But first... from today’s Seattle Times

2011.5.4

“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 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.”

Software complexity

- First, some common software complexity measures
- Then, why they are weak measures and (perhaps) a way forward

Lines of code (LOC, KLOC, MLOC)

- Count the lines, often omitting comments and/or blank lines
- Lines vs. statements
- Delivered vs. total (including tests, etc.)
- Productivity: LOC/person/time
  - I’ve seen published numbers ranging from ~2K-8K LOC/person/year

"I have made this letter longer than usual, because I lack the time to make it short." —Blaise Pascal
Halstead software science metrics

- \( n_1 \) = number of distinct operators
- \( n_2 \) = number of distinct operands
- \( N_1 \) = total # of operators
- \( N_2 \) = total # of operands
- \( N = N_1 + N_2 \) = "length"
- \( V = N \times \log_2(n) \) = Volume
- \( D = (n_1 / 2) \times (N_2 / n_2) \) = Difficulty
- \( E = V \times D \) = Effort

- Volume is intended to capture the size of the implementation
- "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]
- Difficulty is proportional to the unique operators and the ratio of total operands to the number of operands
- The intent of the second part is based on a belief that repeated use of operands is more error-prone
- Effort to implement or understand a program

Cyclomatic complexity (McCabe)

- Take the CFG and find the number of edges (\( E \)), number of nodes (\( N \)), and the number of connected components (\( P \))
- Connected components are subgraphs for which there is a path between any two vertices
- The cyclomatic complexity is \( M = E - N + 2P \) and is intended to measure the number of linearly independent paths through a program’s source code
- \#tests (branch coverage) \( \leq M \leq \#tests \) (path coverage)
- Question: should the complexity include method dispatch in OOP?

Software structure metrics

Henry and Kafura

- Measures complexity in terms of fan-in and fan-out of procedures
- Fan-in: the number of local flows into a procedure plus the number of data structures accessed.
- Fan-out: the number of local flows out a procedure plus the number of data structures that the procedure modifies.
- Complexity is \( L^2 \times FI \times FO \)
- Where \( L \) is the length of a procedure
And many more

- Variants of these
  - Some incremental improvements
  - Some extending to interprocedural complexity

- Others that measure
  - Coupling and cohesion
  - Data complexity
  - Data flow complexity
  - ...

- Function points and feature points — intended to measure the function of a system as perceived by users, without reference to the implementation

So?

- Although there is somewhat mixed data, it appears that most of these measures are proportional to LOC

- “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]

- 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?

A hypothesis

- Every complexity measure I’ve seen is based entirely on the static program (except feature/function points, which don’t consider a program directly)

- If complexity measures are to have any real utility, it seems that they must also consider the relationship between the program and its behaviors
  - That is, the way the developer associates behaviors with a program is material to complexity, but is ignored by the literature

- 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

Project(s)?

- Any attempt at trying to make this notion more precise would be terrific

- Maybe a simple model and some empiric data

- Showing that a reasonable model is proportional to LOC would weaken my hypothesis

- Stop by and chat if you’re interested

- Fits into NSF-funded work with Reid Holmes
  - ICSE 2011: “Identifying Program, Test, and Environmental Changes That Affect Behavior”
  - Potential quals project
What is this?

ASCII
D
Gray code
short 68
Gray scale
mask FFFFFF
Excess-8 60
Java byte-code fstore_1
Color scale

Types

- Without getting precise, types are used to interpret and manipulate the bit patterns – that is, they give them (some level of) meaning
- “Concrete” types manipulate the information in memory directly
- Abstract types define a protocol for manipulating instances of those types, but they do not define an implementation

Abstract data type = objects + operations

- The only operations on objects of the type are those provided by the abstraction
- The implementation is hidden
- We need to show that the abstraction and the implementation are each “correct” … and properly related

Big picture

- For every abstract operation
- It commutes [What is purple and commutes?]
- AF gives an abstract meaning to concrete representations – more soon
An Abelian grape (sorry)

