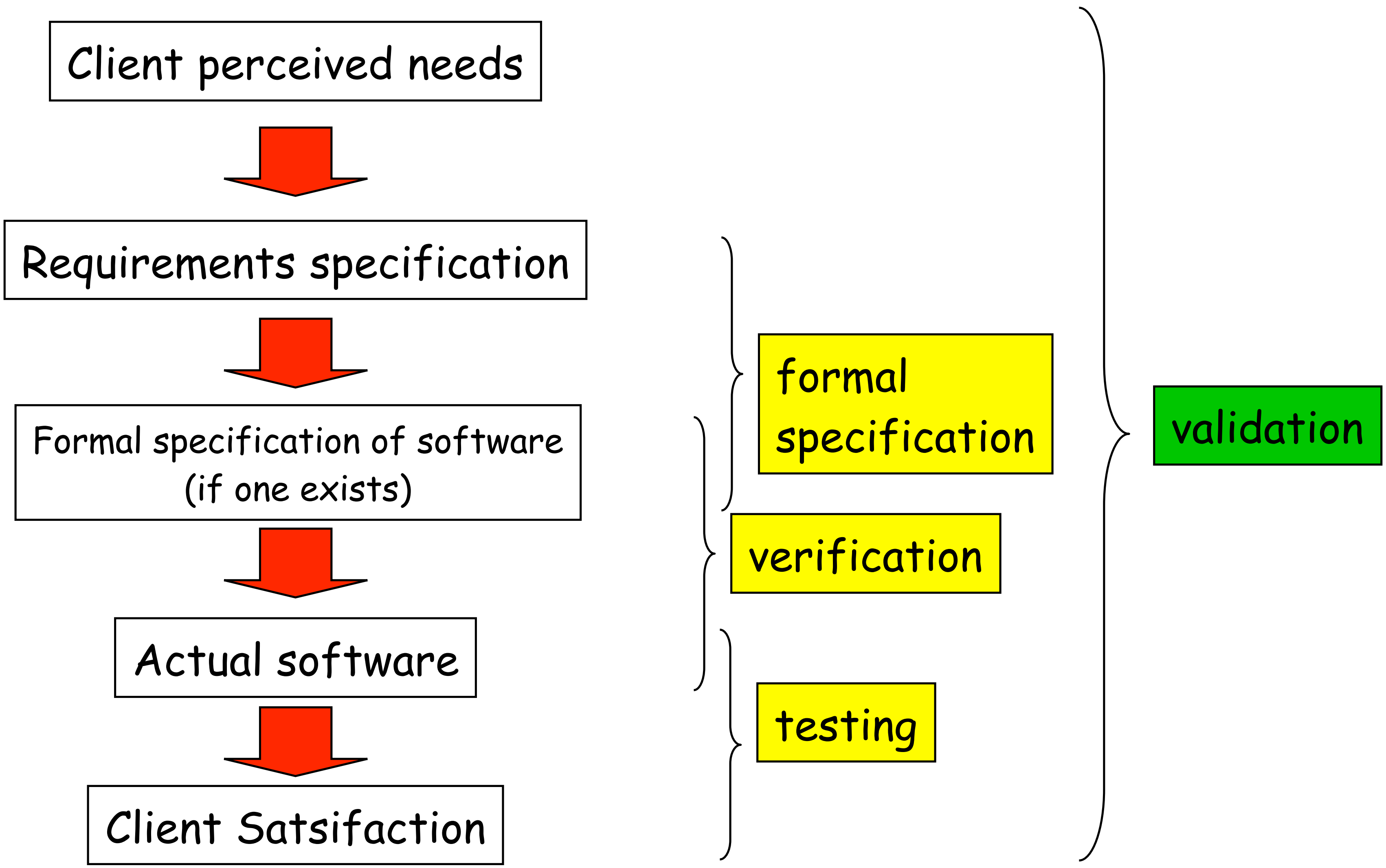
Software Validation and Testing

Verification vs. Validation

- **Verification**: using logical techniques and tools to assess correctness of the product with respect to specifications.

- **Validation**: establishing that the software developed correctly capture the user's needs and intent.

Validation vs. Verification vs. Testing



Sometimes heard...

- **Verification**: Are we building the product right?
- **Validation**: Are we building the right product?
- Apparently due to Barry Boehm (the spiral model guy).

Validation may involve Testing

- **Testing**: trying to find errors in the product
- **Verification**: using specifications, logical techniques and tools to assess correctness of the product.
- The two can be mutually supportive, in ways to be discussed.

Testing is Necessary,
but not really sufficient

Recall Famous Dijkstra Quote:

- "Testing can show the presence of errors, but never their absence."
- **Rare** exception: When the set of all inputs is **finite** and of a reasonable size, **exhaustive testing** can be used.
- Exhaustive testing is only feasible for small systems, typically hardware units, that are finite-state machines.

Exhaustive Testing

- Suppose we wanted to test a 32-bit multiply routine exhaustively. How long would it take?
- $2^{32} \times 2^{32} = 2^{64}$ input combinations
  at, say, 1 combination per nanosecond
- about 585 years, according to Unicalc

Verification alone is not sufficient either

- Creating a formal specification is hard, sometimes as hard as developing the software, or harder.
- A formal specification seldom captures **all** needs and intent.
- It is difficult to ascertain that all needs are covered until the system is built.
- Therefore, validation may not be complete without testing in addtion to verification.

Confusion

- The terminology is not standardized.
- Some people use “verification” to mean one of:
  - validation
  - testing

Standard Testing Terminology

- **Unit testing** tests self-contained units: classes, methods, functions, procedures.
- **Integration testing** tests combinations of units, such as packages, that have already been unit-tested, by having them mutually form an environment similar to actual use.
- **System testing** is top-level integration testing.
- **Acceptance testing** tests the final product according to pre-agreed criteria of the customer.

Regression Testing

- **Regression testing**: when changes are made, re-test previous test cases to ascertain that no new errors were introduced in the changes.

- "**Smoke test**": A coarse form of regression test to determine that the product doesn't simply crash as a result of recent changes.
  [Apply to "daily build" to see if there is any "smoke" (sign of new errors).]

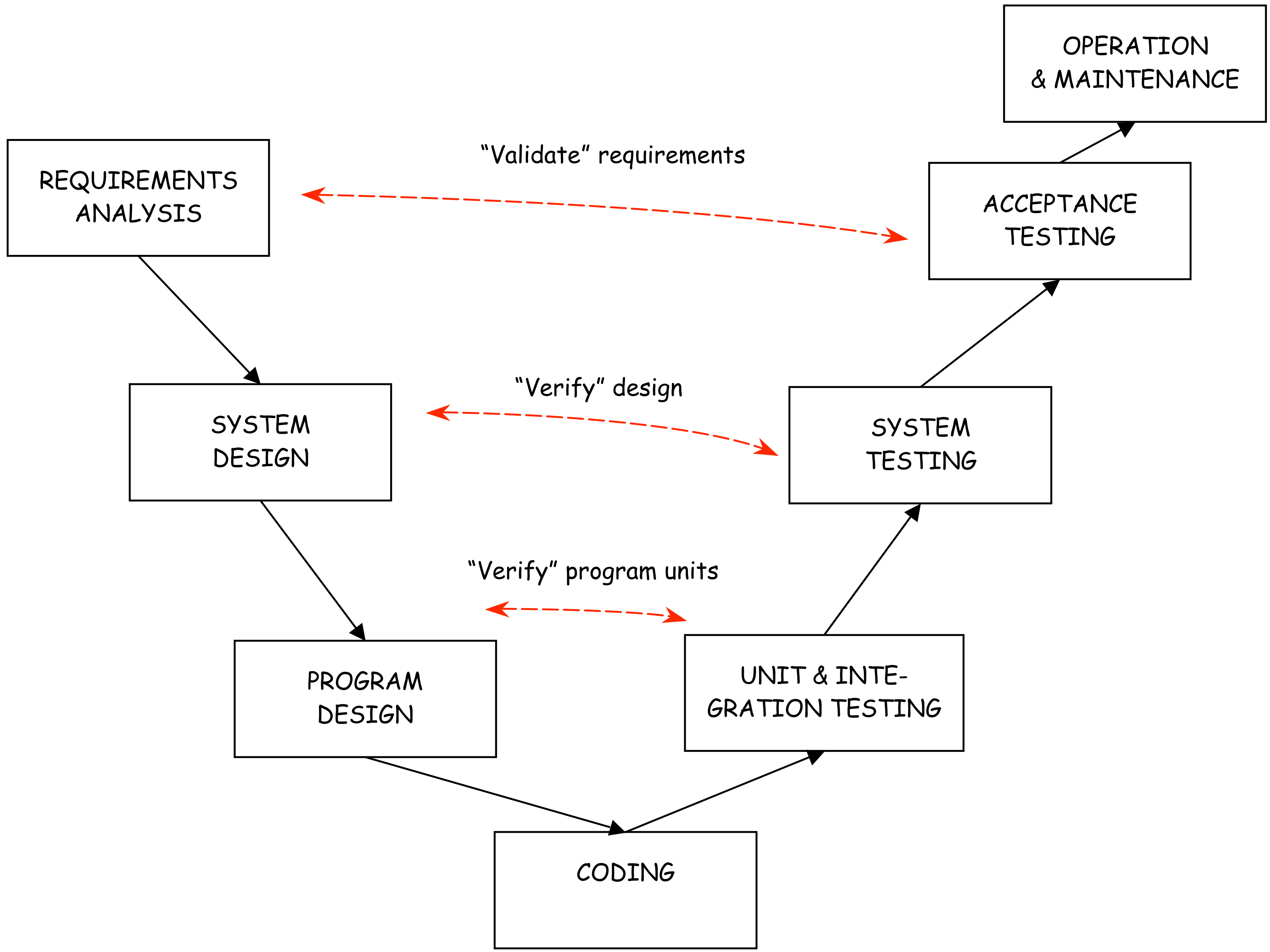
Mutation Testing

- A scheme for **testing tests**, by gauging their ***effectiveness*** of a **given test**.

- Assume that we have a test T which the code passes.

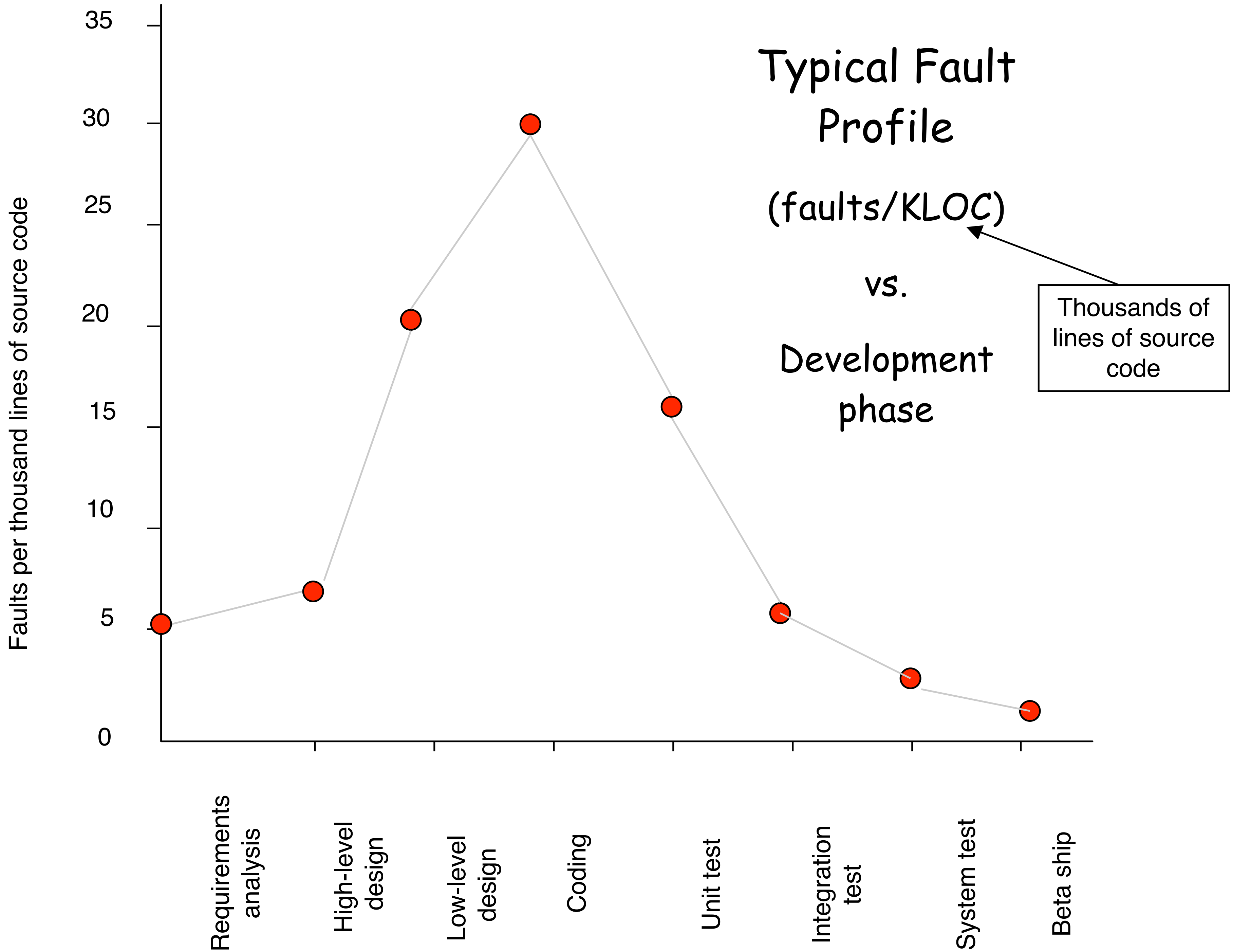
- The code is subjected to “mutation”.
  If the mutated code also *passes* test T, then T is less-likely to be regarded as a good test.

"V": *model*: A possible incorporation of testing in the software life-cycle

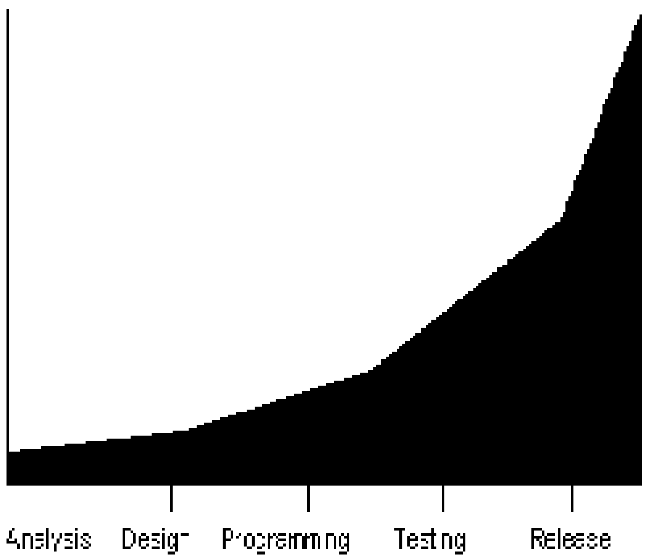


“Fault” vs. “Failure” Terminology

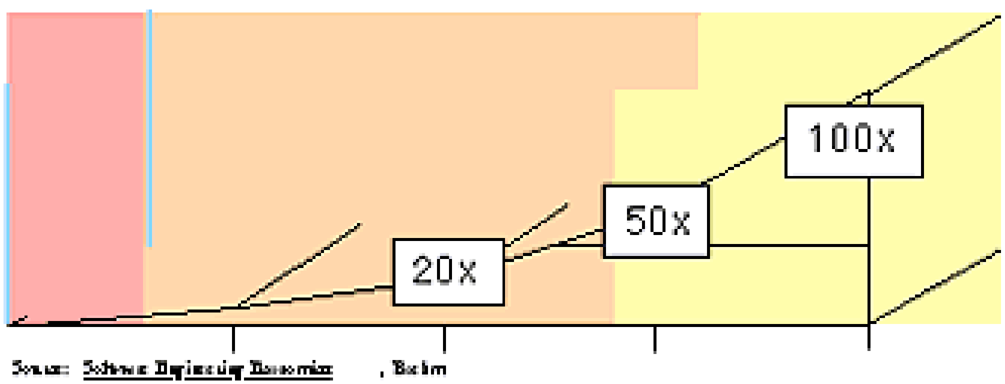
- A **fault** is an error in the code.
- A **failure** is the manifestation of a fault at runtime.
- The mapping from failures to faults is many-to-one.
- (A **flaw** is an error in the design.)


Typical Fault Profile
(faults/KLOC)
vs.
Development phase
Thousands of lines of source code
Faults per thousand lines of source code
35
30
25
20
15
10
5
0
Requirements analysis
High-level design
Low-level design
Coding
Unit test
Integration test
System test
Beta ship

Relative **Cost** of Fixing Errors vs. Phase (typical)



(Ambler)



(Boehm)

An Example

- The "Y2k Problem" probably would have cost 1/1000 (ignoring inflation) as much to fix at design/coding time as it actually took to fix in the field.

- (Then there is the cost of "fixing the fixes".)

Testing Approaches

- Test your own code
- Test code of other people in your group
- One person dedicated to testing
- Outside testing group
- Independent testing company
- Cleanroom approach

“Cleanroom” Approach

- Programmers do *not* test, or otherwise execute, their code.
- Instead, programmers establish correctness by formal reasoning methods and construction techniques.
- The actual testing takes place in the integration phase, which is done by a different team from the programmers.

