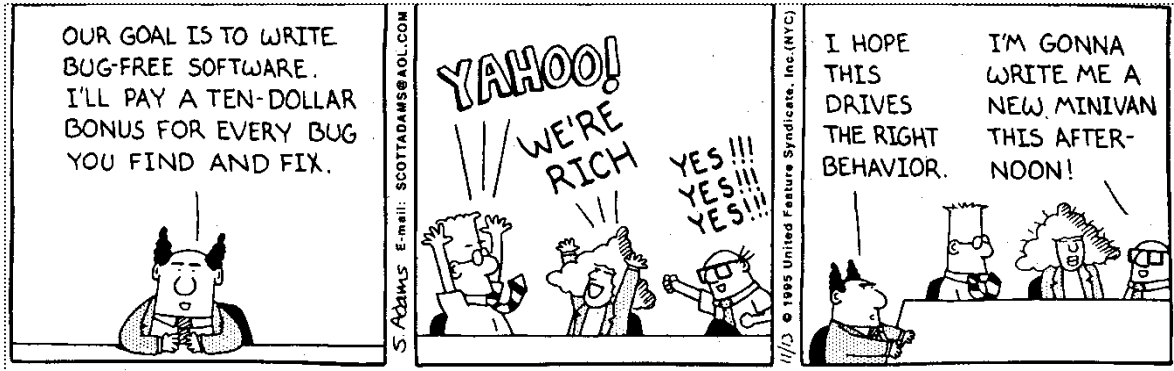
Testing During Implementation (Schach, Chap 15)

GSM2 Shane K. Hightower, a Ft. Pierce, Fla., native, monitors the damage control reporting system aboard USS Yorktown. This computerized system collects data on the status of the ship and feeds it through the local area network to five control stations, eliminating the need for messengers and phone talkers at various damage control stations. Should one control station be disabled, the others can still manage any battle damage problem.

- Software glitches leave Navy "smart ship" dead in the water.
  - USS Yorktown towed to Norfolk due to a database overflow caused by the propulsion system (Slabodkin July '98)



Testing During Implementation

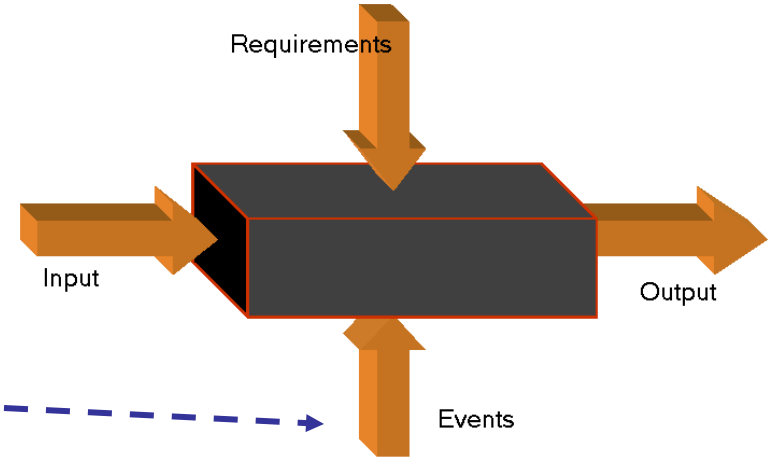
- Once Source Code Available, can test code's Execution
  - ***One way*** — Arbitrary input; see what happens.
  - ***Needed*** — Systematic test case development
- **Regression testing:** Re-run previously passed test cases

  Make sure system modifications didn't break something that used to work.

- ESA lost Ariane 5 rocket due to numerical precision in inertial reference system *(Gleick 96)*
  - *64 bit floating point horizontal velocity converted to a 16 bit signed int.*
  - *Conversion over 32,767 failed 37 seconds after liftoff ($500 Million).*

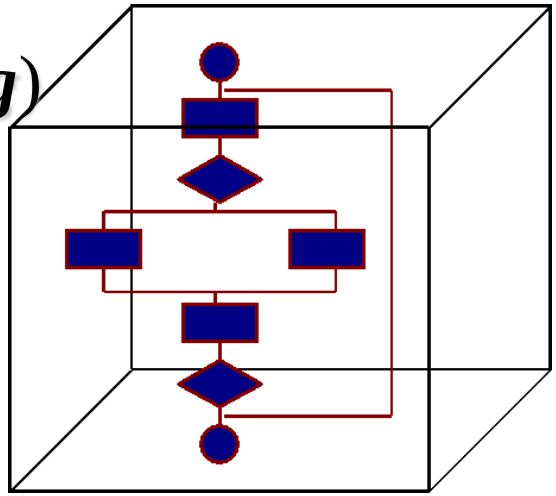
Two Approaches to Testing

- Testing To Specifications (aka ***Black-Box Testing,*** *functional testing)* Focus: what module is *supposed* to do, not how it does it.
  - Info for test cases comes from specification (ie. your acceptance test plan)



