Topics in Metrics for Software Testing

[Reading assignment: Chapter 20, pp. 314-326]

Quantification

- One of the characteristics of a maturing discipline is the replacement of art by science.
- Early physics was dominated by philosophical discussions with no attempt to quantify things.
- Quantification was impossible until the right questions were asked.

Quantification (Cont’d)

- Computer Science is slowly following the quantification path.
- There is skepticism because so much of what we want to quantify it tied to erratic human behavior.

Software quantification

- Software Engineers are still counting lines of code.
- This popular metric is highly inaccurate when used to predict:
  - costs
  - resources
  - schedules

Science begins with quantification

- Physics needs measurements for time, mass, etc.
- Thermodynamics needs measurements for temperature.
- The “size” of software is not obvious.
- We need an objective measure of software size.

Software quantification

- Lines of Code (LOC) is not a good measure software size.
- In software testing we need a notion of size when comparing two testing strategies.
- The number of tests should be normalized to software size, for example:
  - Strategy ***A*** needs 1.4 tests/unit size.

Asking the right questions

- When can we stop testing?
- How many bugs can we expect?
- Which testing technique is more effective?
- Are we testing hard or smart?
- Do we have a strong program or a weak test suite?
- Currently, we are unable to answer these questions satisfactorily.

Lessons from physics

- Measurements lead to Empirical Laws which lead to Physical Laws.
- *E.g.,* Kepler's measurements of planetary movement lead to Newton's Laws which lead to Modern Laws of physics.

Lessons from physics (Cont'd)

- The metrics we are about to discuss aim at getting empirical laws that relate program size to:
  - expected number of bugs
  - expected number of tests required to find bugs
  - testing technique effectiveness

Metrics taxonomy

- **<u>Linguistic Metrics:</u>** Based on measuring properties of program text without interpreting what the text means.
  - *E.g.,* LOC.
- **<u>Structural Metrics:</u>** Based on structural relations between the objects in a program.
  - *E.g.,* number of nodes and links in a control flowgraph.

Lines of code (LOC)

- LOC is used as a measure of software complexity.
- This metric is just as good as source listing weight if we assume consistency w.r.t. paper and font size.
- Makes as much sense (or nonsense) to say:
  - “This is a 2 pound program”
- as it is to say:
  - “This is a 100,000 line program.”

Lines of code paradox

- **Paradox:** If you unroll a loop, you reduce the complexity of your software ...
- Studies show that there is a linear relationship between LOC and error rates for small programs (*i.e.,* LOC < 100).
- The relationship becomes non-linear as programs increases in size.

Halstead's program length

$$H = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

n_1 = the number of distinct operators (keywords) in the program. (Paired operators (begin ... end) are treated as a single operator.)

n_2 = the number of distinct operands (data objects) in the program.

WARNING: Program Length $\neq$ LOC

Example of program length

```
if (y < 0)
      pow = - y;
else
      pow = y;
z = 1.0;
while (pow != 0) {
      z = z * x;
      pow = pow - 1;
      }
if (y < 0)
      z = 1.0 / z;
```

$$n_1 = 9 \ \ (\text{if}, <, =, \text{- (sign)}, \text{while}, \; !=, *, \text{- (minus)}, /)$$

$$n_2 = 7 \ \ (y, 0, \text{pow}, z, x, 1, 1.0)$$

$$H = 9 \log_2 9 + 7 \log_2 7 \approx 48$$

Example of program length

```
for ( j=1; j<N; j++) {
    last = N - j + 1;
    for (k=1; k <last; k ++) {
        if (list[k] > list[k+1]) {
            temp = list[k];
            list[k] = list[k+1];
            list[k+1] = temp;
            }
        }
    }
```

$$n_1 = 9 \;\; (\text{for}, =, <, ++, \text{-}, +, [], >, \text{if})$$

$$n_2 = 7 \;\; (\text{j}, 1, \text{N}, \text{last}, \text{k}, \text{list}, \text{temp})$$

$$H = 9 \log_2 9 + 7 \log_2 7 \approx 48$$

Halstead’s bug prediction

$$B = \frac{(N_1 + N_2)\log_2(n_1 + n_2)}{3000}$$

n_1 = the number of distinct operators

n_2 = the number of distinct operands

N_1 = the total number of operators

N_2 = the total number of operands

Exponentiation Example:

$$B = \frac{(16 + 21)\log_2(9 + 7)}{3000} \approx 0.049 \text{ bugs}$$

Bubble Sort Example:

$$B = \frac{(25 + 31)\log_2(9 + 7)}{3000} \approx 0.075 \text{ bugs}$$

How good are Halstead's metrics?

