**SOFTWARE METRICS**

**METRICS**

A metric can be defined as a quantitative measure of the degree to which a system, component or process possesses a given attribute. It provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process.

Therefore, a **software metric** is a standard of measure of a degree to which a [software](https://en.wikipedia.org/wiki/Software) system or process possesses some property.

Since quantitative measurements are essential in all sciences, there is a continuous effort by [computer science](https://en.wikipedia.org/wiki/Computer_science) practitioners and theoreticians to bring similar approaches to software development. The goal is to obtain an objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

Data generated by static code analysis are used in evaluating various aspects of the software or the development process. The aim of establishing reliable quantitative product or process benchmarks is the creation of a model on the basis of which it would be possible to estimate software cost and production deadlines, measure productivity, as well as software quality. Information thus obtained can prove useful further on in the course of software development.

**Classification of Software Metrics**

Software metrics can be classified as follows:

1. **Product Metrics**

Product metrics evaluate the software product in any one stage of its development. They can, for instance, measure the complexity of the design, the size of the source or the object code, or the amount of documentation generated.

1. **Process Metrics**

Process metrics are related to the very process of software production and serve to estimate the time or effort needed to realize the software, average efficiency of human resources, and so on.

The product metrics can also be grouped into two.

* Metrics which determine the size of the program code:

The number of lines of code and the number of statements,

Function Points (FP),

The number of functions or modules in the program,

* Metrics which determine the logical complexity of the program,

The number of binary decisions, Cyclomatic Complexity,

Logical depth of statement,

Halstead complexity (Vocabulary, Length, Volume),

Operator density in the code,

The number of local or global variables used in the code.

Certain Halstead metrics, Effort and Programming Time among them, belong with Process metrics.

**Program Code Size (Metrics based on token count)**

A large number of software metrics attempt to determine the size of the program. Some of the benchmarks become available only after the product has been developed, while others are an attempt to estimate the scope of the program prior to implementation on the basis of other data. The correlation between the size of the program code and its complexity, between the number of errors and maintenance suitability, is often pointed out, but complexity metrics are considered to be superior at quantifying these aspects of software quality.

**Number of Program Lines in the Code**

The number of lines in the code points to the size and textual structure of the source code. There are rules, based on the number of lines in a particular function or module, which determine the limits of code readability. It is often the case that functions or modules with a large number of lines in the code can be decomposed, thus ensuring better clarity of individual code segments. Also, a well-commented code is, as a rule, easier to understand and less time is required for its maintenance.

The number of lines in the code is counted separately for each function or C module:

* Lines - line number total for any type of line in the code. For functions, lines beginning with the opening "{" parenthesis and ending with the closing "}" parenthesis, including the lines in which the parentheses are positioned, are totaled. Functions defined in a single line are considered to contain a single line of code. For modules, lines beginning with the first line in the file and ending with the last line in the file (the line containing the EOF character) are totaled.
* White lines - percentage of all lines containing only white-space characters " ", "\t", "\v", "\f", "\r", "\b", or "\a"
* Comment lines - percentage of all lines containing only comments and potential white-space characters
* Executable lines - percentage of all lines containing an executable statement or a part of one. Declarations with variable initialization also go towards this total.
* Lines with comments - percentage of all lines containing a comment or part of one, as well as at least one non-white-space character
* Preprocessor lines - percentage of all lines containing preprocessor directives

Some of the named properties are expressed as their absolute values and some in percentages. Properties expressed in percentages are relatively totaled in relation to the total number of lines of code (parameter Lines).

**Number of Statements**

The number of statements is totaled separately for each function or C module:

* **Statements (total)** - total number of statements of any type. The number is totaled as the sum of executable, declaration, label, and compound statements.
* **Compound statements** - percentage of all compound {...} statements. The function body, as it also is contained within the {...} parentheses, goes towards this total. Components belonging to struct, union, or enum statements, which are also in {...} parentheses, do not count.
* **Declaration statements** - percentage of all declarations ending with a semicolon ( ; ).  
  For example, in the following fragment of code, the total number of statements is five:

struct

{

   int a;

   char b;

   struct foo

   {

      int c;

   } d;

} e;

* **Empty statements** - percentage of all statements which are either empty ";" or are made up of inline assembler statements
* **Executable statements** - percentage of all statements which represent one of the following commands: if, while, do-while, for, break, continue, return, switch, or an expression
* **Expression statements** - percentage of all statements which are expressions (expressions which are a part of other statements, for example, conditions, do not go towards this total)
* **Jump statements** - percentage of all statements which are goto, break, or continue
* **Label statements** - percentage of all statements which are label definitions
* **Loop statements** - percentage of all statements which are while, do-while, for
* **Selection statements** - percentage of all statements which are if or case  
  (case 'A': case 'B': counts as one)

Properties expressed in percentages are relatively totaled in relation to the total number of statements (parameter Statements).

