Software Metrics

Martin Ward

Reader in Software Engineering

martin@gkc.org.uk

Software Technology Research Lab
De Montfort University

Classification of Software Metrics

What is measured	
Product	Process

Results of measurement	
Objective	Subjective

Way of measuring	
Primitive	Derived

Possible values				
Nominal	Ordinal	Interval	Proportional	Absolute

Lines of Code Metrics

- SLOC: Source Lines of Code
- CLOC: Comment Lines of Code
- S&CLOC: Source and Comment LOC
- BLOC: Blank Lines of Code
- LOC: Lines of Code
- PLOC: Physical Lines of Code
- LLOC: Logical Lines of Code

Function Points

- A way to assess the complexity of software from functional requirements, rather than source code
- Different standards:
  - IFPUG: International Function Point User Group
  - NESMA: Netherlands Software Metrics Association
  - COSMIC: Common Software Measurement International Consortium
  - MkII: Based on IFPUC
- Function points are used to:
  - Quantify system functionality
  - Measure development and maintenance of software independently of implementation, project and organisation

Function Points — IFPUG

- System functionality divided into components with type:
  - Transactional functions
    - External input
    - External output
    - External inquiry
  - Data functions
    - Internal logical files
    - External interface files
- Components are assigned to a complexity class
  - Low, medium, high

Function Points — IFPUG

Every component is assigned a number of *function points* according to complexity and number of appearances:

$$\text{fp} = \text{appearences} \times \text{complexity}$$

Component type	Low	Medium	High
External input	3	4	6
External output	4	5	7
External inquiry	3	4	6
Internal logical file	7	10	15
External interface file	5	7	10

Unadjusted Function Points (UFP) =

Sum of FPs of all components in the system

Function Points — IFPUG

- Adjusted Function Points (AFP)

$$\text{AFP} = \text{UFP} \times \text{VAF}$$

- Value Adjustment Factor (VAF)

$$\text{VAF} = \text{TDI} \times 0.01 + 0.65$$

- Total Degree of Influence (TDI) =
  Sum of 14 General System Characteristics (GSC)

Function Points — IFPUG

General System Characteristics	
1. Data communication	8. On-line update
2. Distributed data processing	9. Complex processing
3. Performance	10. Reusability
4. Heavily used configuration	11. Installation ease
5. Transaction rate	12. Operational ease
6. Online data entry	13. Multiple sites
7. End-user efficiency	14. Facilitate change

Cyclomatic Complexity

- Cyclomatic Complexity measures the amount of decision logic in a single module. It is defined in terms of the control flow graph of the module.

Cyclomatic Complexity

- Cyclomatic Complexity measures the amount of decision logic in a single module. It is defined in terms of the control flow graph of the module.
- A **Control Flow Graph** is a directed graph with a single entry node and a single exit node. For each node N in the graph there is a path from the entry node to N and a path from N to the exit node.

Cyclomatic Complexity

- Cyclomatic Complexity measures the amount of decision logic in a single module. It is defined in terms of the control flow graph of the module.
- A **Control Flow Graph** is a directed graph with a single entry node and a single exit node. For each node N in the graph there is a path from the entry node to N and a path from N to the exit node.
- An **execution path** is any path from entry to exit.

Cyclomatic Complexity

- Cyclomatic Complexity measures the amount of decision logic in a single module. It is defined in terms of the control flow graph of the module.
- A **Control Flow Graph** is a directed graph with a single entry node and a single exit node. For each node N in the graph there is a path from the entry node to N and a path from N to the exit node.
- An **execution path** is any path from entry to exit.
- Any control flow graph will become **strongly connected** if a single edge is added from exit to entry. Intuitively, this represents the control flow through the rest of the system and the computing environment.

Cyclomatic Complexity

- Cyclomatic complexity is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module

Cyclomatic Complexity

- Cyclomatic complexity is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module
- This turns out to be equal to $e - n + 2$ where e is the number of edges and n is the number of nodes in the graph. This is the cyclomatic number $(e' - n + 1)$ of the strongly connected graph formed from the control flow graph by adding an edge from the exit node to the entry node. ($e' = e + 1$ is the edge set of the graph with this extra node).