- The validity of the metric has been confirmed experimentally many times, independently, over a wide range of programs and languages.
- Lipow compared actual to predicted bug counts to within 8% over a range of program sizes from 300 to 12,000 statements.

Structural metrics

- Linguistic complexity is ignored.
- Attention is focused on control-flow and data-flow complexity.
- Structural metrics are based on the properties of flowgraph models of programs.

Cyclomatic complexity

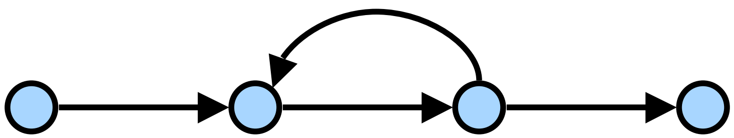
- McCabe's Cyclomatic complexity is defined as: $M = L - N + 2P$
- L = number of links in the flowgraph
- N = number of nodes in the flowgraph
- P = number of disconnected parts of the flowgraph.

Property of McCabe’s metric

- *The complexity of several graphs considered together is equal to the sum of the individual complexities of those graphs.*

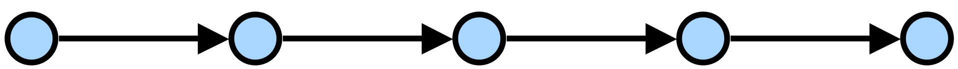
Examples of cyclomatic complexity

L=1, N=2, P=1
M=1-2+2=1



L=4, N=4, P=1
M=4-4+2=2

L=2, N=4, P=2
M=2-4+4=2



L=4, N=5, P=1
M=4-5+2=1

Cyclomatic complexity heuristics

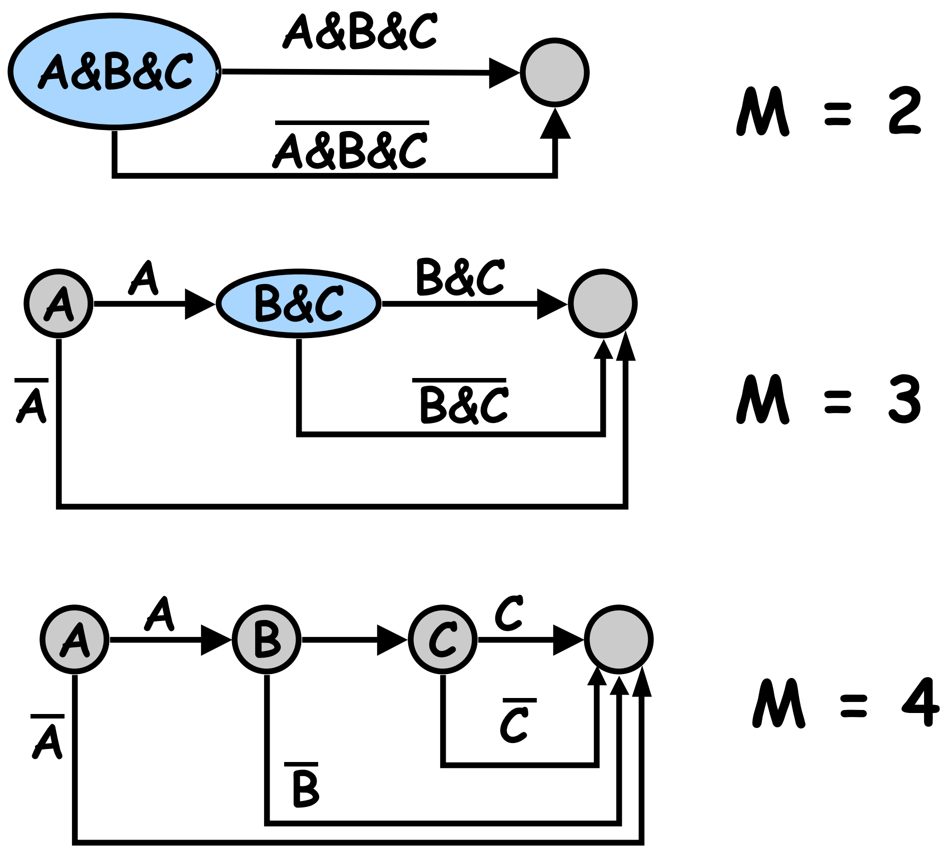
- To compute Cyclomatic complexity of a flowgraph with a single entry and a single exit:

$$M \approx 1 + \text{ total number of binary decisions}$$

- **Note:**
  - Count n-way case statements as *N* binary decisions.
  - Count looping as a single binary decision.

