CSE 503 Winter 2013 • David Notkin

Lecture #3 Notes: software complexity, projects

1. But first… from 2011.5.4 Seattle Times: "Industry experts believed they knew where to look for crack-inducing metal fatigue on aging airplanes, but the in-flight rupture of a Southwest Airlines Boeing 737 on Friday has raised concerns about part of the fuselage they previously thought wasn't vulnerable.

   "A similar hole opened on a Southwest 737 only 21 months ago, and then on an American Airlines 757 last year, raising awareness that metal fatigue can cause the aluminum skin to separate at the so-called lap joints, where panels are spliced together."

2. Software complexity
   a. "I have made this letter longer than usual, because I lack the time to make it short." –Blaise Pascal
   b. Lines of code (LOC, KLOC, MLOC)
      i. Count the lines, often omitting comments and/or omitting blank lines
      ii. Lines vs. statements
      iii. Delivered vs. total (including tests, etc.)
      iv. Productivity: LOC/person/time
         1. I've seen published numbers ranging from ~2K-8K LOC/person/year
         2. Sensible?
   c. Halstead software science metrics
      i. n1 = #distinct operators
      ii. n2 = #distinct operands
      iii. n = n1 + n2 is the "vocabulary"
      iv. N1 = total # of operators
      v. N2 = total# of operands
      vi. N = N1 + N2 is the "length"
      vii. **$V = N \times \log_2(n)$** • *Volume is intended to capture the size of the implementation*
      viii. Making N choices from the vocabulary – assuming that humans do logarithmic search – leads to the formula
      ix. "The volume of a function should be at least 20 and at most 1000. The volume of a parameterless one-line function that is not empty; is about 20. A volume greater than 1000 tells that the function probably does too many things." [verifysoft.com]
      x. **$D = (n1 / 2) \times (N2 / n2)$** • *Difficulty is proportional to the unique operators and the ratio of total operands to the number of operands*

xi. The intent of the second part is based on a belief that repeated use of operands is more error-prone

xii. **E = V × D** • *Effort to implement or understand a program*

d. Cyclomatic complexity (McCabe)

i. Take the CFG and find the number of edges (**E**), number of nodes (**N**), and the number of connected components (**P**)

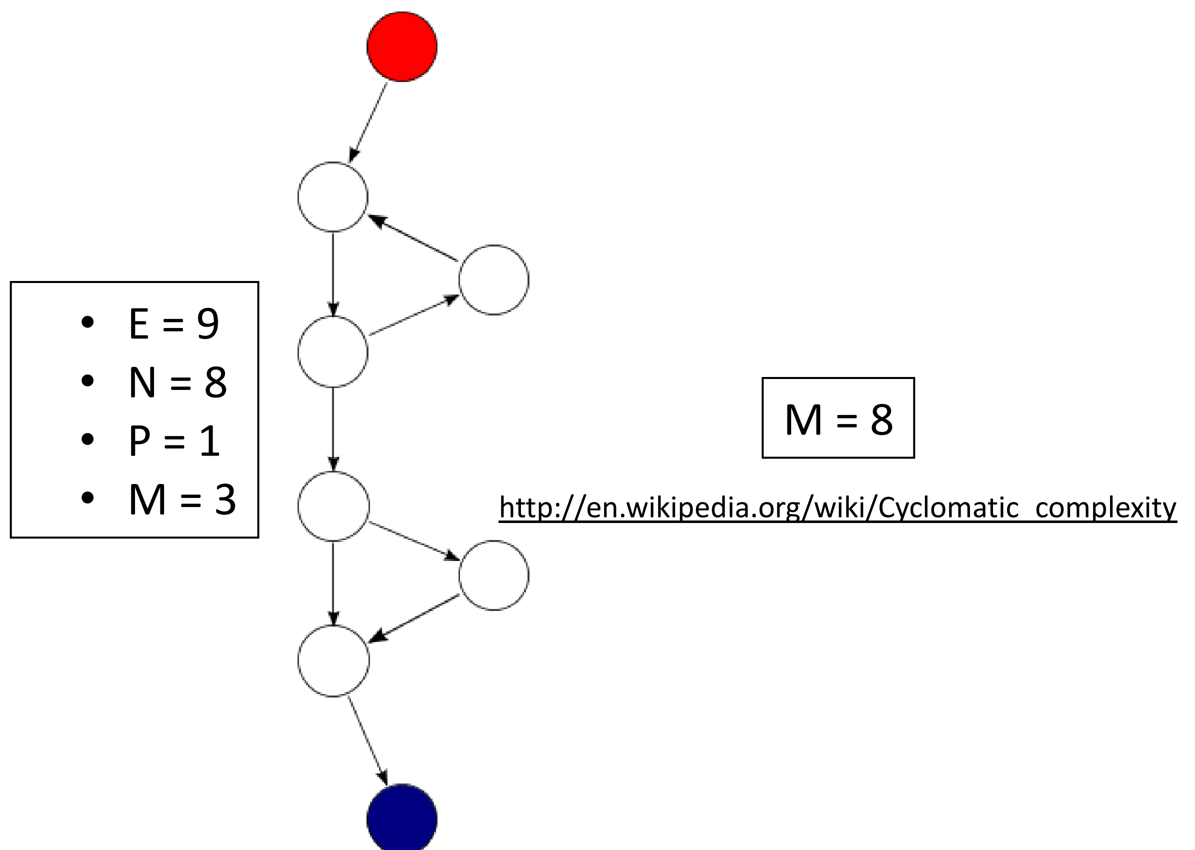
ii. Connected components are subgraphs for which there is a path between any two vertices

