2", and
     "file3" for read access, and returns the number of files
     successfully opened. The file pointers of the opened files
     are put in the global variables.
   */
     int i = 0;
     if(
         ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)
                                                    && (0)) ||
         ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)
                                                    && (0)) ||
         ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
     );
     return(i);
}
```

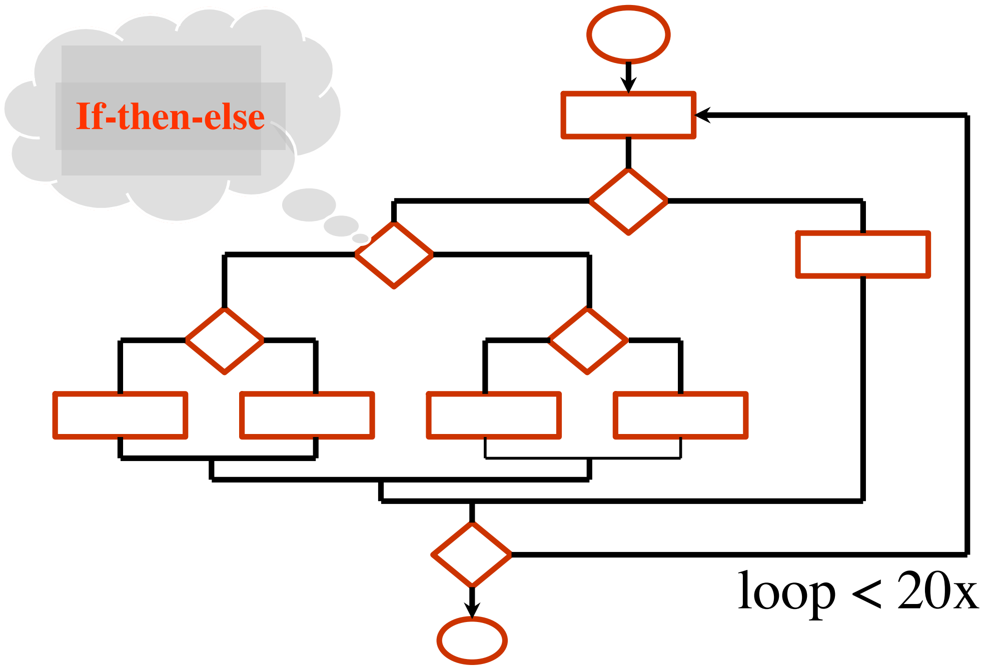
Figure 4.3    Function to open three files.

# Example CFG (Fig 4.4. vs 4.5)



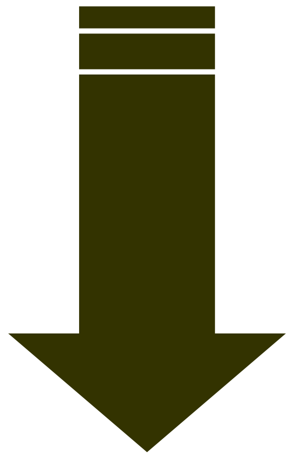



# White-Box Testing (Ch 4, 5)
# Exhaustive Testing



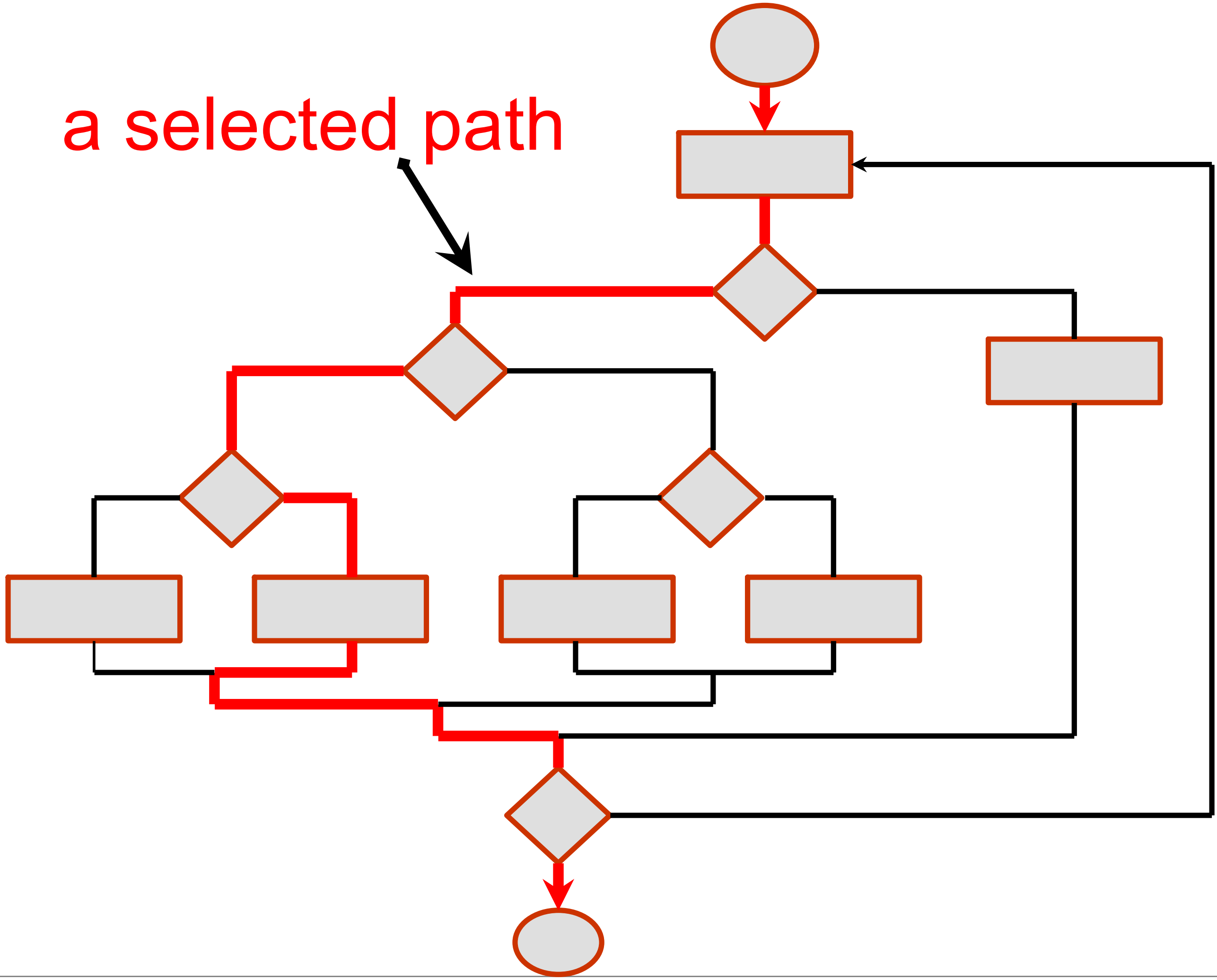

There are many possible paths!

$5^{20}$ (~$10^{14}$ ) different paths



Selective Testing

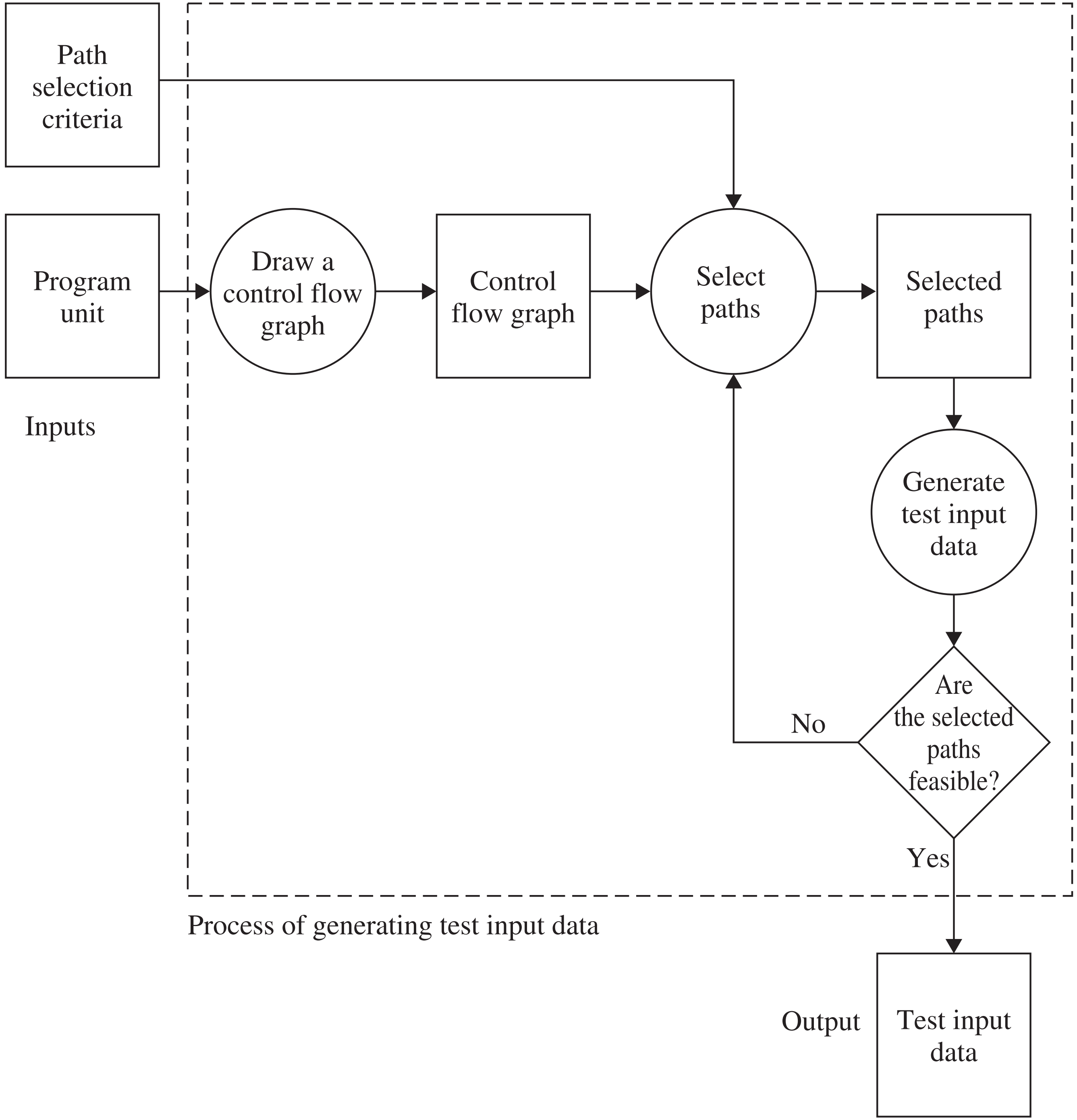
# Selective White-box Testing




**Selective:**

- ✓ Control flow testing
- ✓ Data flow testing
- ✓ Loop testing
- ✓ Fault-based testing

# Work process



# Path selection criteria

- All paths
- Statement coverage
- Branch coverage
- Predicate coverage