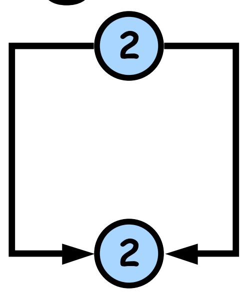
Compound conditionals

- Each predicate of each compound condition must be counted separately. *E.g.,*



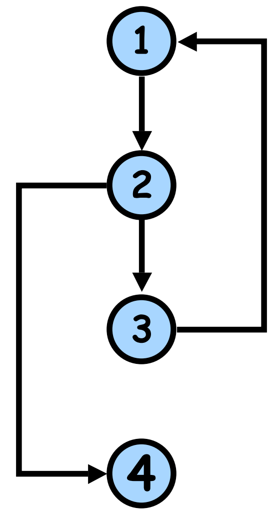
Cyclomatic complexity of programming constructs

1. if E then
   A
   else
   B
2. C



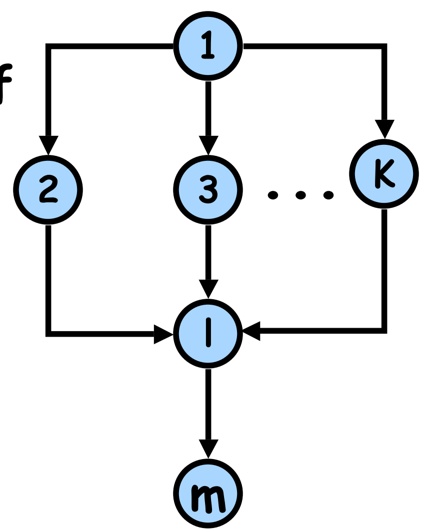
M = 2

1. loop
   A
2. exit when E
   B
3. end loop
4. C



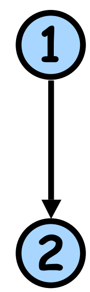
M = 2

1. case E of
2. a: A
3. b: B
   …
k. k-1: N
l. end case
m. L



M = (2(k-1)+1)-(k+2)+2=K-1

1. A
   B
   C
   …
2. Z



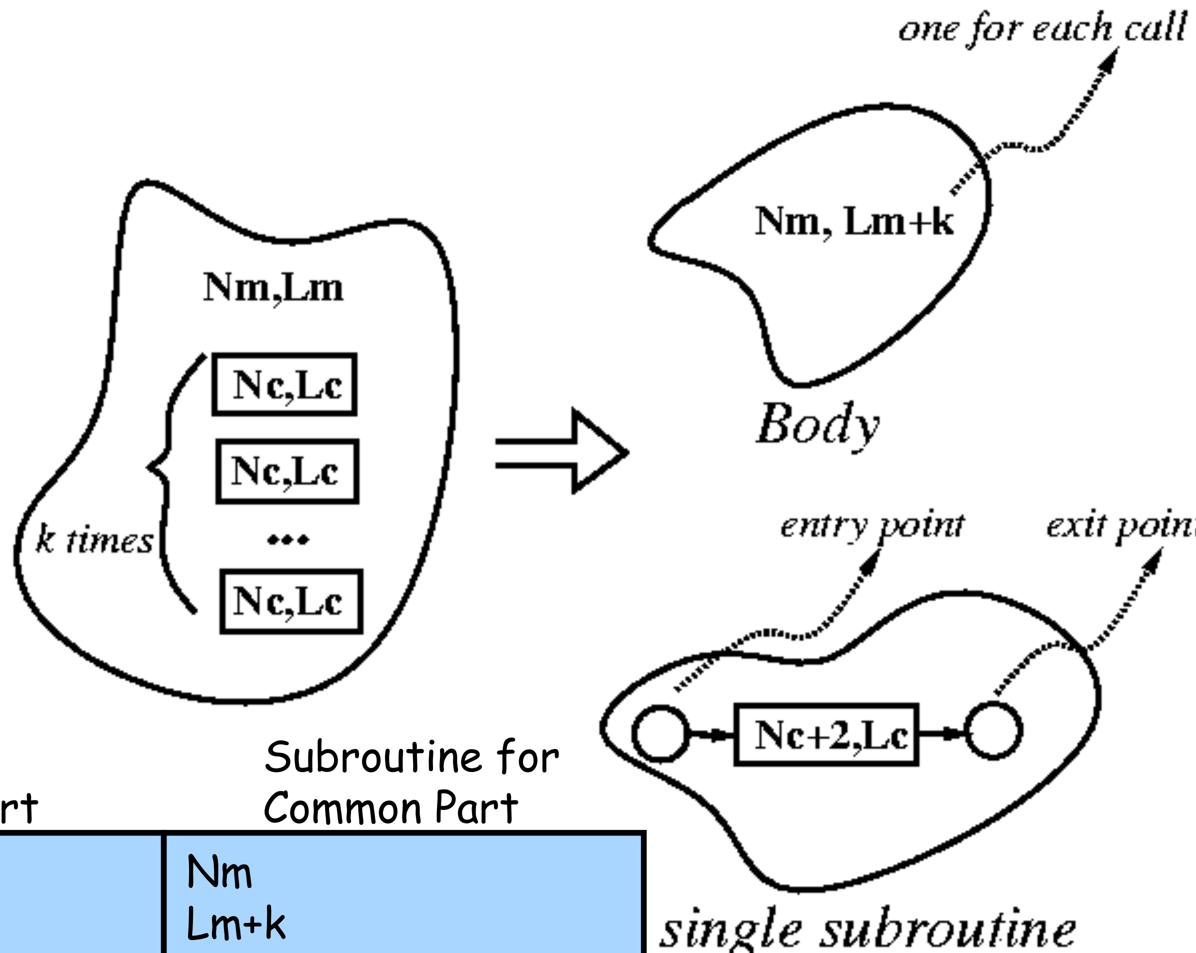
M = 1

Applying cyclomatic complexity to evaluate test plan completeness

- Count how many test cases are intended to provide branch coverage.
- If the number of test cases < *M* then one of the following may be true:
  - You haven't calculated *M* correctly.
  - Coverage isn't complete.
  - Coverage is complete but it can be done with more but simpler paths.
  - It might be possible to simplify the routine.

Warning

