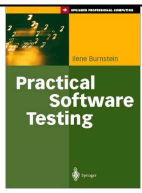
Software Testing

ETS 200

http://cs.lth.se/ets200



Chapter 5

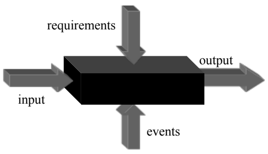
Prof. Per Runeson

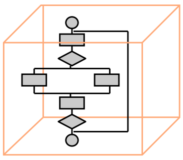
Lecture

- Chapter 5
  – White-box testing techniques (Lab 2)

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Testing Strategies



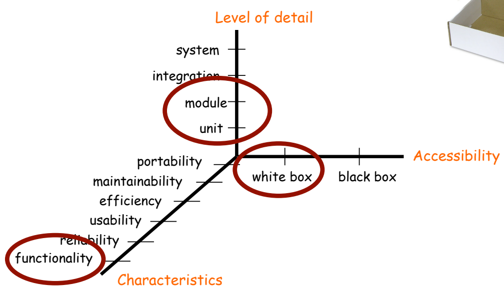


Black Box Testing

White Box Testing

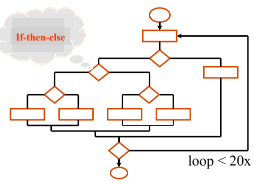
Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Types of Testing



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

White-Box Testing (Ch 5)
Exhaustive Testing



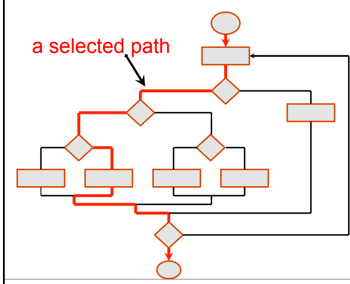
There are many possible paths!

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

Selective Testing

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Selective Testing



Selective:

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

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Code Coverage

- Measures the extent to which certain code items related to a defined test adequacy criterion have been executed (covered) by running a set of test cases (= test suites)
- Goal: Define test suites such that they cover as many (disjoint) code items as possible
- (Note, other types of coverage, such as test case coverage, requirements coverage, usage 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.

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Code Coverage Measure – Example

- Statement Coverage (CV_s)
  - Portion of the statements tested by at least one test case.

$$CV_s = \left(\frac{S_t}{S_p} \right) \times 100\%$$

S_t : number of statements tested

S_p : total number of statements

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Code Coverage Analysis

- Code coverage analysis is the process of:
  - Finding areas of a program not exercised by a set of test cases
  - Creating additional test cases to increase coverage
  - Determining a quantitative measure of code coverage, which is (believed to be) a predictor of code quality
- Code coverage analyzers automate this process
- Additional aspect of code coverage analysis:
  - Identifying redundant test cases that do not increase coverage
  - Identifying "dead code"

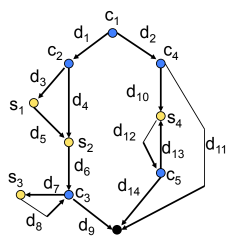
Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Main Classes of Test Adequacy Criteria

- Control Flow Criteria:
  - Statement, decision (branch), condition, and path coverage are examples of control flow criteria
  - They rely solely on syntactic characteristics of the program (ignoring the semantics of the program computation)
- Data Flow Criteria:
  - Require the execution of path segments that connect parts of the code that are intimately connected by the flow of data

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

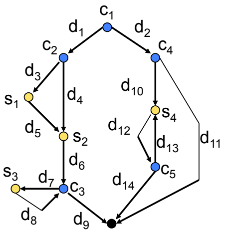
Overview of Control Flow Criteria




```
If c1 then
   if c2 then s1
   s2
   while c3 do s3
else
   if c4 then
      repeat s4 until c5
   endif
endif
```

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

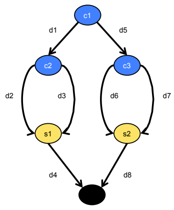
Overview of Control Flow Criteria



- **Statement Coverage**
- **Decision (or Branch) Coverage**
- **Condition Coverage**
- Condition/Decision Coverage
- Multiple Condition Coverage
- Modified Condition Decision Coverage (MC/DC)
- Simple Paths
- **Linearly Independent Paths**
- Linear Code Sequence and Jump (LCSAJ)
- Visit-Each Loop
- All Paths
- …

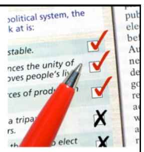
Life Insurance Example

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



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Statement Coverage



- Execute each statement at least once
- Tools are used to monitor execution

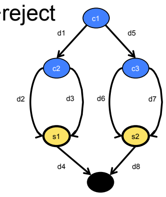
- A possible concern may be:
  – Dead code

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Statement Coverage

AccClient(83, female)->accept
AccClient(83, male) ->reject

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



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Condition Coverage

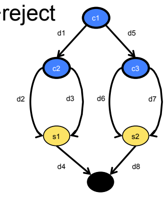
- Test all possible conditions in a program
- A condition in a program may contain:
  – Boolean operators and variables
  – Relational operators
  – Arithmetic expressions
  – ….

