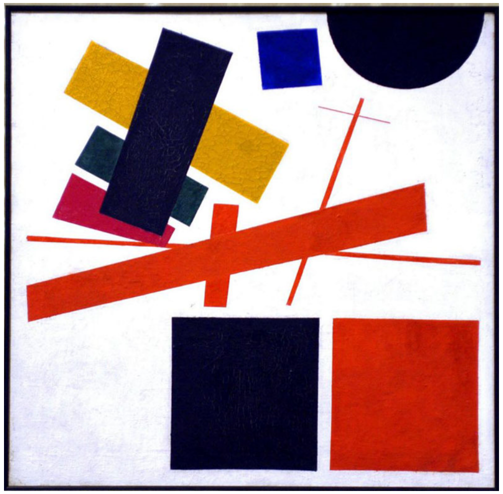
CPEN 422
Software Testing and Analysis



Test Adequacy and Coverage

10.1

Systematic Testing

Functional testing:

- *Test cases come from requirements / user stories.*

Structural testing:

- *Inspect the code / coverage criteria to see if you missed cases*

Model-based testing:

- *Use models of aspects of the system and its behavior to guide test case generation*

Testing Thoroughly

“Each software system should be thoroughly tested”.

- What does thorough mean?
- How can we meaure *test adequacy?*
- When can we stop testing?

Adequacy Criteria as Design Rules

- Many design disciplines employ design rules
  - E.g.: “traces (on a chip, on a circuit board) must be at least ___ wide and separated by at least ___”
  - “Interstate highways must **not** have a grade greater than 6% without special review and approval”

- Design rules do not guarantee good designs
  - Good design depends on talented, creative, disciplined designers; design rules help them avoid or spot flaws
- Test design is no different

Practical (in)Adequacy Criteria

- Criteria that identify **inadequacies** in test suites. Examples
  - If no test in the test suite executes a particular program statement, the test suite is *inadequate* to guard against faults in that particular statement.

- If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about **improving** the test suite.

- If a test suite satisfies all the obligations by all the criteria, we **do not know** definitively that it is an effective test suite, but we have some evidence of its thoroughness.

9.2

Terminology

- **Test case**: a set of inputs, execution conditions, and a pass/fail criterion.
- **Test case specification**: a requirement to be satisfied by one or more test cases.
- **Test obligation**: a *partial* test case specification requiring some property deemed important to thorough testing
- **Test suite**: a set of test cases.
- **Test or test execution**: the activity of executing test cases and evaluating their results.
- **Adequacy criterion**: a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair.
- **Test coverage:** percentage of test obligations met for a given adequacy criterion.

Where do test obligations come from?

- Functional (black box, specification-based): from software specifications
  - Example: If spec requires robust recovery from power failure, test obligations should include simulated power failure

- Structural (white or glass box): from code
  - Example: Traverse each program loop one or more times.

- Model-based: from model of system
  - Models used in specification or design, or derived from code
  - Example: Exercise all transitions in communication protocol model

- Fault-based: from hypothesized faults (common bugs)
  - Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs

