Software Metrics

Disclaimer

There's a plethora of testing tools and static analyzers that compute metrics.
We will not be focusing here on tools but rather on concepts.

General Introduction

Definitions

- *Measure* - quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.
  – E.g., Number of errors

- *Metric* - quantitative measure of degree to which a system, component or process possesses a given (quality?) attribute. "A handle or guess about a given (quality?) attribute."
  – E.g., Number of errors found per person hours

Why Measure Software?

- Determine the quality of the current product or process

- Predict qualities of a product/process

- Improve quality of a product/process

Motivation for Metrics

- Estimate the cost & schedule of future projects
- Evaluate the productivity impacts of new tools and techniques
- Establish productivity trends over time
- Improve software quality
- Forecast future staffing needs
- Anticipate and reduce future maintenance needs

Example Metrics

- Defect rates
  - (where a defect is something less serious than an error wrt reqs)
- Error rates
- Measured by:
  - individual
  - Module/class/procedure
- Errors should be categorized by origin, type, cost
  - It is a luxury to invest in root cause analysis

Product vs. Process

- Process Metrics
  - Insights of process paradigm, software engineering tasks, work product, or milestones
  - Lead to long term process improvement
- Product Metrics
  - Assess the state of the project
  - Track potential risks
  - Uncover problem areas
  - Adjust workflow or tasks
  - Evaluate teams ability to control quality

Types of Measures

- Direct Measures (internal attributes)
  - Cost ($$), effort (in man/days), LOC (lines of code), response speed, memory footprint
  - white box viewpoint
- Indirect Measures (external attributes)
  - Functionality, complexity, efficiency, reliability, maintainability
  - Black box viewpoint
- Both pertain to quality?

Size-Oriented Metrics

Size of the software produced:
- *LOC* - Lines Of Code
- *KLOC* - 1000 Lines Of Code
- *SLOC* – Statement Lines of Code (ignore whitespace)

- Popular because easy to compute
- Typical Measures:
  - Errors/KLOC, Defects/KLOC, Cost/LOC, Documentation Pages/KLOC

COMP 4004 – T2a 9

Complexity Metrics

- LOC - a 'rough' function of complexity
- Halstead's Software Science
  - (entropy measures, ie measures towards (quality?) equilibrium…)
  - n_1 - number of distinct operators
  - n_2 - number of distinct operands
  - N_1 - total number of operators
  - N_2 - total number of operands

COMP 4004 – T2a 10

Example

```
if (k < 2)
{
  if (k > 3)
    x = x*k;
}
```

- Distinct operators: if ( ) { } > < = * ;
- Distinct operands: k 2 3 x
- $n_1 = 10$
- $n_2 = 4$
- $N_1 = 13$
- $N_2 = 7$

Halstead's Metrics

- Amenable to experimental verification [1970s]

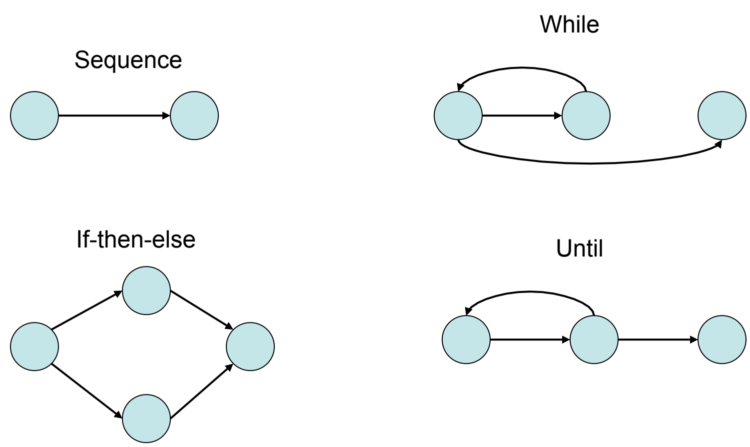
- Program length: $N = N_1 + N_2$ (in ex: 20)
- Program vocabulary: $n = n_1 + n_2$ (in ex: 14)
- Volume $V = N * \log_2 n$ (in ex: 76.14)
- Difficulty D = (n1/2) + (N2/n2) (in ex: 6.75)
- Effort E = D x V (in ex: 514)
- Time to program T = (E / 18) seconds (in ex: 29 s)
- Number of delivered bugs B = V / 3000 (in ex: 0.025)