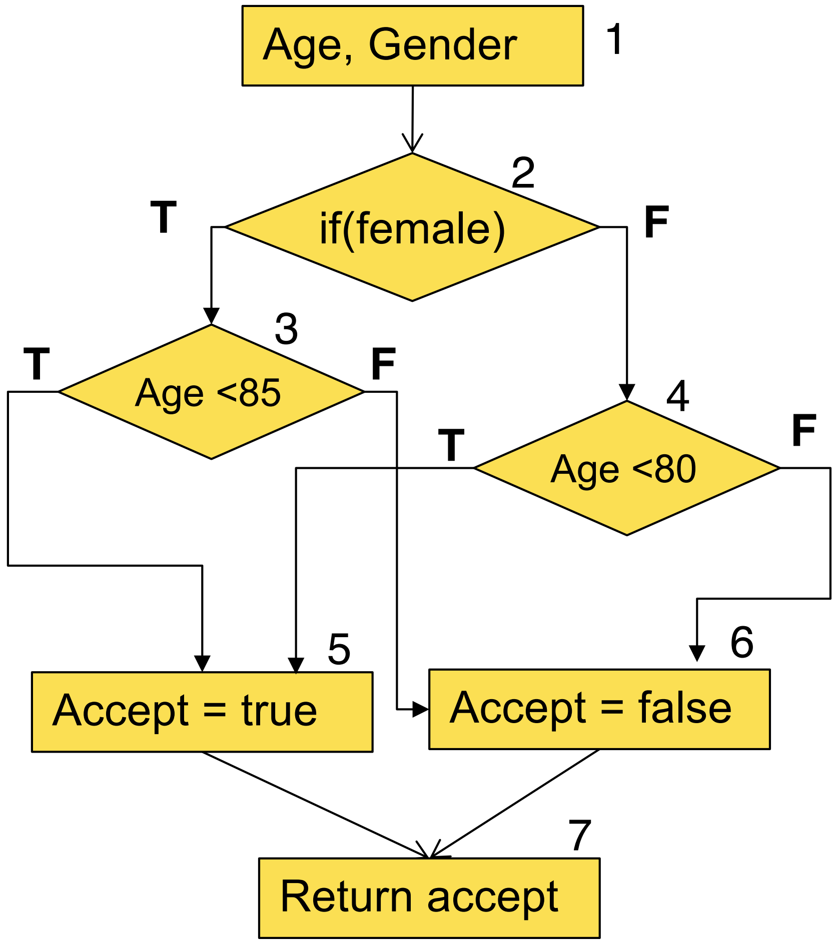
# A Word of Warning

Staats et al, On the Danger of Coverage Directed Test Case Generation, FASE 2012

- First, coverage criteria satisfaction *alone is a poor* indication of test suite effectiveness.
- Second, the use of structural coverage as a *supplement*—not a target—for test generation can have a positive impact.

# Life Insurance Example

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```



# All paths

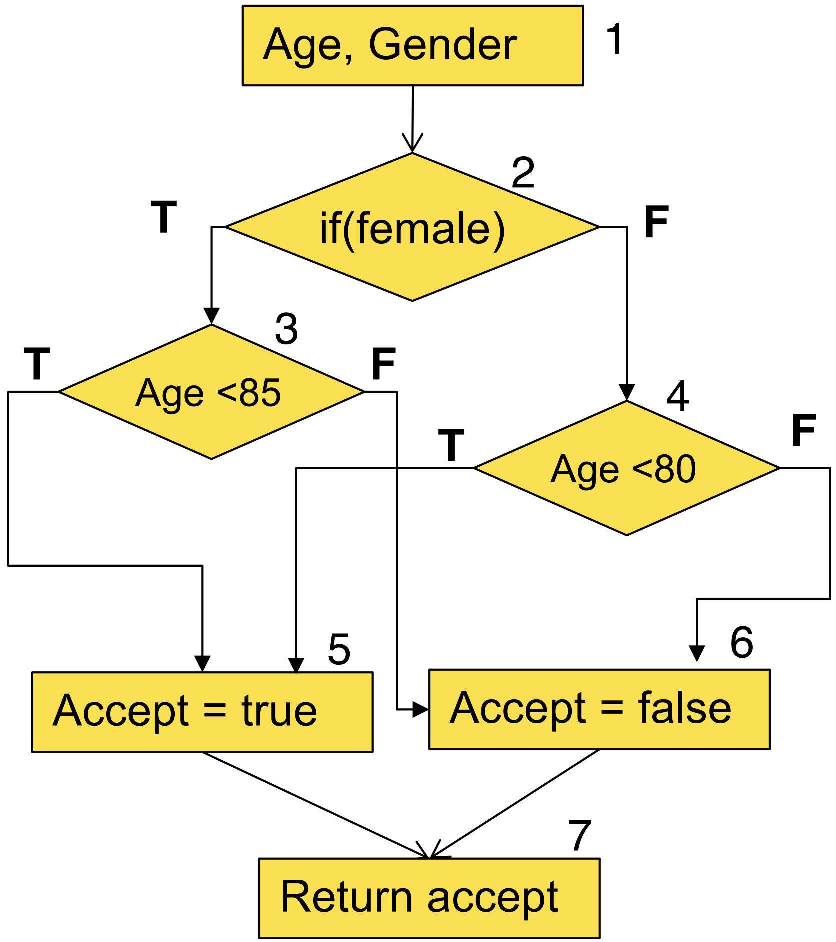
| Female | Age < 85 | Age < 80 |
|---|---|---|
| Yes | Yes | Yes |
| Yes | Yes | No |
| ~~Yes~~ | ~~No~~ | ~~Yes~~ |
| Yes | No | No |
| No | Yes | Yes |
| No | Yes | No |
| ~~No~~ | ~~No~~ | ~~Yes~~ |
| No | No | No |

<Yes,Yes,*> 1-2(T)-3(T)-5-7

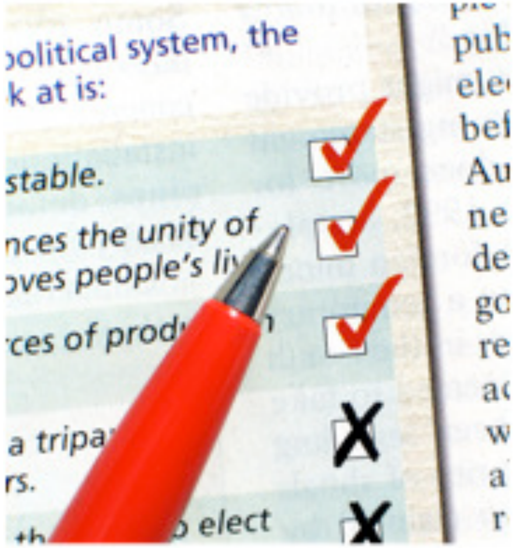
<Yes,No,No> 1-2(T)-3(F)-6-7

<No,Yes,Yes> 1-2(F)-4(T)-5-7

<No,*,No> 1-2(F)-4(F)-6-7



# Statement Coverage

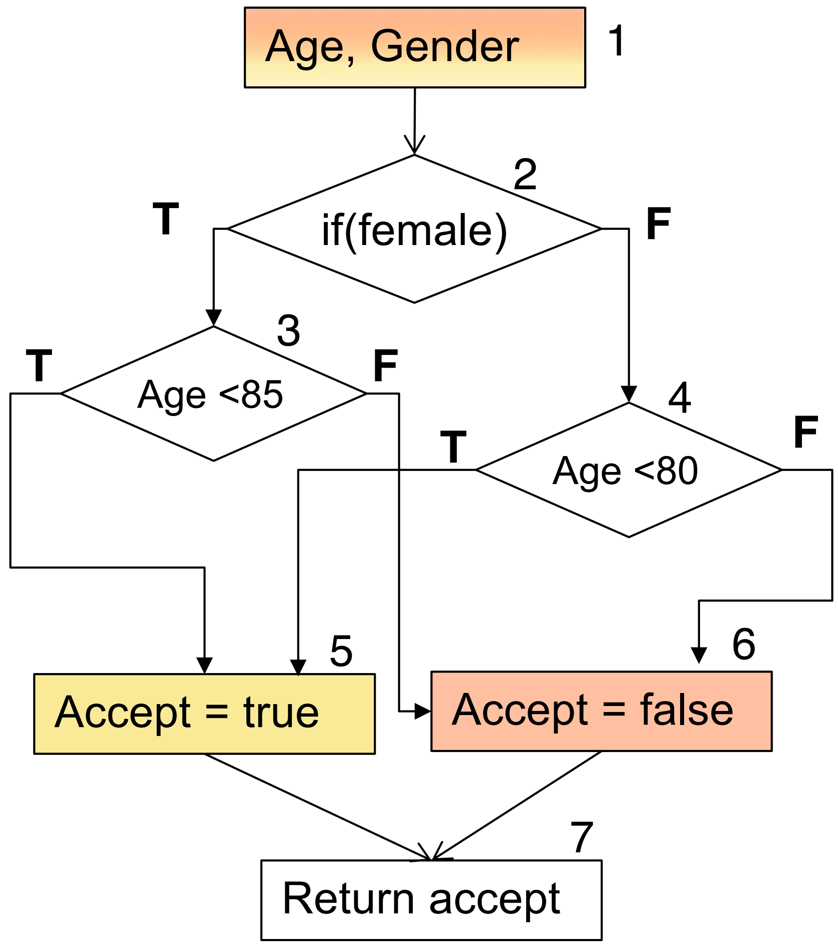


- Execute each statement at least once

- A possible concern may be:
  - Dead code

# Statement Coverage

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
    accept := age < 85;
   else
    accept := age < 80;
return accept
```