Approach testing as an *intellectual challenge* in its own right

- Think as if testing **someone else**'s program, not your own.
- The objective is to find as many ***distinct*** problems as possible.
- Remember that you are testing the ***program*** and not the ***person***.

Things Testing Can't/Shouldn't Do

- Don't expect testing in itself to improve a poor design.

- As a developer, don't *rely* on testing by others as the prime method for identifying your own mistakes.

Testing Example (1)

- A reputable programmer has produced a binary search **method** for searching an array:

  ```
  int search(float* a, int M, int N, float sought);
  ```

  The method is supposed to determine whether the value sought occurs in the array a in the range of indices M to N-1. If the value occurs, its index is returned. If not, -1 is returned.
  (If M > N, the range is considered to be empty.)

  It is to be assumed that the elements of the array are sorted in that range of indices.

Testing Example (2)

- Another reputable programmer has produced a complete program "triangle" that reads triples of numbers at a time and classifies them as to whether they are the lengths of the sides of some triangle, and if so, what kind.
- Negative side-length counts as a side with the length as absolute value.
- The sides are in a specified range between 1E-150 and 1E150. Negative sides are converted to the corresponding positive value.

Testing Example (2)

- The possible outputs are:
  - “not a triangle”
  - “equilateral triangle”
  - “isosceles triangle”
  - “scalene triangle”
  - Any of the last three above preceded by “right”.
  - Or none of the above, with an indication that one or more of the inputs is out of range.
- “All classifications are based on native machine arithmetic”.

Your team's tasks

- Determine how you would test each method or program.
- What essential differences are there between the two testing approaches?
- If you intend to use test cases, what are they or how would they be created?

Testing Terminology

- **Black-box** (or opaque box) testing tests software against the operating environment, without using knowledge of internal structure.
  Also called: **Functional Testing**

- **White-box** (or clear-box) testing uses *knowledge of internal structure* to test specific pieces.
  Also called: **Structural Testing**

Testing Terminology

- **Gray-box**: Functional Testing assisted by structural knowledge to reduce unnecessary blind testing.

Testing Continuum

Pure white-box (structural)	Gray-box (some structural)	Pure black-box (behavioral)	Live tests
Development activity		Testing activity	Technical support, beta testing

Black- vs. White Box

- **Black-box** focuses on the **specification**: What is in the **spec** that the program doesn't do?
- **White-box** focuses on the **program**: What does the **program** do that is not in the spec?
- Normally don't rely on one or the other exclusively. "Mow the grass in two directions" for better cutting.

Basic Testing Tools

- **Plan & Checklists**: tests to perform
  - With each test, pre-condition, post-condition
- **Test matrix/spreadsheet**: listing tests against use cases and potential error areas
- **Logging capability**
  - Note pad
  - Keystroke recorder (for playback)
  - Event logger
  - Screen recorder
  - Video camera
- **Tracking Database**

Testing Matrix

Tested by					
	Test 1	Test 2	Test 3	Test 4	Test 5
Use Case 1	x				
Use Case 2		x	x		
Use Case 3		x		x	x

Test Result Categorization

- Area of defect
- Severity level of defect
- Follow up:
  - Responsibility
  - Time to fix
  - Lines of code affected

10 Sample Severity Levels, with examples (Boris Beizer)

- **Mild**: misspellings in output
- **Moderate**: misleading or redundant behavior
- **Annoying**: truncated names, etc.
- **Disturbing**: some transactions not processed
- **Serious**: lost transaction
- **Very serious**: incorrect output
- **Extreme**: frequent “very serious” errors
- **Intolerable**: data corrupted
- **Catastrophic**: shutdown
- **Infectious**: shutdown spreading to others

4 Possible Reaction Levels to failed tests

- **Defer**: fix as time permits
- **Schedule**: fix by some future date
- **Required**: fix before acceptance
- **Immediate**: fix before testing is continued

Testing Forms and Tracking

- Discrepancy report form
- Error investigation form
- Error reporting/tracking system/database

Discrepancy Report Form

(Sherry Pfleeger)

DISCREPANCY REPORT FORM

DRF Number:____ Tester name:________________________________
Date: ____ Time: _____________________________________
Test Number: ___________________________________
Script step executed when failure occurred: ____
Descripton of failure: ____
____
____
____
Activities before occurrence of failure:
____
____
Expected results:
____
____
Requirements affected:
____
____
Effect of failure on test:
____
____
Effect of failure on system:
____
____
Severity level:
(LOW) 1 2 3 4 5 (HIGH)

FAULT REPORT

S.P0204.6.10.3016

ORIGINATOR: Joe Bloggs

BRIEF TITLE: Exception 1 in dps_c.c line 620 raised by NAS

FULL DESCRIPTION Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above.
(I think the re-enabling is a red herring.)

ASSIGNED FOR EVALUATION TO: **DATE:**

CATEGORISATION: **0 1 2 3 Design Spec Docn**
SEND COPIES FOR INFORMATION TO:
EVALUATOR: **DATE:** 8/7/92

CONFIGURATION ID	ASSIGNED TO	PART
dpo_s.c		

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

CONFIGURATION ID	IMPLEMENTOR/DATE	REVIEWER/DATE	BUILD/ISSUE NUM	INTEGRATOR/DATE
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

COMMENTS:

CLOSED

FAULT CONTROLLER: **DATE: 9/7/92**

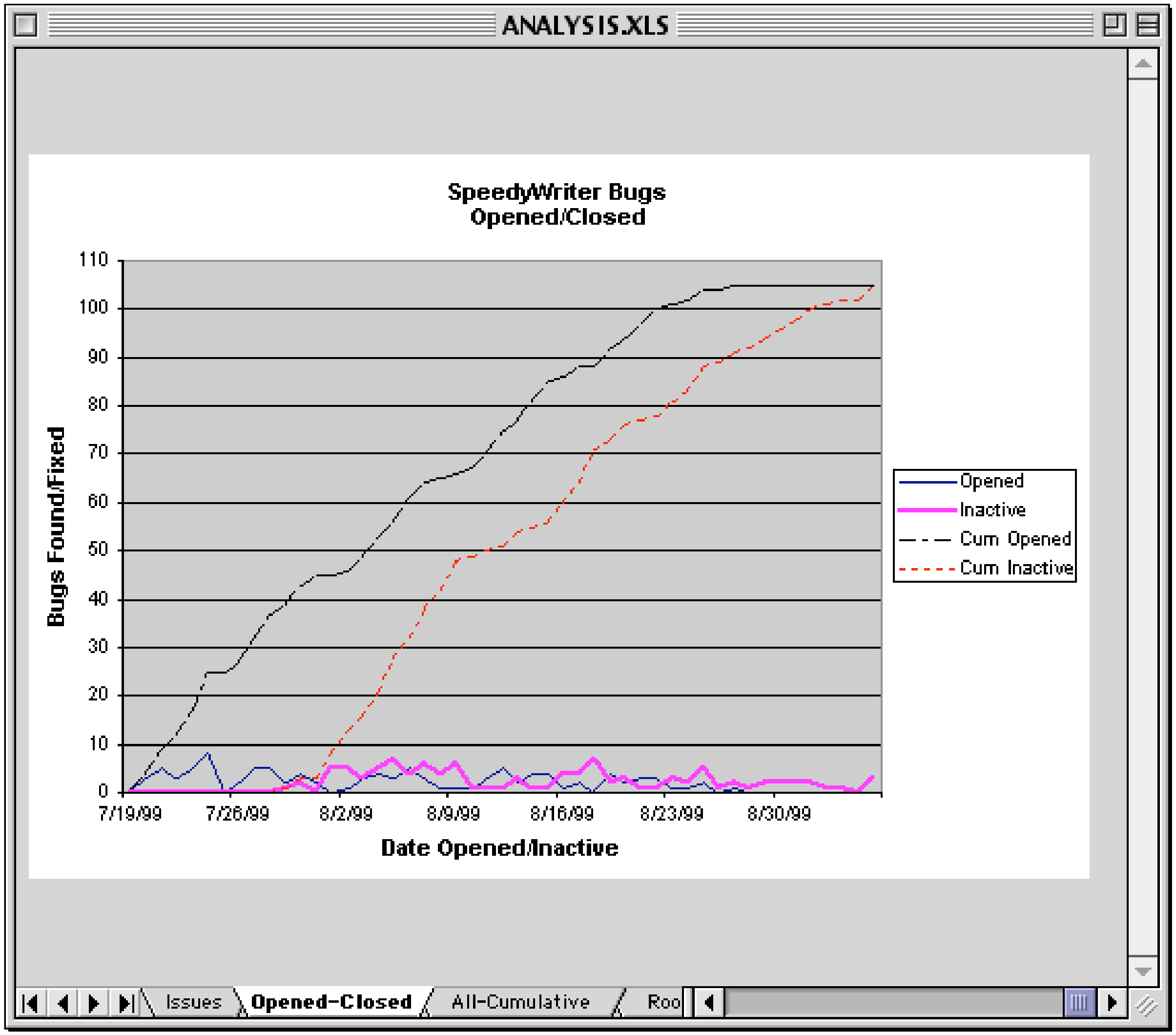
Testing Issue Spreadsheet

ANALYSIS.XLS

	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	Date Open	Sev	Pri	Risk	Owner	Estimated	Su	St	Isc	St	Subsystem	Configuration	Close Date	Resolution	Root Cause	
2	7/21/99	1	1	1	Muhammad Zam	8/12/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/3/99		Functional	
3	7/21/99	2	2	4	Chuck Chavall	7/28/99	du	du	du	du	User Interface	A.X.0.0.001	7/30/99		System	
4	7/24/99	3	3	9	Jenny Chung	8/12/99	du	du	du	du	Edit Engine	A.Y.0.0.001	8/1/99		Process	
5	7/23/99	3	3	9	Muhammad Zam	7/24/99	du	du	du	du	Edit Engine	A.Z.0.0.001	8/2/99		Data	
6	7/24/99	1	4	4	Chuck Chavall	7/31/99	du	du	du	du	User Interface	B.Y.0.0.001	8/1/99		Functional	
7	7/24/99	2	5	10	Jenny Chung	8/17/99	du	du	du	du	Tools	C.X.0.0.001	8/7/99		System	
8	7/23/99	3	3	9	Larry Kanemetsu	8/4/99	du	du	du	du	Edit Engine	C.Y.0.0.001	8/4/99		Process	
9	7/20/99	4	1	4	Frank Currant	8/14/99	du	du	du	du	User Interface	C.Z.0.0.001	8/5/99		Data	
10	7/22/99	5	1	5	Chuck Chavall	8/18/99	du	du	du	du	Edit Engine	D.Z.0.0.001	8/5/99		Functional	
11	7/22/99	1	1	1	Jenny Chung	7/28/99	du	du	du	du	User Interface	A.X.0.0.001	8/4/99		Documentation	
12	7/23/99	2	2	4	Muhammad Zam	8/17/99	du	du	du	du	User Interface	A.X.0.0.001	8/7/99		Functional	
13	7/23/99	1	3	3	Chuck Chavall	7/26/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/3/99		System	
14	7/21/99	2	1	2	Jenny Chung	8/13/99	du	du	du	du	Other	A.X.0.0.001	8/5/99		Process	
15	7/23/99	1	2	2	Larry Kanemetsu	7/27/99	du	du	du	du	Unknown	A.X.0.0.001	8/7/99		Data	
16	7/24/99	2	3	6	Frank Currant	7/25/99	du	du	du	du	N/A	A.Y.0.0.001	8/7/99		System	
17	7/24/99	3	4	12	Chuck Chavall	8/13/99	du	du	du	du	Edit Engine	A.Z.0.0.001	8/2/99		Functional	
18	7/24/99	4	1	4	Jenny Chung	8/19/99	du	du	du	du	User Interface	B.Y.0.0.001	8/1/99		Functional	
19	7/21/99	1	2	2	Muhammad Zam	7/27/99	du	du	du	du	Tools	C.X.0.0.001	7/30/99		System	
20	7/19/99	1	3	3	Chuck Chavall	7/27/99	du	du	du	du	File	C.Y.0.0.001	8/1/99		Process	
21	7/22/99	2	4	8	Jenny Chung	8/18/99	du	du	du	du	User Interface	C.Z.0.0.001	8/1/99		Data	
22	7/21/99	3	5	15	Larry Kanemetsu	7/22/99	du	du	du	du	Edit Engine	D.Z.0.0.001	7/29/99		Process	
23	7/20/99	4	3	12	Muhammad Zam	8/13/99	du	du	du	du	User Interface	C.Y.0.0.001	8/4/99		Documentation	
24	7/20/99	1	2	2	Chuck Chavall	7/30/99	du	du	du	du	Edit Engine	C.Z.0.0.001	8/2/99		Functional	
25	7/24/99	4	3	12	Jenny Chung	8/12/99	du	du	du	du	User Interface	D.Z.0.0.001	8/5/99		System	
26	7/24/99	5	4	20	Larry Kanemetsu	8/8/99	du	du	du	du	User Interface	A.X.0.0.001	8/2/99		Process	
27	7/28/99	1	5	5	Frank Currant	8/22/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/7/99		Data	
28	7/29/99	2	3	6	Chuck Chavall	8/20/99	du	du	du	du	Other	A.X.0.0.001	8/11/99		Code	
29	7/31/99	1	3	3	Jenny Chung	8/5/99	du	du	du	du	Unknown	A.X.0.0.001	8/9/99		Functional	
30	7/28/99	2	3	6	Larry Kanemetsu	8/17/99	du	du	du	du	Edit Engine	B.Y.0.0.001	8/8/99		System	
31	7/27/99	1	1	1	Muhammad Zam	8/1/99	du	du	du	du	User Interface	B.Y.0.0.001	8/10/99		Process	
32	7/30/99	2	2	4	Chuck Chavall	8/20/99	du	du	du	du	Tools	C.X.0.0.001	8/6/99		Data	
33	7/27/99	3	3	9	Jenny Chung	8/3/99	du	du	du	du	File	C.Y.0.0.001	8/5/99		Code	
34	7/26/99	4	4	16	Muhammad Zam	8/14/99	du	du	du	du	Install/Config	C.Z.0.0.001	8/2/99		System	
35	7/27/99	1	5	5	Chuck Chavall	8/12/99	du	du	du	du	Edit Engine	D.Z.0.0.001	8/4/99		Functional	
36	7/31/99	1	1	1	Jenny Chung	8/3/99	du	du	du	du	Edit Engine	A.X.0.0.001	8/8/99		System	
37	7/30/99	2	2	4	Larry Kanemetsu	8/24/99	du	du	du	du	User Interface	A.X.0.0.001	8/15/99		Process	
38	7/30/99	3	1	3	Frank Currant	8/11/99	du	du	du	du	Tools	D.Z.0.0.001	8/8/99		Data	
39	7/27/99	4	2	8	Chuck Chavall	8/2/99	du	du	du	du	Edit Engine	B.Y.0.0.001	8/6/99		Functional	
40	7/28/99	5	1	5	Jenny Chung	8/5/99	du	du	du	du	User Interface	C.X.0.0.001	8/5/99		Process	

Issues / Opened-Closed / All-Cumulative / Root Cause Breakdown / Roo

Open/Closed Bug Tracking



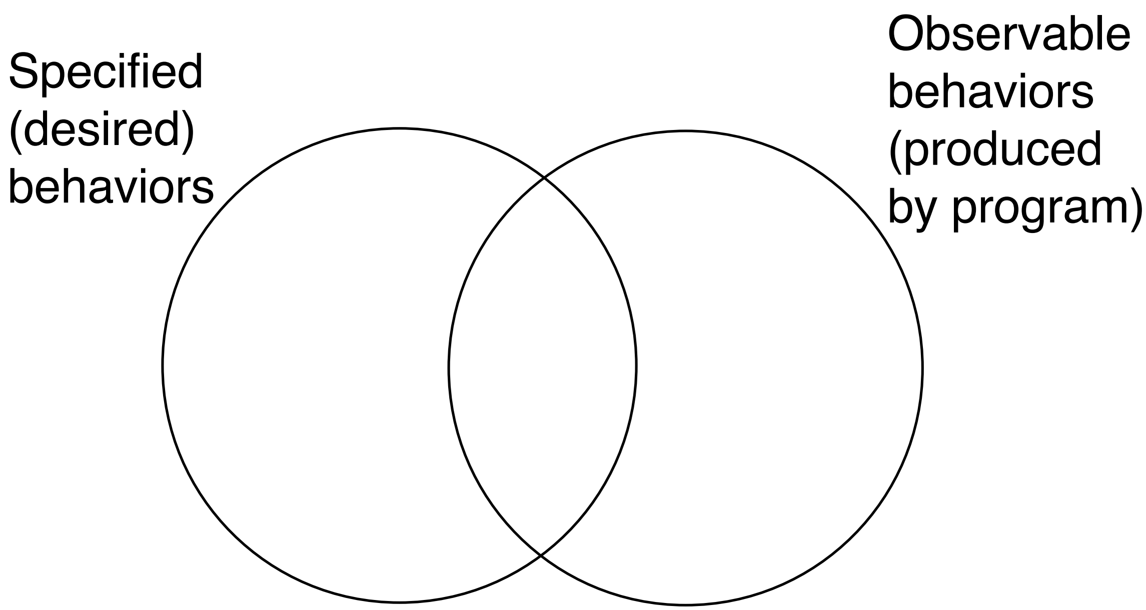
General Testing Approach

- Create lists of **potential** problems and categorize them by area.
- Design **repeatable** tests, for proof of problems and problem resolutions.