- Welcome to the science of metrics, for which interpretation is often an art… For example, D and E are taken to pertain to understandability…

McCabe's Complexity Measures

- McCabe's metrics are based on a control flow representation of the program.
- A control flow graph is used to depict control flow.
- Nodes represent processing tasks (one or more code statements)
- Edges represent control flow between nodes

Flow Graph Notation



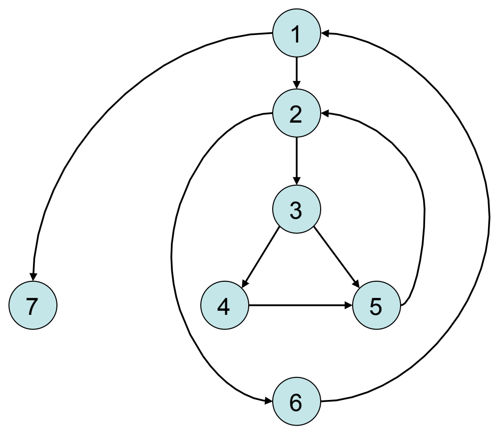
Cyclomatic Complexity

- Defined as the set of independent paths through the control flow graph
- $V(G) = E - N + 2$
  - E is the number of flow graph edges
  - N is the number of nodes
- $V(G) = P + 1$
  - P is the number of predicate nodes

Example

```
i = 0;
while (i<n-1) do
  j = i + 1;
  while (j<n) do
    if A[i]<A[j]
     then
      swap(A[i], A[j]);
  end do;
  i=i+1;
end do;
```

Flow Graph



Computing V(G)

- V(G) = 9 – 7 + 2 = 4
- V(G) = 3 + 1 = 4
- Basic paths are:
  - 1, 7
  - 1, 2, 6, 1, 7
  - 1, 2, 3, 4, 5, 2, 6, 1, 7
  - 1, 2, 3, 5, 2, 6, 1, 7

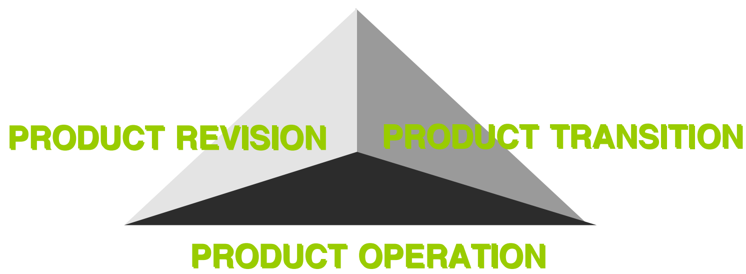
Meaning of V(G)

- Complexity increases with the number of decision paths and loops
- V(G) is a quantitative measure of the testing difficulty and, ultimately, an indication of reliability
- Experimental data shows value of V(G) should be no more then 10 - testing is very difficult above this value

McClure's Complexity Metric

- Complexity = C + V
  - C is the number of comparisons in a module
  - V is the number of control variables referenced in the module (from ifs and loops)
  - Targets decisional complexity
    - Somewhat pertains to path sensitization
- Similar to McCabe's but with regard to control variables.
  - Can this be correlated to software quality?

