Software Measures and Metrics

Inquiry

Consider something *physical* (tangible) in the real world

1. How do you measure it?
2. How do you interpret “goodness” or “badness” of it

Consider something non-*physical* (intangible)

1. How do you measure it?
2. How do you interpret “goodness” or “badness” of it

What are measures of size?

1. What are everyday measures of size?
2. Do any of those measures apply to software?

What are measures of complexity?

1. Think of an system you interact with, how complex is it?
2. Is it a complex problem or a complex solution/.

Measurement Basics

Measurement Scales

- Nominal - Categorical; either equal or not equal
  - Example: Types of programs: “desktop, web, pda”
- Ordinal - Ranking; ordered but not ratio’d; <, >
  - Example: Programmer experience: “low, medium, high”
- Interval - Ranked and Additive but not ratio’d; <,>,+,-
  - Example: RPI Rating. Kentucky’s is .6731and WVU’s is .6021, but does this mean Kentucky is “12% better” than West Virginia?
- Ratio - Existence of absolute zero ; <,>,+,-,/
  - Example: Losses: Wisconsin lost 3 games, VCU lost 9, so in fact VCU lost three times as many games as Wisconsin.

Many “counting” values are ratio, while many derived or interpreted values are in fact Interval (but not always)

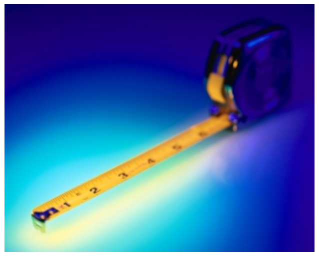
Metrics and Measures

What is a metric?

- A **metric** is a standard of **measure** (so why the different term?)
- A software metric is *interpreted* according to a quality attribute
  - Size , Complexity, Dependency, Cycle Time
  - In other words, it has an *interpretation* based on a *target*

Metrics assumptions

- A **software property** can be measured
- A relationship exists between *what we can measure* and *what we want to know*
- This **relationship** has been formalized and validated
- It may be difficult to relate what can be measured to **desirable quality attributes**



Measurement Targets

You can measure a number of things, we will focus on four types of software properties:

1. *Size* – something we can actually measure
2. *Complexity* – exploring more advanced ways to describe how hard our problems (and solutions) are
3. *Coupling* – metrics describing the internal dependencies in our software
4. *Cohesion* – how well our low-level design structures "works together"

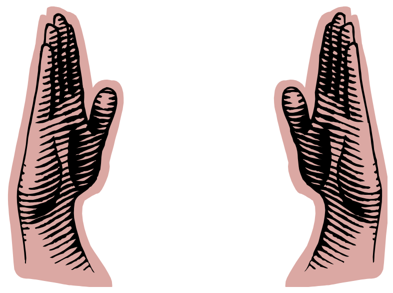
Code Size Metrics

Size measures are “controversial”

- Traditionally, lines of code has been used
  - But, a lot of issues: Language differences, verbosity, reuse, what to count?
  - Is it a proxy for complexity or for LOE?

Why does size matter so much in software?

- Typically the measure against which cost and quality are normalized
  - defects per KLOC (thousand lines of code)
  - $ per LOC
  - page of documentation per KLOC
  - LOC per person-month
- An indirect measure of effort



Lines of Code (LOC)

The most popular size metric. Different methods for counting LOC exist

- *Method 1*: Count each statement as 1, everything else 0
- *Method 2*: Same as 1 except add ½ for each comment line
- *Method 3*: Same as 2, except add 2 for special constructs

Example

```
/**
 * Draw an oval to the canvas
 * @param x1 The x coordinate of the top left corner of a box containing the arc
 * @param y1 The y coordinate of the top left corner of a box containing the arc
 * @param width  The width of the arc
 * @param height The height of the arc
 * @param segments The number of line segments to use when drawing the arc
 * @param start  The angle the arc starts at
 * @param end    The angle the arc ends at
 */
public void drawArc(float x1, float y1, float width, float height,
                    int segments, float start, float end) {
          predraw();
          TextureImpl.bindNone();
          currentColor.bind();

          while (end < start) {
                    end += 360;
          }
          float cx = x1 + (width / 2.0f);
          float cy = y1 + (height / 2.0f);
          LSR.start();
          int step = 360 / segments;
          for (int a = (int) start; a < (int) (end + step); a += step) {
                    float ang = a;
                    if (ang > end)
                              ang = end;
                    float x = (float) (cx + (FastTrig.cos(Math.toRadians(ang)) * width / 2.0f));
                    float y = (float) (cy + (FastTrig.sin(Math.toRadians(ang)) * height / 2.0f));
                    LSR.vertex(x,y);
          }
          LSR.end();
          postdraw();
}
```