- Use the relationship between *M* and the number of covering test cases as a guideline not an immutable fact.



	Embedded Common Part	Subroutine for Common Part
Main Nodes	Nm+kNc	Nm
Main Links	Lm+kLc	Lm+k
Subnodes	0	Nc+2
Sublinks	0	Lc
Main M	Lm+kLc-Nm-kNc+2	Lm+k-Nm+2
Subroutine M	0	Lc-Nc-2+2=Lc-Nc=Mc
Total M	Lm+kLc-Nm-kNc+2	Lm+Lc-Nm-Nc+k+2

When is the creation of a subroutine cost effective?

- *Break Even Point* occurs when the total complexities are equal:
- The break even point is independent of the main routine's complexity.

$$L_m + kL_c - N_m - kN_c + 2 = L_m + L_c - N_m - N_c + k + 2$$

$$k(L_c - N_c) = L_c - N_c + k$$

$$k(L_c - N_c - 1) = L_c - N_c$$

$$k(M_c - 1) = M_c$$

$$kM_c - k = M_c$$

$$kM_c - M_c = k$$

$$M_c(k - 1) = k$$

$$M_c = \frac{k}{k - 1}$$

Example

- If the typical number of calls to a subroutine is 1.1 (k=1.1), the subroutine being called must have a complexity of 11 or greater if the net complexity of the program is to be reduced.

$$M_c = \frac{1.1}{1.1 - 1} = 11$$

Cost effective subroutines (Cont'd)