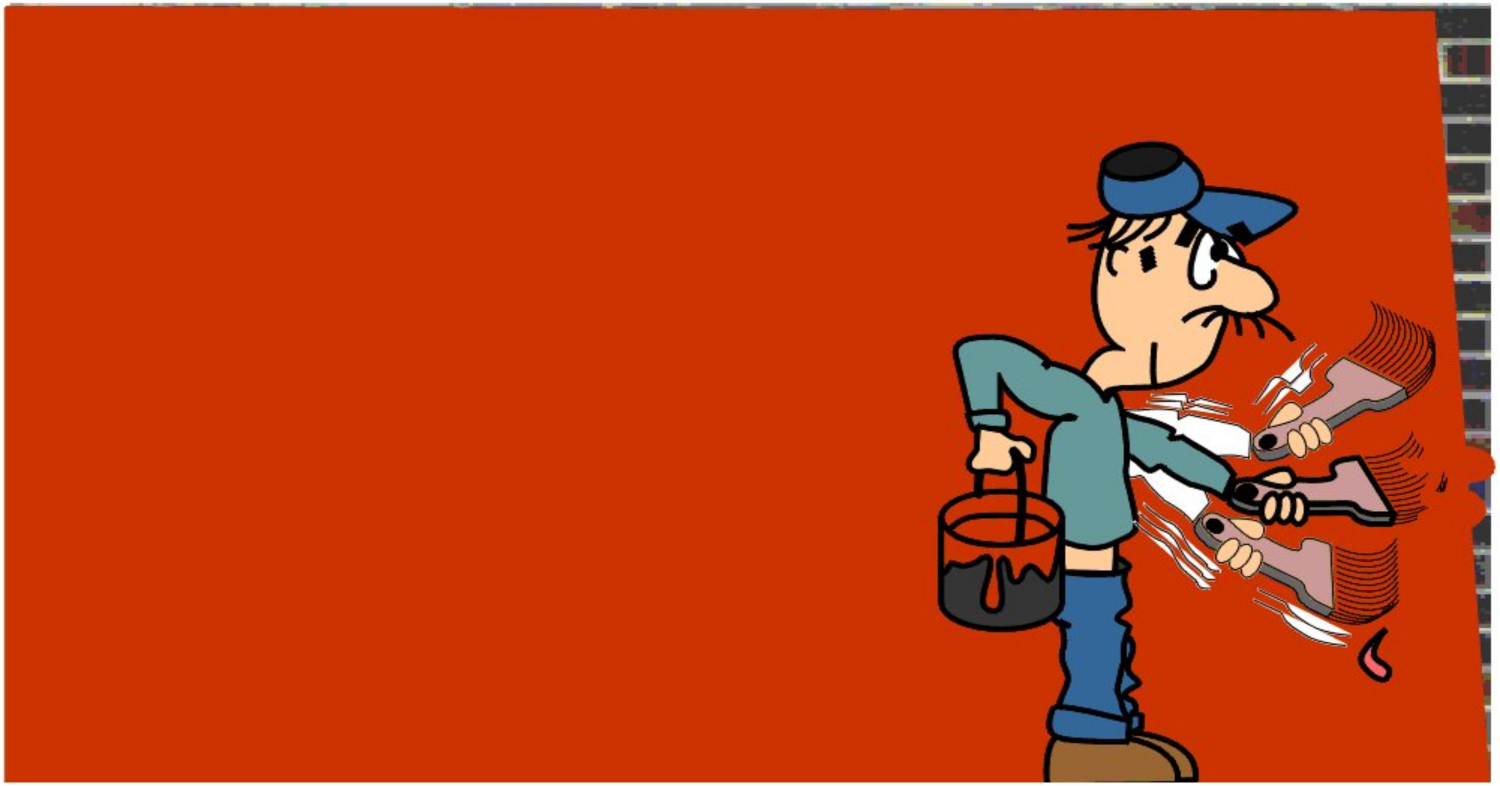
Adequacy criteria

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if
  - all the tests succeed (pass)
  - every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
- Example:

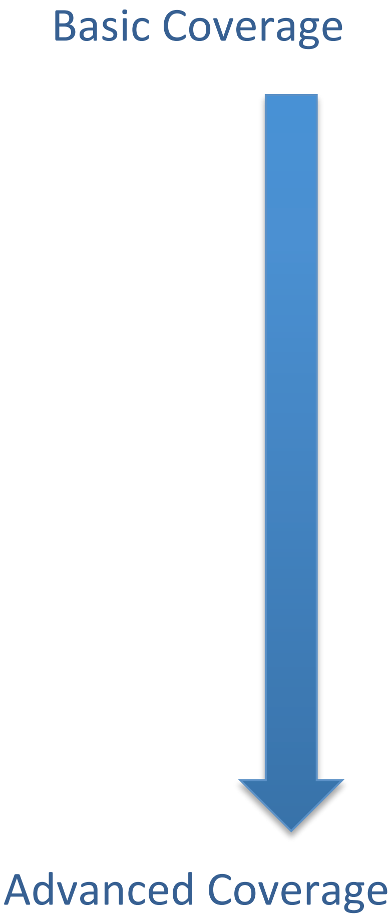
the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was "pass"

Code Coverage

Introduced by Miller and Maloney in 1963



Coverage Criteria



Basic Coverage

Advanced Coverage

- Line coverage
- Statement
- Function/Method coverage
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- Path coverage
- Loop coverage
- Mutation adequacy
- …

Line Coverage

- Percentage of source code lines executed by test cases.
  - For developer easiest to work with
  - Precise percentage depends on layout?
    - var x = 10; if (z++ < x) y = x+z;
  - Requires mapping back from binary?
- In practice, coverage not based on lines, but on *control flow graph*