AccClient(83, female)->accept

AccClient(83, male) ->reject

# Branch Coverage

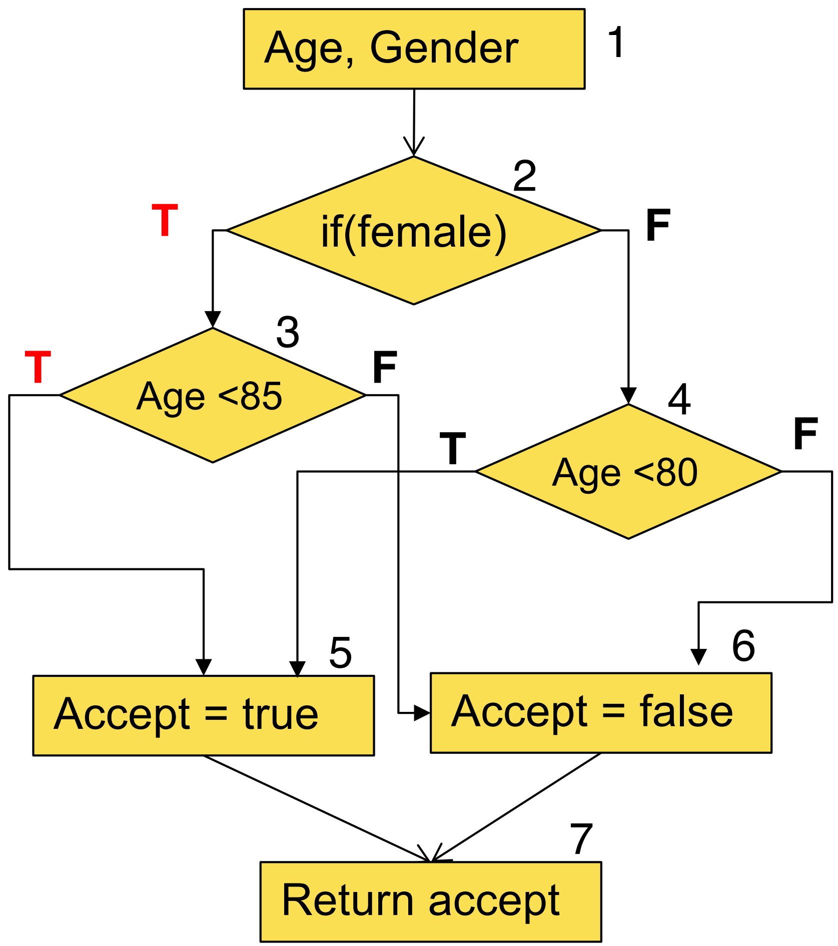
- Tests cover each decision element in the code with all possible outcomes
- A decision element in a program may be:
  - If-then
  - Case
  - Loop

