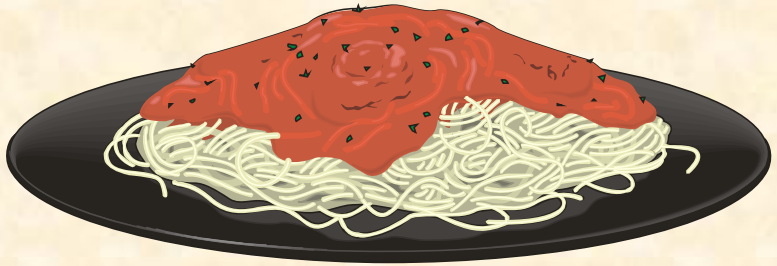
Prof. Philip Koopman

Carnegie Mellon University

Electrical & Computer ENGINEERING

Avoiding Spaghetti Code



"The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."

— Edsgar Dijkstra, 1968

Spaghetti Code

Carnegie Mellon University

- Anti-Patterns:
  - Deeply nested conditionals
  - "Switch" nesting
  - High Cyclomatic Complexity (too many paths through the code)

Hacker DICTIONARY

http://www.hacker-dictionary.com/terms/spaghetti-code

spaghetti code

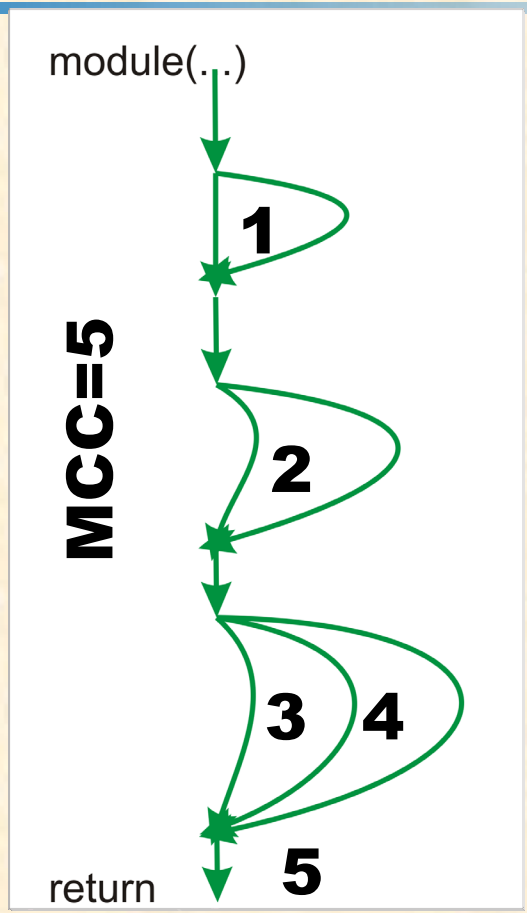
n. Code with a complex and tangled control structure, esp. one using many GOTOs, exceptions, or other `unstructured' branching constructs. Pejorative. The synonym `kangaroo code' has been reported, doubtless because such code has many jumps in it.

- Unstructured code leads to bugs
  - Unstructured code is generally hard to understand, test, and review
    - But, even structured code can be problematic if it is too complex
  - Want to limit complexity within each unit (e.g., subroutine, method)
    - Complex code is difficult to review – you will miss bugs during review
    - Complex code can be difficult or impossible to test