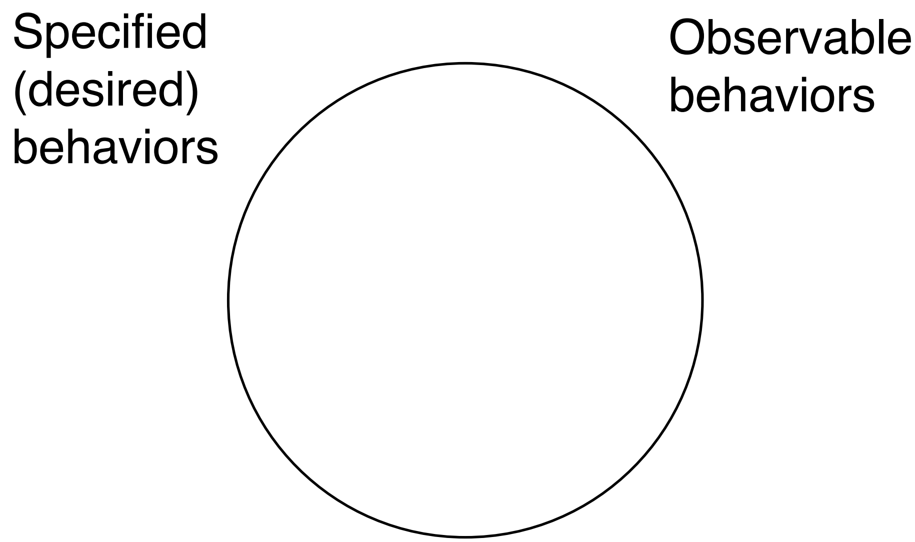
Broad Categories for Errors

- Errors in interpreting requirements
- Errors in translating requirements into design, i.e. in programming
- Errors in implementing design
- Errors in the testing process itself

Program Behavior Views
(sets of behaviors taken over all inputs)



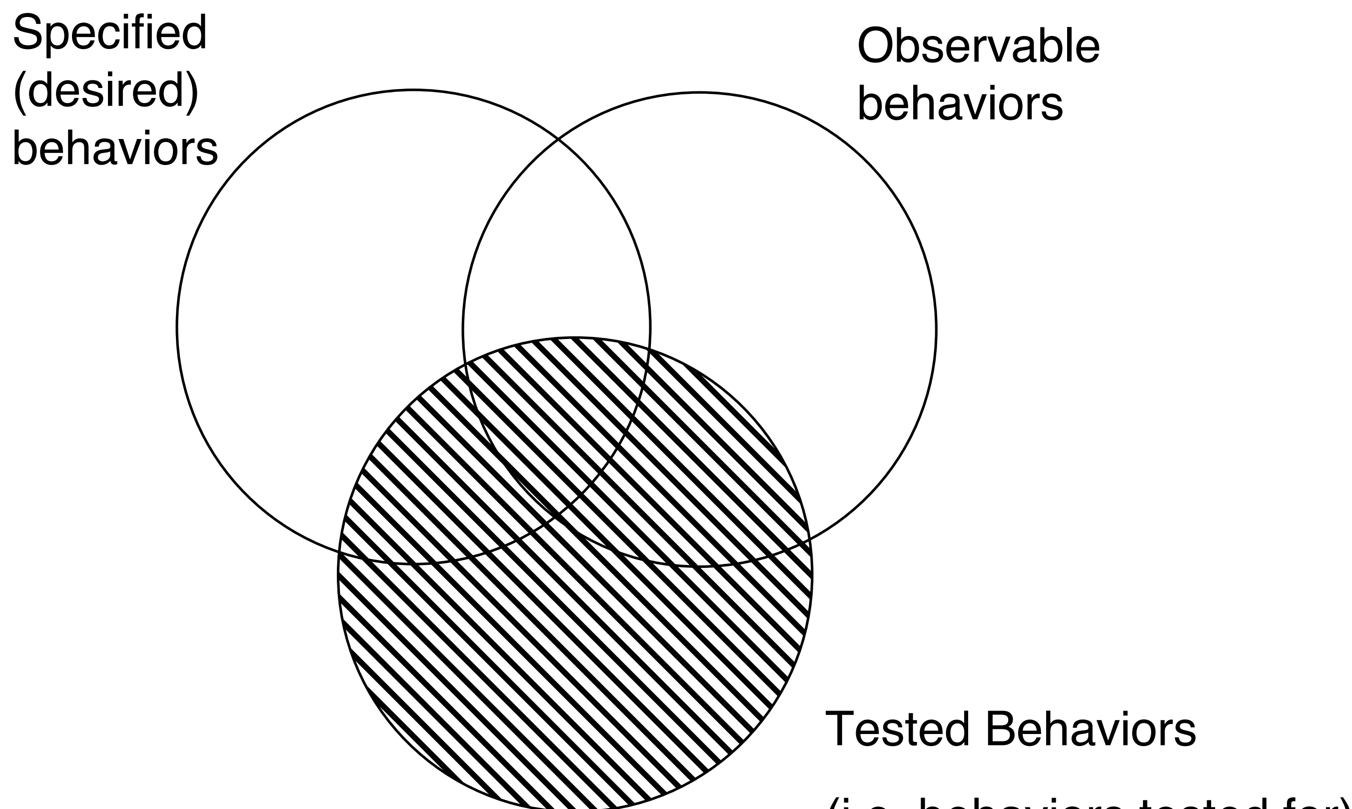
The Ideal



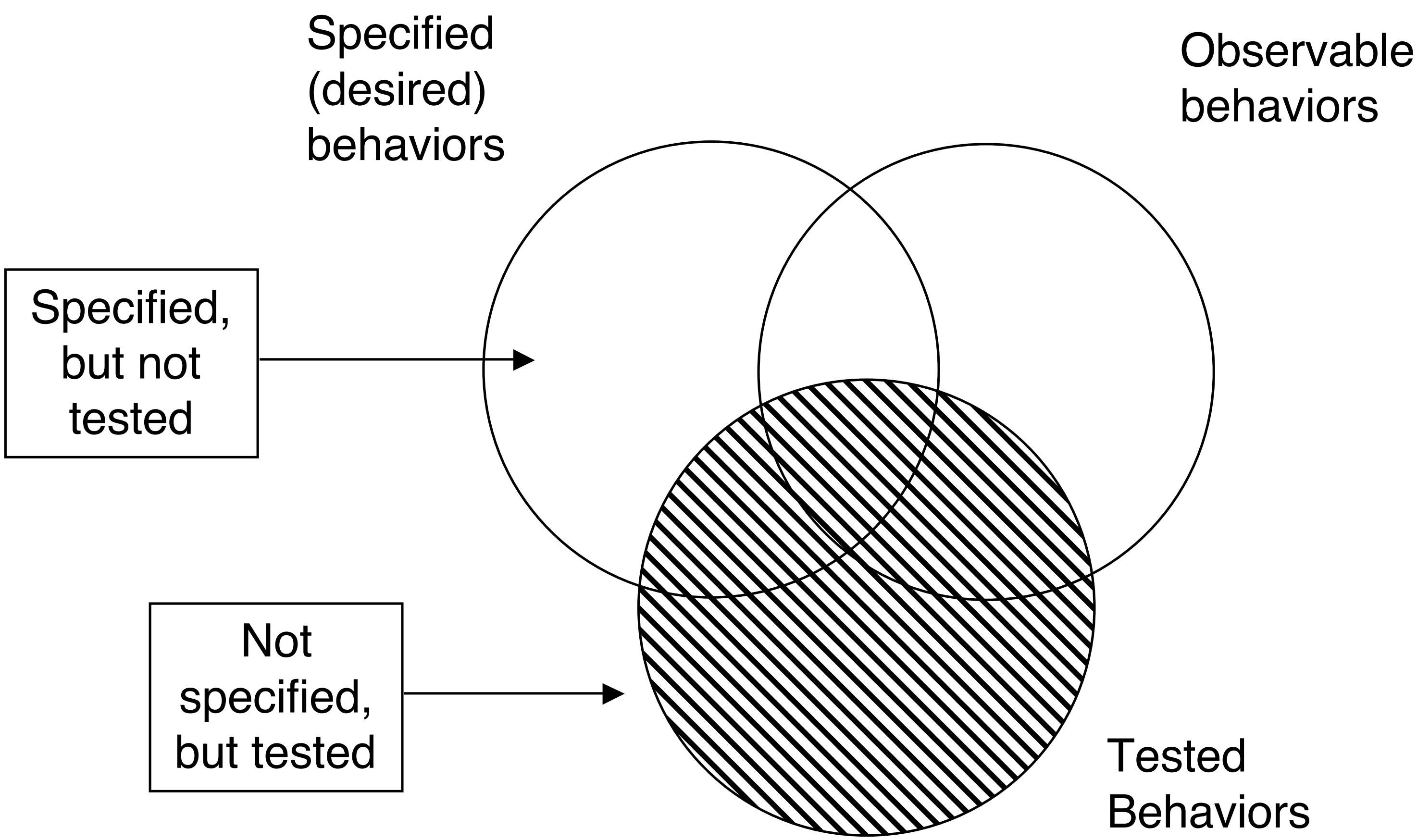
The ideal might not be fully realizable because

- Some aspects of a specification may be left arbitrary, unspecified.
- Either:
  - The specification is to be regarded as incomplete, or
  - Any behavior ***consistent with*** the specification will be accepted.

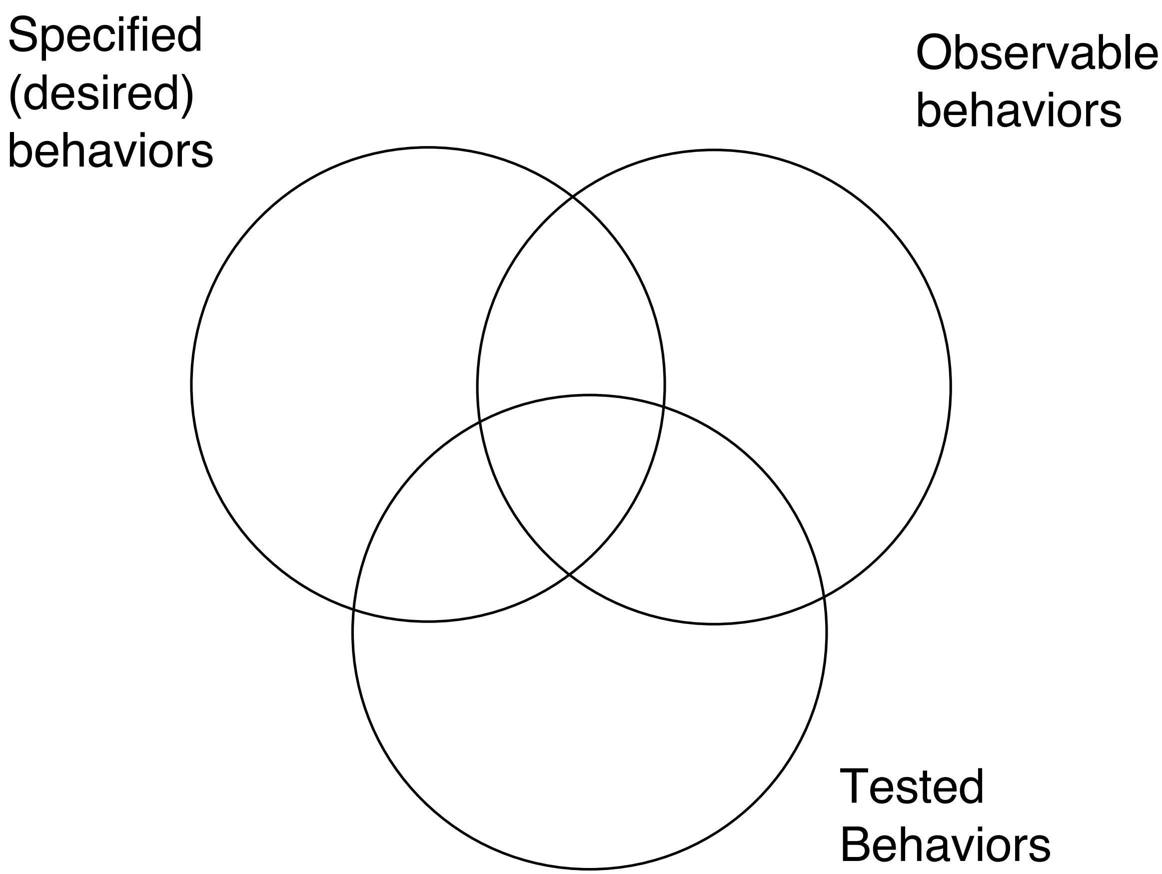
Testing asks questions:
Does a behavior occur?



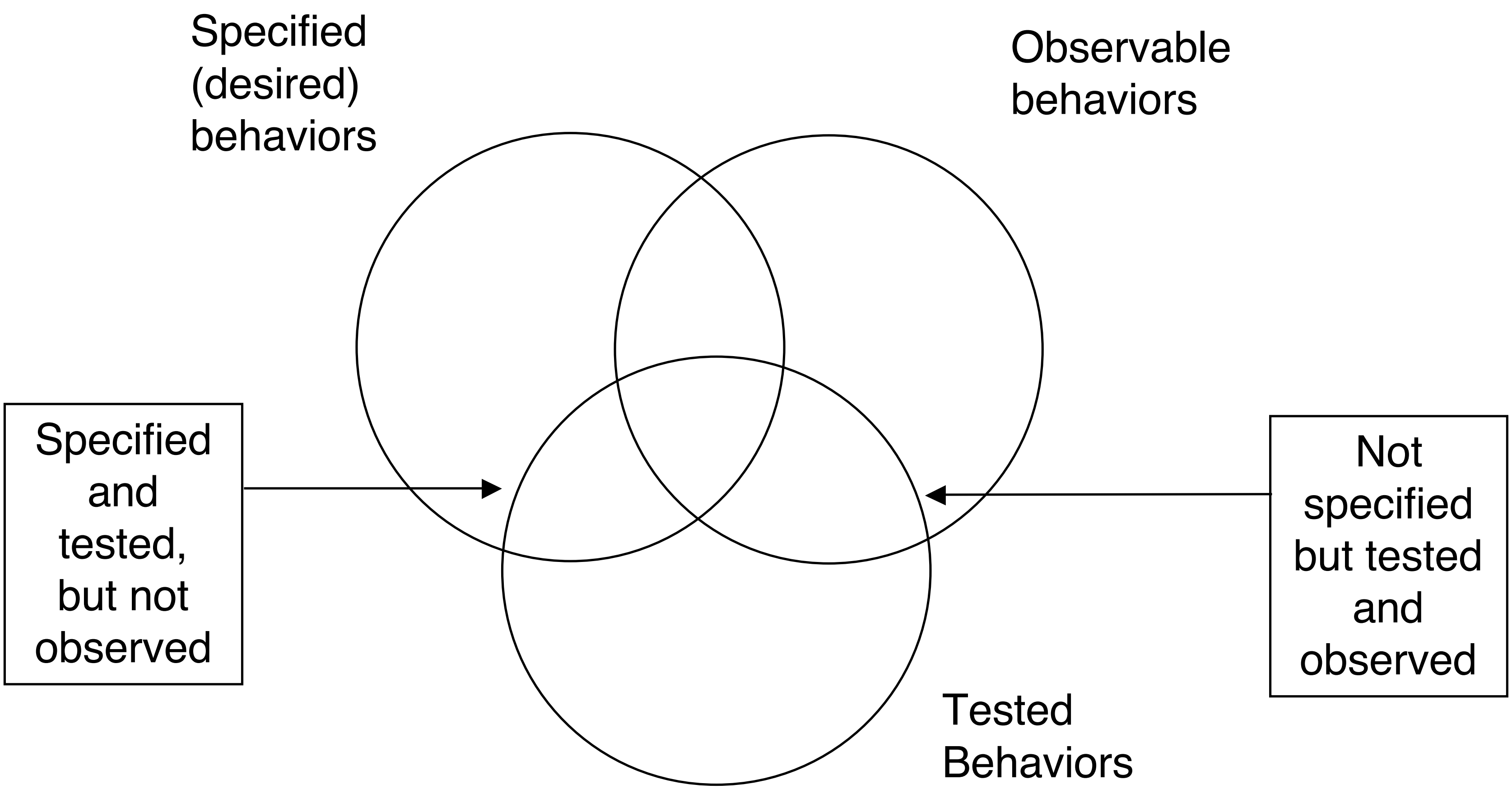
Testing


Specified (desired) behaviors
Observable behaviors
Specified, but not tested
Not specified, but tested
Tested Behaviors

Which regions of the diagram indicate errors?



Which regions of the diagram indicate errors?