# Branch Coverage /1

AccClient(83, female)->accept

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

true

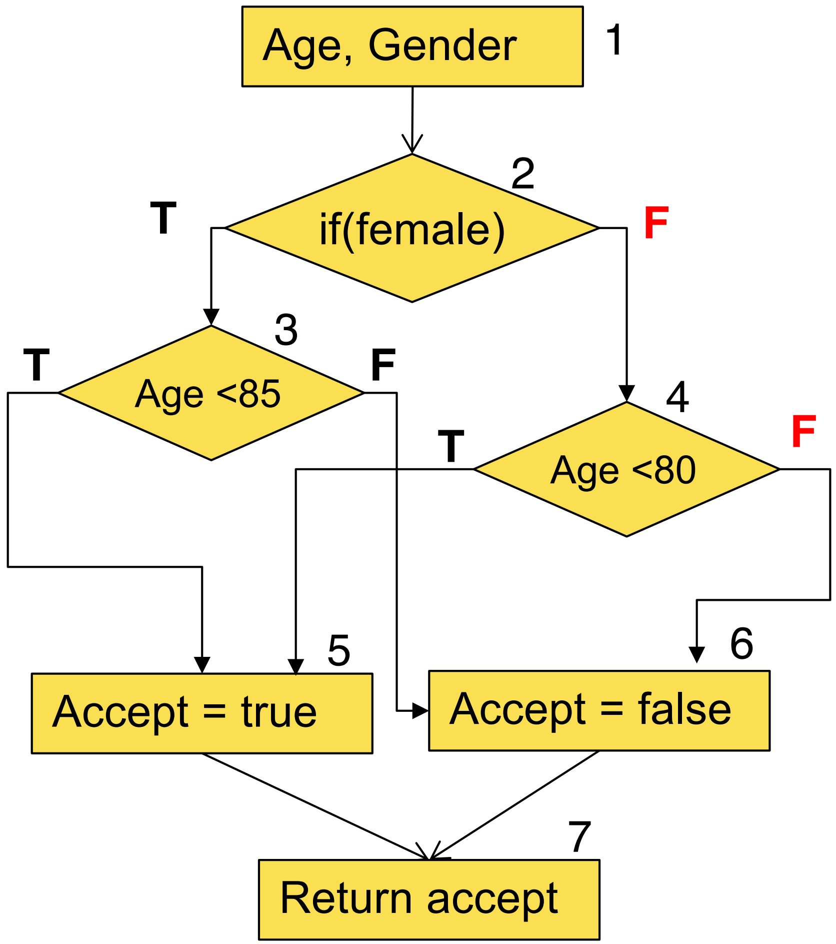


# Branch Coverage /2

AccClient(83, male)
->reject

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
    else
     accept := age < 80;
return accept
```

false



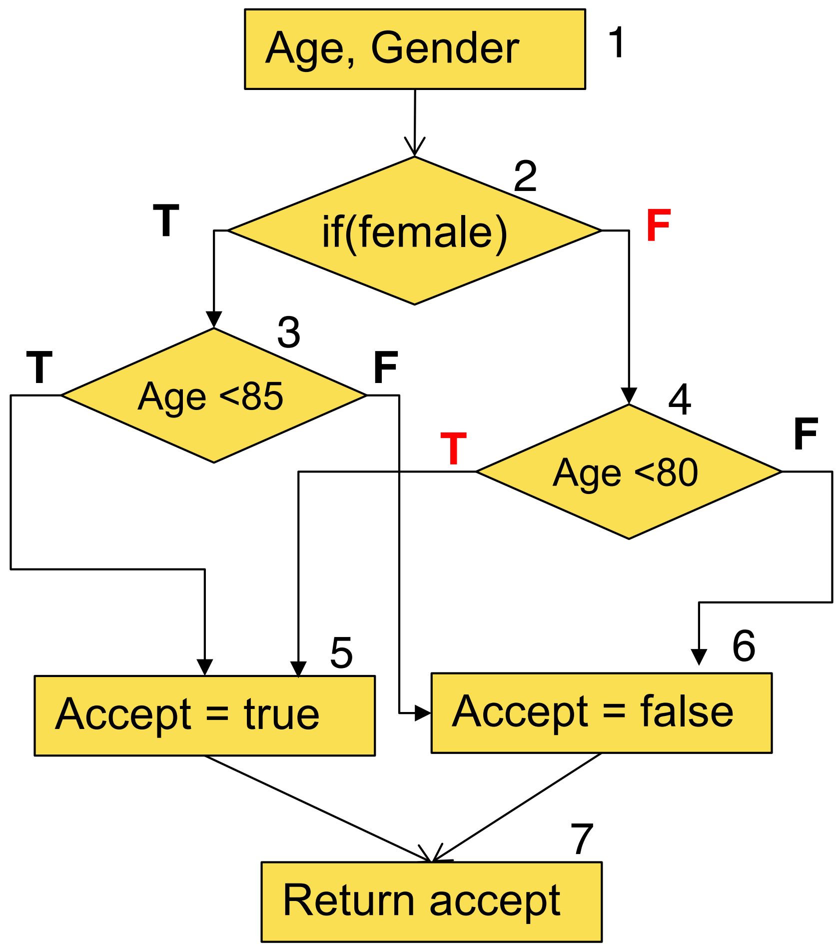
# Branch Coverage /3

AccClient(78, male)->accept

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

true

true



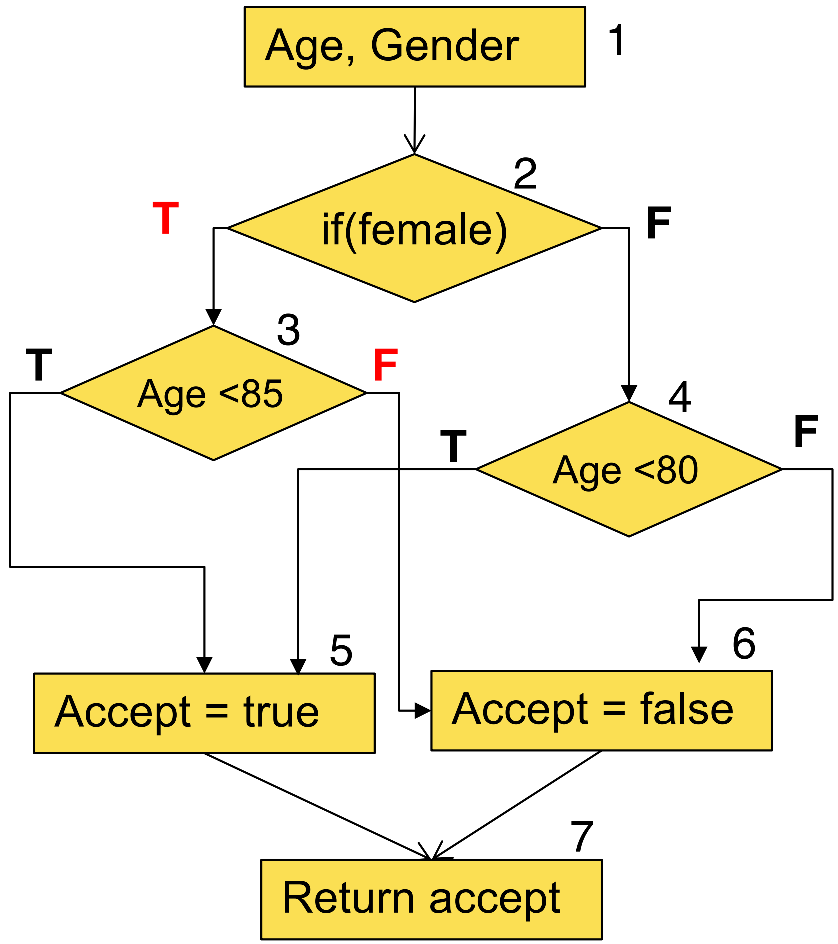
# Branch Coverage /4

AccClient(88, female) ->reject

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

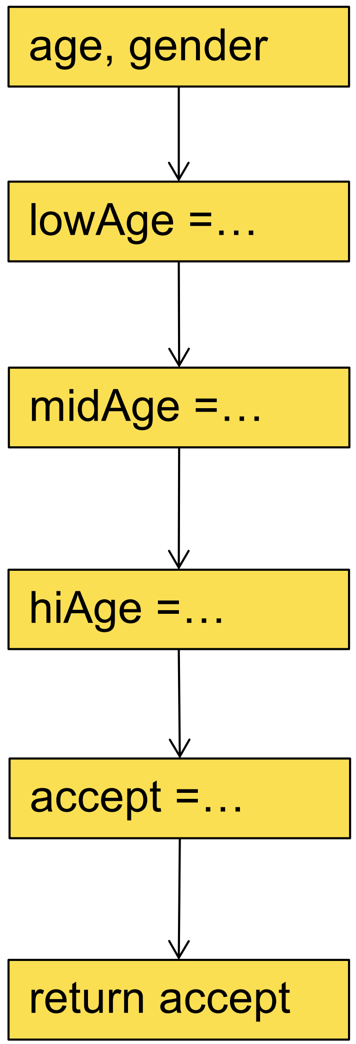
false

false



# Predicate Coverage