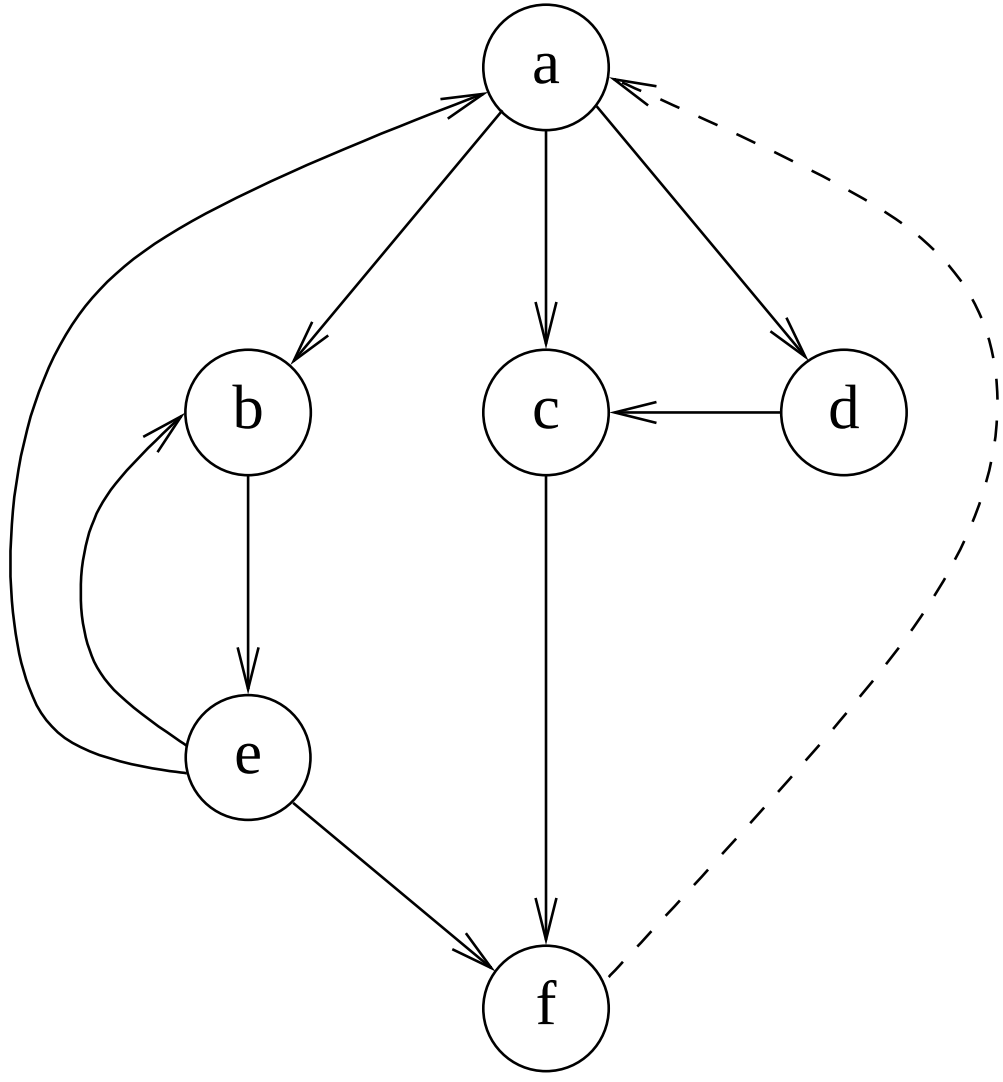
Cyclomatic Complexity

- Cyclomatic complexity is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module
- This turns out to be equal to $e - n + 2$ where e is the number of edges and n is the number of nodes in the graph. This is the cyclomatic number $(e' - n + 1)$ of the strongly connected graph formed from the control flow graph by adding an edge from the exit node to the entry node. ($e' = e + 1$ is the edge set of the graph with this extra node).
- A simple way to compute cyclomatic complexity is:

$$1 + \sum_{n} (\mathsf{out\_edges}(n) - 1)$$

Cyclomatic Complexity

For this graph: $v(G) = e - n + 2 = 9 - 6 + 2 = 5$



Cyclomatic Complexity

- If a flowchart has no edges crossing each other, and divides the plane into R regions (including the infinite region "outside" the graph), then the complexity is just R. This follows from Euler's formula, that for planar graphs:

$$n - e + R = 2$$

Cyclomatic Complexity

- If a flowchart has no edges crossing each other, and divides the plane into R regions (including the infinite region "outside" the graph), then the complexity is just R. This follows from Euler's formula, that for planar graphs:

$$n - e + R = 2$$

- The most efficient and reliable way to determine complexity is through use of an automated tool.

Cyclomatic Complexity

"Straight line code" with no branches or loops has complexity 1.

Cyclomatic Complexity

"Straight line code" with no branches or loops has complexity 1.