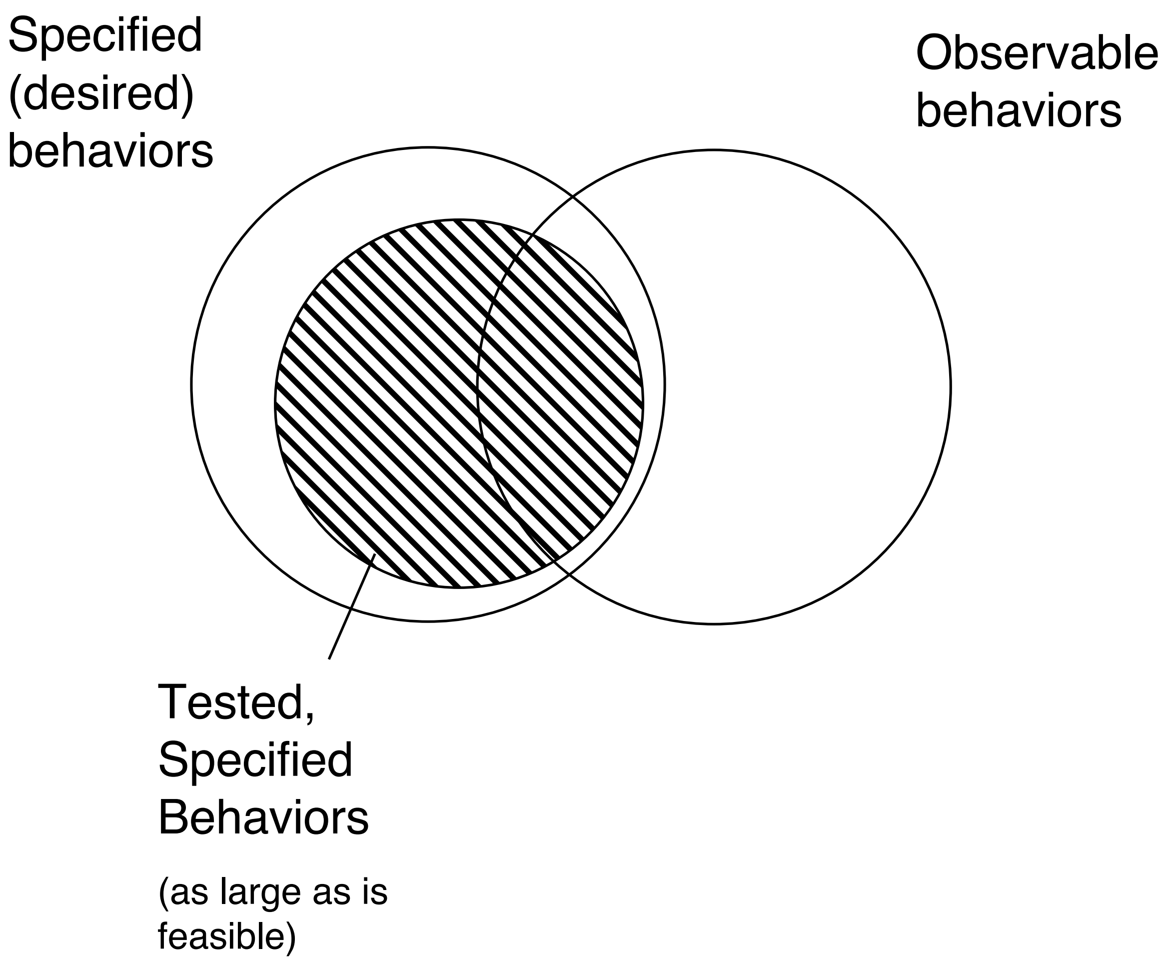


Black- vs. White Box
(recap)

- **Black-box** focuses on the **specification**: What is in the **spec** that the program doesn't do?
- **White-box** focuses on the **program**: What does the **program** do that is not in the spec?
- Normally don't rely on one or the other exclusively. "Mow the grass in two directions" for better cutting.

Black-Box Testing

Good Black-Box Test Plan



Black-Box Techniques

- Recall that “black box” means we do not get to see the code; we only have access to an installation of the product.
- Also called “functional testing” (vs. “structural testing”, which would be “white box”)
- Driven by requirements, use cases

Black-Box Techniques

- **Equivalence Partitioning**:
  - Use a small number of test **equivalence classes**, rather than a large number of individual test data points.
  - The actual tests are representatives of the equivalence classes.
  - Partitioning based on their relative likelihood of exposing logic errors in the code.
  - Example: Partition a number space into:
    - less than 0
    - equal to 0
    - greater than 0, less than 100
    - greater than or equal to 100

Black-Box Techniques

- Equivalence Partitioning Examples (cont'd):
  - Partition a two-dimensional number space into (x, y) where:
    - $x < y$
    - $x == y$
    - $x > y$
  - Partition a String space into
    - length = 0
    - length = 1
    - length = 2
    - length > 2
    - length extremely large

Black-Box Techniques

- How would you equivalence-partition the triangle program input space?

Decision Table

A "declarative" means to categorize input

- Partitions input space (triples of numbers) into equivalence classes.
- All inputs in a cell should have the same anticipated equivalence class.

Input Categories

Output Categories

a, b, c a triangle?	N	Y							
a = b?	-	Y				N			
a = c?	-	Y		N		Y		N	
b = c?	-	Y	N	Y	N	Y	N	Y	N
not a triangle	x								
scalene									x
isosceles					x		x	x	
equilateral		x							
should not occur			x	x		x			

Decision Table Variant

- Slightly more condensed, based on logical equivalences
- Fewer or no "should not occur" entries

Input Categories

<table>
<tr><td>a, b, c a triangle?</td><td>N</td><td colspan="5">Y</td></tr>
<tr><td>a = b?</td><td>-</td><td colspan="2">Y</td><td colspan="3">N</td></tr>
<tr><td>a = c?</td><td>-</td><td>Y</td><td>N</td><td>Y</td><td colspan="2">N</td></tr>
<tr><td>b = c?</td><td>-</td><td>Y</td><td>N</td><td>N</td><td>Y</td><td>N</td></tr>
<tr><td>not a triangle</td><td>x</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>scalene</td><td></td><td></td><td></td><td></td><td></td><td>x</td></tr>
<tr><td>isosceles</td><td></td><td></td><td>x</td><td>x</td><td>x</td><td></td></tr>
<tr><td>equilateral</td><td></td><td>x</td><td></td><td></td><td></td><td></td></tr>
</table>

Output Categories

Black-Box Techniques

- **Boundary-value testing**: Pick test cases near to "natural" boundaries in data space, so as to test whether the program performs the correct classification of borderline cases.

Black-Box Techniques

- **Logarithmic testing**: Repeatedly split the data space into two, testing sample points in the upper and lower halves of the split, then repeat on the lower half only.

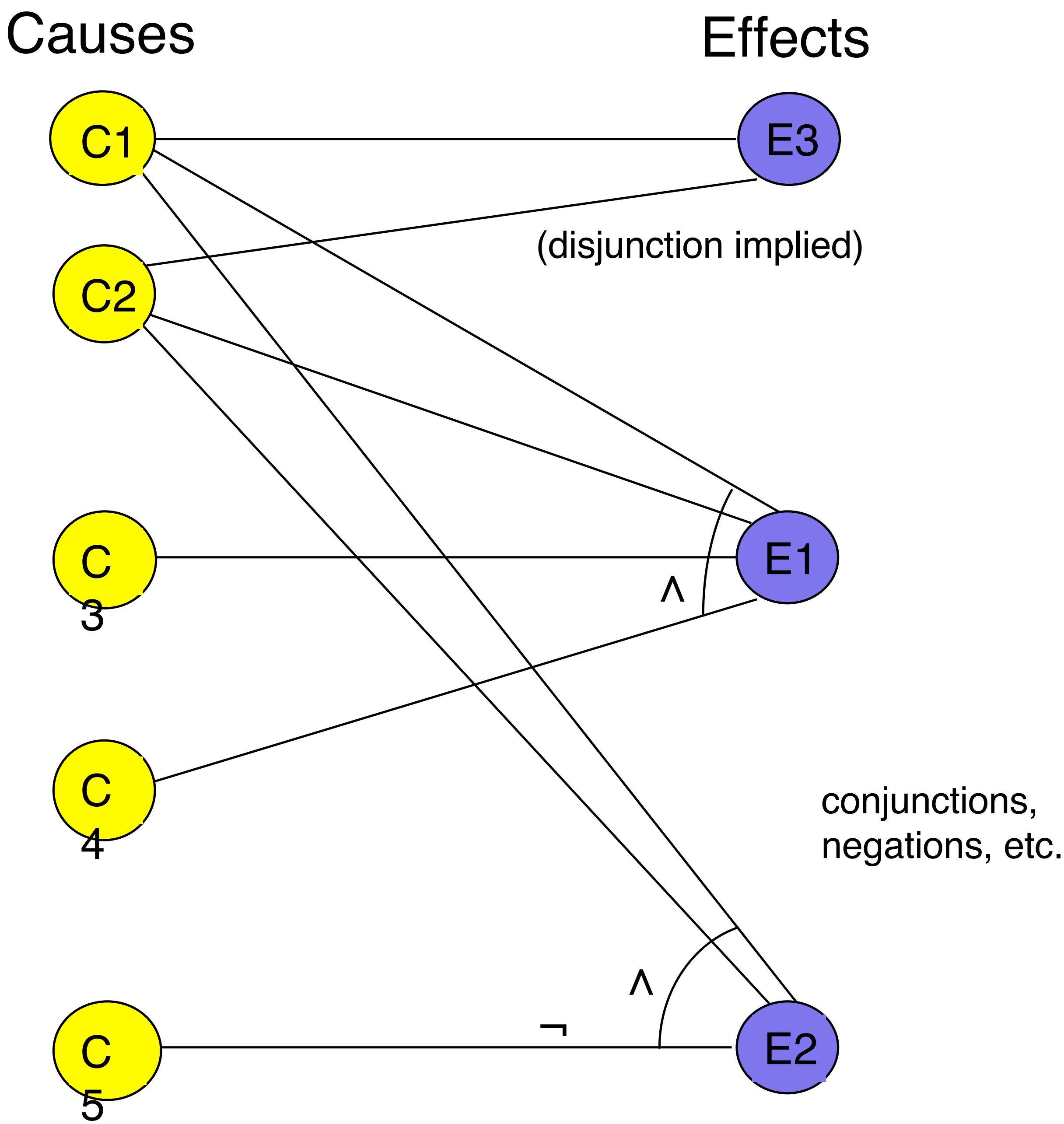
Black-Box Techniques

- **Cause-Effect Graphing**: Examine requirements specification for **chains of conditions**; develop test cases that check whether these chains are actually observed
  - Example: “If the clipboard is empty, then the paste menu option should *not* be selectable.”
  - Therefore: Develop a test in which the clipboard should be empty, and check that the menu option is not selectable.

Black-Box Techniques

- **Cause-Effect Graphing**, possible relationships:
  - A condition *implies* an action
  - A condition *precludes* an action
  - Two actions are *mutually* exclusive
  - A *combination* (conjunction) of two conditions implies an action
  - etc.

Example Cause-Effect Graph



Exercise

- Discuss cause-effect graphing for the Traffic Jam program.

Black-Box Techniques

- **Comparison Testing**:

  Test product side-by-side with a **"gold standard"**,

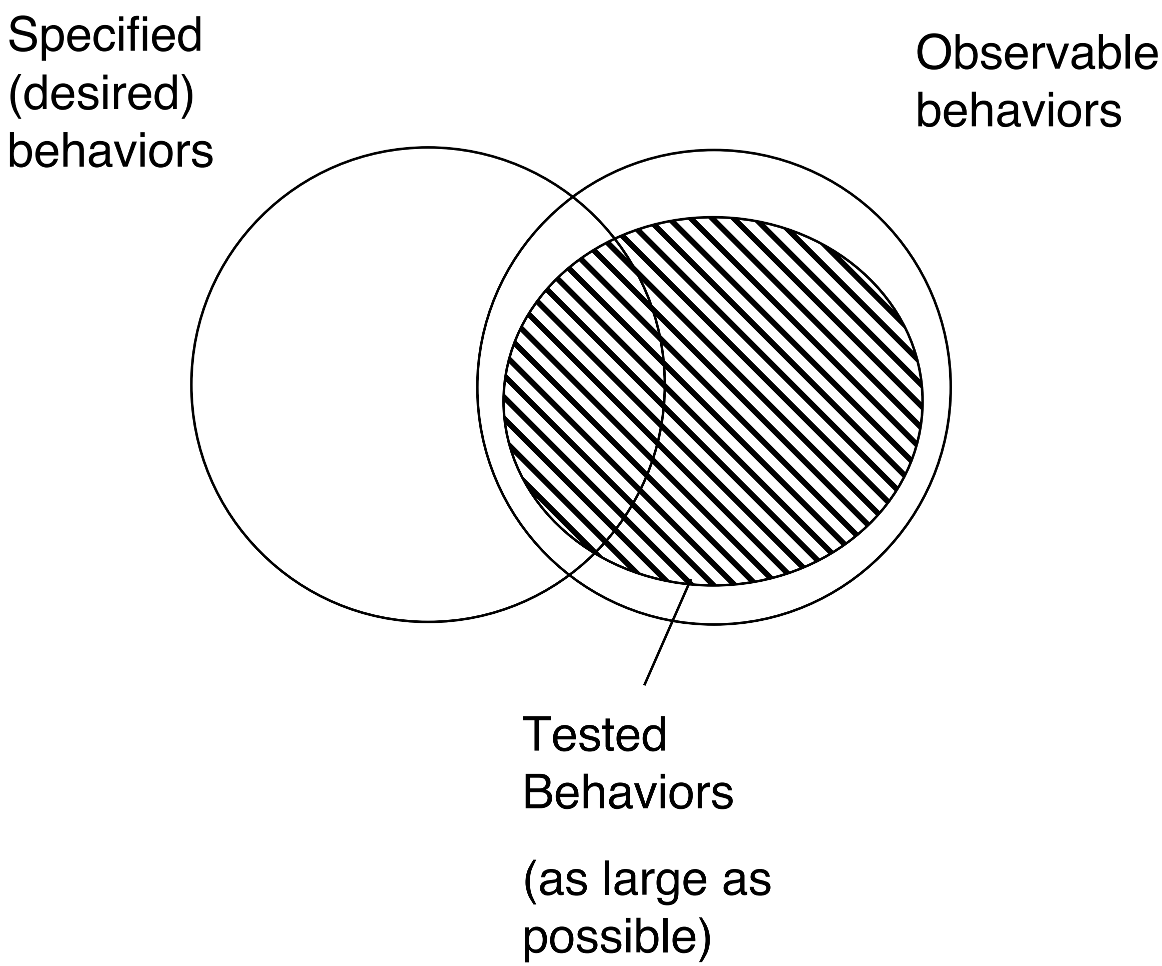
  a program believed to be correctly operating (such as an earlier version having most, if not all, of the features)

Black-Box Techniques

- **Garbage-In Test**: See if unusual input characters, click sequences, etc. can force the system into inconsistent states.
- **Open-Book Test**
- **Data-Quantity Stress Test**: See if unusually large amounts of data cause nominal values to be exceeded, revealing untested overflow conditions, etc.

White-Box Testing

Good White-Box Test Plan



Consider the "triangle" program

- You are now given source code for the triangle program.
- What tests would you perform?

Code Inspection and Walkthrough

- Used for:
  - Identifying errors
  - Identifying test cases
  - Helping yourself and others to understand how the program works.

Code for the triangle program (page 1 of 4).

```
class Conversion
{
public:
/** top of acceptable numeric range */
static const double MAX = 1e150;

/** bottom of acceptable numeric range */
static const double MIN = 1e-150;

/** Check whether argument is within range **/
inline static bool inRange(double value) {return MIN <= value && value <= MAX;}


/**
 * Check whether an input string represents a valid floating point numeral
 * (as opposed to some non-digit characters, etc.)
 * and, if so, whether the number is in the specified range.
 *
 * @param inputString string to be checked for being a numeral, and possibly
 *         converted to number result.
 *
 * @param result number to which string was converted, if successful
 *
 * @return true if the input represents a valid number in the specified
 *         range and result is the numeric value,
 *         otherwise return false and 'result' is undefined.
 *
 * pre-condition: true <br>
 *
 * post-condition: If returned value is true, then result is the numeric
 *                 representation of the argument inputString.
 *                 If returned value is false, then 'result' is undefined.
 **/

static bool convert(const string inputString, double& result)
  {
  char* endptr;

  // After strtod, endptr will point to end of converted string.
  // Conversion is successful iff *endptr is the null character (0).

  result = strtod(inputString.c_str(), &endptr);

  if( *endptr == 0 )
    {
    result = fabs(result);

    return inRange(result);
    }

  return false;
  }
};  // Conversion
```

```
/**
 * TriangleTester deals analyze three sides as a triangle and reading
 * sets of three sides from an input stream.
 **/

class TriangleTester
{
public:
static const int SIDES = 3;             // This is about triangles.

static enum{ NOT_A_TRIANGLE,
             EQUILATERAL_TRIANGLE,
             SCALENE_TRIANGLE,
             RIGHT_SCALENE_TRIANGLE,
             ISOSCELES_TRIANGLE,
             RIGHT_ISOSCELES_TRIANGLE} Classification;

/**
 * Return the type of triangle, if any, that is formed by three sides.
 * @param a first side
 * @param b second side
 * @param c third side
 *
 * pre-condition: all sides are positive and in-range <br>
 *
 * post-condition: the correct classification is returned
 **/

static int analyze(double a, double b, double c)
  {
  // Arrange the three sides to simplify subsequent analysis.

  order(a, b);
  order(b, c);
  order(a, c);

  // assert( a <= b && b << c );

  if( a + b <= c )  // The only test needed for triangle-ness, due to ordering.
    {
    return NOT_A_TRIANGLE;
    }
  if( a == b && b == c )
    {
    return EQUILATERAL_TRIANGLE;
    }

  bool right = isRight(a, b, c);

  if( a == b || b == c )
    {
    return right ? RIGHT_ISOSCELES_TRIANGLE : ISOSCELES_TRIANGLE;
    }

  return right ? RIGHT_SCALENE_TRIANGLE : SCALENE_TRIANGLE;
  }
```

```
/**
 * Return 1 if the triangle is a right triangle, otherwise return 0.
 * @param a length of the first side
 * @param b length of the second side
 * @param c length of the third side
 * @return 1 if the triangle is a right triangle, otherwise return 0.
 *
 * pre-condition: a <= b && b <= c    <br>
 *
 * post-condition: return value indicates right triangle
 **/

static inline int isRight(double a, double b, double c)
  {
  return a*a + b*b == c*c ? 1 : 0;
  }


/**
 * Order numbers a and b so that a <= b.
 * @param a one of the two numbers to be ordered
 * @param b the other of the two numbers to be ordered
 *
 * pre-condition: true <br>
 *
 * post-condition: a <= b
 **/

static inline void order(double& a, double& b)
  {
  if( a > b )
    {
    double temp = a;
    a = b;
    b = temp;
    }
  // assert( a <= b );
  }
```

```
/**
 * Until end-of-file, read groups of three numbers and classify whether they
 * are all in range and could be the sides of a triangle, and if so,
 * what kind:
 *    [right] {equilateral, isosceles, scalene}
 *
 * @param in istream containing groups of three sides
 * @param out ostream on which results are shown
 **/

static void test(istream& in, ostream& out)
  {
  string inputSide[SIDES];
  double side[SIDES];

  // read numbers as strings, exit loop if end-of-file

  while( in >> inputSide[0] >> inputSide[1] >> inputSide[2] )
    {
    for( int i = 0; i < SIDES; i++ )    // echo sides
      {
      out << inputSide[i] << " ";
      }

    // convert inputs to numeric and show any bad ones

    bool inputOk = true;

    for( int i = 0; i < SIDES; i++ )
      {
      if( !Conversion::convert(inputSide[i], side[i]) )
        {
        inputOk = false;
        out << "\nbad input: " << inputSide[i];
        }
      }

    if( inputOk )
      {
      switch( analyze(side[0], side[1], side[2]) )
        {
        case NOT_A_TRIANGLE:           out << "not a triangle.";          break;
        case SCALENE_TRIANGLE:         out << "scalene triangle.";        break;
        case RIGHT_SCALENE_TRIANGLE:   out << "right scalene triangle.";  break;
        case ISOSCELES_TRIANGLE:       out << "isosceles triangle.";      break;
        case RIGHT_ISOSCELES_TRIANGLE: out << "right isosceles triangle.";break;
        case EQUILATERAL_TRIANGLE:     out << "equilateral triangle.";    break;
        }
      }
    out << endl;
    }
  }
};  // TriangleTester
```

White-Box in Conjunction with Verification Techniques

- Verification is based on techniques for reasoning about programs.
- These techniques can also be used to simplify the number of white-box test cases.