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

(Simple) Condition Coverage

AccClient(83, female)->accept
AccClient(83, male) ->reject

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



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

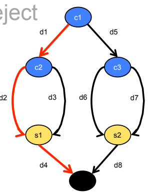
Decision (Branch) Coverage /1

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

true

AccClient(83, female)->accept

AccClient(83, male) ->reject



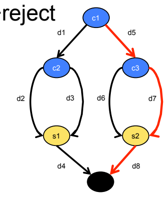
Decision (Branch) Coverage /2

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

false

AccClient(83, female)->accept

AccClient(83, male) ->reject



Decision (Branch) Coverage /3

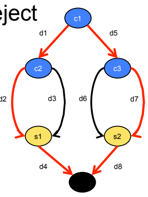
```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
    accept := age < 85;
   else
    accept := age < 80;
return accept
```

true

false

AccClient(83, female)->accept

AccClient(83, male) ->reject



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

CC, DC, C/DC, MC/DC, M-CC Examples

```
If (A<10 and B>250) then …
```

Condition:
(TT) A = 2; B = 300 (True)

Decision:
(TT) A = 2; B = 300 (True)
(FT) A = 12; B = 300 (False)

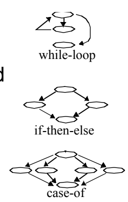
Decision/Condition:
(TT) A = 2; B = 300 (True)
(FF) A = 12; B = 200 (False)

Multiple Condition:
(TT) A = 2; B = 300 (True)
(FT) A = 12; B = 300 (False)
(TF) A = 2; B = 200 (False)
(FF) A = 12; B = 200 (False)

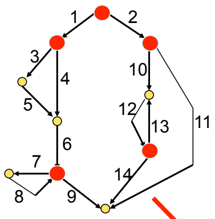
Modified Condition/Decision:
(TT) A = 2; B = 300 (True)
(FT) A = 12; B = 300 (False)
(TF) A = 2; B = 200 (False)

Independent Path Coverage

- 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



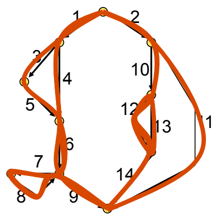
Independent Paths Coverage – Example



- 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.

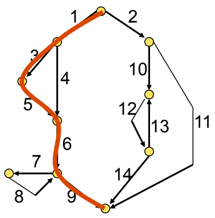
Cyclomatic Complexity = 5 + 1 = 6
= 14 – 10 + 2

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

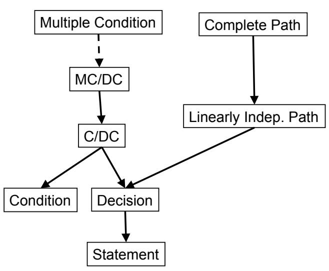
How to find Test Cases?

- 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 exection.

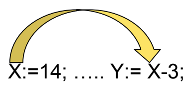
Control-Flow Coverage Relationships

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



Data Flow Testing

- 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.



Data Flow Testing – Definitions

- **Def** – assigned or changed
- **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)-> *

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

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(*, *)-> *

Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

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
```

Combined for both variables:

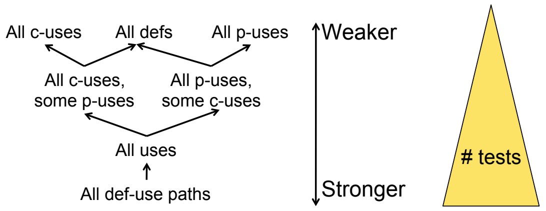
AccClient(*, female)-> *

AccClient(*, male)-> *

AccClient(*, *)-> *

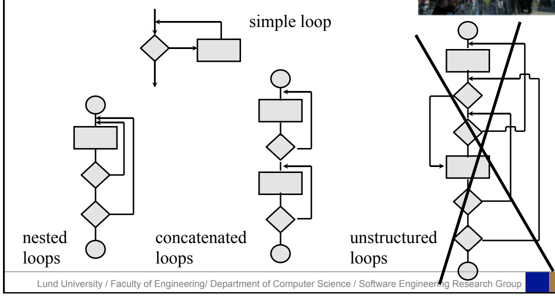
Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Data Flow Criteria



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

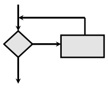
Loop Testing [Beizer 1990]



Lund University / Faculty of Engineering/ Department of Computer Science / Software Engineering Research Group

Loop Testing: Simple Loops

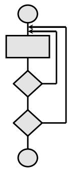
Minimum conditions - simple loops



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

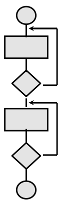
where n is the maximum number of allowable passes

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



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

Fault-Based Testing (Mutation Testing)

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

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

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

Why is white-box testing not enough?

- Missing features
  - Missing code
- Different states of the software
  - Infinite amount of different paths through the software
  - Different paths can reveal different defects
- Variations of perspectives
  - Control flow/Data flow
  - Input/Output behaviour
- Test data generation
- Quality attributes
  - Performance
  - Robustness
  - Reliability
  - Usability
  - …

This Week

- Project
  - Find/read literature
- Lab 1
  - Thursday & Friday: Black-box testing

Next Week

- Project
  - Report outline (Feb 4)
- Lab 2
  - Thursday & Friday: White-box testing
  - Report of Lab 1

Recommended exercises

- Chapter 5
  - 2, 5, 6, 9, 10, 11, 14