$k = 1, M_c = \infty$

(creating a subroutine you only call once is not cost effective)

$$k = 2, M_c = \frac{2}{1} = 2$$

(break even occurs when $M_c = 2$)

$$k = 3, M_c = \frac{3}{2} = 1.5$$

$$k = 1000, M_c = \frac{1000}{999} \approx 1$$

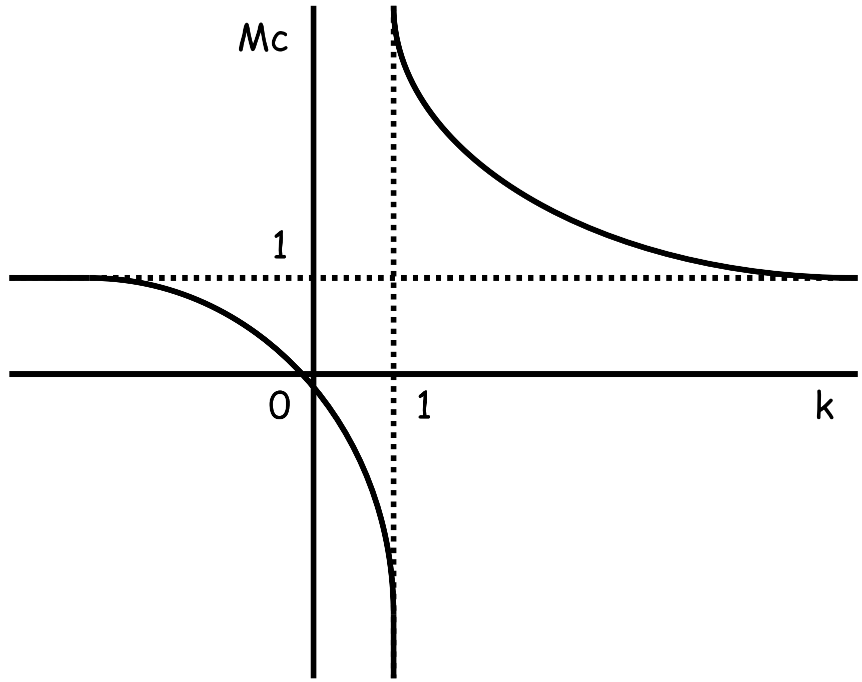
(for more calls, M_c decreases asymptotically to 1)

Cost effective subroutines (Cont'd)

The relationship between M_c and k :

$$M_c = \frac{k}{k - 1} = 1 + \frac{1}{k - 1}$$

Relationship plotted as a function



- Note that the function does not make sense for values of $0 < k < 1$ because $Mc < 0$!
- Therefore we need to mention that $k > 1$.

How good is M?

- A military software project applied the metric and found that routines with $M > 10$ (23% of all routines) accounted for 53% of the bugs.
- Also, of 276 routines, the ones with $M > 10$ had 21% more errors per LOC than those with $M <= 10$.
- McCabe advises partitioning routines with $M > 10$.

Pitfalls

- *if ... then ... else* has the same *M* as a loop!
- *case* statements, which are highly regular structures, have a high *M*.
- <u>**Warning:**</u> McCabe's metric should be used as a rule of thumb at best.

Rules of thumb based on M

- Bugs/LOC increases discontinuously for $M > 10$
- M is better than LOC in judging life-cycle efforts.
- Routines with a high M (say > 40) should be scrutinized.
- M establishes a useful lower-bound rule of thumb for the number of test cases required to achieve branch coverage.

Software testing process metrics

- Bug tracking tools enable the extraction of several useful metrics about the software and the testing process.
- Test managers can see if any trends in the data show areas that:
  - may need more testing
  - are on track for its scheduled release date
- Examples of software testing process metrics:
  - Average number of bugs per tester per day
  - Number of bugs found per module
  - The ratio of Severity 1 bugs to Severity 4 bugs
  - …

Example queries applied to a bug tracking database

- What areas of the software have the most bugs? The fewest bugs?
- How many resolved bugs are currently assigned to John?
- Mary is leaving for vacation soon. How many bugs does she have to fix before she leaves?
- Which tester has found the most bugs?
- What are the open Priority 1 bugs?

Example data plots

- Number of bugs versus:
  - fixed bugs
  - deferred bugs
  - duplicate bugs
  - non-bugs
- Number of bugs versus each major functional area of the software:
  - GUI
  - documentation
  - floating-point arithmetic
  - etc

Example data plots (cont'd)

- Bugs opened versus date opened over time:
  - This view can show:
    - bugs opened each day
    - cumulative opened bugs
- On the same plot we can plot resolved bugs, closed bugs, etc to compare the trends.

You now know …

- … the importance of quantification
- … various software metrics
- … various software testing process metrics and views