A structured program with binary **if** statements and **while** loops has complexity:

$$\#(\textbf{if}\ \text{statements}) \ + \ \#(\textbf{while}\ \text{loops}) \ + \ 1$$

Cyclomatic Complexity

"Straight line code" with no branches or loops has complexity 1.

A structured program with binary **if** statements and **while** loops has complexity:

$$\#(\textbf{if statements}) \; + \; \#(\textbf{while loops}) \; + \; 1$$

if $x = y$ **then** $z := 1$ **fi**

has complexity 2.

Cyclomatic Complexity

"Straight line code" with no branches or loops has complexity 1.

A structured program with binary **if** statements and **while** loops has complexity:

$$\#(\textbf{if statements}) \ + \ \#(\textbf{while loops}) \ + \ 1$$

if $x = y$ **then** $z := 1$ **fi**

has complexity 2.

if $x = y$ **then** $z := 1$
elsif $p = q$ **then** $z := 2$
elsif $r = s$ **then** $z := 3$
else $z := 4$ **fi**

has complexity 4.

Cyclomatic Complexity

Cyclomatic Complexity

while $n > 0$ **do** $n := n - 1$ **od**

has complexity 2.

Cyclomatic Complexity

while $n > 0$ **do** $n := n - 1$ **od**

has complexity 2.

do $n := n + 1$;
    if $A[n] = x$ **then exit**(1) **fi**;
    if $n = N$ **then exit**(1) **fi od**

has complexity 3.

Note that the **do** ... **od** loop itself does not contribute anything to the cyclomatic complexity, since it is an unconditional loop.

Example

actions prog :

prog $\equiv$

line := “”; $m := 0$; $i := 1$; **call** inhere **end**

loop $\equiv$

$i := i + 1$;

if $i = n + 1$ **then call** alldone **fi**;

$m := 1$;

if item$[i] \neq$ last

then !P write(line **var** os);

line := “”; $m := 0$; **call** inhere **fi**;

call more **end**

inhere $\equiv$

$p :=$ number$[i]$; line := item$[i]$;

line := line $+\!\!+$ “” $+\!\!+$ p; **call** more **end**

more $\equiv$

if $m = 1$

then $p :=$ number$[i]$;

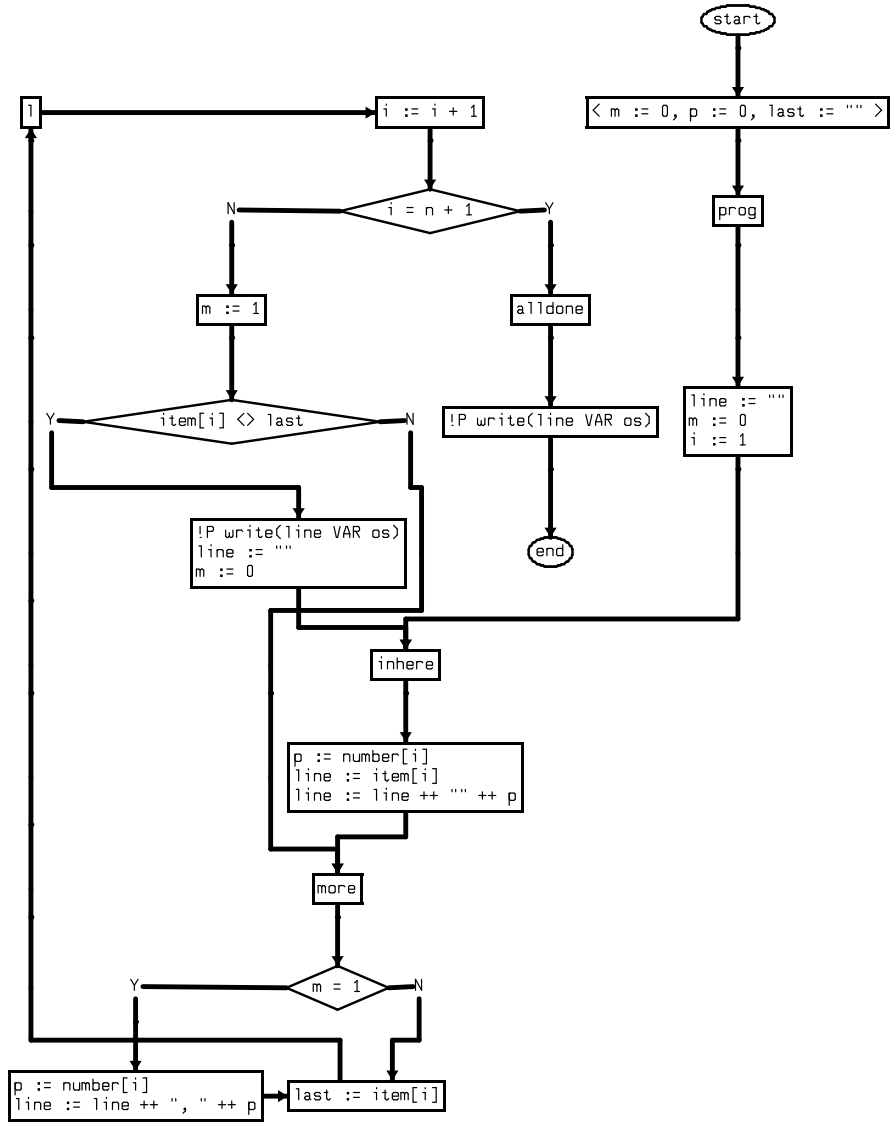
line := line $+\!\!+$ “, ” $+\!\!+$ p **fi**;