```
bool AccClient(agetype age;
  gndrtype gender)
bool loAge, midAge, hiAge
  lowAge := age < 80
  midAge := age>=80 and age<85
  hiAge := age >= 85
  accept := lowAge or
   (gender=female and midAge)
return accept
```



# Advanced Condition Coverage

- Condition/Decision Coverage (C/DC)
  - as DC plus: every condition in each decision is tested in each possible outcome
- Modified Condition/Decision coverage (MC/DC)
  - as above plus, every condition shown to independently affect a decision outcome (by varying that condition only)
  - a condition independently affects a decision when, by flipping that condition and holding all the others fixed, the decision changes
  - this criterion was created at Boeing and is required for aviation software according to RCTA/DO-178B
- Multiple-Condition Coverage (M-CC)
  - all possible combinations of conditions within each decision taken

# Independent Path Coverage (Table 3.3)

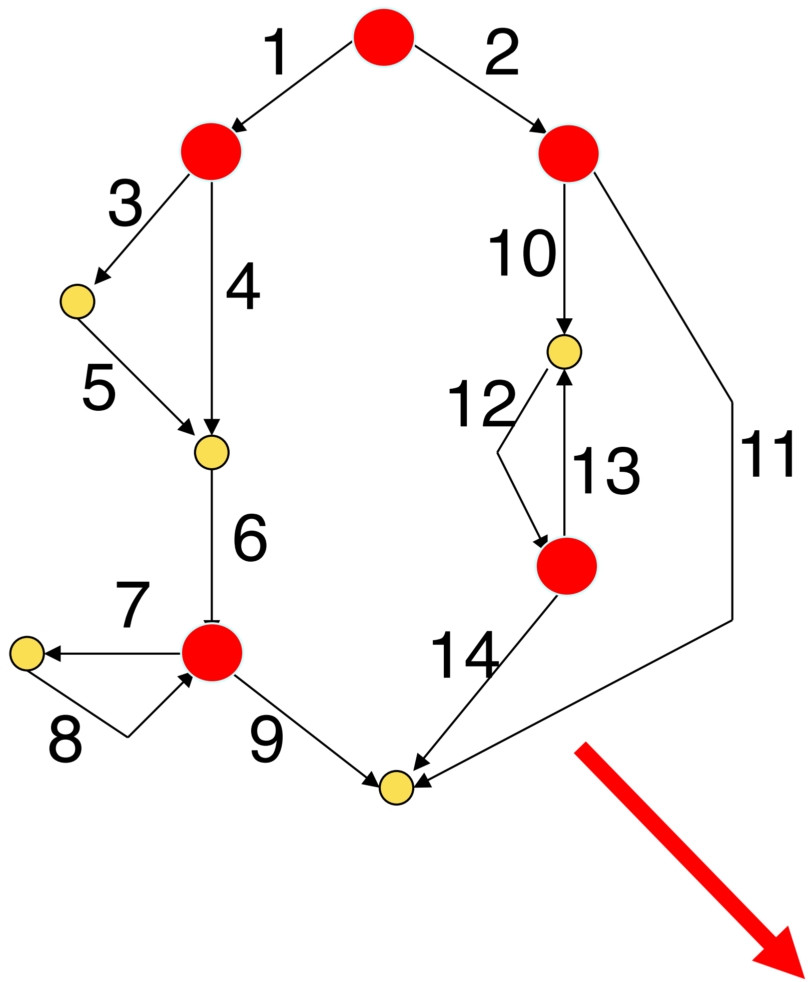
- McCabe cyclomatic complexity estimates number of test cases needed
- The number of independent paths needed to cover all paths at least once in a program
  - Visualize by drawing a flow graph
  - CC = #(edges) – #(nodes) + 2
  - CC = #(decisions) + 1

while-loop

if-then-else

case-of

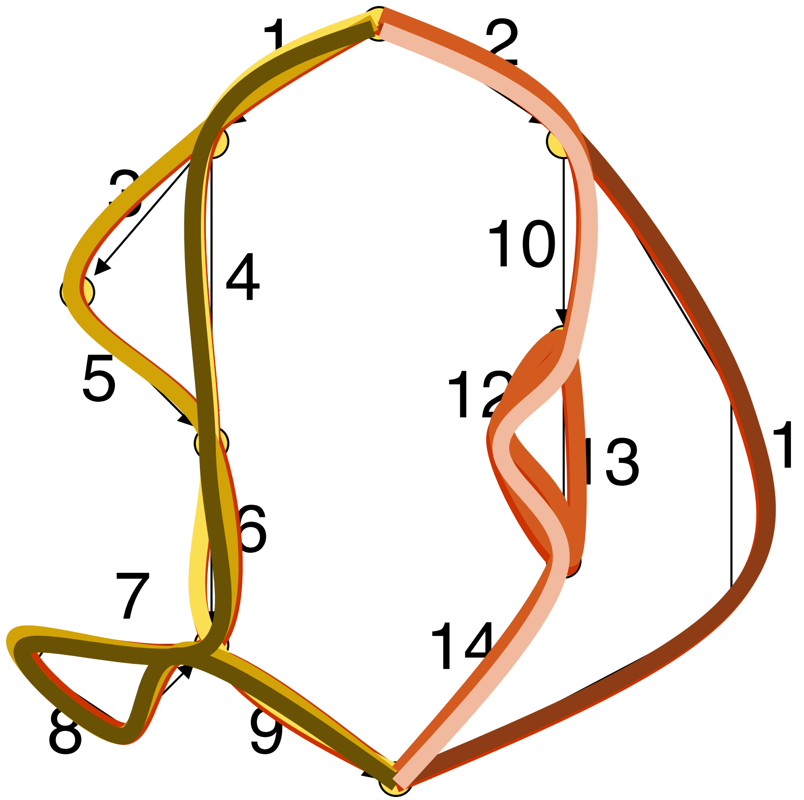
# Independent Paths Coverage – Example




- Independent Paths Coverage
  - Requires that a minimum set of linearly independent paths through the program flow-graph be executed
- This test strategy is the rationale for McCabe's cyclomatic number (McCabe 1976) …
  - … which is equal to the number of test cases required to satisfy the strategy.

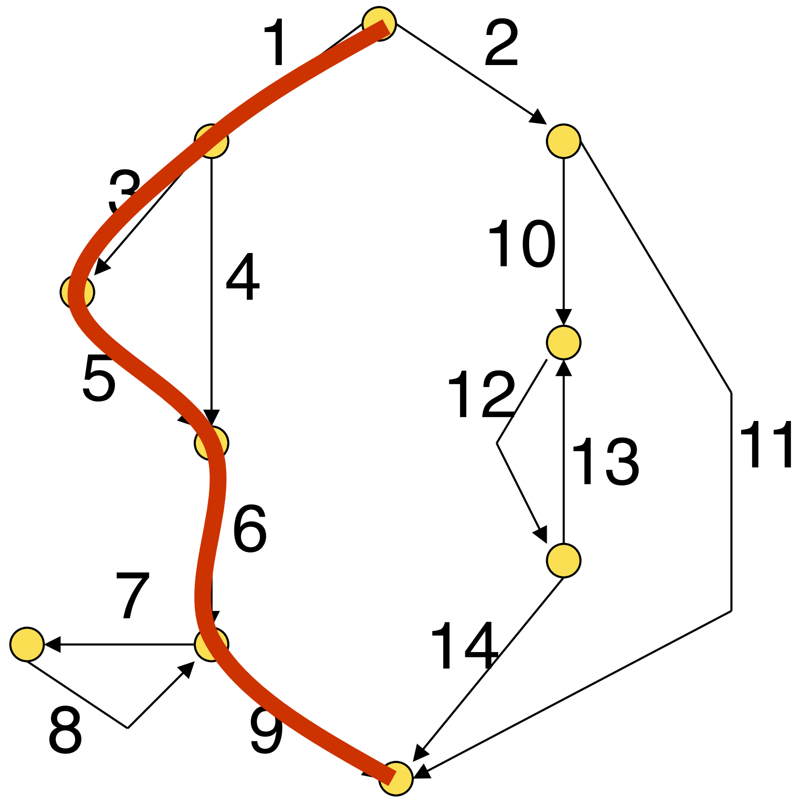
**Cyclomatic Complexity = 5 + 1 = 6**
**= 14 – 10 + 2**

# Independent Paths Coverage – Example




- Edges: 1-2-3-4-5-6-7-8-9-10-11-12-13-14