McCall's Triangle of Quality

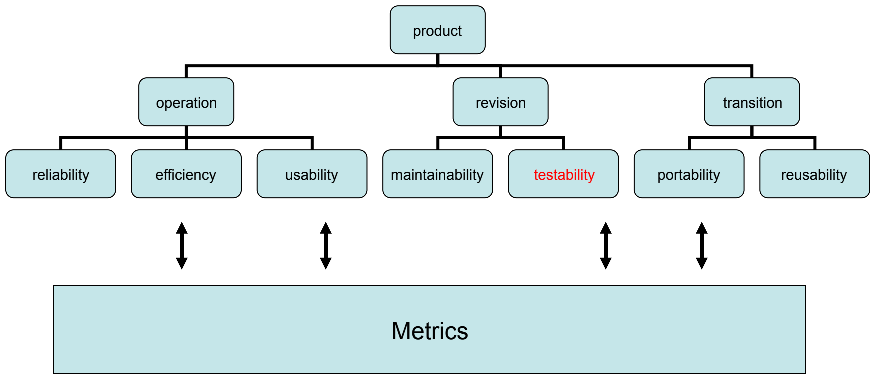


A Comment

McCall's quality factors were proposed in the early 1970s. They appear to be as valid today as they were at that time. It's likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

Quality Model

(is ISO 9126 relevant to this?)



High level Design Metrics

- Structural Complexity
- Data Complexity
- System Complexity

- Structural Complexity S(i) of a module i.
  - $S(i) = f_{out}^2(i)$
  - Fan out is the number of modules immediately subordinate (directly invoked).

Design Metrics

- Data Complexity D(i)
  - $D(i) = v(i)/[f_{out}(i)+1]$
  - v(i) is the number of inputs and outputs passed to and from i

- System Complexity C(i)
  - C(i) = S(i) + D(i)
  - As each C(i) increases the overall complexity of the architecture increases

System Complexity Metric

- Another metric:
  - $length(i) * [f_{in}(i) + f_{out}(i)]^2$
  - Length is LOC
  - Fan in is the number of modules that invoke i

- The real question: what are ‘good’ numbers for each of these metrics?

Coupling for a module

- Data and control flow: (a key distinction)
  - d_i input data parameters
  - c_i input control parameters
  - d_o output data parameters
  - c_o output control parameters
- Global
  - g_d global variables for data
  - g_c global variables for control
- Environmental
  - w fan in
  - r fan out

Metrics for Coupling

- $M_c = k/m$, $k=1$
  - $m = d_i + ac_i + d_o + bc_o + g_d + cg_c + w + r$
  - Key point: a, b, c, k can be adjusted based on actual data… but how is this done?
    - This computation can be so subjective that most will rely on simpler (if not simplistic) metrics…

Component Level Metrics

- Cohesion (internal interaction) – pertains to data members
- Coupling (external interaction) - a function of input and output parameters, global variables, and modules called
- Complexity of program flow - hundreds have been proposed (e.g., cyclomatic complexity)
- Cohesion – difficult to measure
  - Bieman '94, TSE 20(8)

Using Metrics

- The Process
  - Select appropriate (??) metrics for problem
  - Use metrics on problem
  - Assess and generate feedback
- Steps:
  - Formulation
  - Collection
  - Analysis
  - Interpretation
  - Feedback

Metrics for OO Software

Metrics for the Object Oriented

- Chidamber & Kemerer '94 TSE 20(6)
- Metrics specifically designed to address object-oriented software
- Class-oriented metrics:
  - No need for procedure level metrics
  - Cluster level metrics is simply too complex
  - Simple direct measures

Chidamber and Kemerer Metrics

- Weighted methods per class (MWC)
- Depth of inheritance tree (DIT)
- Number of children (NOC)
- Coupling between object classes (CBO)
- Response for class (RFC)
- Lack of cohesion metric (LCOM)

33

Weighted methods per class (WMC)

$$WMC = \sum_{i=1}^{n} c_i$$

- c_i is the *complexity* of each method M_i of the class
  - Often, only public methods are considered
- Complexity may be the McCabe complexity of the method
- Smaller values are better
- Perhaps the average complexity per method is a better metric

34

Weighted Methods per Class

- Viewpoints from Chidamber and Kemerer:

-The number of methods and complexity of methods is an indicator of ***how much time and effort is required to develop and maintain*** the object

-The ***larger the number of methods*** *in an object, the greater the potential* ***impact on the children***

-Objects with ***large number of methods*** are likely to be more application specific, ***limiting possible reuse***

Depth of inheritance tree (DIT)

- For the system under examination, consider the hierarchy of classes
- DIT is the *length of the maximum path from the node to the root of the tree*
- Relates to the scope of the properties
  - How many ancestor classes can potentially affect a class
- **Smaller values are better**

Number of children (NOC)

- For any class in the inheritance tree, NOC is the number of *immediate* children of the class
  - The number of direct subclasses
- How would you interpret this number?