Events: External Conditions that set the Context of the module

- Testing To Code (aka ***Glass-Box Testing***)
  - Focus: how code in module is structured, not what its supposed to do
  - The code itself is tested, w/o regard to specifications



Feasibility of Complete Black-Box/Glass-Box Testing

- Dijkstra [1972]: "Testing can show the Presence of Bugs, but is hopelessly inadequate for showing the Absence of Bugs."

- The **Art** of Testing:
  - Want: A small, manageable set of test cases:
    - Maximize Chances of Detecting Fault, While
    - Minimizing Chances of Wasting Testing $$$

Goal: Construct every test case so as to Detect Previously Undetected Fault (ie., minimize overlap between test cases).

Equivalence Classes and Edge Cases

- Acceptance test (ex: black box testing) => handle any number of input values in the range 1 … 16,383
  - **Basic idea**: If system works for one test case in the range (1..16,383), then will probably work for any other test case in that range;
    - so, don't waste $$$ with nearly redundant testing.
    - Instead focus on *equivalence classes* and *edge cases*

Break 1..16,383 into three equivalence classes:

{ …, -1, 0,} {1, 2, …, 16382, 16383,} {16384, … }

Equivalence Class 1: Fewer than 1 record

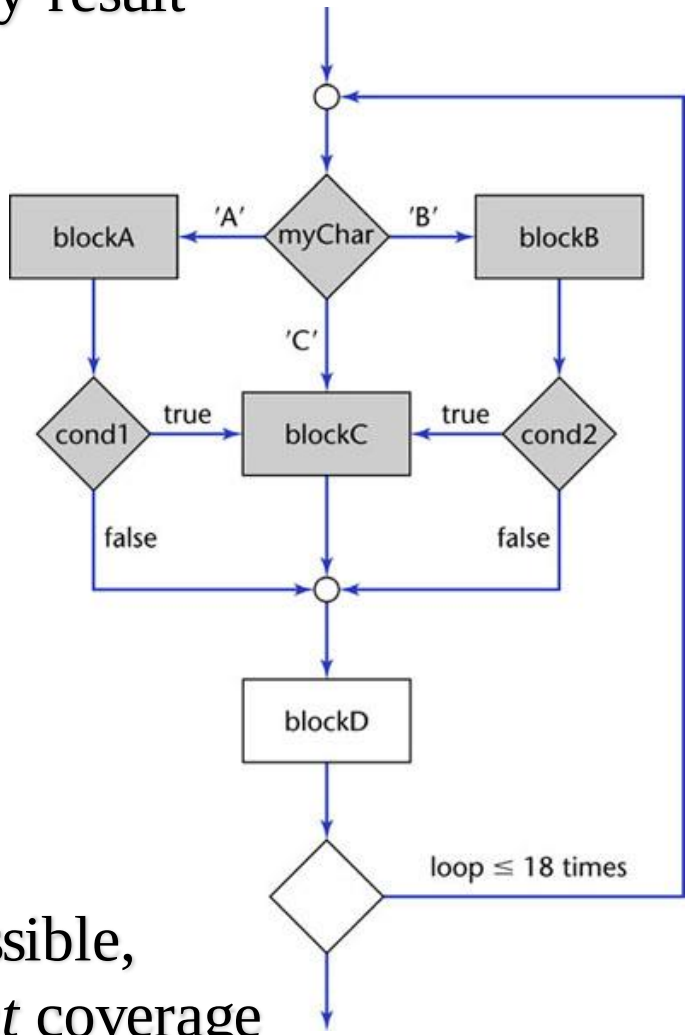
Class 2: Between 1 and 16,383 records

Class 3: More than 16,383 records

Glass Box testing (testing to code)