Using Live Assertions

- ```
  order(a, b);
  order(b, c);
  order(a, c);

  assert( a <= b && b <= c );
  ```

- The program will now tell us when the assumption is wrong.
- It will provide an indication of what has to be rethought.
  ```

Verification + W.B. Testing

- Could use verification to help fix the problem.
- Having *verified* the assertion

```
assert( a <= b && b <= c );
```

  will testing be simplified?
- How?

Some Ideas for More Effective Testability

Ideas for More Effective Testability

- **Instrument** your code as you build it; this could help with unit tests
  - Code-in traces, explanations, indications, …
  - Being able to turn instrumentation on or off can help understand whether sub-systems are working correctly.

Example: C++ Instrumentation

```
class Trace
{
static int currentLevel;

static void trace(string message, int level)
     {
     if( level >= currentLevel )
            cerr << message << endl;
     }

static void setLevel(int level)
     {
     currentLevel = level;
     }
};
```

Ideas for More Effective Testability

- Build **testing interfaces** into the code. These are interfaces that can be seen by the developer but not the user.

User Interface | Testing Interface

"Model" aspect of the product

- These interfaces allow greater automation in the testing process, since they can be driven by a testing program more readily than one requiring a user interface (such as a GUI or CLI).

Testing Interface

<table>
<tr><td>User Interface</td><td>Testing Interface</td></tr>
<tr><td colspan="2">“Model” aspect of the product</td></tr>
</table>

Build Self-Test Routines into the Code

- **Self-tests** can employ data generators that will stress test the code.

Classifying Programming Errors

Categories of Programming Errors

- Logic errors
- Pointer errors
- Numeric accuracy/precision/roundoff errors
- Input/output representation errors
- Data structure usage errors
- Memory errors
- User-interface errors (windows, etc.)
- Environment errors, such as misuse of file-system, devices, etc.
- Timing errors, such as in a real-time system

Exercise

- Each team take one category of error.
- List as many specific sub-types of this error as you can.
- **Example**: Logic Errors:

Logic Error Categories (1)

- Numeric and character boundary
  - Off-by-1
- Array-out-of-bound
  - Insufficient space allocated
  - Input or output buffer overflow
- Inequality comparison (used > instead of >= or <)
- Used ++ instead of --.
- Used pre-incrementation rather than post.

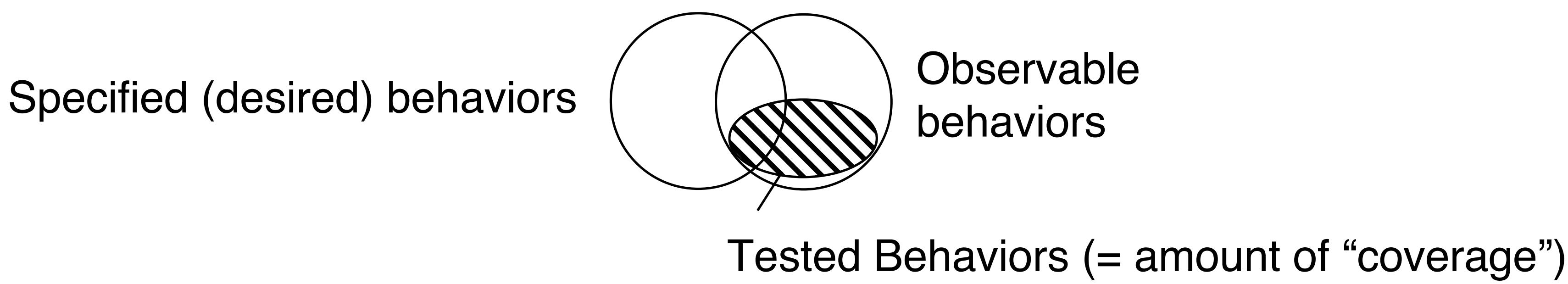
Logic Error Categories (2)

- Negation error
- Loop continuation criterion
  - Infinite loops
- Flag not cleared when used
- Flag, count, or sum not initialized
- Routine not reinitialized before subsequent use
- Dynamic type-casting exception

Theoretical Aspects of Whitebox Testing

White-Box Coverage

- **“Coverage” notion:**
  - View the program as a directed graph (flowchart or data-flow diagram)
  - Develop (external or internal) tests that “cover”, i.e. exercise program to various degrees.



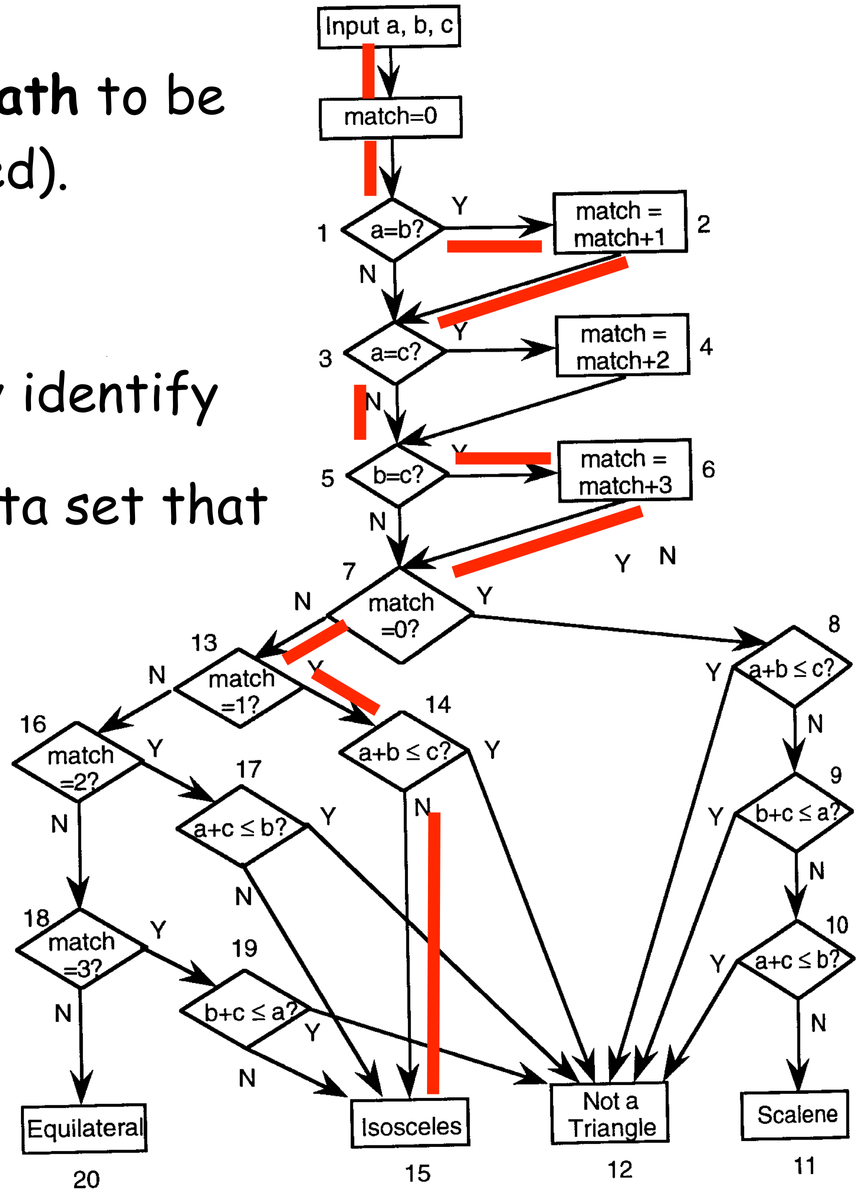
Minimal Coverage

- Tests should exercise, at least once, every:
  - variable
  - assignment box
  - decision box
  - simple path through flow chart

Testing Based on Paths

Example of **a single path** to be covered by test (in red).

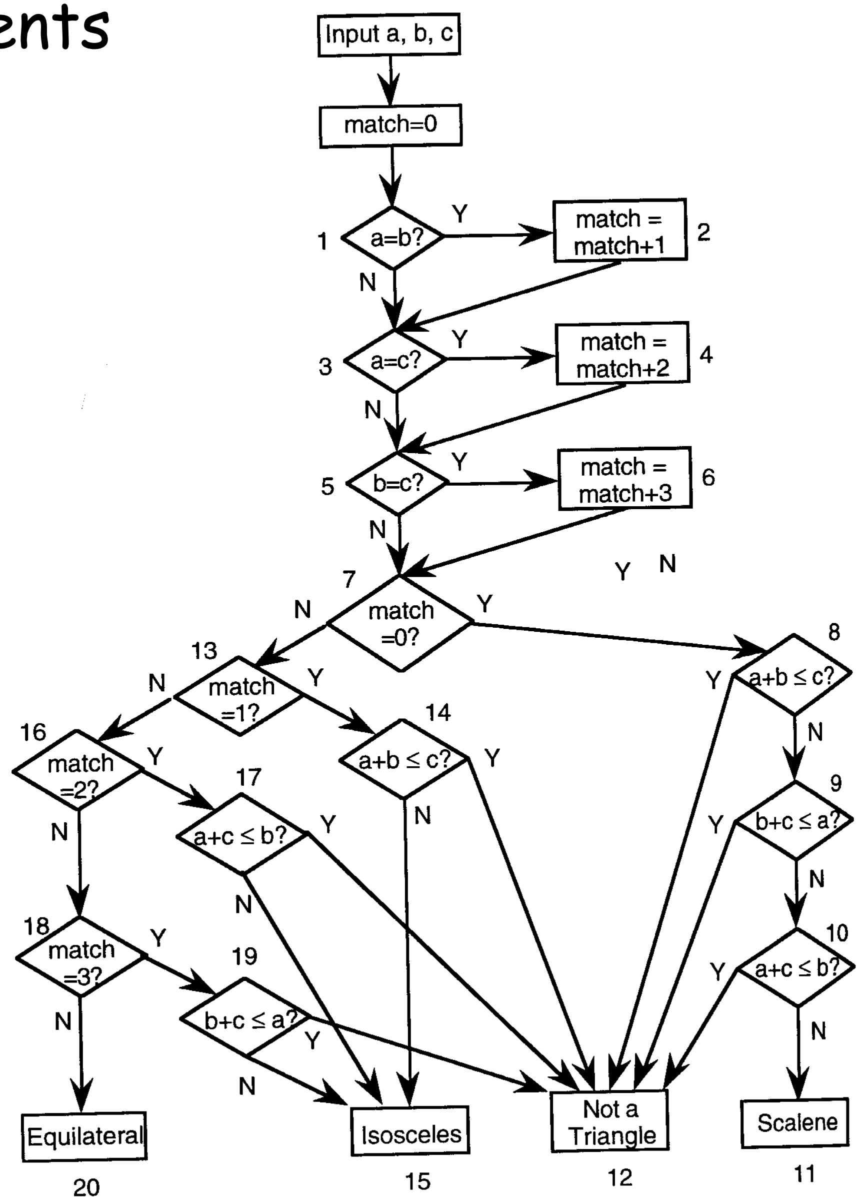
Insufficient to simply identify path; need to have data set that will exercise it.



Coverage "Basis"