5.3

The Control Flow Graph

- Node:
  - Regions of source code (basic blocks)
  - Basic block = maximal program region with **single entry** and **single exit** point
- Directed edges:
  - *possibility* that program execution proceeds from the end of one region directly to the beginning of another
- Intra-procedural:
  - *within one* procedure / method
  - Extra nodes: *single entry, single exit for full procedure*

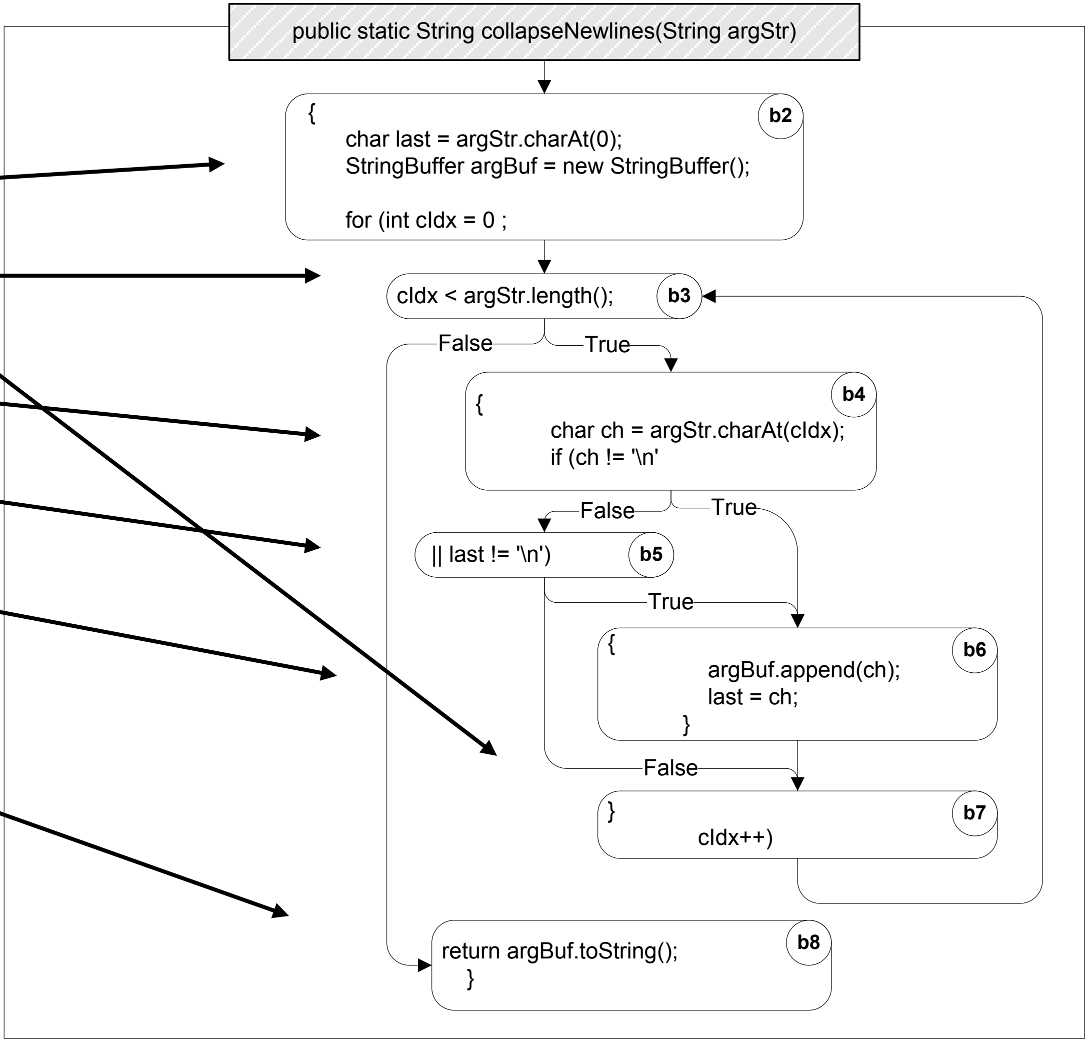
5.3

Deriving a Control Flow Graph

```
public static String collapseNewlines(String argStr)
  {
     char last = argStr.charAt(0);
     StringBuffer argBuf = new StringBuffer();

     for (int cIdx = 0; cIdx < argStr.length(); cIdx++)
     {
        char ch = argStr.charAt(cIdx);
        if (ch != '\n' || last != '\n')
        {
           argBuf.append(ch);
           last = ch;
        }
     }

     return argBuf.toString();
  }
```