Measuring Complexity

Carnegie Mellon University

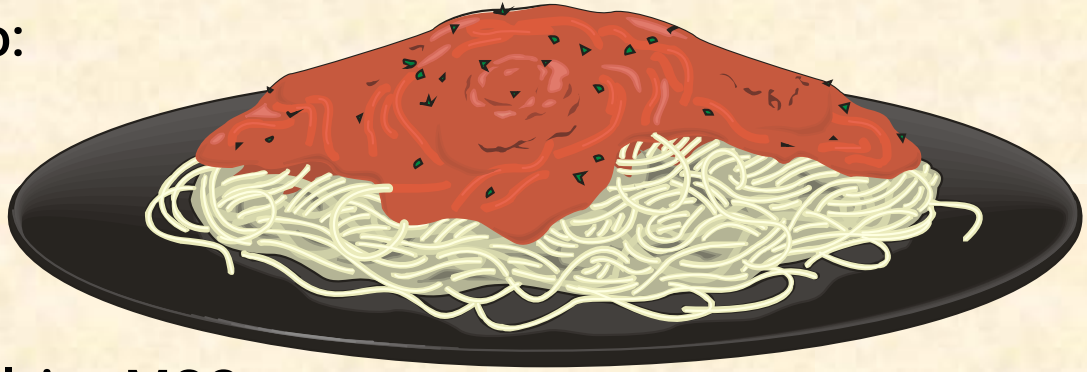
- McCabe Cyclomatic Complexity (MCC)
  - Measure each module (subroutine, method, …)
  - Draw a control flow graph
    - Graph has an arc for each path through the module
  - MCC is # of "holes" in graph + 1
    - Worst case number of unit tests to cover all paths
    - Might need more tests – it's a guideline
- Strict Cyclomatic Complexity (SCC)
  - For complex "if" tests, each condition counts
    - "if ((x == 0) || (y == 0)) …"
      counts as +2, because need to test x!=0 and y!=0
    - MCDC testing requires this type of coverage

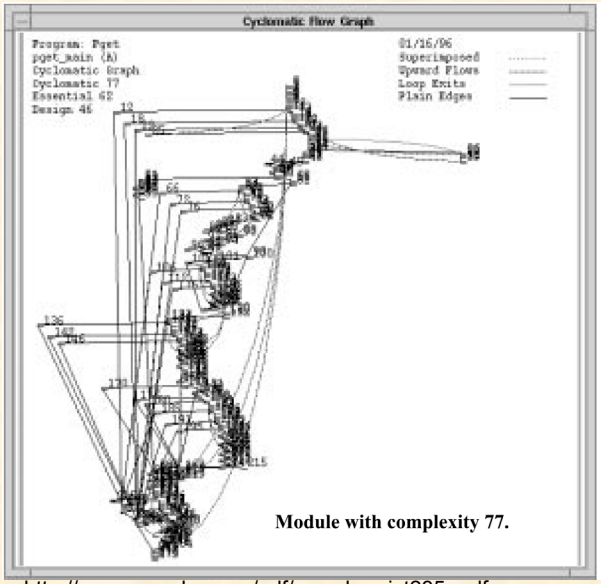


High MCC Results in Tangled Code

Carnegie Mellon University

- Complexity beyond GOTO due to:
  - Nested conditionals
  - Overly complex lines of code
  - Multiple return points
  - Nested exceptions





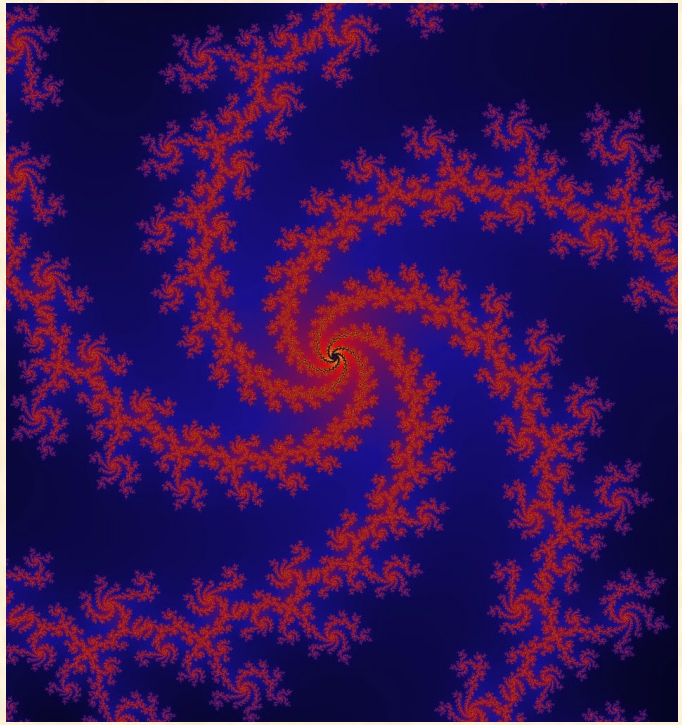
http://www.mccabe.com/pdf/mccabe-nist235r.pdf