Method 1:

Method 2:

Method 3:

Other Size Measures

Halstead's Measures

- *Program Vocabulary* - the number of unique tokens
- *Program Length* - size metric based on the number of times the unique tokens appear in the code
- *Program volume* - storage required for the program

Function Point Analysis

- A unit of measurement to express the amount of business functionality an information system provides to a user. (Wikipedia)
- Great in concept in that it tries to remove the bias created by language or coding style. But difficult in practice as it requires *subjective* forms of rating *complexity* of the problem space

Some fairly obvious ones have also been proposed:

- Counting the number of objects in a package or system
- Counting files, methods, amount of documentation, etc

So are the Size instruments we've discussed *measures* or *metrics?*

Code Complexity

Does Size → Complexity or vice-versa?

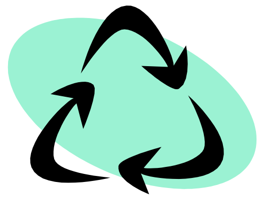
- *Well, generally the bigger it is, the more complex it is*
  - But not necessarily the case; small amounts of code can be complex too, just as big sections of code can be trivial (e.g. Javabeans)
- Again, size is typically used as the basis for estimating developer effort and number of defects

Complexity metrics

- Attempt to capture how “hard” a section of code is
- More complex code generally takes longer to implement, requires greater or specialized expertise, is harder to test, and is usually a source of higher defect injection rates

Code Complexity Metrics

McCabe's Cyclomatic Complexity



- Goal: determine the complexity of a code module independent of its size.
- The *number of linear segments* of code in a code fragment
  - Very useful for constructing unit test cases
- To compute, construct a graph G where:
  - Each statement is a node, or *vertice* in the graph
  - Each conditional statement with multiple outcomes has an arc or *edge* connecting the statement to the next statement if that conditional branch is followed.
  - After constructing the graph, compute:

```
CC(G) = E – N + 2
```

  - Where G is the graph, |E| the # edges, |N| the # of nodes
  - Note CC(G) of a code segment with no conditionals is 1.

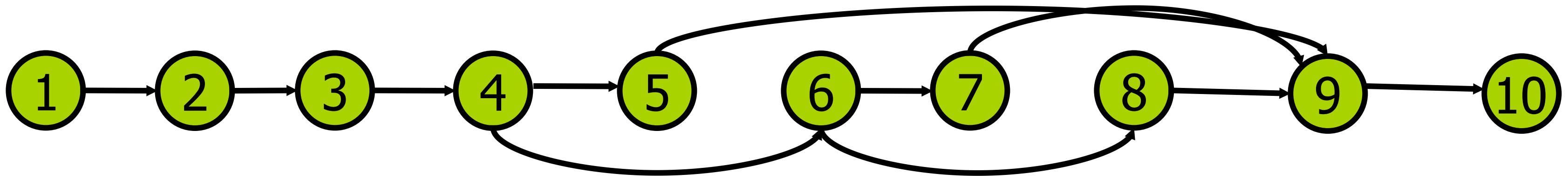
Cyclomatic Complexity - Example

Doesn’t this look familiar???

```
1   Node getSecondElement() {
2     Node head = getHead();
3     Node rval;;
4     if (head == null)
5       rval = null;
6     else if (head.next == null)
7       rval = null;
8     else rval = head.next.node;
9     return rval;
10  }
```

What is CC(G)?

Can you rewrite to make CC(G) less?



```
1. public void drawArc(float x1, float y1, float width, float height,
            int segments, float start, float end) {
2.     predraw();
3.     TextureImpl.bindNone();
4.     currentColor.bind();

5.     while (end < start) {
6.         end += 360;
       }
7.     float cx = x1 + (width / 2.0f);
8.     float cy = y1 + (height / 2.0f);
9.     LSR.start();
10.    int step = 360 / segments;
11.    for (int a = (int) start; a < (int) (end + step); a += step) {
12.        float ang = a;
13.        if (ang > end)
14.            ang = end;
15.        float x = (float) (cx + (FastTrig.cos(Math.toRadians(ang))
* width / 2.0f));
16.        float y = (float) (cy + (FastTrig.sin(Math.toRadians(ang))
* height / 2.0f));

17.        LSR.vertex(x,y);
       }
18.    LSR.end();
19.    postdraw();
}
```

What is CC(G)? 4

21 edges - 19 nodes +2 = 4

How does this map to "linearly independent segments"?