last := item$[i]$; **call** loop **end**

alldone $\equiv$

!P write(line **var** os); **call** Z **end endactions**

Example



What is the McCabe complexity of this program?

McCabe Complexity

The complexity is 4.

The same as for the program:

if $x = y$ **then** $z := 1$
elsif $p = q$ **then** $z := 2$
elsif $r = s$ **then** $z := 3$
else $z := 4$ **fi**

McCabe complexity does not take into account the "structuredness" or "unstructuredness" of the code.

This point led to the development of the **essential complexity** metric.

Cyclomatic Compexity Variants

- **Cyclomatic Complexity Metric (v(G))**: a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and therefore, the minimum number of paths that should be tested.

Cyclomatic Compexity Variants

- **Cyclomatic Complexity Metric (v(G))**: a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and therefore, the minimum number of paths that should be tested.
- **Actual Complexity Metric (ac)**: the number of independent paths traversed during testing.

Cyclomatic Compexity Variants

- **Cyclomatic Complexity Metric (v(G))**: a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and therefore, the minimum number of paths that should be tested.
- **Actual Complexity Metric (ac)**: the number of independent paths traversed during testing.
- **Module Design Complexity Metric (iv(G))**: is the complexity of the design-reduced module and reflects the complexity of the module's calling patterns to its immediate subordinate modules.

Metrics

- **Essential Complexity Metric (ev(G))**: a measure of the degree to which a module contains unstructured constructs. This metric measures the degree of structuredness and the quality of the code. It is used to predict the maintenance effort and to help in the modularization process.

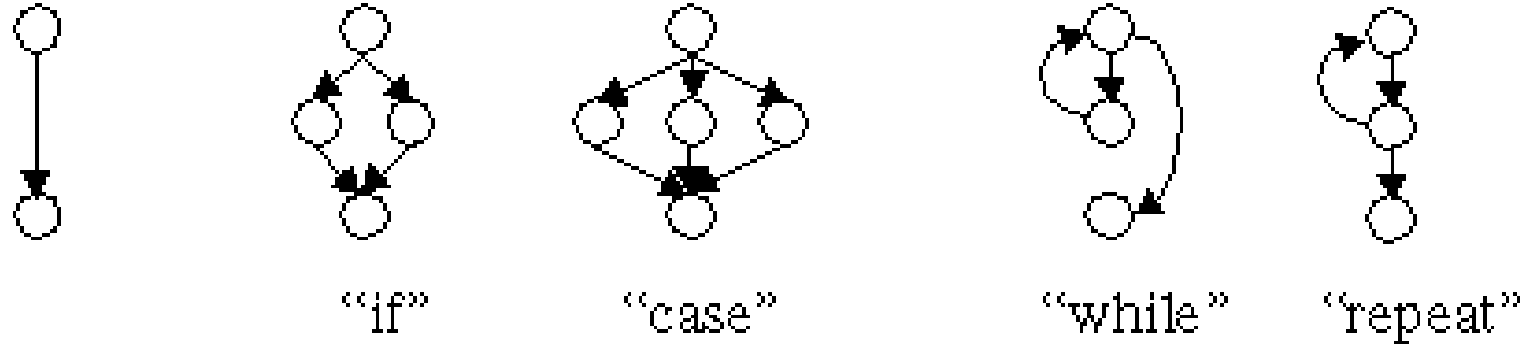
Metrics

- **Essential Complexity Metric (ev(G))**: a measure of the degree to which a module contains unstructured constructs. This metric measures the degree of structuredness and the quality of the code. It is used to predict the maintenance effort and to help in the modularization process.
- **Global Data Complexity Metric (gdv(G))**: quantifies the cyclomatic complexity of a module's structure as it relates to global/parameter data. It can be no less than one and no more than the cyclomatic complexity of the original flowgraph.