- Applying MCC
  - Want maximum MCC to be 10 or 15
    - Above 30 is highly suspect
    - Above 50 is untestable in practice
      - » Too tangled to reasonably test each path
      - » Exception for flat switch statements
    - Above 75 predicts bug farms
      - » Each fix breaks something else

Strict Cyclomatic Complexity (SCC)

Carnegie Mellon University

- Problem: complex conditionals used to game complexity
  - If (x) { if (y) { if (z) {…} } } ➔ MCC of +3
  - If (x && y && z) {…} ➔ MCC of +1
    - But same number of test cases…
      … and same complexity



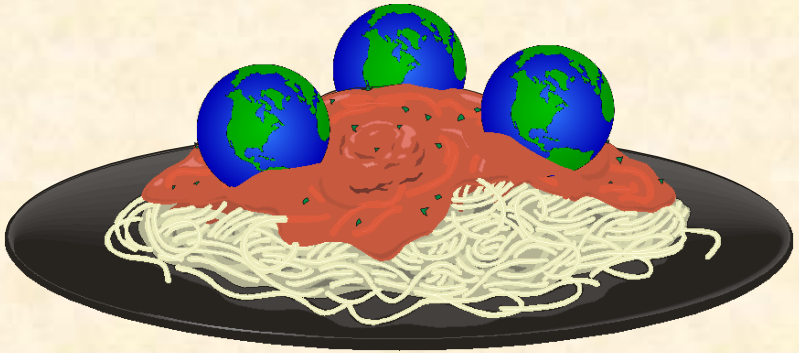
- Solution: SCC (also known as CC2)
  - Every extra conditional Boolean term counts +1
    - if (x) { if (y) { if (z) {…} } } ➔ SCC of +3
    - if (x && y && z) {…} ➔ SCC of +3
- Important notes on applying MCC/SCC:
  - This is a per-subroutine metric, not whole .c file
  - The point is to encourage breaking up complex code into small pieces

Spaghetti Factor (SF) Metric

Carnegie Mellon University

SF = SCC + (Globals*5) + (SLOC/20)

- SCC = Strict Cyclomatic Complexity
- Globals = # of read/write global variables referenced
- SLOC = # source lines of code (e.g., C statements)
- Scoring:
  - 5-10 - This is the sweet spot for most code
  - 15 - Don't go above this for most modules
  - 20 - Look closely; possibly refactor
  - 30 - Refactor the design
  - 50 - Untestable; throw the module away and redesign
  - 75 - Unmaintainable; throw the module and its design away; start over
  - 100 - Nightmare; throw it out and re-architect



Code Complexity Best Practices

Carnegie Mellon University

- Keep MCC below 10 to 15
  - Even better, keep SCC below 10 to 15
    - Exception: easy to test flat switch statements are OK
  - This enables thorough unit test
- Additional signs of complexity issues
  - "If" statements nested more than 2 or 3 deep
  - Nested "if" and "switch" statements
  - Excessive "break," "continue," multiple "return"

- If your module is too complex, it's time to break it up!
  - Focus on worst offenders & break pieces of logic out into helper functions
  - The point of this is to enable good peer review and good unit test
- Complexity pitfalls:
  - Creeping complexity over time … at some point, refactor!

https://xkcd.com/292/

Not thinking about how much pain this is going to cause in the future

Essential

Rationalizing Your Awful Hackjob

O RLY?

@ThePracticalDev