Abelian groups of order 8, with subgroup lattices

Specifying ADTs

- A common way is to define the abstract effect of each operation (including constructors) using formal/informal pre- and post-conditions
- Might see this using an extended JavaDoc

Example

// Overview: An IntSet is a mutable, unbounded set of integers.
class IntSet {
    // effects: makes a new IntSet = {}
    public IntSet() {
        // returns: true if x ∈ this
        // else returns false
        public boolean contains(int x) {
            // effects: this_post = this_pre ∪ {x}
            public void add(int x) {
                // effects: this_post = this_pre ∖ {x}
                public void remove(int x)

Algebraic specifications

From Stotts (http://www.cs.unc.edu/~stotts/723/adt.html)

- Define a sort – give signatures of operations (you’ve seen this kind of thing before in typed OO and functional languages)

    sort IntSet imports Int, Bool
    signatures
    new : -> IntSet
    insert : IntSet × Int -> IntSet
    member : IntSet × Int -> Bool
    remove : IntSet × Int -> IntSet
Define axioms

- "Just" like high school algebra

  variables $i, j : \text{Int}; s : \text{IntSet}$

  axioms
  
  member(new(), $i) = \text{false}$
  member(insert($s, j), i) =
  if $i = j$ then $\text{true}$ else member($s, i)$
  remove(new(), $i) = \text{new}()$
  remove(insert($s, j), i) =
  if $i = j$ then remove($s, i)$
  else insert(remove($s, i), j)$

Are these really sets?

- Posit stuff like...
  
  insert(insert($s, i), j) =
  insert(insert($s, j), i)$
  insert(insert($s, i), i) = insert($s, i)$

- Prove from axioms

- Tons of issues about completeness, consistency, equality (initial vs. final algebras), etc.

- But again, "just" like high school algebra

Proving specification properties

- Regardless of the style of specification, proofs are usually done inductively

- No information about the concrete representation and implementation — rather, showing the correctness of the protocol over the ADT’s operations

LetterSet

case-insensitive character set [from Ernst]

  // effects: creates an empty LetterSet
  public LetterSet();

  // effects: this.post =
  // if ($\exists c \in \text{this.pre} | \text{toLowerCase}(c) = \text{toLowerCase}(\text{c})$)
  // then this.pre else this.pre $\cup \{c\}$
  public void insert($\text{char}\ c$);

  // effects: this.pre = this.pre $\setminus \{c\}$
  public void delete($\text{char}\ c$);

  // returns: ($c \in \text{this})$
  public boolean member($\text{char}\ c$);
Prove desirable property of LetterSet
Large enough LetterSet contains two distinct characters

Prove: |S|>1 ⇒ (∃c₁,c₂ ∈ S | toLowerCase(c₁) ≠ toLowerCase(c₂))

- Base case: S = ∅, vacuously true
- Inductive case: S was produced by a call of the form T.insert(c)

Assume: |T|>1 ⇒ (∃c₁,c₂ ∈ T | toLowerCase(c₁) ≠ toLowerCase(c₂))

Show: |S|>1 ⇒ (∃c₁,c₂ ∈ S | toLowerCase(c₁) ≠ toLowerCase(c₂))

Remember insert's post-condition:

* For inductive case, consider the two possibilities for S
  - If S = T, the theorem holds by induction
  - If S = T ∪ {c}, there are three cases
    - |T| = 0: Vacuously true
    - |T| ≥ 1: T did not contain a char of toLowerCase(c), so the theorem holds by the meaning of union
    - |T| > 1: By inductive assumption, T contains different letters, so by the meaning of union, T ∪ {c} also contains different letters

Now: Assume abstraction is correct

- Abstraction function (AF): E ∈ Eₐ
  - Maps a concrete object to an abstract value
  - Defines how the data structure is to be interpreted
  - Oh, that’s a “D”, that’s an fstore_1, that’s a 68, etc.
- Representation invariant (RI): a boolean predicate characterizing legal concrete representations
  - States data structure well-formedness
  - In essence, defines the domain of AF
  - Captures information that must be shared across implementations of multiple operations