public static String collapseNewlines(String argStr)
{
char last = argStr.charAt(0);
StringBuffer argBuf = new StringBuffer();
for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
{
char ch = argStr.charAt(cIdx);
if (ch != '\n'
b4
False
True
|| last != '\n')
b5
True
{
argBuf.append(ch);
last = ch;
}
b6
False
}
cIdx++)
b7
return argBuf.toString();
}
b8

Splitting multiple conditions depends on goal of analysis

CFG Abstraction Level?

- Loop conditions? (yes)
- Individual statements? (no)
- Exception handling? (no)

- *What's best depends on type of analysis to be conducted*

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
 }
}
```

- Coverage:

$$\frac{\#\ \text{executed statements}}{\#\ \text{statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
 }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
@Test
void testFoo() {
  foo(5);
}
// 100% statement coverage
```

- Coverage:

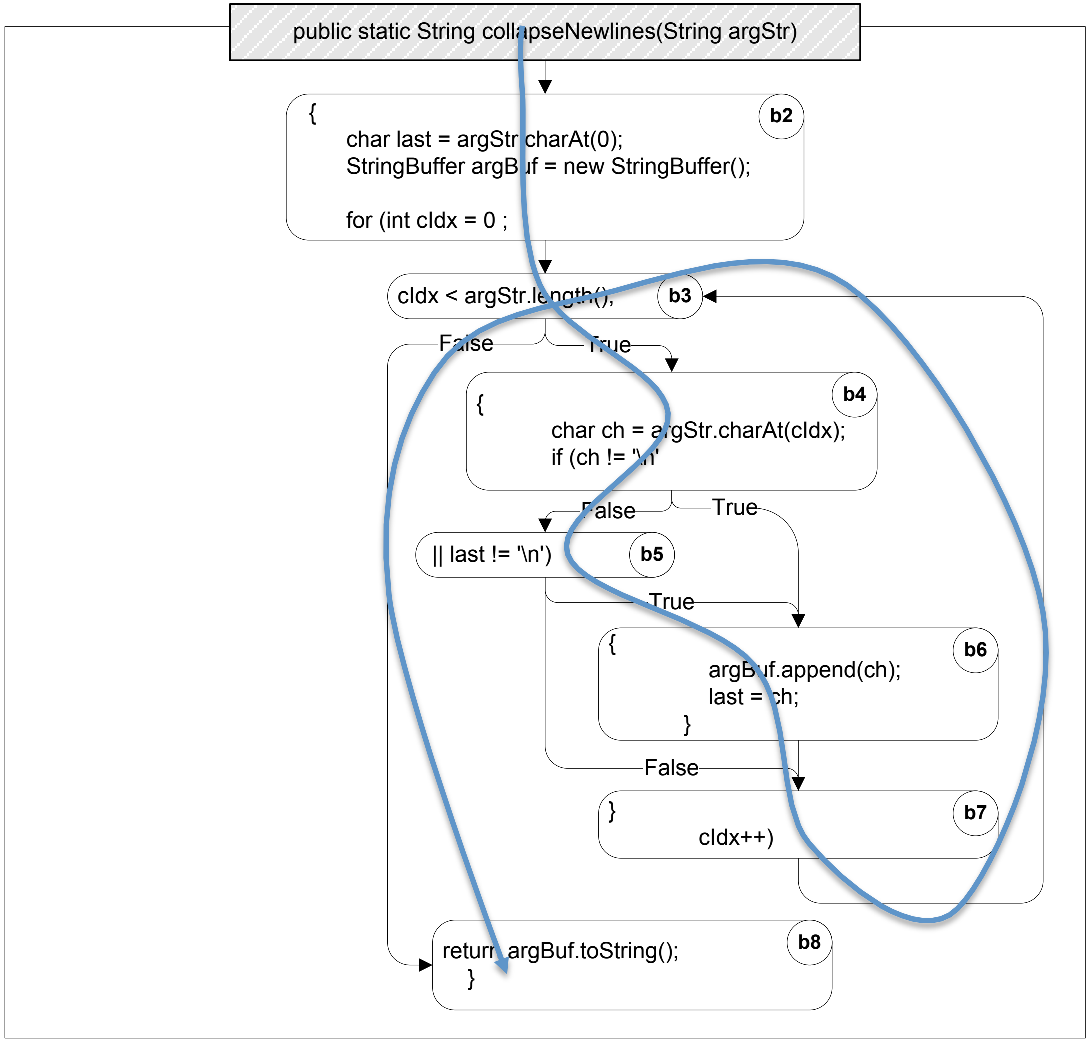
$$\frac{\#\ \text{executed statements}}{\#\ \text{statements}}$$