- Path1: 1-0-0-1-0-1-0-0-1-0---0---0---0---0
- Path2: 1-0-1-0-1-1-1-1-1-0---0---0---0---0
- Path3: 1-0-0-1-0-1-1-1-1-0---0---0---0---0
- Path4: 0-1-0-0-0-0-0-0-0-1---0---1---0---1
- Path5: 0-1-0-0-0-0-0-0-0-1---0---1---1---1
- Path6: 0-1-0-0-0-0-0-0-0---1---0---0---1

# Independent Paths Coverage – Example




- Edges: 1-2-3-4-5-6-7-8-9-10-11-12-13-14
- Why no need to cover Path7 below ???
- Path7: 1-0-1-0-1-1-0-0-1-0---0---0---0---0

- Path1: 1-0-0-1-0-1-0-0-1-0---0---0---0---0
- Path2: 1-0-1-0-1-1-1-1-1-0---0---0---0---0

- Path3: 1-0-0-1-0-1-1-1-1-0---0---0---0---0

# MCC – a word of warning

MCC ≠ path coverage

Use MCC as an approximation

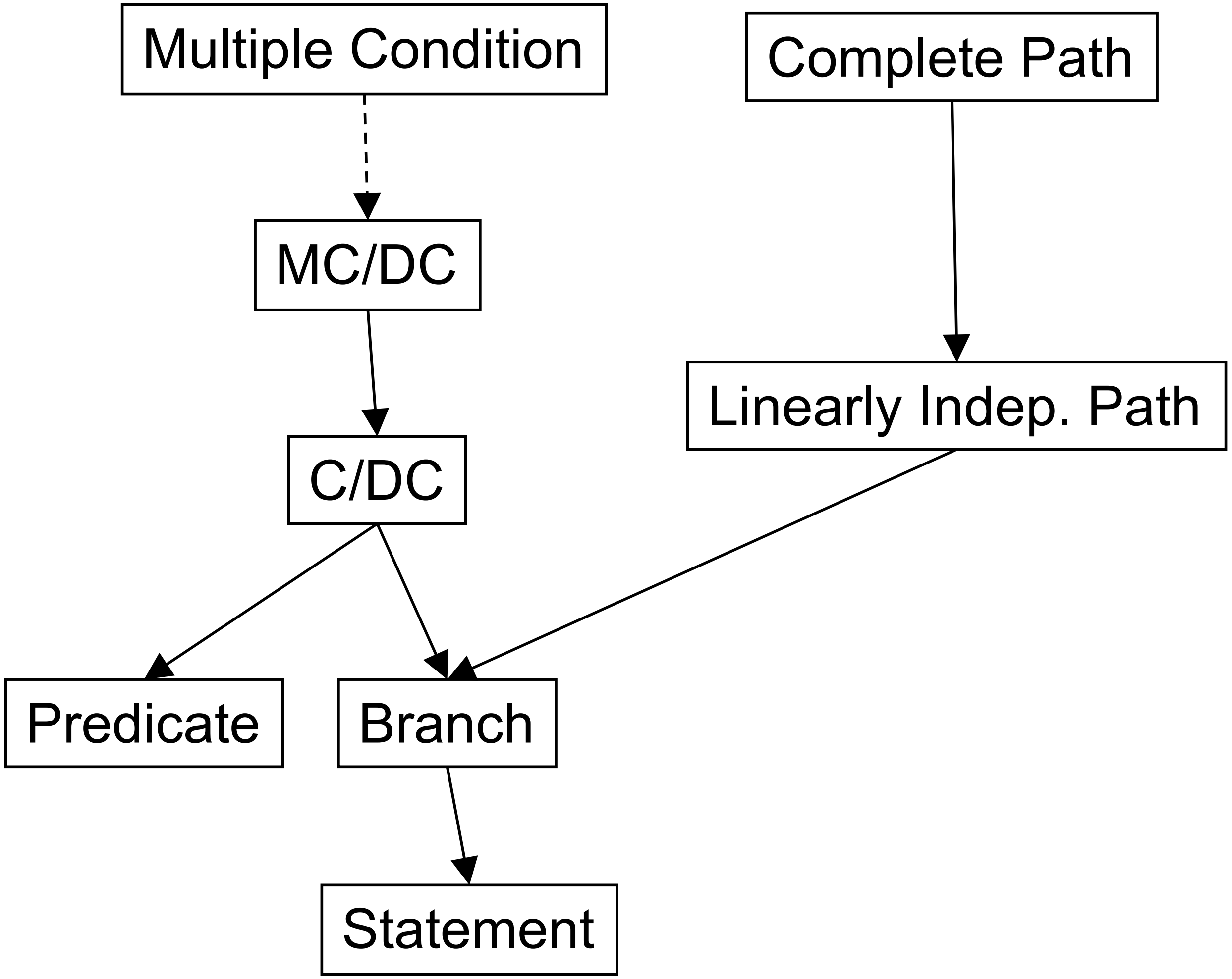
# How to find Test Cases?

- Required outcome at each predicate node contained in a path
- Consider all requirements together
- Guess a value that will satisfy these requirements

- Only feasible for small tasks. For real systems guidance by e.g. symbolic execution.

# Control-Flow Coverage Relationships

- *Subsumption*:
  a criterion C1 subsumes another criterion C2, if any test set {T} that satisfies C1 also satisfies C2



# Data Flow Testing (Chapter 5)

- Identifies paths in the program that go from the **assignment** of a value to a variable to the **use** of such variable, to make sure that the variable is properly used.

X:=14; ….. Y:= X-3;

# Data Flow Testing – Definitions

- **Def** – assigned or changed
- **Undef, Kill** – unassigned
- **Uses** – utilized (not changed)
  - **C-use** (Computation) e.g. right-hand side of an assignment, an index of an array, parameter of a function.
  - **P-use** (Predicate) branching the execution flow, e.g. in an if statement, while statement, for statement.
- Example: All **def-use paths** (DU) requires that each DU chain is covered at least once

# Data Flow Testing – Example