- **A moderate (??) value indicates scope for reuse and high values may indicate an inappropriate abstraction in the design**

37

Number of Children

- Viewpoints:
- As *NOC grows*, *reuse increases* - but the abstraction may be diluted

- *Depth is generally better* than breadth in class hierarchy, since it *promotes reuse* of methods through inheritance
  - Really?? Open-closed principle? Does this not contradict heuristic for DIT?

- Classes *higher up in the hierarchy* should have *more sub-classes* then those lower down

- NOC gives an idea of the potential *influence a class has on the design*: classes with large number of children may require more testing

COMP 4004 – T2a 38

Coupling between Classes

- CBO is the number of collaborations between two classes (fan-out of a class C)
  - the number of other classes that are referenced in the class C (where a *reference* to another class, A, is a reference to a method or a data member of class A)
- Viewpoints:
- High fan-outs denote class coupling to other classes/objects and thus are undesirable. High fan-ins denote good designs and a high level of reuse
- Not possible to maintain high fan-in and low fan outs across the entire system
- Excessive coupling indicates weakness of class encapsulation and may inhibit reuse
- High coupling also indicates that more faults may be introduced due to inter-class activities

Response for class (RFC)

- Mc_i: # of methods called in response to a message that invokes method M_i
  - Fully nested set of calls
- Smaller numbers are better
  - Larger numbers indicate increased complexity and debugging difficulties

$$\sum_{i=1}^{n} Mc_i$$

If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated

Lack of cohesion metric (LCOM)

- Number of methods in a class that reference a specific instance variable
- A measure of the "tightness" of the code
- If a method references many instance variables, then it is more complex and less cohesive
- The larger the number of similar methods in a class the more cohesive the class is
- "Cohesion of methods within a class is desirable, since it promotes encapsulation" (??)

41

Lack of Cohesion in Methods

- LCOM – poorly described in Pressman
- Class C_k with n methods $M_1, \ldots M_n$
- I_j is the set of instance variables used by M_j

COMP 4004 – T2a 42

LCOM

- There are n such sets I_1 ,…, I_n
  - $P = \{(I_i, I_j) \mid (I_i \cap I_j) = \emptyset\}$
  - $Q = \{(I_i, I_j) \mid (I_i \cap I_j) \neq \emptyset\}$
- If all n sets I_i are $\emptyset$ then $P = \emptyset$

- LCOM = $|P| - |Q|$, if $|P| > |Q|$
- LCOM = 0 otherwise

Example LCOM

- Take class C with M_1, M_2, M_3
- $I_1 = \{a, b, c, d, e\}$
- $I_2 = \{a, b, e\}$
- $I_3 = \{x, y, z\}$
- $P = \{(I_1, I_3), (I_2, I_3)\}$ //those do not intersect
- $Q = \{(I_1, I_2)\}$ //those that do

- Thus LCOM = 1

Explanation

- LCOM is the number of empty intersections minus the number of non-empty intersections
- This is a notion of degree of similarity of methods
- If two methods use common instance variables then they are (??) similar
- LCOM of zero is not maximally cohesive
- $|P| = |Q|$ or $|P| < |Q|$