- Infeasible to test ***all*** paths, etc.
- Instead, identify a "**basis**" from which all paths can be constructed.
- Make sure every element of basis is covered by ***some*** test
- **Example**: Basis could be set of all edges; Compute a set of tests that covers all.

Estimate the number of elements in a basis set of paths that will cover all edges.



D-D Path Nomenclature (Ed Miller 1977)

- D-D = "Decision to Decision"
- Path between two decisions, that contains no decision itself
- *Dependence* among D-D paths, e.g. a variable defined in one path is referenced in another.

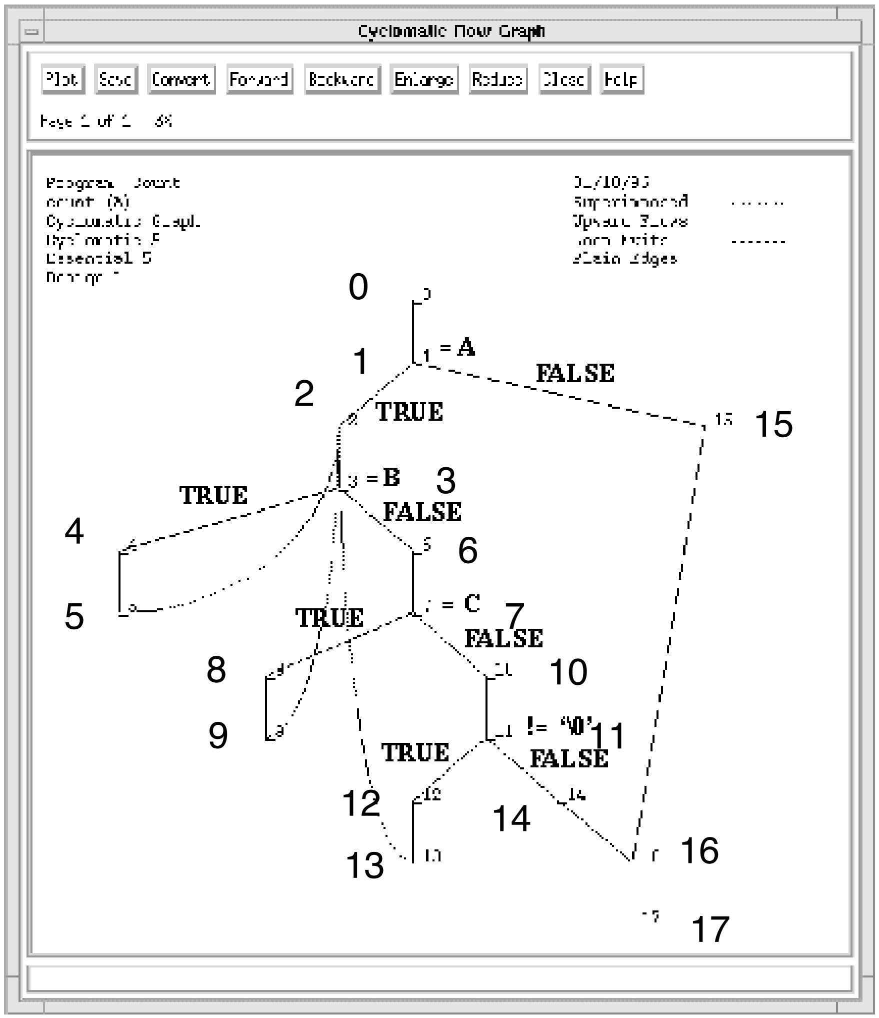
Classification of Structural Test Coverage

- C_0 Every statement tested
- C_1 Every D-D path tested
- C_1p Every **p**redicate & outcome tested
- C_d C_1 + every *inter-dependent pair* of D-D paths tested
- C_i^k Every path that contains up to k repetitions of a loop (e.g. k = 2) tested
- C_∞ Every path tested

"Baseline"-based method for constructing a basis

- Pick a single linear path through the program, the "baseline".
- Pick the next path by taking decisions alternate to the baseline.
- Repeat picking other alternates to the alternates, etc.
- Eventually a basis number of tests will be reached.

Baseline Testing Method
(in McCabe's Cyclomatic Tool)



Test Path 1 (baseline): 0 1 2 3 4 5 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

Test Path 2: 0 1 15 16 17

11(1): string[index]=='A' ==> FALSE

Test Path 3: 0 1 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

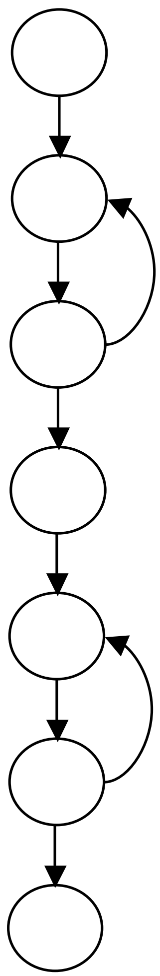
Test Path 4: 0 1 2 3 4 5 2 3 6 7 8 9 2 3 6 7 10 11 14 16 17

Complexity-Based Testing

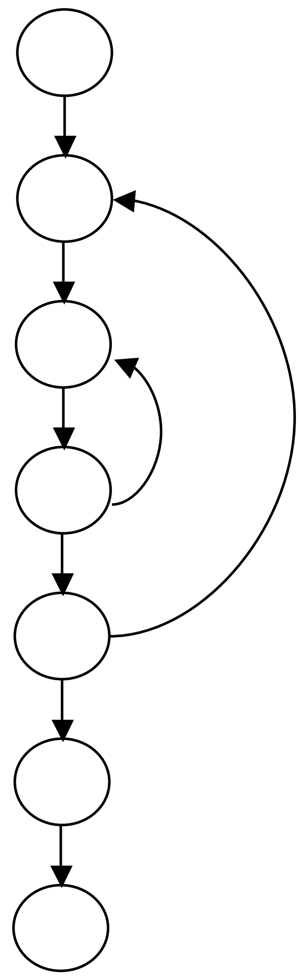
- Construct the flowchart for the program.
- Calculate the graph complexity C.
- **Find C independent paths** and corresponding test data for each.
- Execute program on test data.
- Check results with what is expected.

Classifying Loop Complexity (Informal)

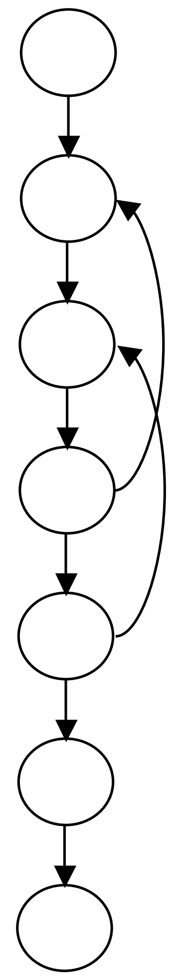
Concatenated



Nested



Intersecting



Program Graph
Complexity Measures

- Used to determine ***how much*** testing required for a given sub-system

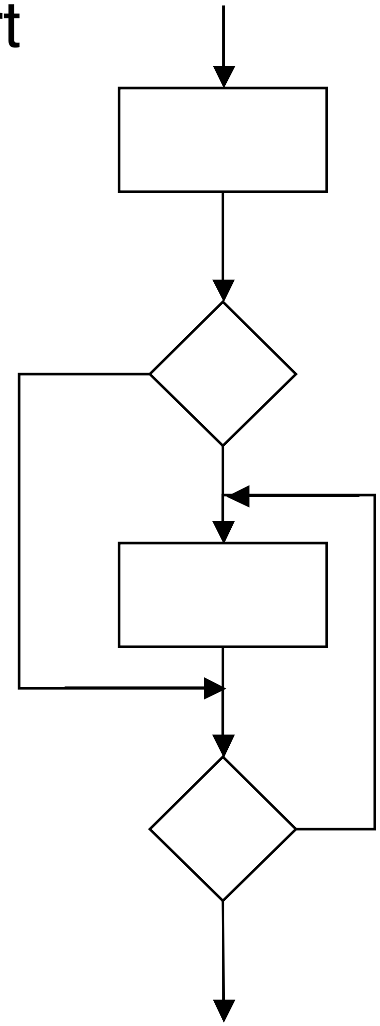
- Examples
  - McCabe Cyclomatic complexity
  - Halstead software metric

McCabe Cyclomatic complexity for a program graph

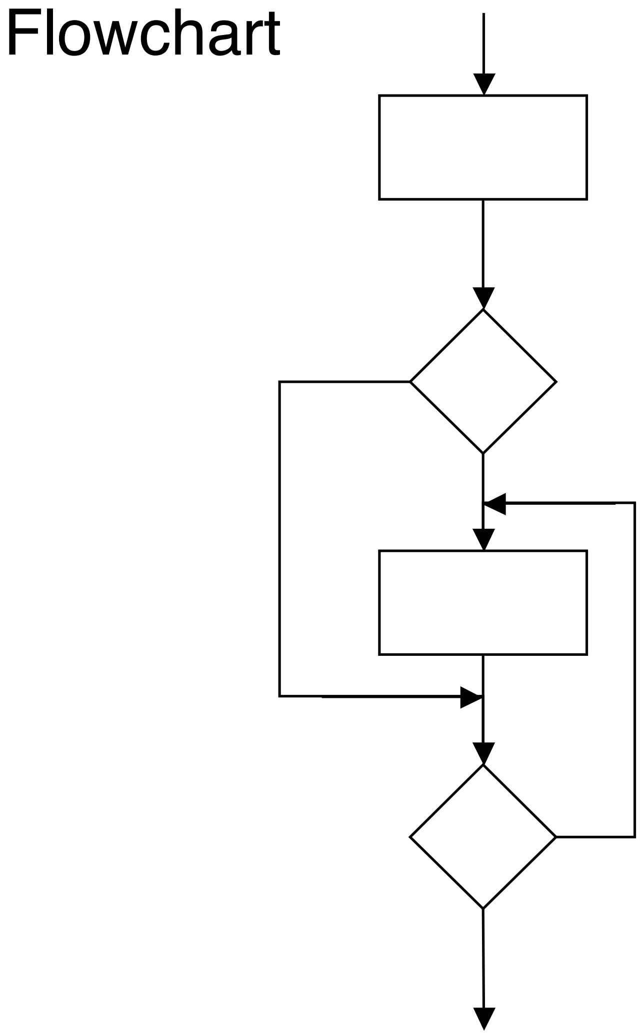
- IEEE Trans. Softw. Engrg., Dec. 1976
- Popular, but theoretically questionable
- $V(G) = e - n + p$
  - e is the number of **edges** (arcs), which represent data processing **boxes**
  - n is the number of **nodes**, which represent **points** where edges are connected
  - p is the number of **separate parts** (connected sub-graphs)

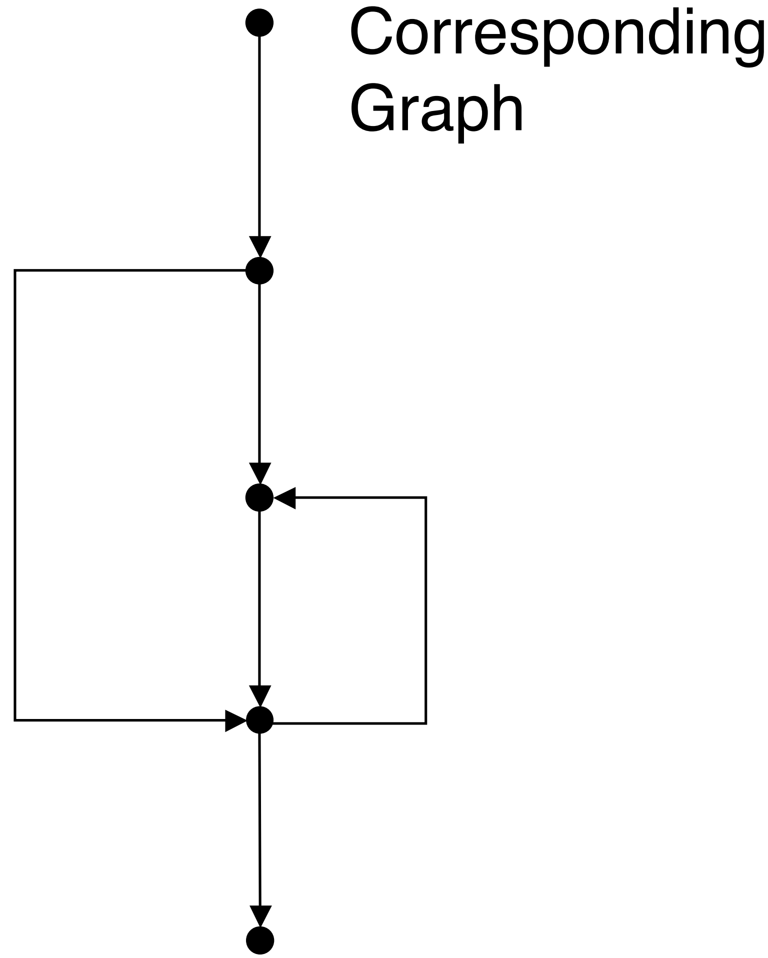
Example

Flowchart

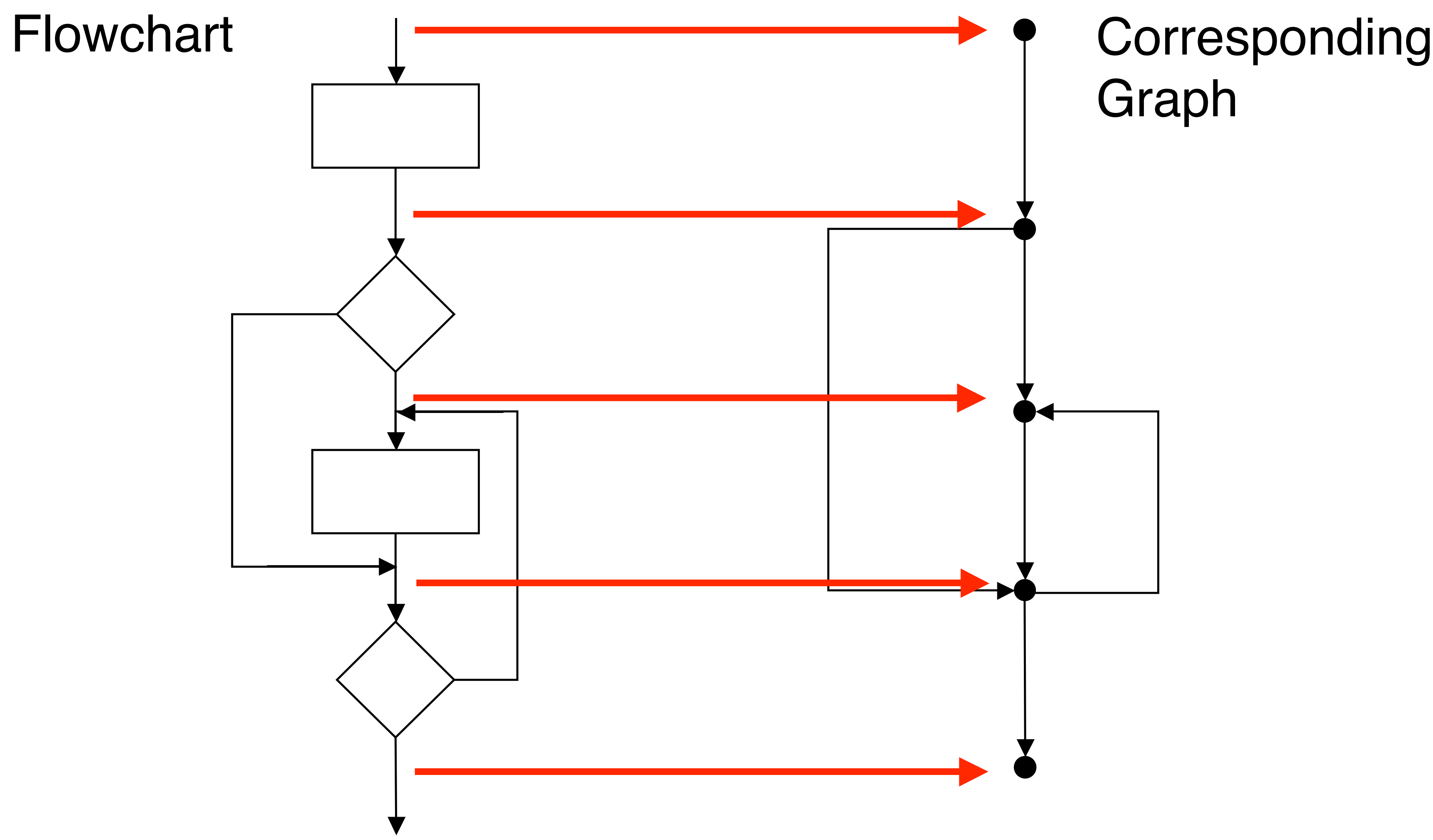


Example

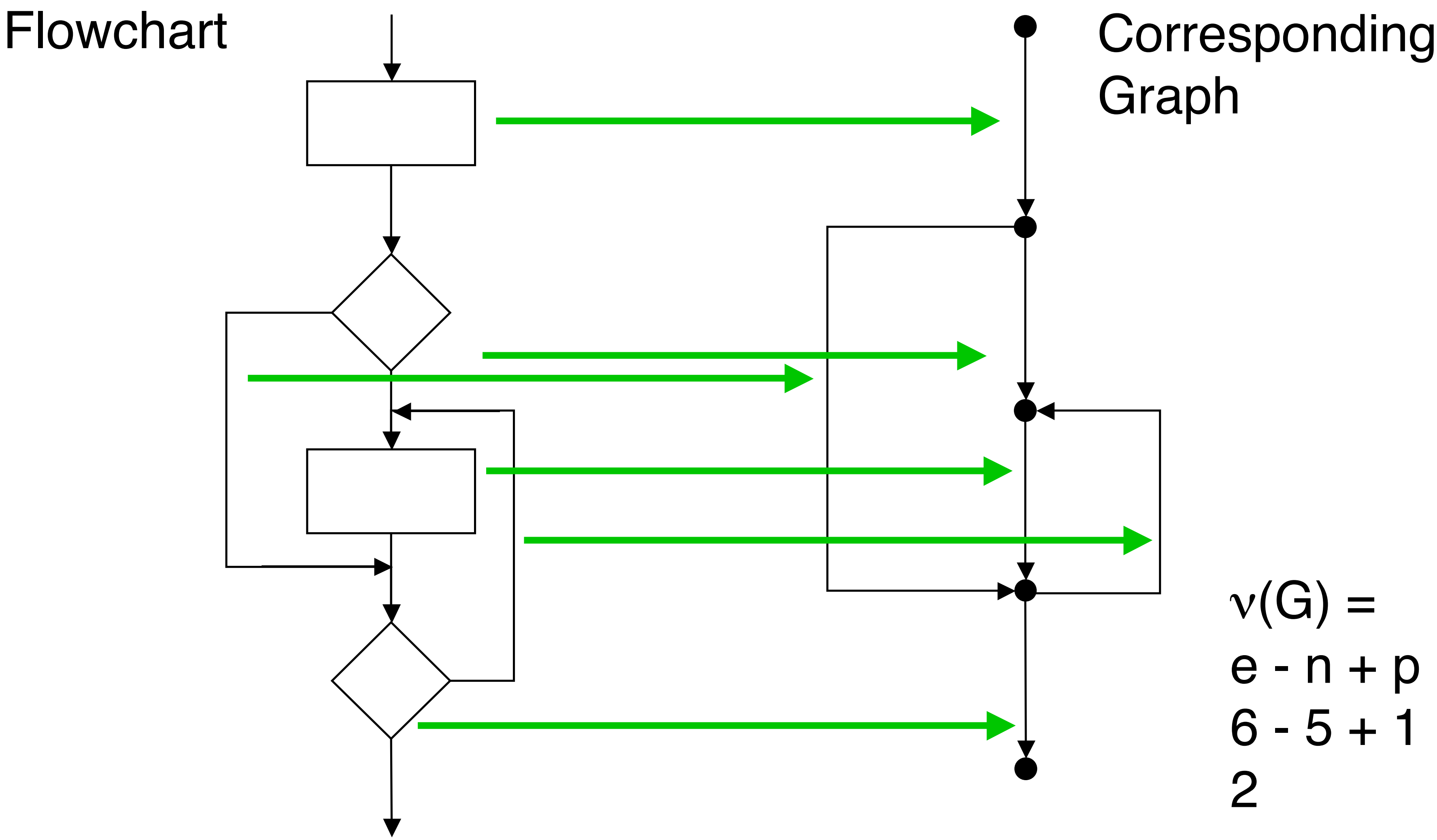




Node Correspondence



Edge Correspondence



$\nu(G) =$
$e - n + p =$
$6 - 5 + 1 =$
2

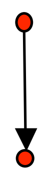
McCabe's measure drawn from Cyclomatic Number of a Graph

- Defined by Berge (*Theory of graphs and its applications*, 1962), motivated by Euler
- Originally defined for *undirected* graphs
- $\nu(G)$ = edges - nodes + separate parts
- **Property**: *Connecting* two nodes increases ν value by 1 if there was a path between the two nodes; otherwise it leaves ν value the same.

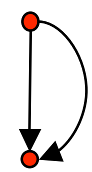
Illustration of Property

Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.

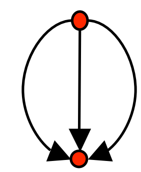