Structured Programming

Structured programming avoids unmaintainable "spaghetti code" by restricting the usage of control structures to those that are easily analyzed and decomposed.

Each primitive construct has a single entry and a single exit:



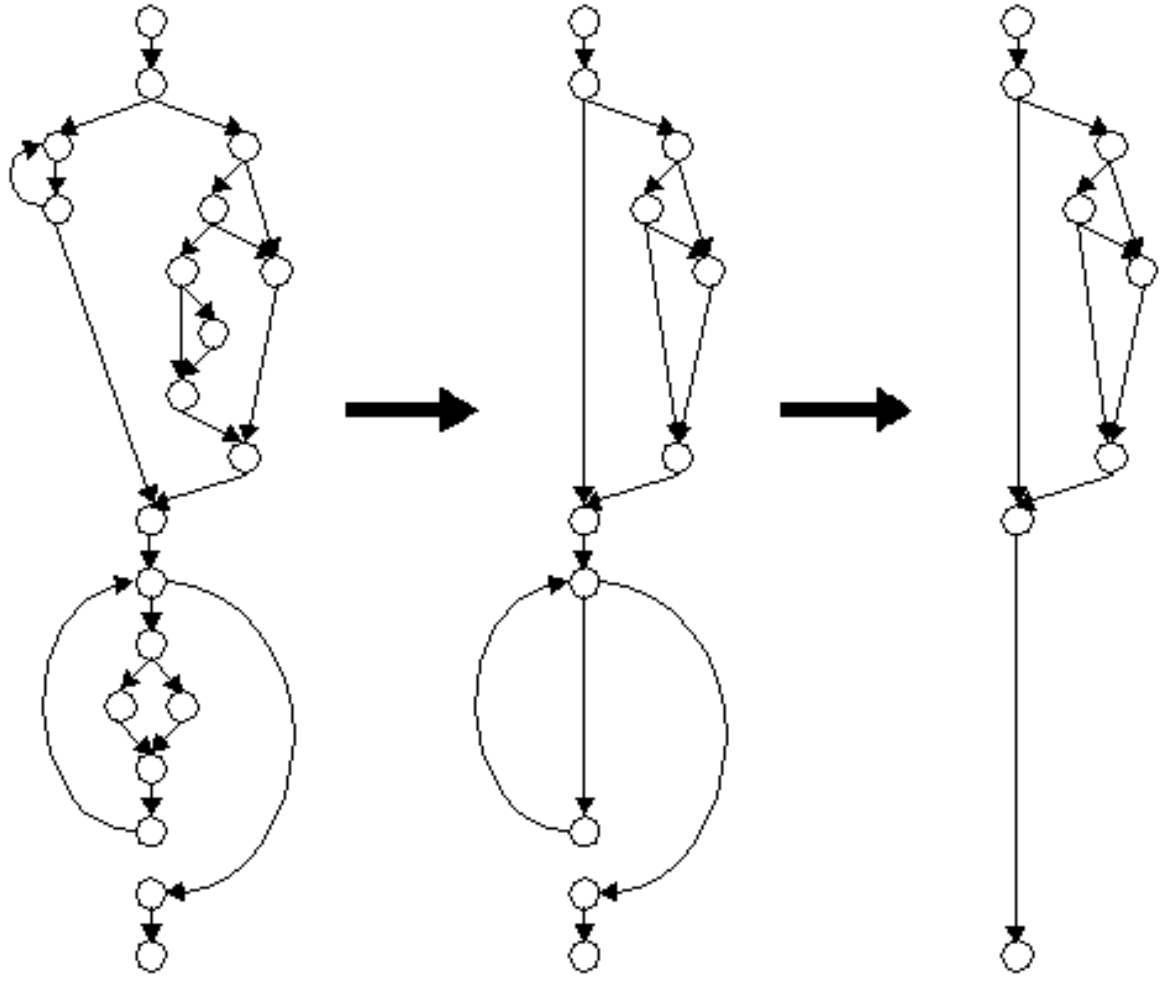
Sequence **Selection** **Iteration**

Essential Complexity

The **essential complexity, ev(G)**, of a module is calculated by repeatedly removing structured programming primitives from the module's control flow graph until the graph cannot be reduced any further, and then calculating the cyclomatic complexity of the reduced graph.

Any fully structured program therefore has an essential complexity of 1. This is true even if the structures are actually implemented using labels and **goto** statements.