```
[1] bool AccClient(agetype
        age; gndrtype gender)
[2] bool accept
[3]    if(gender=female)
[4]      accept := age < 85;
[5]     else
[6]      accept := age < 80;
[7] return accept
```

Considering `age`, there are two DU paths:

(a)[1]-[4]

(b)[1]-[6]

Test case conditions:

AccClient(*, female)-> *

AccClient(*, male)-> *

# Data Flow Testing – Example

```
[1] bool AccClient(agetype
         age; gndrtype gender)
[2] bool accept
[3]    if(gender=female)
[4]      accept := age < 85;
[5]     else
[6]      accept := age < 80;
[7] return accept
```

Considering `gender`,
there is one DU path:

(a) [1]-[3]

Test case conditions:

AccClient(*, *)-> *

# Data Flow Testing – Example

```
bool AccClient(agetype
        age; gndrtype gender)
bool accept
    if(gender=female)
        accept := age < 85;
     else
        accept := age < 80;
return accept
```

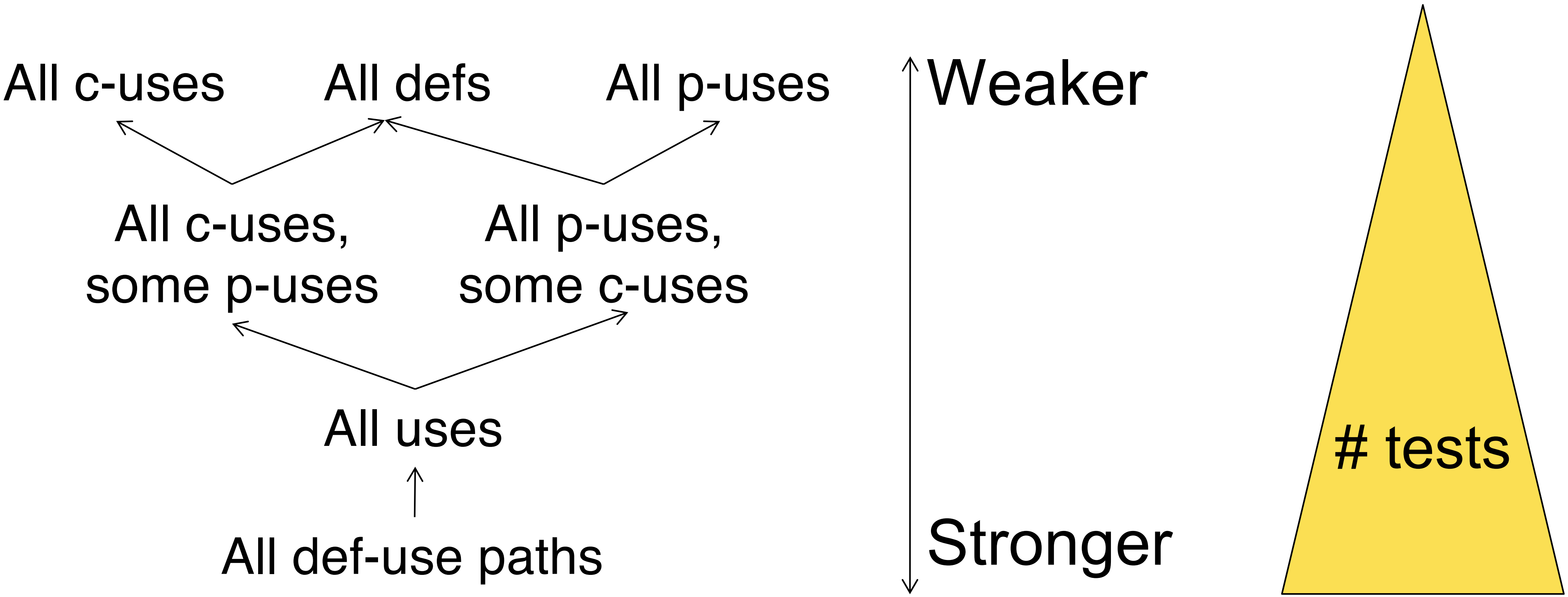
Combined for both variables:

AccClient(*, female)-> *

AccClient(*, male)-> *

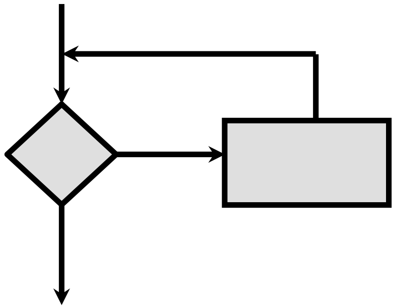
AccClient(*, *)-> *

# Data Flow Criteria

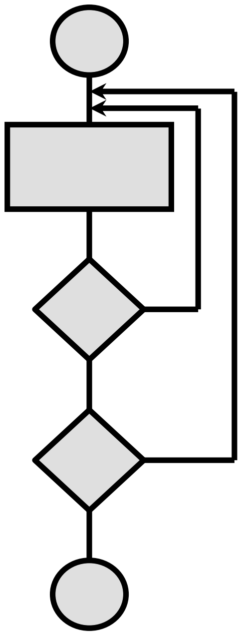


# Loop Testing

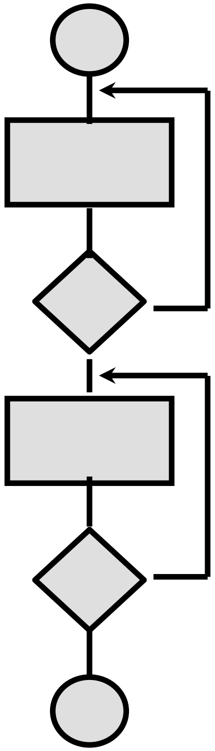
Types of loops [Beizer 1990]



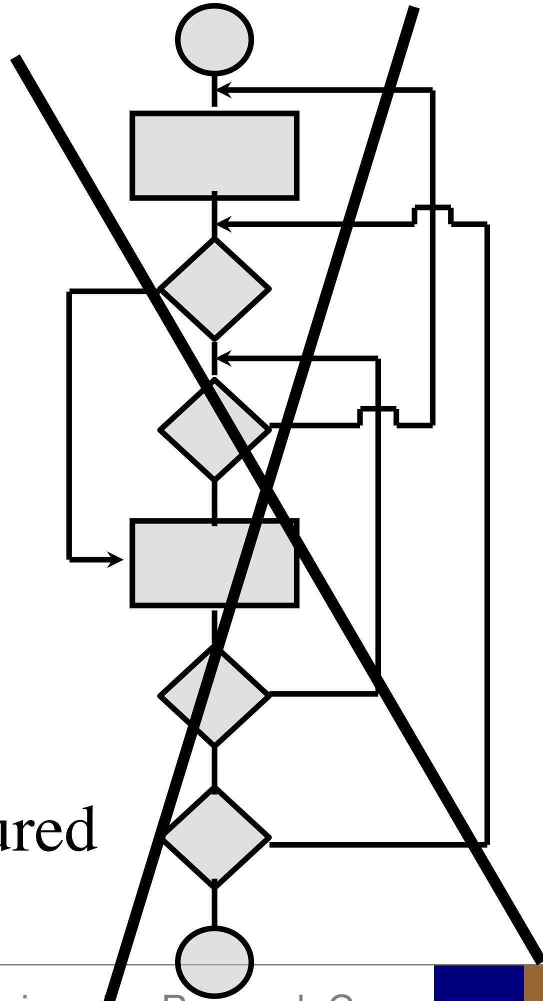
simple loop



nested loops



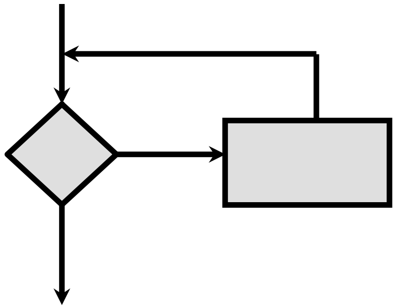
concatenated loops



unstructured loops

# Loop Testing: Simple Loops

## Heuristics



Minimum conditions - simple loops

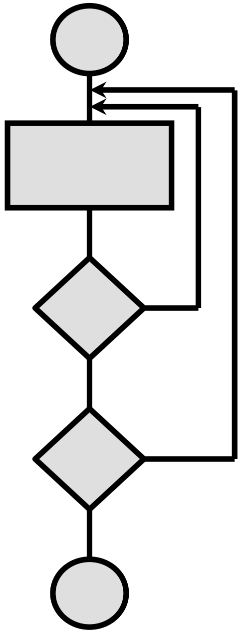
1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

# Nested Loops

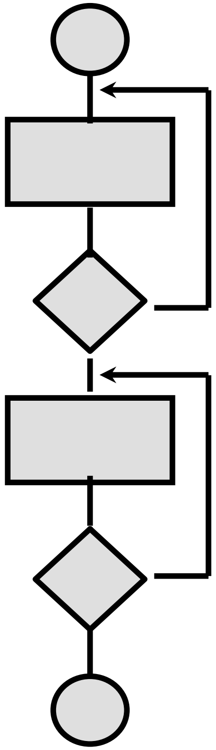
## Heuristics

- 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

## Heuristics



- If loop counters are independent:
  - Same strategies as simple loops
- If loop counters depend on each other:
  - Same strategies as nested loops

# How can I know if a test case is good?

# Mutation Testing, Section 3.5

**Terminology**

- **Mutant** – new version of the program with a small deviation (=fault) from the original version
- **Killed** mutant – version detected by the test set
- **Live** mutant – version *not* detected by the test set

# Mutation Testing

**A method for evaluation of test suite effectiveness – not for designing test cases!**

1. Take a program and test data generated for that program
2. Create a number of *similar* programs (mutants), each differing from the original in a small way
3. The original test data are then run through the *mutants*
4. If tests detect all changes in mutants, then the mutants are dead and the test suite adequate

   Otherwise: Create more test cases and iterate 2-4 until a sufficiently high number of mutants is killed

# Example Mutation Operations

- Change relational operator (<,>, …)
- Change logical operator (II, &, …)
- Change arithmetic operator (*, +, -,…)
- Change constant name / value
- Change variable name / initialisation
- Change (or even delete) statement
- …

# Example Mutants