CharSet Abstraction
A finite mutable set of Characters [From Ernst]

// Overview: A CharSet is a finite mutable set of Characters
// effects: creates a fresh, empty CharSet
public CharSet () { }

// effects: this ∈ this ∪ {c}
public void insert (Character c) { elts.add(c); }

// effects: this ∈ this ∖ {c}
public void delete (Character c) { elts.remove(c); }

// returns: (c ∈ this)
public boolean member (Character c) { return elts.contains(c); }

// returns: cardinality of this
public int size () { return elts.size(); }

A CharSet implementation

class CharSet {
  private List<Character> elts = new ArrayList<Character>();
  public void insert(Character c) { elts.add(c); }
  public void delete(Character c) { elts.remove(c); }
  public boolean member(Character c) { return elts.contains(c); }
  public int size() { return elts.size(); }
}
The RI can help identify an error

- Perhaps **delete** is wrong
  - It should remove all occurrences
- Perhaps **insert** is wrong
  - It should not insert a character that is already there
    
    ```java
    class CharSet {
        // Rep invariant: els has no nulls and no duplicates
        private List<Character> els;
        ...
    }
    ```

- Or...
  - ∀ indices i of els . els.elementAt(i) ≠ null
  - ∀ indices i, j of els . i ≠ j ⇒ ¬els.elementAt(i).equals(els.elementAt(j))

Where's the error?

- // Rep invariant: els has no nulls and no duplicates
  ```java
  public void insert(Character c) {
      els.add(c);
  }
  public void delete(Character c) {
      els.remove(c);
  }
  ```

The RI constrains structure, not meaning

- Another implementation of **insert** that preserves the RI
  ```java
  public void insert(Character c) {
      Character cc = new Character(encrypt(c));
      if (!els.contains(cc))
          els.addElement(cc);
  }
  public boolean member(Character c) {
      return els.contains(c);
  }
  ```

- The program is wrong ... call on the AF!

Abstraction function

**concrete to abstract value mapping**

- **AF( CharSet this) = { c | c is contained in this.els }**
  - set of Characters represented by elements contained in this.els
  - Typically not executable, but useful to reason about client behavior
- Helps reason about the semantics of **insert**
  ```java
  // effects: this_post = this_pre ∪ {c}
  public void insert (Character c);
  ```
  - Helps identify a problem
  - Applying the AF to the result of the call to insert yields
    **AF(els) ∪ {encrypt('a')}**
  - Consider the following reasonable AF
    - **AF(this) = { c | encrypt(c) is contained in this.els }**
    - **AF(this) = { decrypt(c) | c is contained in this.els }**
“Placing blame” using AF

- $\text{AF( CharSet \ this}) = \{ c | c \text{ is contained in } this.\text{elts} \} \$
- Consider a call to insert:
  - On entry, the meaning is $\text{AF(this}_{\text{pre}}) = elts_{\text{pre}}$
  - On exit, the meaning is $\text{AF(this}_{\text{post}}) = \text{AF(this}_{\text{pre}}) \cup \{\text{encrypt('a')}\} $

- Does this AF fix things?
  - $\text{AF(this}} = \{ c | \text{encrypt(c) is contained in this.elts} \}$
    - $\equiv \{ \text{decrypt(c)} | c \text{ is contained in this.elts} \}$

Some final odds and ends

- Looking at these examples using the commutative diagram may help clarify any confusions
  - Or ask!
- AF’s can be maintained across fairly complicated implementations that (for example) reorganize dynamically for performance
  - Multiple concrete values still map to the same abstract value
- Why map concrete to abstract?
  - It’s not a function in the other direction
    - Ex: lists \([a,b]\) and \([b,a]\) each represent the set \([a, b]\)
  - It’s not as useful in the other direction