iii. The cyclomatic complexity is **M = E − N + 2P** and is intended to measure the number of linearly independent paths through a program's source cod

iv. #tests (branch coverage) ≤ **M** ≤ #tests (path coverage)

v. Question: should the complexity include method dispatch in OOP?

vi. Example



e. Software structure metrics [Henry and Kafura]

i. Measures complexity in terms of fan-in and fan-out of procedures

ii. *fan-in*: the number of local flows into a procedure plus the number of data structures accessed.

iii. *fan-out*: the number of local flows out a procedure plus the number of data structures that the procedure modifies.

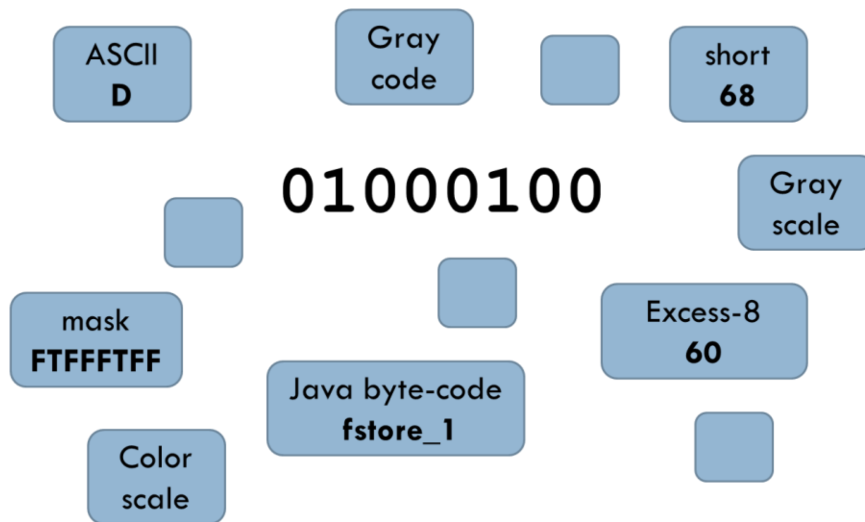
    iv. Complexity is $L^2 \times FI \times FO$ here L is the length of a procedure

f. Coupling and cohesion
    i. Cohesion: why elements are placed together in a module
    ii. Coupling: interactions between modules in a design
    iii. Function points and feature points – intended to measure the function of a system as perceived by users, without reference to the implementation

g. And many more and many variants of these (interprocedural complexity, etc.) So?
    i. Although there is somewhat mixed data, it appears that most of these measures are proportional to LOC
    ii. "Les Hatton claimed recently (Keynote at TAIC-PART 2008, Windsor, UK, Sept 2008) that McCabe Cyclomatic Complexity has the same prediction ability as lines of code." –Wikipedia [cyclomatic complexity]
    iii. Also, how "actionable" the information is has always confused me: if you are told your program is an "8" what are you supposed to do?

h. A hypothesis
    i. Every complexity measure I've seen is based entirely on the static program (except feature/function points, which don't consider a program directly)
    ii. ***If*** complexity measures are to have any real utility, it seems that they must also consider the relationship between the program and its behaviors
    iii. That is, the way the developer associates behaviors with a program is material to complexity, but is ignored by the literature
    iv. It is also imaginable that this measure would be "actionable" by identifying specific dependences that make this mapping complex – they could perhaps be addressed similarly to dependences that preclude parallelization
    v. Project(s)?
        1. Any attempt at trying to make this notion more precise would be terrific
        2. Maybe a simple model and some empiric data
        3. Showing that a reasonable model is proportional to LOC would weaken my hypothesis
        4. Stop by and chat if you're interested