Essential Complexity Calculation



Limiting Complexity

McCabe proposed that organisations should limit the complexity of a module to a maximum of 10 (with significant supporting evidence), but limits as high as 15 have been used successfully as well.

A single **switch** or **case** statement with N branches has a complexity of N, but is conceptually simple: so McCabe recommended exempting modules consisting of single multiway decision (**switch** or **case**) statements from the complexity limit.

Rewriting a single multiway decision to cross a module boundary is a clear violation of structured design. Each decision branch can be understood and maintained in isolation, so the module is likely to be reliable and maintainable. Therefore, it is reasonable to exempt modules consisting of a single multiway decision statement from a complexity limit.

Cyclomatic Compexity

Rules of thumb:

$v(G)$	Procedure type	Risk level
1–4	Simple	Low
5–10	Well structured, stable	Low
11–20	Moderately Complex	Medium
21–50	Complex	High
51–	Very complex	Very high

Information Flow

Cyclomatic compexity measures control flow, but not *data flow*, such as parameter passing and variable access.

- **Fan-in:** the amount of information that flows into a procedure
- **Fan-out:** the amount of information that flows out of a procedure

There exists information flow from procedure A to procedure B if:

- A calls B
- B calls A and uses its return value
- Both A and B are called by C, which passes the return value of A to B

Information Flow Complexity

Henry & Kafura (1981)

$$
\begin{aligned}
\text{IFC} &= (\text{fanin} \times \text{fanout})^2 \\
\text{WIFC} &= \text{length} \times \text{IFC} \\
\text{fanin} &= \text{procedures\_called} + \text{parameters\_read} \\
&\quad + \text{global\_vars\_read} \\
\text{fanout} &= \text{procedures\_calling\_this\_procedure} + \text{output\_parameters} \\
&\quad + \text{global\_vars\_written\_to} \\
\text{length} &= \text{logical\_SLOC or cyclomatic\_complexity}
\end{aligned}
$$

Information Flow Complexity

```
char * strncat(char *ret, const char *s2, size_t n)
{
  char *s1 = ret;
  if (n > 0) {
    while (*s1)
      s1++;
    while (*s1++ = *s2++)  {
      if (--n == 0) {
        *s1 = '\0';
        break;
      }
    }
  }
  return ret;
}
```

$$\text{fanin} = 3$$

$$\text{fanout} = 1$$

$$\text{IFC} = 3^2 = 9$$

$$\text{WIFC} = 10 \times 9 = 90$$

Halstead's Metrics

- Operands
  - Variables
  - Constants
- Operators: Symbols, keywords and names that affect operands
  - Arithmetic operators
  - Logical operators
  - Assignments
  - Special symbols
  - Parenthesis
  - If, while, do...
  - Function names

Halstead's Metrics

Basic Attributes:

$$
\begin{aligned}
n_1 &= \text{Number of distinct operators} \\
n_2 &= \text{Number of distinct operands} \\
N_1 &= \text{Total number of operators} \\
N_2 &= \text{Total number of operands}
\end{aligned}
$$

For the `strncat` example:

- Operators: `{}`, `*`, `=`, `if`, `while`, `++`, `--`, `==`, `break`, `return`, `;`
- Operands: `ret`, `s1`, `s2`, `n`, `0`, `'\0'`

Halstead's Metrics

Operators	
{}	4
*	5
=	3
if	2
while	2
++	3
--	1
==	1
break	1
return	1
;	5

Operands	
ret	2
s1	5
s2	1
n	2
0	1
'\0'	1

Attributes	
n_1	11
n_2	6
N_1	28
N_1	12

Halstead's Metrics

Metric	Formula	Value
Program length	$N = N_1 + N_2$	40
Vocabulary size	$n = n_1 + n_2$	17
Program volume	$V = N \cdot \log_2 n$	163.5
Difficulty level	$D = \frac{n_1 \cdot N_2}{2n_2}$	11
Effort to implement	$E = D \cdot V$	1798.5
Time to implement (secs)	$T = \frac{E}{18}$	99.92

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)
- **Number of Nodes** The number of nodes in the abstract syntax tree

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)
- **Number of Nodes** The number of nodes in the abstract syntax tree
- **Control Flow and Data Flow (CFDF)** The number of edges in the flowgraph (CF) plus the number of times that variables are defined and used (DF)

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)
- **Number of Nodes** The number of nodes in the abstract syntax tree
- **Control Flow and Data Flow (CFDF)** The number of edges in the flowgraph (CF) plus the number of times that variables are defined and used (DF)
- **Branch-Loop Complexity (BL)** The number of non-loop predicates plus the number of loops