Control Flow Based Adequacy Criteria

12.2
12.3

- Every block / Statement?

One test case: b2,3,4,5,6,7,**3,**8
Input: “a”



public static String collapseNewlines(String argStr)

b2
```
{
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for (int cIdx = 0 ;
```

b3
```
cIdx < argStr.length();
```

False

True

b4
```
{
      char ch = argStr.charAt(cIdx);
      if (ch != '\n'
```

False

True

b5
```
|| last != '\n')
```

True

b6
```
{
      argBuf.append(ch);
      last = ch;
}
```

False

b7
```
}
      cIdx++)
```

b8
```
return argBuf.toString();
}
```

12.3

Branch Coverage

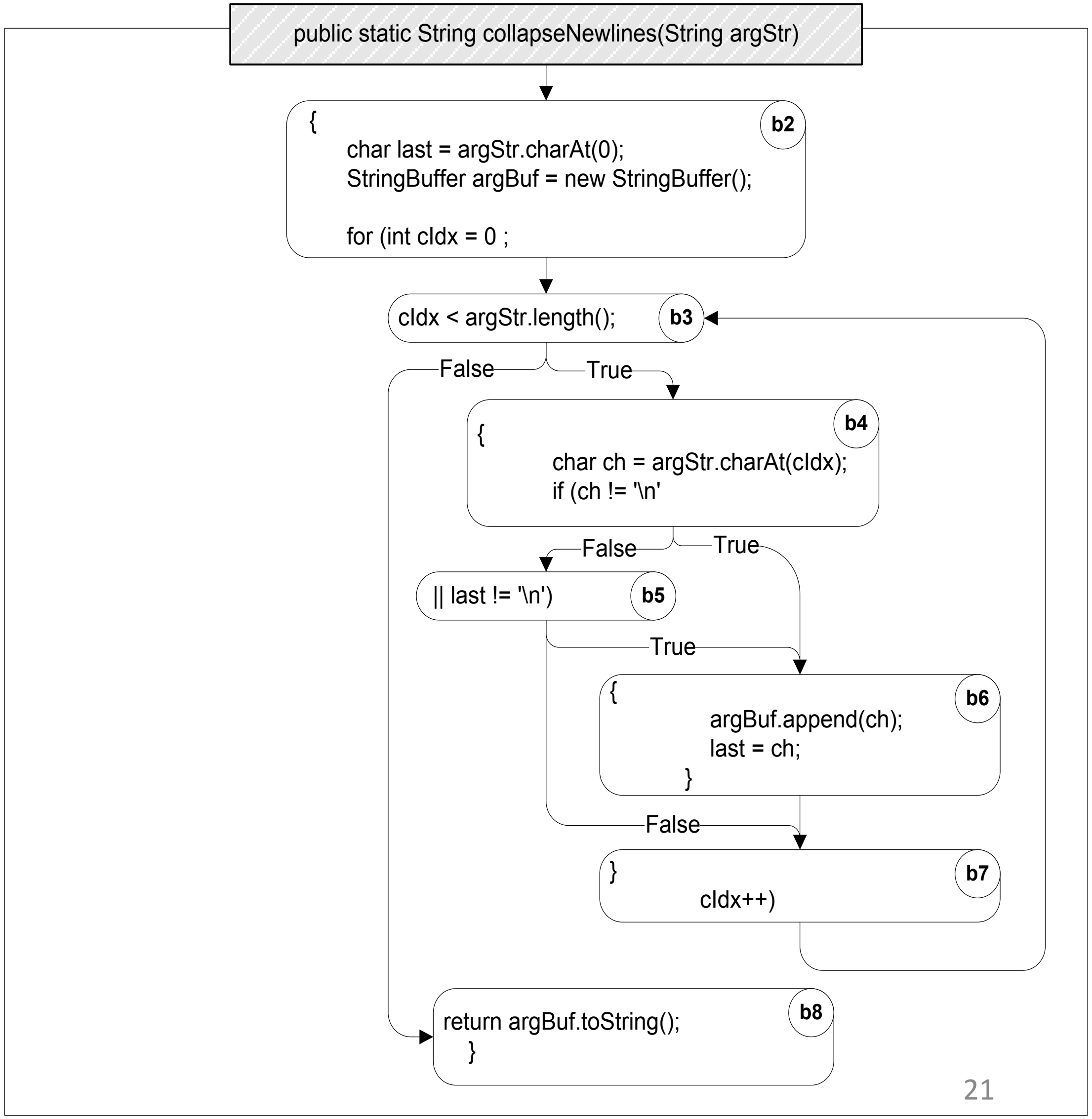
- Every path going out of node executed at least once
  - Decision-, all-edges-, coverage
  - Coverage: percentage of edges hit.