Some other cohesion metrics

LCOM3	Consider an undirected graph G, where the vertices are the methods of a class, and there is an edge between two vertices if the corresponding methods use at least an attribute in common. LCOM3 is then defined as the number of connected components of G.
LCOM4	Like LCOM3, where graph G additionally has an edge between vertices representing methods m and n, if m invokes n or vice versa.
Co (connectivity)	Let V be the number of vertices of graph G from measure LCOM4, and E the number of its edges. Then $Co = 2 \cdot \frac{\lvert E\rvert - (\lvert V\rvert - 1)}{(\lvert V\rvert - 1) \cdot (\lvert V\rvert - 2)}$.
LCOM5	Consider a set of methods $\{M_i\}$ ($i=1,...,m$) accessing a set of attributes $\{A_j\}$ ($j=1,...,a$). Let $\mu(A_j)$ be the number of methods which reference attribute A_j. Then $LCOM5 = \frac{\frac{1}{a}\left(\sum_{j=1}^{a} \mu(A_j)\right) - m}{1 - m}$

Class Size

- CS
  - Total number of operations (inherited, private, public)
  - Number of attributes (inherited, private, public)
- May be an indication of too much responsibility for a class

Number of Operations Overridden

- NOO
- A large number for NOO indicates possible problems with the design
- Poor abstraction in inheritance hierarchy

Number of Operations Added

- NOA
- The number of operations added by a subclass
- As operations are added the subclass 'moves away' from the parent class
- As depth increases NOA should decrease

Method Inheritance Factor

$$\text{MIF} = \frac{\sum_{i=1}^{n} M_i(C_i)}{\sum_{i=1}^{n} M_a(C_i)} \quad .$$

- $M_i(C_i)$ is the number of methods inherited and not overridden in C_i
- $M_a(C_i)$ is the number of methods that can be invoked with C_i
- $M_d(C_i)$ is the number of methods declared in C_i

MIF

- $M_a(C_i) = M_d(C_i) + M_i(C_i)$
- All that can be invoked = new or overloaded + things inherited

- MIF is [0,1]
- MIF near 1 means little specialization
- MIF near 0 means large change

Coupling Factor

$$CF = \frac{\sum_i \sum_j is\_client(C_i, C_j)}{(TC^2 - TC)} \quad .$$

- is_client(x,y) = 1 iff a relationship exists between the client class and the server class. 0 otherwise

- (TC^2-TC) is the total number of relationships possible (??)

- CF is [0,1] with 1 meaning high coupling

Polymorphism Factor

$$PF = \frac{\sum_i M_o(C_i)}{\sum_i \left[M_n(C_i) * DC(C_i) \right]}.$$

- $M_n()$ is the number of new methods
- $M_o()$ is the number of overriding methods
- DC() is the number of descendent classes of a base class
- The factor is computed as the number of methods that redefine inherited methods, divided by maximum number of possible distinct polymorphic situations

Operational Oriented Metrics

- Average operation size (LOC, volume)
- Number of messages sent by an operator
- Operation complexity – cyclomatic
- Average number of parameters/operation
  - The larger the number the more complex the collaboration

Measuring Encapsulation?

- Lack of cohesion indicates potential lack of encapsulation
- Consider % of public and protected
  – What would this indicate??
- Public access to data members
  – What would this indicate??

Inheritance

- Number of root classes
- Fan in – multiple inheritance
- NOC, DIT, etc.

Appendix

Main Results

- Metric definitions – first suite:

Chidamber and Kemerer's (CK) Metric Suite (Class Metrics Only) [3]	
Metric Name Value	**Definition**
Weighted Methods Per Class (WMC)	*Sum of complexities of local methods of a class. For simple WMC, when all complexities are unity, same as number of class methods.*
Depth of Inheritance Tree (DIT)	*Max number of edges between a given class and a root class in an inheritance graph (0 for a class which has no base classes).*
Num.Children(NOC)	*A count of the number of direct children of a given class.*
Coupling Between Objects (CBO)	*Counts other classes whose attributes or methods are used by the given class plus those that use the attributes or methods of the given class.*
Response For a Class (RFC)	*A count of all of local methods of a class plus all of methods on other classes directly called by any of the methods on the class.*
Lack of Cohesion of Methods (LCOM)	*Num. of disjoint sets of local methods, no two sets intersect, any two methods on same set share at least one local variable* (1998 definition).