3. What is this?



4. Types
    a. Without getting precise, types are used to interpret and manipulate the bit patterns – that is, they give them (some level of) meaning
    b. “Concrete” types manipulate the information in memory directly
    c. Abstract types define a protocol for manipulating instances of those types, but they do not define an implementation
    d. Abstract data type = objects + operations
    e. The only operations on objects of the type are those provided by the abstraction
    f. The implementation is hidden
    g. We need to show that the abstraction and the implementation are each “correct” … and properly related … to be continued
5. Projects
    a. For those with significant development experience, this should be an aggressive use of a model checker, of selected tools, of some new technology, etc. There should be a focus to the effort; not "just playing around" with the tool or technology. The intent is to have you learn, in some depth, about an approach to build better software or to build software better. The specific project must be approved by me. The results of the project will be a web page that describes the effort, assess the strengths/weakness of the approach, lists problems you faced (you might do this by starting a mini-blog right away), links to the tools, etc. You can work individually or in pairs.
    b. Model checkers: take a finite state machine and a temporal logic formula and return either “true,” “false with a counterexample,” or “can’t finish the computation.”

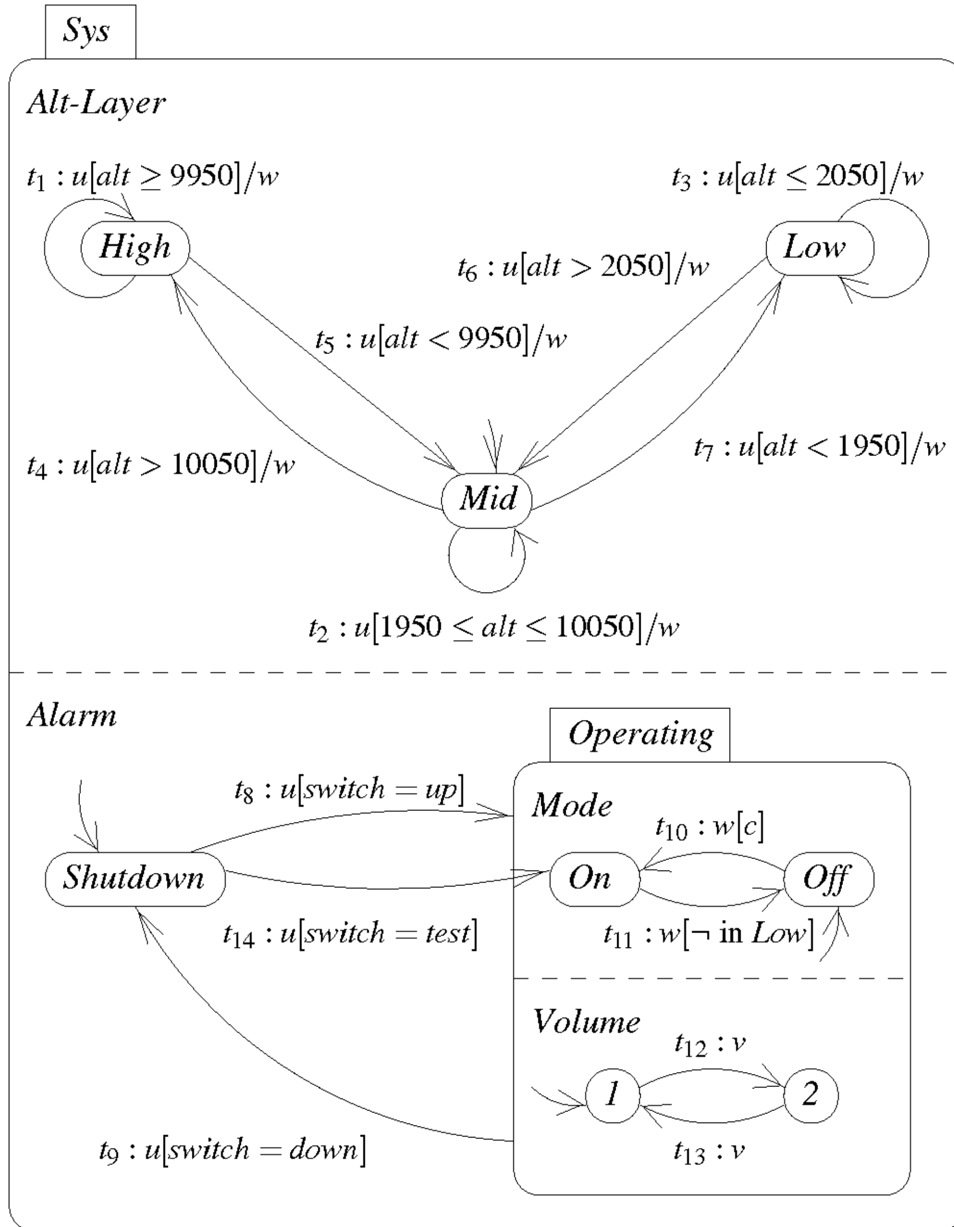
c. Wikipedia example using CTL – trivial finite state machine
Let "P" mean "I like chocolate"

i. **AG**.P
"I will like chocolate from now on, no matter what happens."