- *Each predicate must be both true and false*

12.2
12.3

Branch Coverage

- One longer input:
  - “a\n\n”
- Alternatively:
  - Block (“a”) and
  - “\n” and
  - “\n\n”



public static String collapseNewlines(String argStr)

```
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ;
```
b2

```
cIdx < argStr.length();
```
b3

```
{
        char ch = argStr.charAt(cIdx);
        if (ch != '\n'
```
b4

```
|| last != '\n')
```
b5

```
{
            argBuf.append(ch);
            last = ch;
        }
```
b6

```
}
        cIdx++)
```
b7

```
return argBuf.toString();
  }
```
b8

Condition Testing

12.4

- **Compound predicates:**
  - – (((a || b) && c) || d) && e

- Should we test the effect of *individual* conditions on the outcome?

1. *Basic* ***condition***: each cond. true, false
2. *Branch and condition*: same, + branch
3. *Compound condition*: each combination, 2^N (costly)
4. *Modified Condition / Decision Coverage* (MC/DC)

MC/DC: Modified Condition + Decision Coverage

12.4

- Basic condition + decision coverage + ...
  - *each basic condition should* ***independently*** *affect outcome of each decision*
- Requires:
  - For each basic condition C, two test cases,
  - values of all *evaluated* conditions except C are the same
  - compound condition as a whole evaluates to *true* for one and *false* for the other
  - N + 1 cases, for N conditions.

Example: Basic Condition Coverage

```
foo (A, B, C) {
   if ( (A || B) && C ) {
      /* statements*/
   }
   else {
     /* statements*/
   }
}
```

In order to ensure **Condition coverage** criteria for this example, A, B and C should be evaluated at least one time "true" and one time "false" during tests:

```
T1: foo(true, true, true)
       // A = true, B = true, C = true
T2: foo(false, false, false)
       // A = false, B = false, C = false
```

Example: Decision Coverage

```
if ( (A || B) && C ) {
   /* instructions */
}
else {
/* instructions */
}
```

In order to ensure **Decision coverage** criteria, the condition ((A or B) and C) should also be evaluated at least one time to "true" and one time to "false”:

```
A = true, B = true, C = true   ---> "true"
A = false, B = false, C = false  ---> "false"
```

Example: MCDC

```
if ( (A || B) && C ) {
   /* instructions */
}
else {
/* instructions */
}
```

In order to ensure **MCDC** criteria, each *boolean variable* should be evaluated one time to "true" and one time to "false", and this with *affecting* the decision's outcome:

```
A = true  / B = false / C = true   --->  "true”
A = false / B = false / C = true   --->  "false”
A = false / B = true  / C = true   --->  "true"
A = false / B = true  / C = false  --->  "false”
```

```
(((a || b) && c) || d) && e
```

#tc	a	b	c	d	e	outcome
t1	*T*	F	T	F	T	T
t2						
t3						
t4						
t5						
t6						
t7						
t8						
t9						
t10						

12.4

(((a || b) && c) || d) && e

#tc	a	b	c	d	e	outcome	
t1	***T***	F	T	F	T	T	
t2	***F***	F	T	F	T	F	
t3	F	***T***	T	F	T	T	
t4	F	***F***	T	F	T	F	=t2
t5	T	F	***T***	F	T	T	=t1
t6	T	F	***F***	F	T	F	
t7	-	-	F	***T***	T	T	
t8	T	F	F	**F**	T	F	=t6
t9	-	-	-	T	**T**	T	=t7
t10	-	-	-	-	**F**	F	