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)
- **Number of Nodes** The number of nodes in the abstract syntax tree
- **Control Flow and Data Flow (CFDF)** The number of edges in the flowgraph (CF) plus the number of times that variables are defined and used (DF)
- **Branch-Loop Complexity (BL)** The number of non-loop predicates plus the number of loops
- **Recursion and Nesting Complexity (RNC)** The number of instances of recursion and nesting in the program

More Metrics

- **Lines of Code (LOC)** The number of executable lines of code (excluding blank lines and comments)
- **Number of Nodes** The number of nodes in the abstract syntax tree
- **Control Flow and Data Flow (CFDF)** The number of edges in the flowgraph (CF) plus the number of times that variables are defined and used (DF)
- **Branch-Loop Complexity (BL)** The number of non-loop predicates plus the number of loops
- **Recursion and Nesting Complexity (RNC)** The number of instances of recursion and nesting in the program
- **Function Points (FPs) Interface Complexity (FPIC)**

Structural Complexity

The sum of the weights of every construct in the program. The construct is defined subjectively according to experience gained by engineers and managers. For example:

Construct	Weight	Construct	Weight	Construct	Weight
+	1	-	2	*	2
/	3	**	3	=	0
<>	0	<	0	>	0
<=	0	>=	0	Min	1
Max	1	Div	2	Mod	2
If	4	And	1	Or	2
Not	2	Push	10	Abort	10
Array	0	Proc	20	For	10

Metrics for Object Oriented Systems

Object Orientedness Metric 1: Weighted Methods per Class (WMC)

Definition Consider a Class C_1 , with methods M_1,...M_n that are defined in the class. Let c_1,...c_n be the complexity of the methods. Then :

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

If all method complexities are considered to be unity, then WMC $= n$, the number of methods.

Metrics for Object Oriented Systems

Object Orientedness Metric 2: Depth of Inheritance Tree (DIT)

Definition The depth of inheritance of the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

A class with small DIT, has much potential for reuse. (i.e. it tends to be a general abstract class). On the other side, as a class gets deeper into a class hierarchy, it becomes more difficult to maintain.

Metrics for Object Oriented Systems

Object Orientedness Metric 2: Depth of Inheritance Tree (DIT)

Definition The depth of inheritance of the class. In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree.

A class with small DIT, has much potential for reuse. (i.e. it tends to be a general abstract class). On the other side, as a class gets deeper into a class hierarchy, it becomes more difficult to maintain.

Object Orientedness Metric 3: Number of Children (NOC)

Definition The number of immediate sub-classes subordinated to a class in the class hierarchy.

Classes with many children are considered a bad design habit that occurs frequently.

Metrics for Object Oriented Systems

Object Orientedness Metric 4: Coupling Between Object Classes (CBO)

Definition The number of other classes to which this class is coupled. Two classes are coupled when methods in one class use methods or instance variables defined by another class.

A modular and encapsulated design shall yield a low CBO, and this is a desired situation. The more independent the class is, the easier to test and/or reuse it.

Metrics for Object Oriented Systems

Object Orientedness Metric 4: Coupling Between Object Classes (CBO)

Definition The number of other classes to which this class is coupled. Two classes are coupled when methods in one class use methods or instance variables defined by another class.

A modular and encapsulated design shall yield a low CBO, and this is a desired situation. The more independent the class is, the easier to test and/or reuse it.

Object Orientedness Metric 5: Response For a Class (RFC)

Definition The number of methods that can potentially be executed in response to a message received by an object of that class.

Metrics for Object Oriented Systems

Object Orientedness Metric 6: Number of Variables per Class (NVC)

Definition The average number of public variables and private variables per class.

Metrics for Object Oriented Systems

Object Orientedness Metric 6: Number of Variables per Class (NVC)

Definition The average number of public variables and private variables per class.

Object Orientedness Metric 7: Average Parameters per Method (APM)

Definition The number of method parameters divided by the total number of methods.

Lorenz and Kidd argue that APM should not exceed 0.7

Metrics for Object Oriented Systems

Object Orientedness Metric 6: Number of Variables per Class (NVC)

Definition The average number of public variables and private variables per class.

Object Orientedness Metric 7: Average Parameters per Method (APM)

Definition The number of method parameters divided by the total number of methods.

Lorenz and Kidd argue that APM should not exceed 0.7

Object Orientedness Metric 8: Number of Objects (NOO)

Definition The number of objects extracted from source code.