```
if (a || b)
    c = a + b;
else
    c = 0;
```

⇒

```
if (a && b)
    c = a + b;
else
    c = 0;
```

---

```
if (a || b)
    c = a + b;
else
    c = 0;
```

⇒

```
if (a || b)
    c = a * b;
else
    c = 0;
```

# Types of Mutants

- Stillborn mutants: Syntactically incorrect – killed by compiler, e.g., x = a ++ b
- Trivial mutants: Killed by almost any test case
- Equivalent mutant: Always acts in the same behavior as the original program, e.g., x = a + b and x = a – (–b)

- None of the above are interesting from a mutation testing perspective
- Those mutants are interesting which behave differently than the original program, and we do not (yet) have test cases to identify them (i.e., to cover those specific changes)

# Equivalent Mutants

```
if (a == 2 && b == 2)
    c = a + b;
else
    c = 0;
```

→

```
if (a == 2 && b == 2)
    c = a * b;
else
    c = 0;
```

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

→

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

# Program Example

```
nbrs = new int[range]
public int max(int[] a) {
  int imax := 0;
  for (int i = 1; i < range; i++)
     if a[i] > a[imax]
        imax:= i;
return imax;
}
```

| | a[0] | a[1] | a[2] | imax |
|---|---|---|---|---|
| TC1 | 1 | 2 | 3 | 2 |
| TC2 | 1 | 3 | 2 | 1 |
| TC3 | 3 | 1 | 2 | 0 |

# Relational Operator Mutant

```
nbrs = new int[range]

public int max(int[] a) {

    int imax := 0;

    for (int i = 1; i < range; i++)

        if a[i] >= a[imax]

            imax:= i;

return imax;

}
```

| | a[0] | a[1] | a[2] | imax |
|---|---|---|---|---|
| TC1 | 1 | 2 | 3 | 2 |
| TC2 | 1 | 3 | 2 | 1 |
| TC3 | 3 | 1 | 2 | 0 |

Need a test case with two identical max entries in a[.] to be detected

# Variable Operator Mutant

```
nbrs = new int[range]

public int max(int[] a) {

    int imax := 0;

    for (int i = 1; i < range; i++)

        if i > a[imax]

            imax:= i;

return imax;

}
```

| | a[0] | a[1] | a[2] | imax |
|---|---|---|---|---|
| TC1 | 1 | 2 | 3 | 2 |
| TC2 | 1 | 3 | 2 | 0 |
| TC3 | 3 | 1 | 2 | 0 |

# Variable Operator Mutant

```
nbrs = new int[range]

public int max(int[] a) {

    int imax := 0;

    for (int i = 0; i < range; i++)

        if a[i] > a[imax]

            imax:= i;

return imax;

}
```

| | a[0] | a[1] | a[2] | imax |
|---|---|---|---|---|
| TC1 | 1 | 2 | 3 | 2 |
| TC2 | 1 | 3 | 2 | 1 |
| TC3 | 3 | 1 | 2 | 0 |

Need a test case counting loops to be detected

# This Week / Next Week

- This week
  - Read project description thoroughly, decide subject
- Next week
  - Lab 1 – White-box testing (Thu)

# Recommended exercises

- Chapter 4
  - 1, 2, 3, 4, 7, 8
- Chapter 5
  - 1, 2, 4, 5