12.4

`(((a || b) && c) || d) && e`

#tc	a	b	c	d	e	outcome
t1	***T***	F	T	F	T	T
t2	***F***	F	T	F	T	F
t3	F	***T***	T	F	T	T
t6	T	F	***F***	F	T	F
t7	-	-	F	***T***	T	T
t10	-	-	-	-	**F**	F

DO-178B/ED-12B Software Considerations in Airborne Systems and Equipment Certification

Failure Condition	Software Level	Coverage
Catastrophic	A	MC/DC
Hazardous / Severe	B	Decision Coverage
Major	C	Statement Coverage
Minor	D	
No Effect	E	

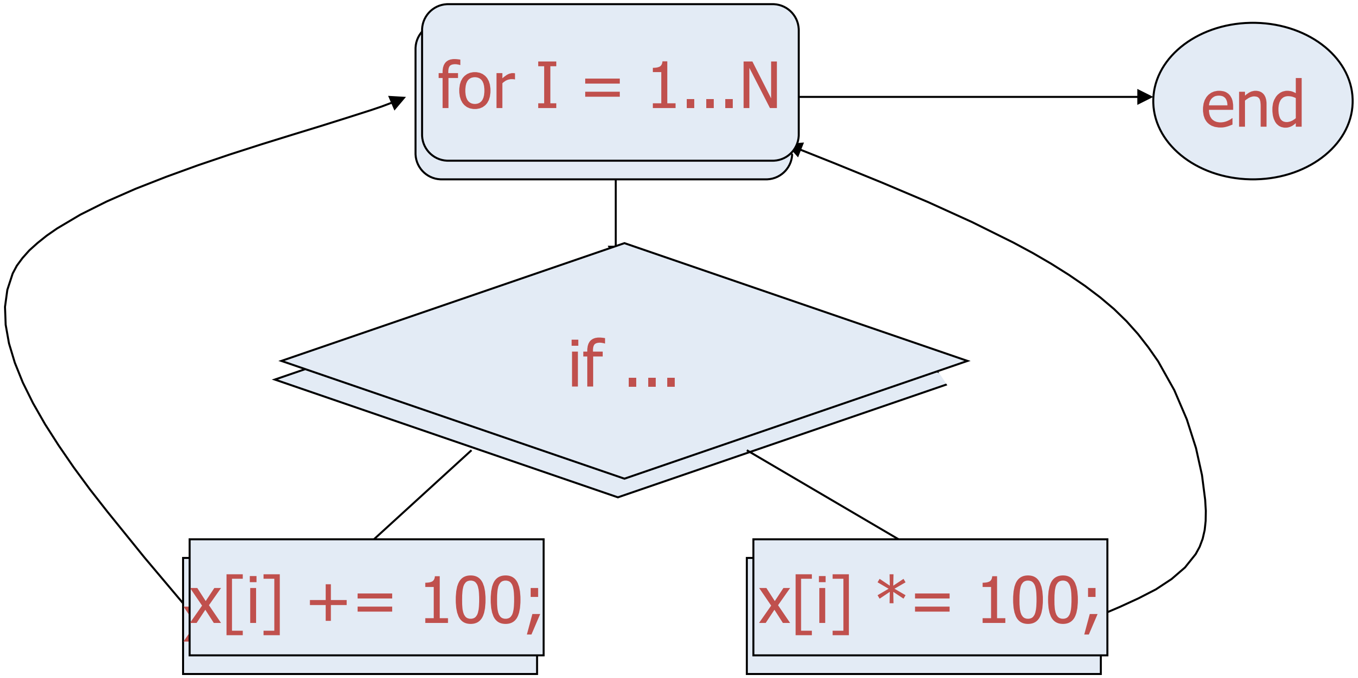
- The worldwide avionics software standard which all airborne software is required to comply.
- The world's strictest software standard
- Influences other domains including medical devices, transportation, and telecommunications.