Main Results

• Metric definitions – second suite:

Fernando Brito e Abreu's MOOD Metric Suite (Class Metrics Only) [2]	
Attribute Hiding Factor (AHF)	*[1-total num. visible (can be accessed)) attributes in a set of classes] / total num. attributes in the set. Measures visibility of a class definition.*
Method Hiding Factor (MHF)	*[1-total num. visible (can be called) methods in a set of classes] / total num. methods in the set. Measures visibility of a class definition.*
Attribute Inheritance Factor (AIF)	*The ratio of inherited attributes to the total number of attributes in a class.*
Method Inheritance Factor (MIF)	*The ratio of inherited methods to the total number of methods in a class.*

59

Main Results

• Metric definitions – third suite:

Bansiya and Davis' QMOOD Metric Suite (Class Metrics Only) [1]	
Avg. Num. Ancestors (QMOOD_ANA)	*Average of DIT for all classes in the system.*
Cohesion Among Methods (QMOOD_CAM)	*A measure of cohesion that is based on the similarity of method signatures in a class. Included for completeness; not implemented in this research.*
Class Interface Size (QMOOD_CIS)	*The count of public methods in a class.*
Data Access Metric (QMOOD_DAM)	*The ratio of private or protected attributes to the total number of attributes declared in a class.*
Direct Class Coupling (QMOOD_DCC)	*A count of classes that accept instances of a given class as a parameter plus classes including attributes of the given class' type.*
Measure of Aggregation (QMOOD_MOA)	*The percentage of data declarations in the system whose types are of user defined classes, as opposed to those of system defined classes such as integers, real numbers, etc.*
Measure of Fnctnl. Abstraction (QMOOD_MFA)	*Same as MOOD_MIF.*
Number of Methods (QMOOD_NOM)	*The number of methods in a class. Same as WMC when weights of the methods in the class equal unity.*

Main Results

- Software examined: Mozilla Rhino – an open source implementation of JavaScript written in Java
- An example of the use of the agile software development in open source software
- Six Rhino versions were analyzed in this case study
- Delivery cycle time from 2 to 16 months

61

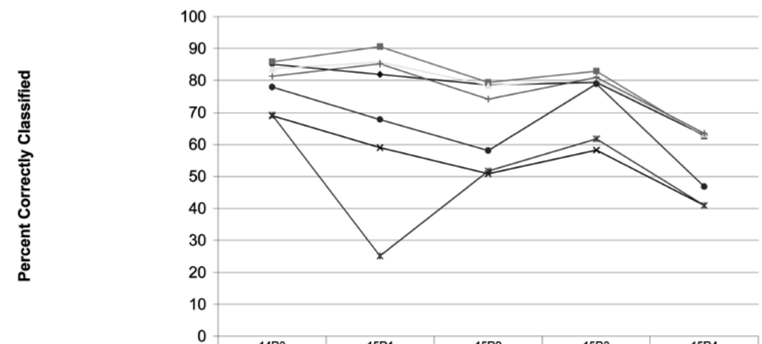
Main Results

- Hypotheses:
- Hypothesis 1: OO metrics can identify fault-prone classes in traditional and highly iterative or agile developed OO software during its initial delivery
- Hypothesis 2: OO metrics can identify fault-prone classes in multiple sequential releases of OO software systems developed and using highly iterative or agile software development process

62

Main Results

- Model validation:



	14R3	15R1	15R2	15R3	15R4
CK With WMC (All)	85.1	81.9	78.7	79.4	62.9
CK With WMC-McCabe (All)	85.9	90.6	79.4	83	62.6
CK With RFC (All)	83.7	85.7	78.5	80.1	63.2
MOOD With AIF (All)	69	59	50.9	58.3	41
MOOD With MIF (All)	69.2	25.1	51.7	61.8	41
QMOOD With CIS (All)	78	67.8	58.1	79	46.8
QMOOD With NOM (All)	81.3	85.2	74.2	81	63.5

63

Main Results

- CK and QMOOD suites contain similar components and produce statistical models that are effective in detecting error-prone classes
- MOOD metrics suite are not good class fault-proneness predictors
- The produced models can be useful in assessing quality in OO classes developed using modern highly iterative or agile software development processes

64