Object Oriented Metrics

MOOD Metrics [Abreu 1994]. All factors range from 0% to 100%:

- Encapsulation
  - MHF: Method Hiding Factor
  - AHF: Attribute Hiding Factor
- Inheritance
  - MIF: Method Inheritance Factor
  - AIF: Attribute Inheritance Factor
- Polymorphism
  - PF: Polymorphism Factor
- Method coupling
  - CF: Coupling Factor

MOOD Metrics: Encapsulation

Method and attribute hiding factor measure how variables and methods are encapsulated in a class. Visibility is iwith respect to other classes. MHF and AHF represent the average amount of hiding among all classes in the system. A private method/attribute is fully hidden.

$$\text{MHF} = \frac{\sum_{i=1}^{M}(1 - V(M_i))}{M}$$

where M is the total number of methods and for each method M_i, $V(M_i)$ is the visibility of this method:

$$V(M_i) = \frac{\#\{\, C_j \mid \text{class } C_j \text{ may call } M_i \text{ and } M_i \text{ is not in } C_j \,\}}{C - 1}$$

where C is the number of classes in the whole system.

MOOD Metrics: Encapsulation

If all methods are private, MHF $= 100\%$. If all methods are public, MHF $= 0\%$.

Method hiding increases reusability and decreases complexity. If there is a need to change the functionality of a particular method, corrective actions will have to be taken in all the objects accessing that method, if the method is not hidden.

A low MHF indicates insufficiently abstracted implementation. A large proportion of methods are unprotected and the probability of errors is high.

A high MHF indicates very little functionality. It may also indicate that the design includes a high proportion of specialized methods that are not available for reuse.

MOOD Metrics: Encapsulation

Research shows that increased MHF decreases bug-density and increases quality.

Increased MHF also decreases defect density and rework effort to find and correct defects.

An acceptable MHF range of 8% to 25% has been suggested.

Similarly, attributes should be hidden by being declared "private". Ideally, all attributes should be hidden, and thus AHF $= 100\%$ is the ideal value. Very low values of AHF should trigger attention.

MOOD Metrics: Inheritance

Method Inheritance Factor:

$$\text{MIF} = \frac{\sum_{i=1}^{C} \text{Mi}(C_i)}{\sum_{i=1}^{C} \text{Ma}(C_i)}$$

where:

- $\text{Mi}(C_i)$ is the number of methods inherited in class C_i, excluding overridden methods
- $\text{Ma}(C_i)$ is the total number of methods available in class C_i (locally defined plus inherited)

Attribute Inheritance Factor, AIF, is defined similarly.

According to one source, the acceptable MIF range is 20% to 80% and the acceptable AIF range is 0% to 48%

MOOD Metrics: Polymorphism

Polymorphism Factor, PF, measures the degree of method overriding in the class inheritance tree

$$\mathrm{PF} = \frac{\sum_{i=1}^{C} \mathrm{Mo}(C_i)}{\sum_{i=1}^{C} (\mathrm{Mn}(C_i) \times \mathrm{DC}(C_i))}$$

where:

- $\mathrm{Mo}(C_i)$ is the number of overriding methods in class C_i
- $\mathrm{Mn}(C_i)$ is the number of new methods in class C_i
- $\mathrm{DC}(C_i)$ is the number of descendants in class C_i

So $\mathrm{Mn}(C_i) \times \mathrm{DC}(C_i))$ is the total number of opportunities for overriding in C_i

PF is an indirect measure of the relative amount of dynamic binding in a system.

MOOD Metrics: Coupling

Class A is *coupled to* class B if A calls methods or accesses variables of B.

Couplings due to inheritance are not included in CF.

$$\mathsf{CF} = \frac{\sum_{i=1}^{C} \sum_{j=1}^{C} \mathsf{is\_client}(C_i, C_j)}{C(C-1)}$$

where:

$$\mathsf{is\_client}(A, B) = \begin{cases} 1 & \text{if } A \neq B \text{ and } A \text{ is coupled to } B \\ 0 & \text{otherwise} \end{cases}$$

Research indicates that increased CF increases defect density and rework effort to find and correct defects.

MOOD reference values

Here are some reference MOOD values for comparison:

System:	MFC	GNU	ET+	Motif
MHF	24.6%	13.3%	9.6%	39.2%
AHF	68.4%	84.1%	69.4%	100.0%
MIF	83.2%	63.1%	83.9%	64.3%
AIF	59.6%	62.6%	51.8%	50.3%
PF	2.7%	3.5%	4.5%	9.8%
CF	9.0%	2.8%	7.7%	7.6%