ii. **EF**.P
"It's possible I may like chocolate some day, at least for one day."

iii. **AF**.**EG**.P
"It's always possible (AF) that I will suddenly start liking chocolate for the rest of time."
(Note: not just the rest of my life, since my life is finite, while **G** is infinite).

d. Partial examples



$$
\text{Displayed-Model-Goal} = \left\{ \begin{array}{lll}
0 & \textbf{if } \text{Composite-RA } \textbf{not in state} \text{ Positive} & \text{/* case 1 */} \\[1em]
\begin{array}{l} \textbf{Max}(\text{Own-Track-Alt-Rate}, \\ \textsc{Prev}(\text{Displayed-Model-Goal}), \\ 1500 \text{ ft/min}) \end{array} & \begin{array}{l} \textbf{if } (\text{New-Climb } \textbf{or} \text{ New-Threat}) \textbf{ and} \\ \textbf{not } \text{New-Increase-Climb } \textbf{and} \\ \textbf{not } (\text{Increase-Climb-Cancelled } \textbf{or} \\ \text{Increase-Descend-Cancelled}) \textbf{ and} \\ \text{Composite-RA } \textbf{in state} \text{ Climb} \end{array} & \text{/* case 2 */} \\[1em]
\begin{array}{l} \textbf{Min}(\text{Own-Track-Alt-Rate}, \\ \textsc{Prev}(\text{Displayed-Model-Goal}), \\ -1500 \text{ ft/min}) \end{array} & \begin{array}{l} \textbf{if } (\text{New-Descend } \textbf{or} \text{ New-Threat}) \textbf{ and} \\ \textbf{not } \text{New-Increase-Descend } \textbf{and} \\ \textbf{not } (\text{Increase-Climb-Cancelled } \textbf{or} \\ \text{Increase-Descend-Cancelled}) \textbf{ and} \\ \text{Composite-RA } \textbf{in state} \text{ Descend} \end{array} & \text{/* case 3 */} \\[1em]
2500 \text{ ft/min} & \textbf{if } \text{New-Increase-Climb} & \text{/* case 4 */} \\[1em]
-2500 \text{ ft/min} & \textbf{if } \text{New-Increase-Descend} & \text{/* case 5 */} \\[1em]
\begin{array}{l} \textbf{Max}(\text{Own-Track-Alt-Rate}, \\ 1500 \text{ ft/min}) \end{array} & \begin{array}{l} \textbf{if } \text{Increase-Climb-Cancelled } \textbf{and} \\ \textbf{not } \text{New-Increase-Climb } \textbf{and} \\ \text{Composite-RA } \textbf{in state} \text{ Positive} \end{array} & \text{/* case 6 */} \\[1em]
\begin{array}{l} \textbf{Min}(\text{Own-Track-Alt-Rate}, \\ -1500 \text{ ft/min}) \end{array} & \begin{array}{l} \textbf{if } \text{Increase-Descend-Cancelled } \textbf{and} \\ \textbf{not } \text{New-Increase-Descend } \textbf{and} \\ \text{Composite-RA } \textbf{in state} \text{ Positive} \end{array} & \text{/* case 7 */} \\[1em]
\textsc{Prev}(\text{Displayed-Model-Goal}) & \text{Otherwise} & \text{/* case 8 */}
\end{array} \right.
$$

AG ((stable & Radio-Altimeter-Status = Valid & Own-Alt-Radio <= 1450) –> !Increase-Descend)

e. Bounded model checking – when the state space isn't finite, chop it off at a particular depth. Weakens the guarantees, makes some checking possible, relies in part on the small scope hypothesis
    i. Alloy is the classic example
    ii. http://alloy.mit.edu/alloy/citations/case-studies.html
f. Test case generators
    i. **Pex automatically generates test suites with high code coverage.** Right from the Visual Studio code editor, Pex finds interesting input-output values of your methods, which you can save as a small test suite with high code coverage. Microsoft Pex is a Visual Studio add-in for testing .NET Framework applications.
    ii. http://code.google.com/p/randoop/
g. Other Microsoft Research tools http://research.microsoft.com/en-us/groups/rise/default.aspx
h. Ernst tools: http://homes.cs.washington.edu/~mernst/software/