You have 4 ways of going through
This method: straight through, then
5 -> 7, 11 -> 18, 13 -> 15
This gives us a foundation as to why edge coverage is so important!

Other Complexity Metrics

Just a smattering…

- Fog index – the average lengths of words and sentences in documentation. The longer it is the more complex the solution.
- Depth of conditionals – Nesting of if/else or loop constructs even harder to check than CC measure indicates.
- Length of identifiers – if a developer practices good mnemonic naming, then presumably s/he writes more understandable code.

Summary:

- Complexity metrics try to put a number on that inherent property of *hard problems*.
  - And that is hard to do!!!
- It is helpful to understand if your measure represents the *problem* or your *solution* (or, *essential vs. accidental complexity)*
- Size and Complexity are intertwined
  - *If size → complexity, you may have a complex solution*
  - *If complexity → size, you may be working on a hard problem*

Cohesion & Coupling

How well does your project team work together?

- Can you put a number on it?
- Would there be a risk of failure if a specific member of your project team left the team?
- Will another team be able to pick up your project?

Definitions:

- Cohesion is the extent to which things work well together
- Coupling is the degree to which one thing can work independently of another (or not).

Cohesion Metrics

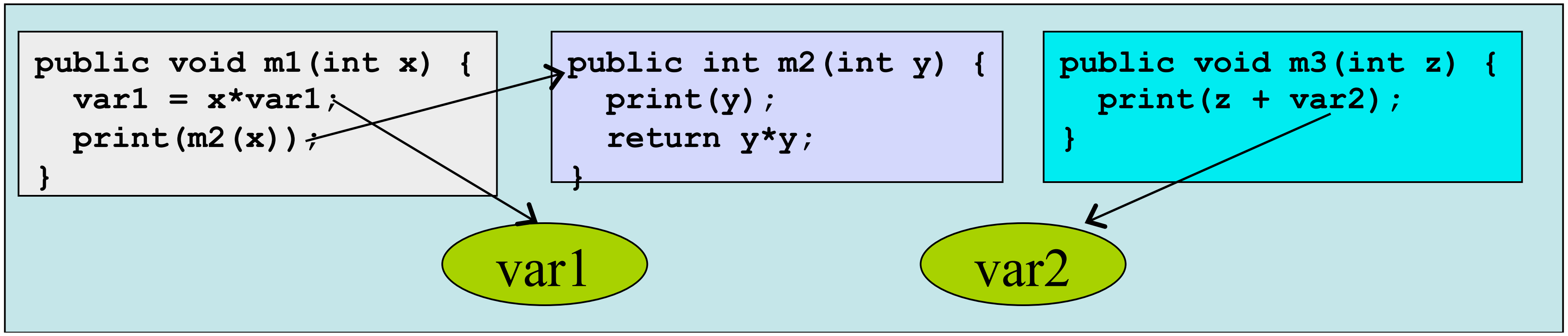
Cohesion: *extent to which a module has a single responsibility*

- On approach is to try and count whether a module uses all of its properties in a connected way
- Another is depth of inheritance / # of overriding methods – a deep and complex inheritance hierarchy leads to defects and less reuse.

A Concrete Example: *Hitz & Montazeri (LCOM4)* - measure the number of *connected components* in a class.

- For any pair of methods m1 & m2 in M, add 1 if:

1. m1 and m2 do not call each other --AND--
2. m1 and m2 do not reference common class variables --AND--
3. Transitivity: any method m1 references & any method m2 references do not reference each other
    - Any number greater than 0 is bad.



```
public void m1(int x) {
  var1 = x*var1;
  print(m2(x));
}
```

```
public int m2(int y) {
  print(y);
  return y*y;
}
```

```
public void m3(int z) {
  print(z + var2);
}
```

Coupling Metrics

Coupling: extent to which the existence and/or operation of a module relies on the existence (operation) of another module

Measure Goal: the number of types the class knows about, and the nature of dependency between classes

- Coupling is considered poor OO as it inhibits the ability to reuse or delegate.