Path Coverage

Adequacy criterion: each path must be executed at least once

Coverage:

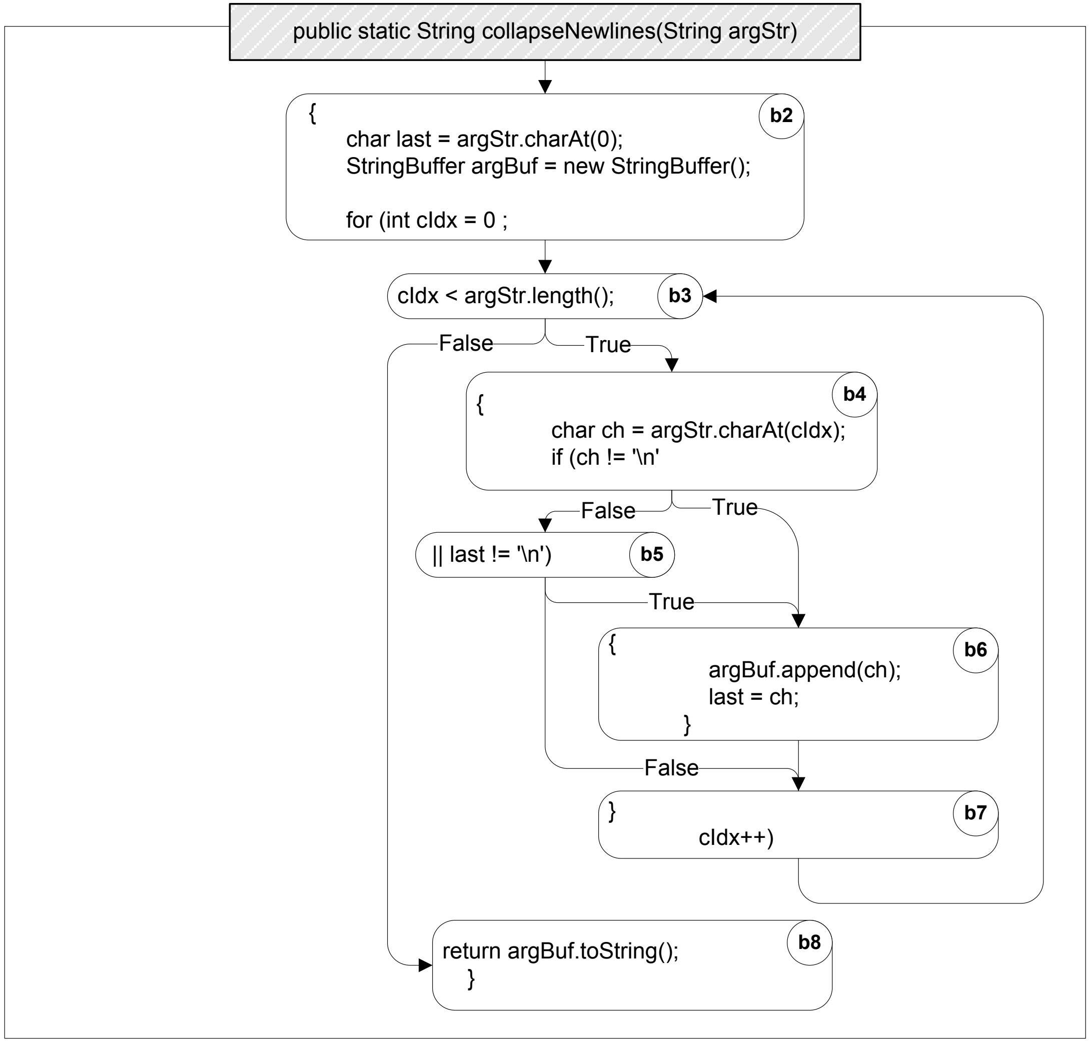
$$\frac{\text{\# executed paths}}{\text{\# paths}}$$



12.5

Path-based criteria?

- All paths?
- Which paths?
- Loop coverage?



public static String collapseNewlines(String argStr)

b2

```
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ;
```

b3

```
cIdx < argStr.length();
```

b4

```
{
        char ch = argStr.charAt(cIdx);
        if (ch != '\n'
```

b5

```
|| last != '\n')
```

b6

```
{
            argBuf.append(ch);
            last = ch;
        }
```

b7

```
}
        cIdx++)
```

b8

```
return argBuf.toString();
   }
```

Path Coverage

- "Loop boundary" testing:
  - Limit the number of traversals of loops: Zero, once, many
- "Boundary interior" testing:
  - Unfold loop as tree
- "Linear Code Sequence and Jump", LCSJ
  - Limit the length of the paths to be traversed
- "Cyclomatic complexity" / McCabe
  - "Linearly independent paths"