- If it is desired that *each* path through module be executed at least once, combinatorial explosion may result

```
read (kmax)        // kmax in an int between 1..18
for (k = 0; k < kmax; k++) do
{
   read (myChar)   // myChar is A, or B, or C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```



- Note: other coverage is possible, such as *branch* or *statement* coverage

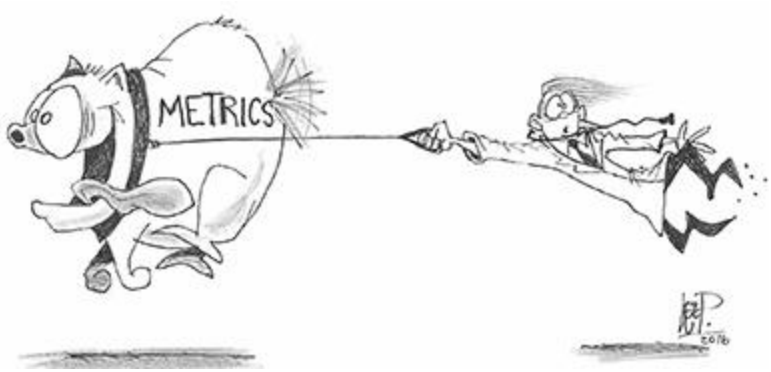
Fault Distribution is not Uniform

- [Myers]: 47% of faults in OS/370 were in only 4% of the modules

- [Endres]: DOS/VS (Release 28):
  - 512 faults in a total of 202 modules
    - 112 of the modules had only one fault
    - There were modules with 14, 15, 19 and 28 faults, respectively
      - The latter three were the largest modules in the product, with over 30000 lines of DOS macro assembler language
      - The module with 14 faults was relatively small, and very unstable. What should be done with this module?

Complexity Metrics: Making Testing Manageable

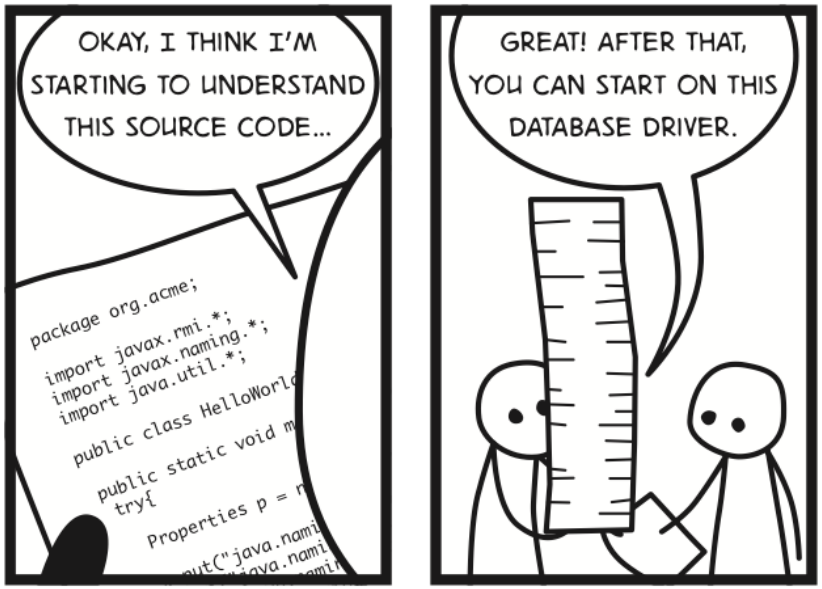
- **Goal** of Using a Software Complexity Metric:
  - Highlight Modules Mostly Likely To Have Faults



- Quality Assurance approach to Testing
  - Would be beneficial to be able to say, "Module M1 is More "Complex" than Module M2"
- Problem: what do you do when you discover an unreasonably high Complexity Value for a Module?

Lines of Code as a Complexity Metric

- Simplest Complexity Measure; Underlying Assumption:
  - There exists a Constant Probability *p* that Line of Code Contains Fault. Based on the idea that the past can be used to predict the future.
  - Example:
    - Tester Believes Line of Code Has 2% Chance of Containing Fault.
    - Module Under Test is 100 Lines Long, *Probably* Contains 2 Faults



McCabe's Cyclomatic Complexity Metric

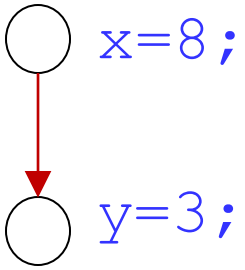
- Cyclomatic Complexity Metric M (McCabe, 76)
  - Essentially the # of Decisions in Module