Example:

- Brito & Abreu (CF): Actual coupling / Max possible
  - Take 2 classes C1, C2. C1 is coupled to C2 iff C1 accesses a method or variable on C2.
  - Note the relationship is directed, so max = $\binom{n}{n-2}$

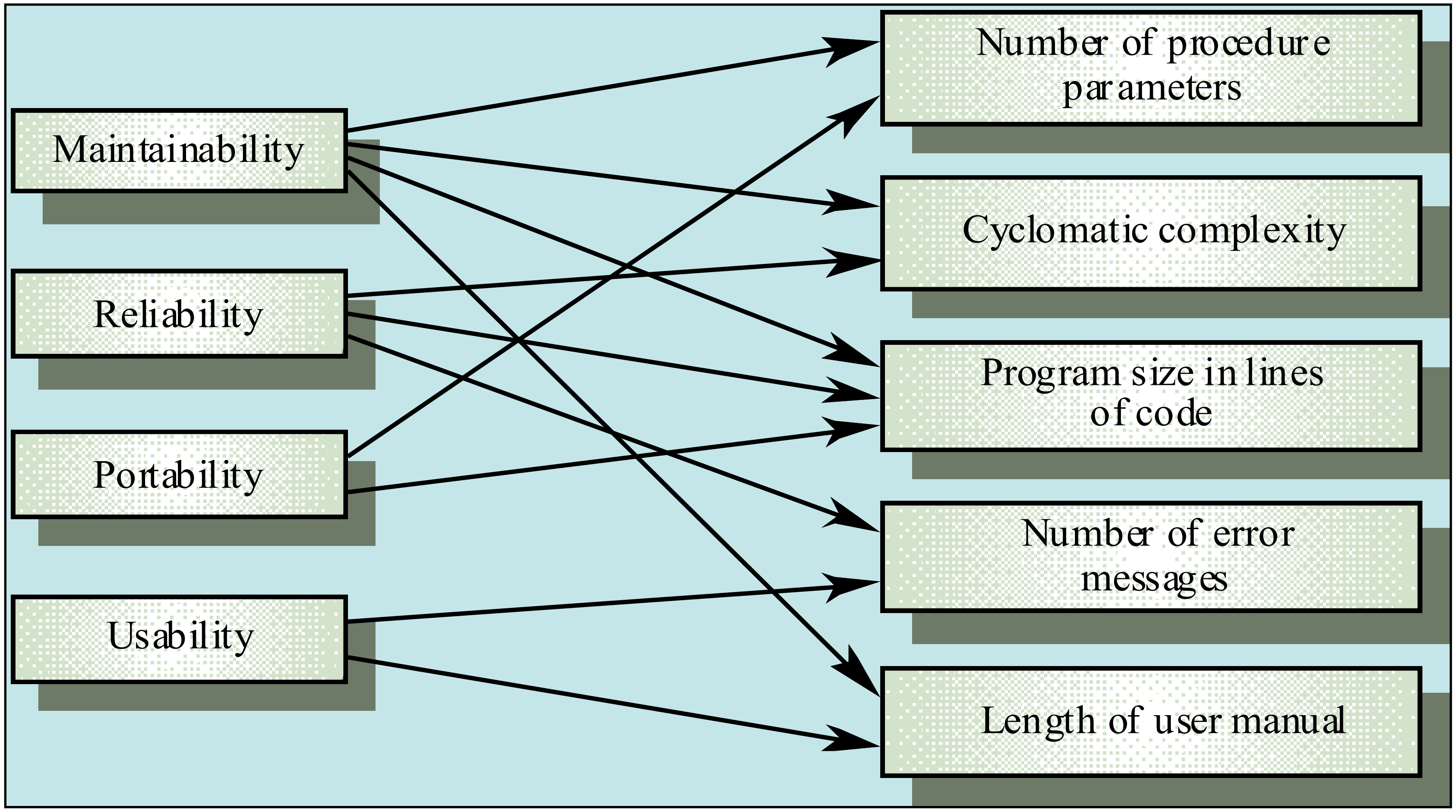
Other Coupling Metrics:

Fan-in / Fan-out – ratio of number of functions that call this function versus the number of functions this function calls.

Several other coupling metrics exist, most all based on counting the number of times a type appears within another class' scope

Why are we measuring again?

We are trying to tie it back to quality attributes!
Don’t say it if you can’t back it up with quantifiable data!!!



Summary: Software Measurement

Software measurement is concerned with quantifying a software project, product, or process

- The systematic use of measurement in software dev is still uncommon
- Allows for objective comparisons between them
- There are few standards in this area

Why do we measure?
- To characterize
- To evaluate
- To predict
- To improve

Metrics Assumptions

- A software property can be measured
- Relationship between measures & what we want to know
- This relationship has been formalized and validated
- This validated interpretation represents our quality attributes

Measures, Metrics, and Agility

- Measures and Metrics still not up there with unit tests, pair programming and static analysis, but interest is growing
- M & M can be automated and included in our CI & Test process
- M & M can provide a basis for sprint or project retrospectives
- M & M can help define a team’s velocity
- M & M help identify “hotspots” – high code risk - quickly