$\nu = 0 - 2 + 2 = 0$



$\nu = 1 - 2 + 1 = 0$

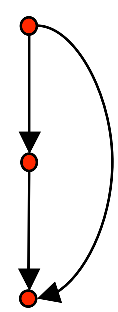


$\nu = 2 - 2 + 1 = 1$

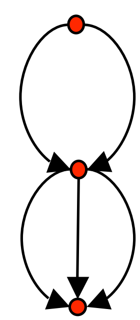


$\nu = 3 - 2 + 1 = 2$

$\nu = 2 - 3 + 1 = 0$



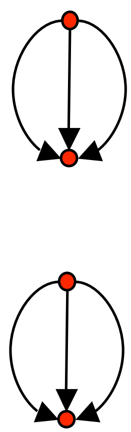
$\nu = 3 - 3 + 1 = 1$

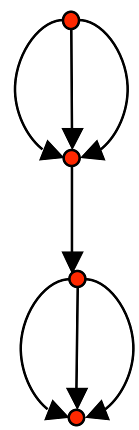


$\nu = 5 - 3 + 1 = 3$

Illustration of Property

Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.





$\nu = 6 - 4 + 2 = 4$

$\nu = 7 - 4 + 1 = 4$

Further Properties of the Cyclomatic *Number* of a Graph

- $\nu(G) = 0$ iff G contains no ***undirected*** cycles

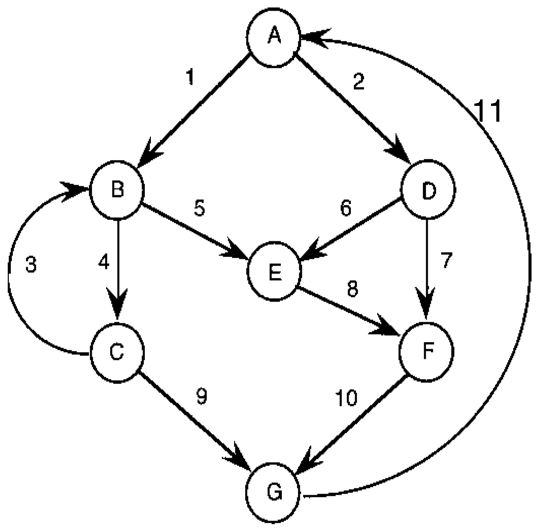
- If G is strongly connected (has only one component), then $\nu(G)$ is the maximum number of linearly- independent undirected cycles.

  [This can be used to compute size of a basis.]

Linear Independent Circuits?

- Think of the edges of a graph as components of a **vector**.
- A circuit is abstracted by the number of times each edge is traversed in the (undirected) circuit.
- One circuit can be constructed from others by *adding* (mod 2) the corresponding vectors.

Adding Circuits



c1 = 1-4-9-11 = (1 0 0 1 0 0 0 0 1 0 1)

c2 = 2-7-10-11 = (0 1 0 0 0 0 1 0 0 1 1)

c3 = 1-4-9-10-2-7 = (1 1 0 1 0 0 1 0 1 1 0)

$\nu(G) = e - n + p$

$= 11 - 7 + 1$

= **5**

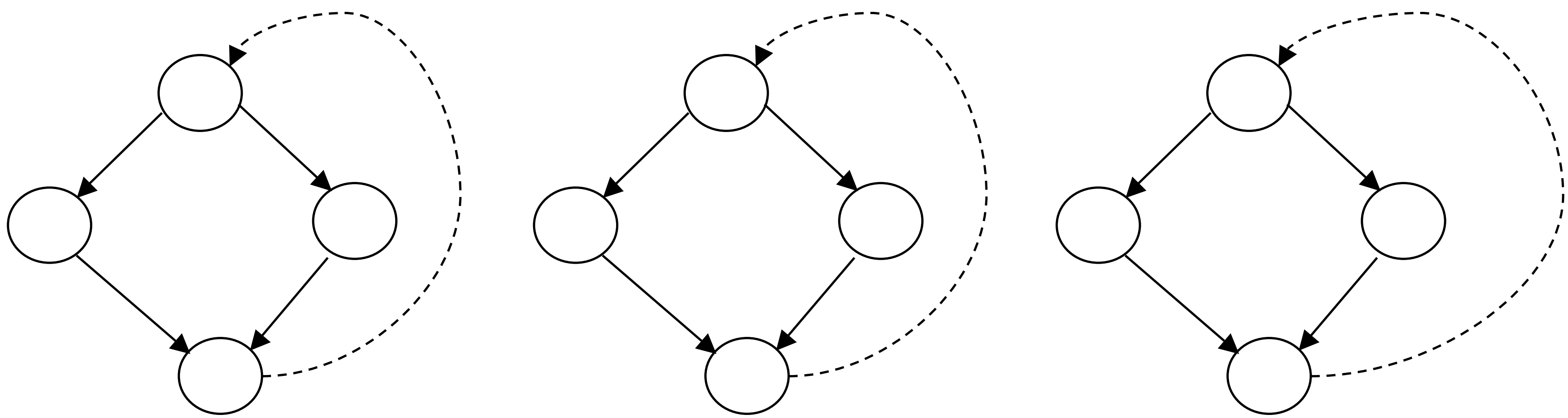
c3 = c1 + c2
c1 and c2 are linearly independent, etc.

5 circuits form a **basis**

McCabe Complexities of Typical Program Graphs

- McCabe: If a part of a graph is not strongly connected, add a **phantom arc** from finish to start;
- *Alternatively*, use the *modified* formula
  $\mu(G) = e - n + 2p$
- Above, 2p accounts for one *phantom* arc in each separate part.

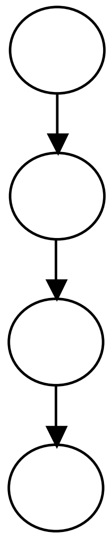
Adding one phantom edge per separate part



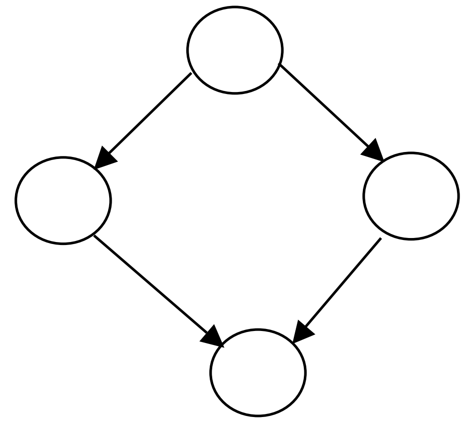
$\mu(G) = e - n + \mathbf{p}$ if phantom edges *counted* in e

$\mu(G) = e - n + \mathbf{2p}$ if phantom edges *not counted* in e

Complexities of Typical Program Graphs



$$\mu(G) = e - n + 2p$$
$$= 3 - 4 + 2$$
$$= 1$$

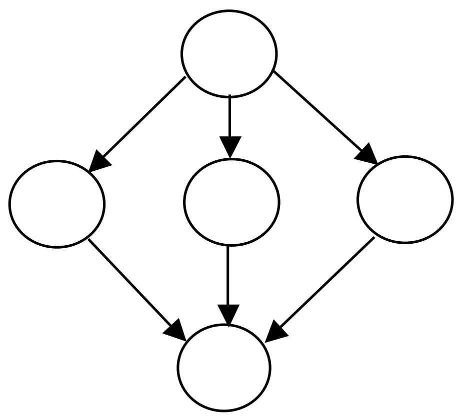


$$\mu(G) = e - n + 2p$$
$$= 4 - 4 + 2$$
$$= 2$$

Straight-line programs have $\mu(G) = 1$.

Two-way branch programs have $\mu(G) = 2$.

Complexities of Typical Program Graphs



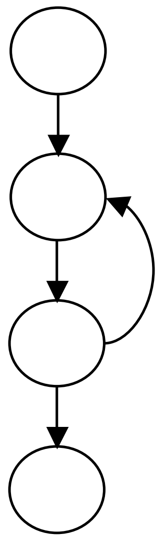
$\mu(G) = e - n + 2p$
$= 6 - 5 + 2$
$= 3$

Adding a branch increases e-n by 1, so

Simple k-way branch programs have $\mu(G) = k$.

Three-way branch programs have $\mu(G) = 3$.

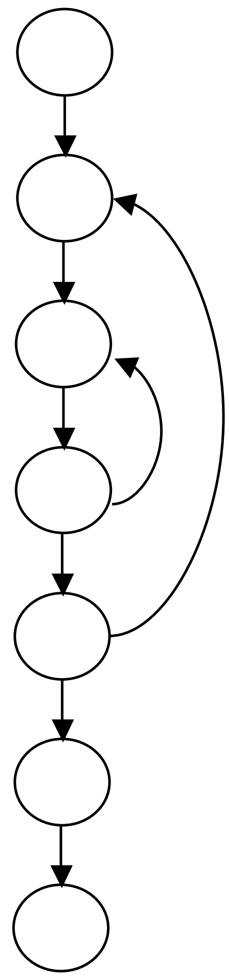
Complexities of Typical Program Graphs



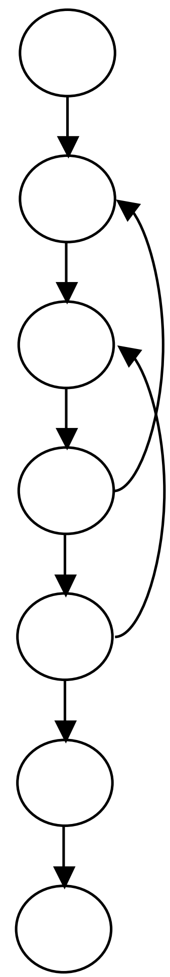
$$\mu(G) = e - n + 2p = 4 - 4 + 2 = 2$$

'while' programs have $\mu(G) = 2$.

Nested



Intersecting

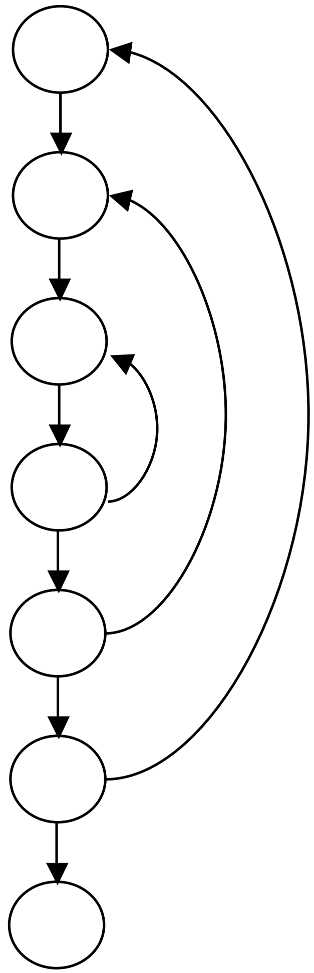


These have the same complexity. This may be undesirable.

$$\mu(G) = e - n + 2p = 8 - 7 + 2 = 3$$

Complexities of Typical Program Graphs

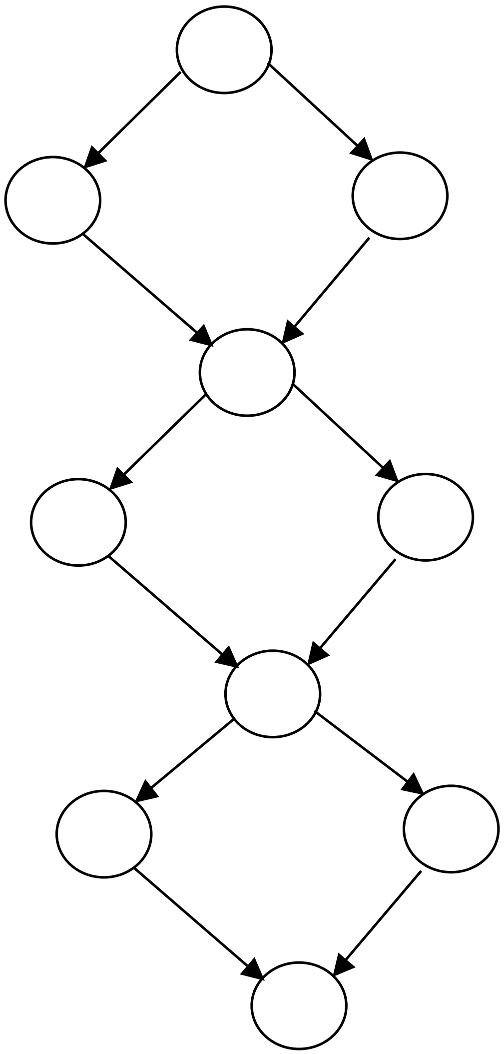
Triply-Nested



$$\mu(G) = e - n + 2p$$
$$= 9 - 7 + 2$$
$$= 4$$

Simple k-tuply nested programs have $\mu(G) = k+1$.

Complexities of Typical Program Graphs



$\mu(G) = e - n + 2p$

$= 12 \quad 10 + 2$

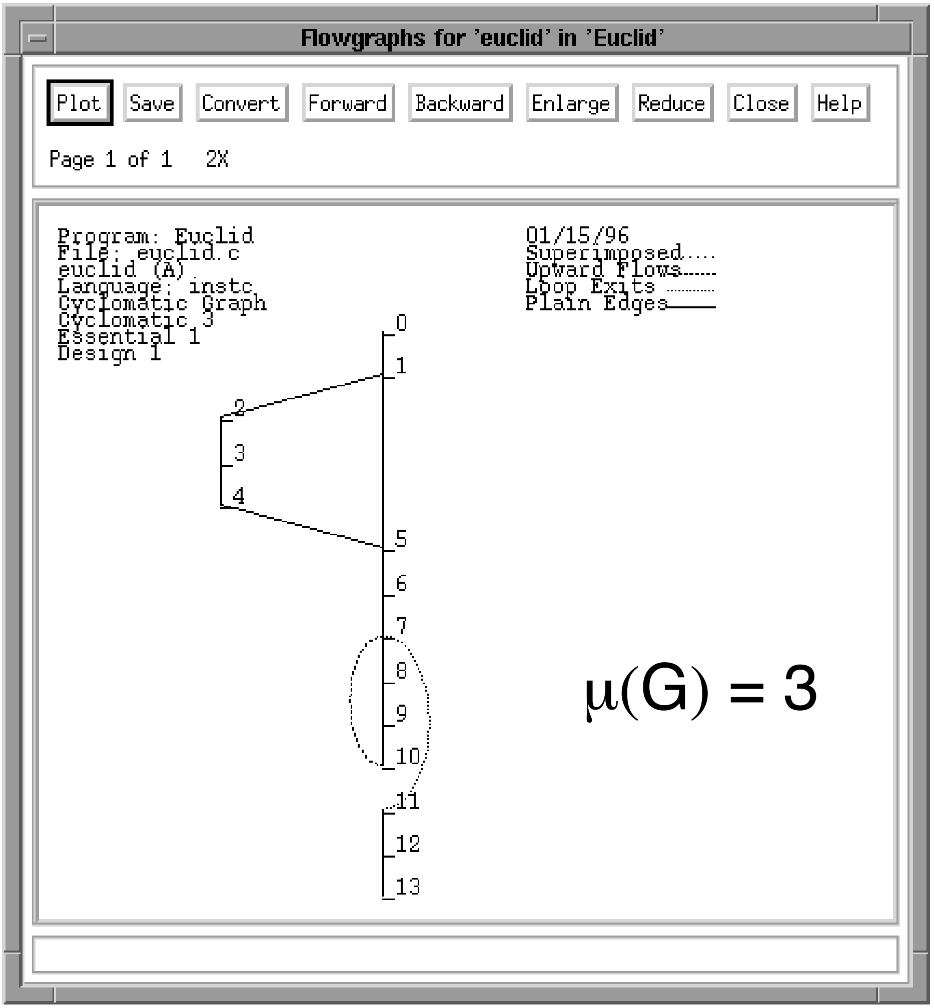
$= 4$

k-section "lattices" have $\mu(G) = k+1$.

Uses of complexity $\mu(G)$

- $\mu(G)$ can be used to indicate the **minimum number of test cases required**.
- Keep $\mu(G)$ small (e.g. < 10) for "understandable" programs.
- Certain constructs (e.g. switch) are exempted from the count.

Cyclomatic Tools