    - M = #edges - #nodes +2
  - Can be used as a Metric for predicting the # of Test Cases needed for Branch Coverage

- M Value for Aegis System (Walsh,79)
  - 276 modules in Aegis
  - 23% of modules with M > 10 contained 53% of detected faults
  - Modules with M > 10 had 21% more faults per line of code

- Industry consensus: Re-design/implement modules with M > 10

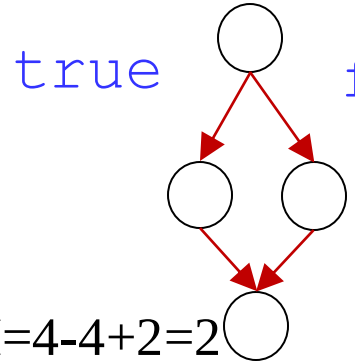
Statement to Graph Conversions

sequence



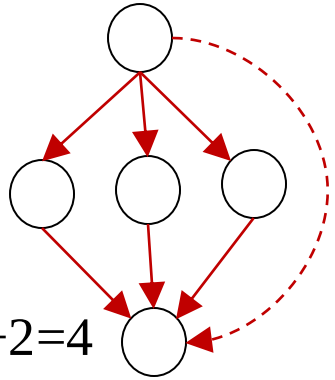
M=1-2+2=1

selection (if-else)



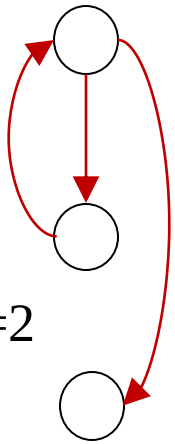
M=4-4+2=2

switch (3 explicit cases + implicit default)



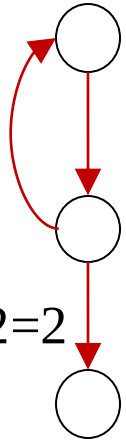
M=7-5+2=4

while loop



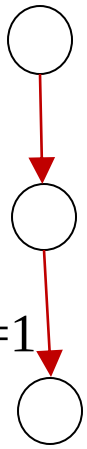
M=3-3+2=2

repeat – until loop



M=3-3+2=2

function call `foo()`



M=2-3+2=1

McCabe's Metric -> M = #edges - #nodes +2

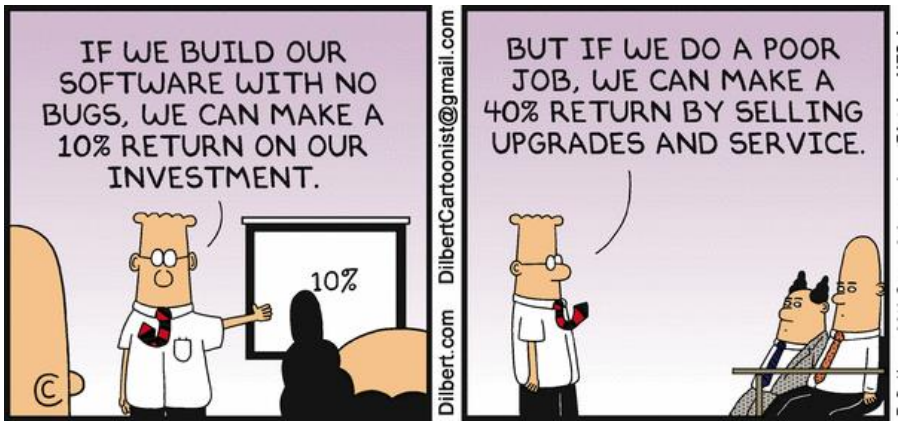
ICE: Applying McCabe's Metric

1. Use statement to graph conversions to build a graph representing the source code
2. Count num edges (#e), num nodes (#n)
3. Compute McCabe's Metric M = #e-#n+2
4. M > 10 is overly complex. Consider Re-designing Module
5. Graph gives insight on how to reduce complexity.
6. M value gives the recommended number of test cases needed for branch coverage.

```
switch a {
    case 1:  x =3;
             break;
    case 2: if (b == 0)
                x=2;
            else
                x=4;
            break;
    case 3: while (c>0)
              process(c);
            break;
}
```

What does detection of a fault tell us?

- What does the detection of a fault within a module tell us about the probability of the existence of additional faults in the same module?
  - Does finding a fault have any bearing on whether other faults are present?



- [Myers]: When a module has too many faults =>
  - It is cheaper to redesign, recode module than to try to fix its faults