```
Module: euclid
Basis Test Paths: 3 Paths

Test Path B1: 0 1 5 6 7 11 12 13

        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> FALSE

Test Path B2: 0 1 2 3 4 5 6 7 11 12 13

        8(    1): n>m ==> TRUE

        14(    7): r!=0 ==> FALSE

Test Path B3: 0 1 5 6 7 8 9 10 7 11 12 13

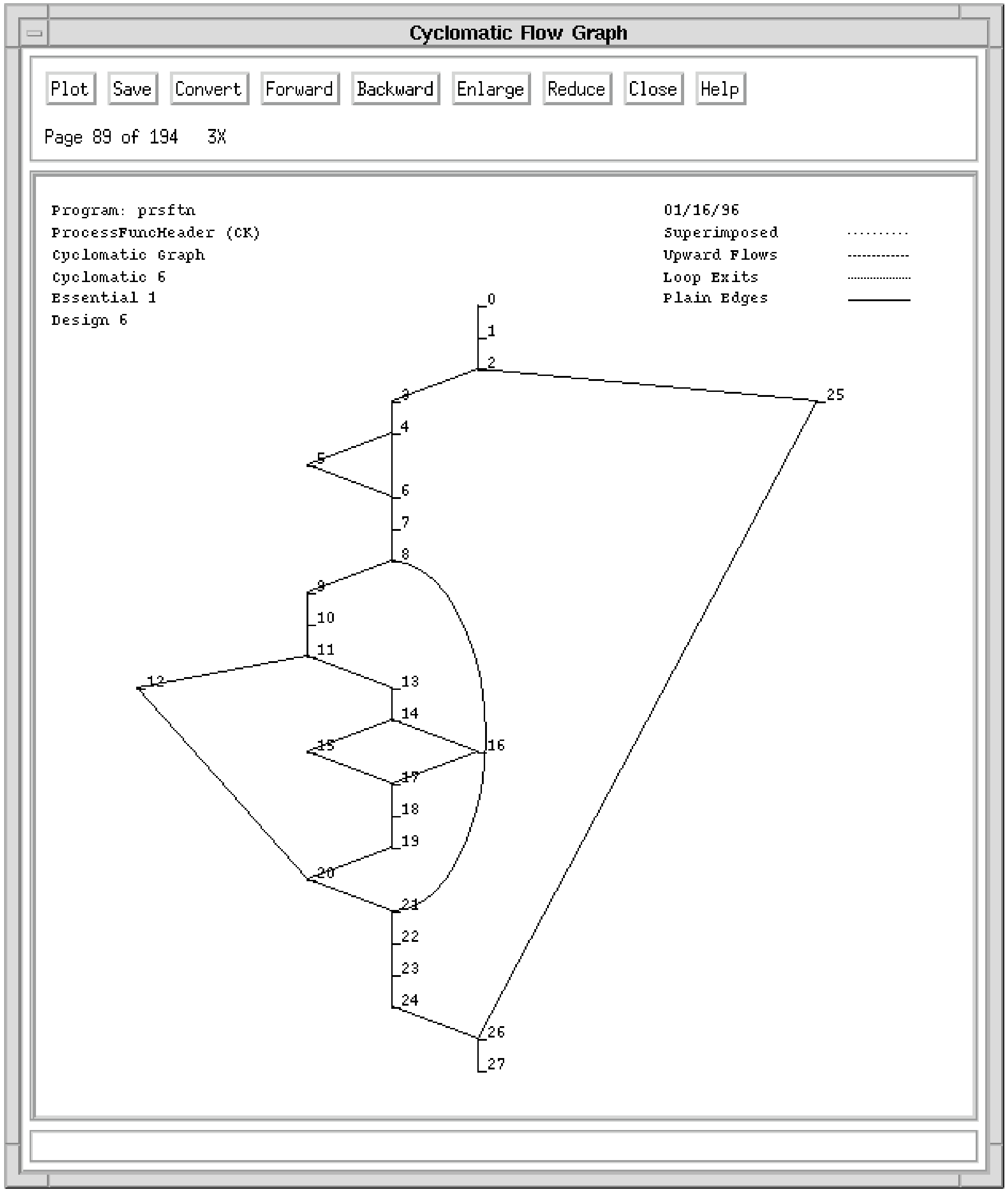
        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> TRUE

        14(    7): r!=0 ==> FALSE
```

Source: Watson & McCabe, 1996

$\mu(G) = 6$

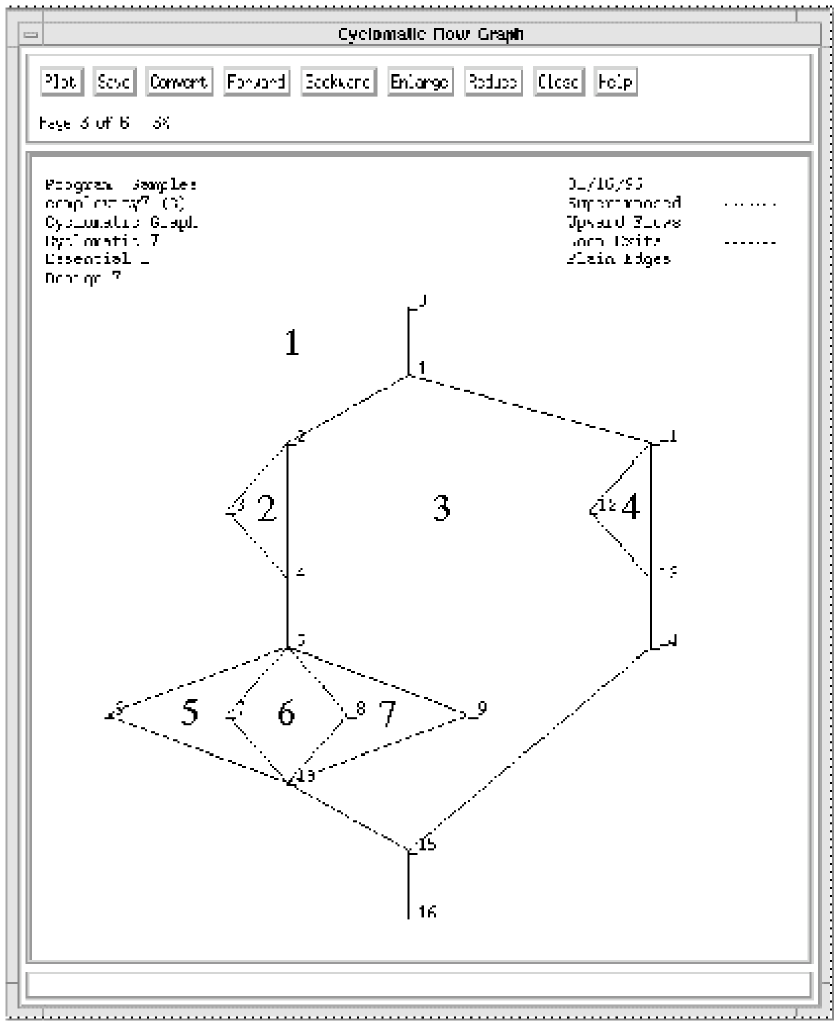

Cyclomatic Flow Graph
Plot
Save
Convert
Forward
Backward
Enlarge
Reduce
Close
Help
Page 89 of 194 3X
Program: prsftn
ProcessFuncHeader (CK)
Cyclomatic Graph
Cyclomatic 6
Essential 1
Design 6
01/16/96
Superimposed
Upward Flows
Loop Exits
Plain Edges
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Simplified Complexity Measures

- Counting binary predicate tests only:
  $\eta(G)$ = predicates + 1

- Counting ***regions*** = e - n + 2p (Euler's formula)

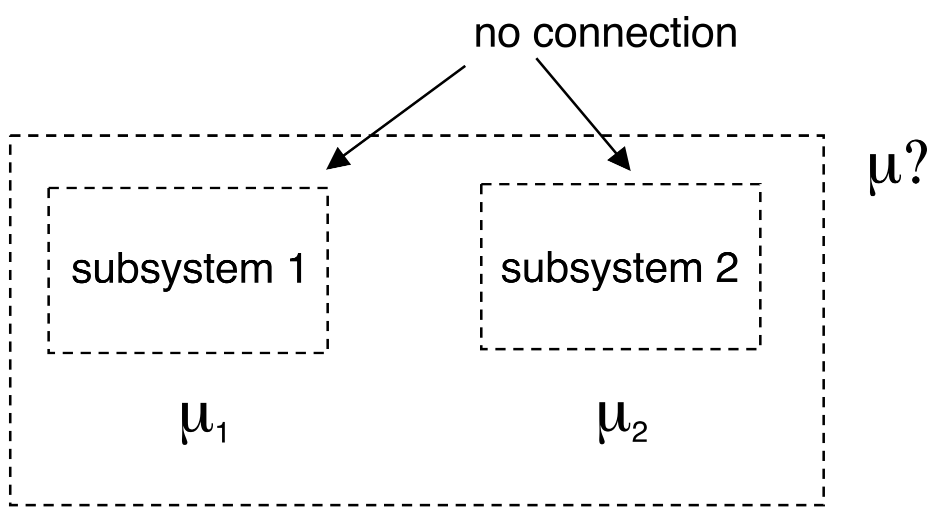
Counting Regions



regions numbered

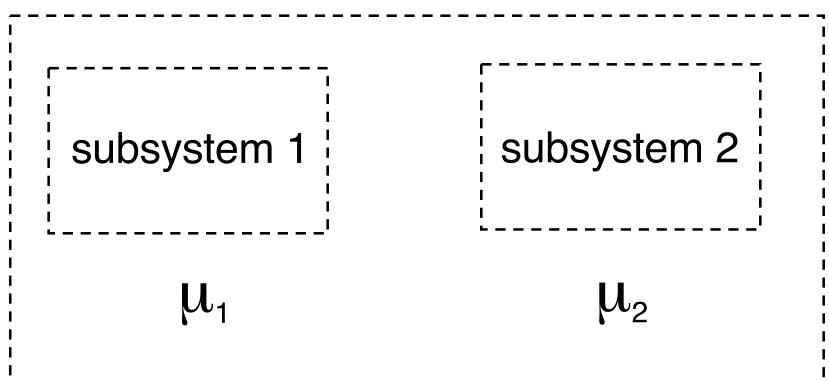
Use of $\mu(G)$ in
Integration Testing

- Composition of modules' complexity:
  what is μ of overall system in terms of individual μ's?



Use of $\mu(G)$ in Integration Testing

- Composition of modules' complexity: what is μ of overall system in terms of individual μ's?



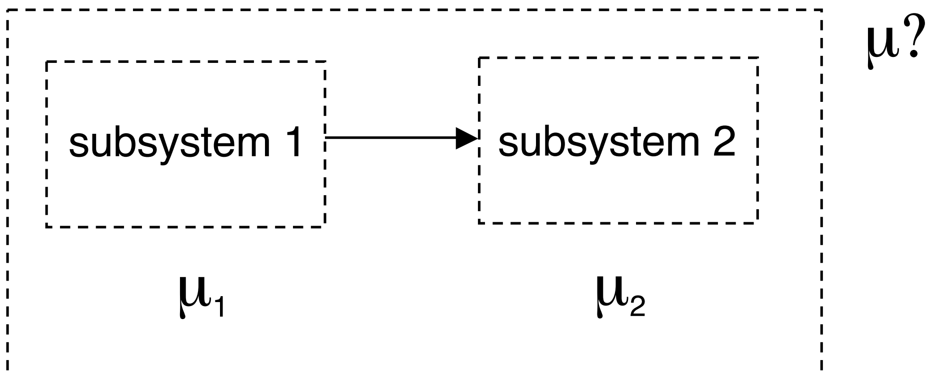
μ?

$$e = e_1 + e_2$$
$$n = n_1 + n_2$$
$$p = p_1 + p_2$$
$$\overline{\mu = \mu_1 + \mu_2}$$

$\mu_1(G) = e_1 - n_1 + 2p_1$ $\mu_2(G) = e_2 - n_2 + 2p_2$

Use of μ(G) in Integration Testing

- What if connected by one edge?



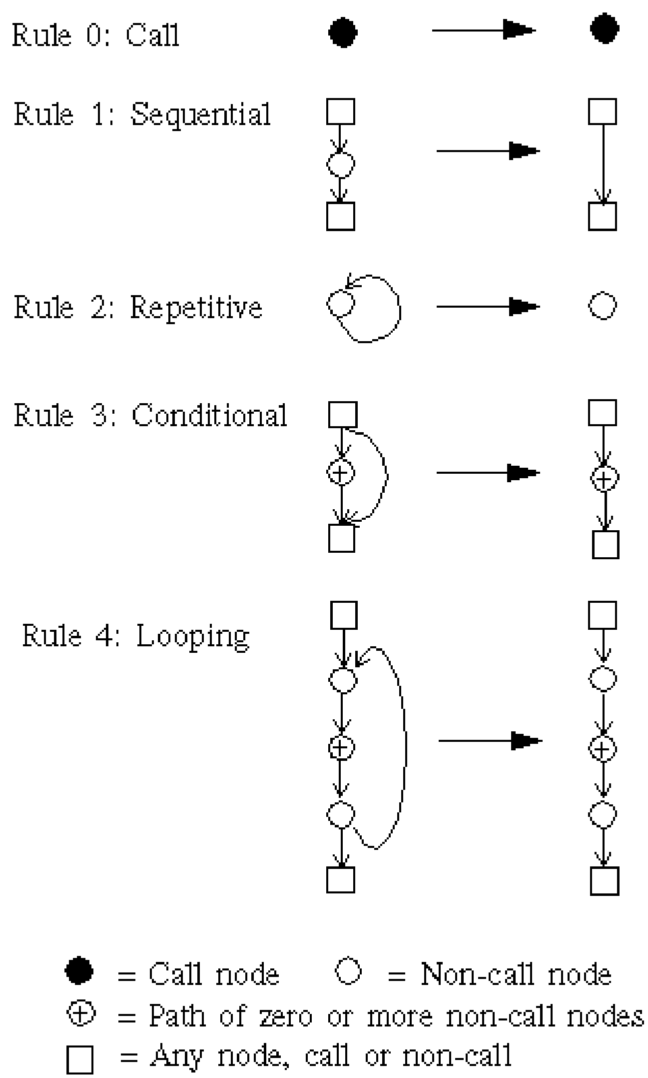
$$\mu = \mu_1 + \mu_2 + 1 \ -1$$

one fewer separate part

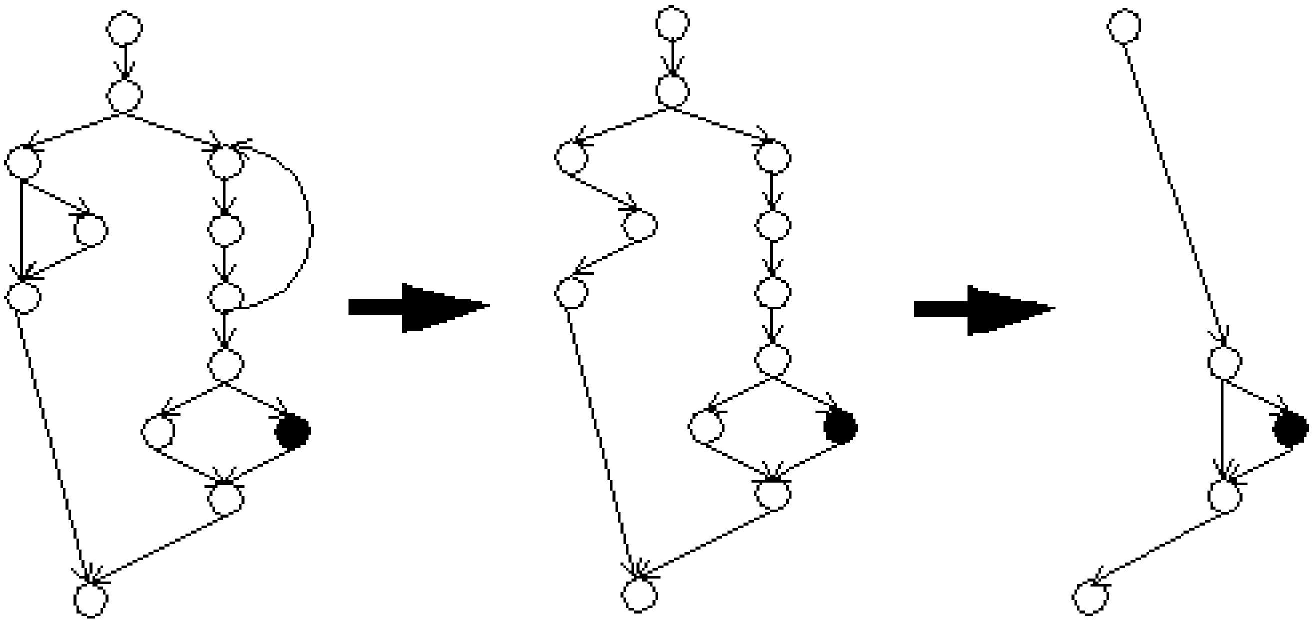
one more edge

So $\mu = \mu_1 + \mu_2$, even if the subsystems are singly connected.

"Design-Reduction" Technique: Lumping hierarchically



“Design-Reduction” Technique



Original graph,
v(G) = 5

Intermediate graph,
after rules 3 and 4
have been applied

Reduced graph,
v(G) = 2

○ = Non-call node ● = Call node

Software Test Standards

(Order from http://www.12207.com/test1.htm)

IEEE 1008	Software Unit Testing
IEEE 1044	Classification for Software Anomalies
ISO/IEC 12207	Information Technology--Software Life Cycle Processes
ISO/IEC TR 15271	Guide for ISO/IEC 12207 (Software Life Cycle Processes)
AECL CE-1001-STD REV.1	Standard for Software Engineering of Safety Critical Software
BSI BS-7738	Specification for Information Systems Products Using SSADM (Structured Systems Analysis and Design Method)
BSI BS-7925-1	Software Testing - Vocabulary
BSI BS-7925-2	Standard for Software Component Testing
DEF 00-55 (Part 1)/1	The Procurement of Safety Critical Software in Defence Equipment-Requirements
DIN VDE 0801	Principles for Computers in Safety-Related Systems
ESA PSS-05-01	Guide to the Software Engineering Standards, Issue 1
German Process-Model (V-Model)	Software Life-Cycle Process Model
IAEA TRS-372	Development and Implementation of Computerized Operator Support Systems in Nuclear Installations
IEC 60601-1-4	Medical Electrical Equipment--Part 1: General Requirements for Safety-4. Collateral Standard: Programmable Electrical Medical Systems
IEC 60880	Software for Computers in the Safety Systems of Nuclear Power Stations
IEE 5	Software Inspection Handbook
NCC	STARTS Purchasers' Handbook--Procuring Software-Based Systems
NRC NUREG/BR-0167	Software Quality Assurance Program and Guidelines
RTCA DO-178B/ED-12B	Software Considerations in Airborne Systems and Equipment Certification