**Number of Function Points**

Function Points are a subjective evaluation of a project. The method used to evaluate Function Points is known in the literature as backfiring, and was borrowed from the book Applied Software Measurement, by Caspers Jones[1](http://www.ristancase.com/html/dac/manual/2.12.01-Software-Metrics-Classification.html#wp69084).

Function Points represent an approximate number of functionalities which exist in a project. Function Points are obtained when the number of statements in the entire program is divided by the average number of statements necessary for a functionality to be realized in a programming language. For C programming language, a value of 128 is adopted. Therefore, Function Points are obtained by dividing the total number of statements (executable statements, declarations, and preprocessor directives) by this number. Function Points thus obtained are termed 'unadjusted'. Basically, there are two types of Function Points, Adjusted and Unadjusted Function Points.

Adjusting is introduced because it was noted that a smaller number of statements is required to write a functionality when the project is simple, than when it is complex.

There are two possible ways to adjust Function Points. The first would be to ask the user a number of questions on ranking the complexity of the project (whose Function Points are being measured), and then to adjust Function Points according to the answers thus obtained; the second is based on Cyclomatic Complexity. In order to simplify using this measure, the second approach was adopted.

Based on the value of the Cyclomatic Complexity of the project, the adjusting is carried out by obtaining the Adjusting factor according to following table.

Table 5: Adjusting factors for adjusting Function Points

|  |  |  |  |
| --- | --- | --- | --- |
| Cyclomatic complexity | Adjusting factors | Cyclomatic complexity | Adjusting factors |
| 1 | 0.70 | 12-16 | 1.05 |
| 2 | 0.80 | 17-21 | 1.10 |
| 3 | 0.85 | 22-27 | 1.15 |
| 4-5 | 0.90 | 28-36 | 1.20 |
| 6-8 | 0.95 | 37-53 | 1.25 |
| 9-11 | 1.00 | 54- | 1.30 |

The Adjusted Function Points are now totaled by dividing the original Function Points by the Adjusting factor.

[1](http://www.ristancase.com/html/dac/manual/2.12.01-Software-Metrics-Classification.html#wp69057)Jones, Caspers: Applied Software Measurement, 2nd edition; McGraw Hill, 1996

**Number of Functions and Modules in the Program**

The number of functions and modules can provide insight into program size and suggest potential physical reorganization of the code. The following metrics are measured:

**1. Functions** - The number of different functions defined in the module or group of modules.

**2. Number of modules** - The number of different C modules in the group of modules.

**Logical Complexity of the Program**

Metrics which measure the logical complexity of the program show how involved the program is in terms of handling and maintenance. These metrics can often provide a precise estimate of the scope of testing to be undertaken and they are correlated with the number of errors which appear in the program.

**McCabe's Cyclomatic Complexity**

The Cyclomatic Complexity parameter is used to locate areas of key importance to testing, to plan the testing process, and to delimit program complexity during the development phase.

**Number of Binary Decisions in the Program**

When dealing with the Cyclomatic Complexity metrics parameter, it is necessary to define the concept of the number of binary decisions in the program. The number of binary decisions is viewed at function level.

The following instances count as single binary decisions:

1. "?": operators,

2. if statements, not taking into account occurrences of else,

3. while statements,

4. do-while statements,

5. for statements (if there is a condition).

For example:

/\*this statement counts\*/

for ( ;i<12; ){/\* loop body\*/}

/\*this for statement does not count\*/

for ( i=0; ; i++ ){/\* loop body\*/}

6. case statements, while connected case statements count as a single decision, whereas default does not count. For example:

switch ( c )  
{  
   case 1:  
   case 2:  
      a += b;  
      break;

   case 3:

      b += a;

      break;  
   default:  
      a -= b;  
      break;  
}

This example contains two binary decisions.

7. Every occurrence of the logical operators OR, AND, because every use of these logical operators is actually equivalent to an if statement stipulating a simple condition.

For example: The following function has two binary decisions, one stemming from the if statement, and the other from the "&&" operator:

Example (a, b)

{

   if ( a < 3 && b>5 ) {/\* ... \*/}

}

There are certain situations in C language in which calculating binary decisions can be problematic. DAC interprets such constructions in a way which least influences the programmer.

- Conditional expression (?: operator) counts as a binary decision because the programmer would otherwise be led to use insufficiently clear conditional expressions instead of if commands in order to achieve a lesser degree of Cyclomatic Complexity. Adjacent case commands are analogous to the logical operator or (||). If DAC were to calculate each case as a distinct binary decision, the programmer could avoid the switch command and use the if command, simulating it with a string of or conditions.

**Cyclomatic Complexity**

The Cyclomatic Complexity of a program module (function) is the greatest number of linearly independent paths through the specified module (function). It measures the number of tests necessary for reasonable protection from errors in the code.

McCabe defined Cyclomatic Complexity in the following form:

v(G) = + e – n + (2 \* p);

Where:

e - is the number of branches of the control flow graph

n - is the number of nodes of the control flow graph

p - is the number of connected components of the graph ( in our case, it is p = 1 )

There is an alternative way to calculate v(G), which is based on the analysis of the source code. It consists of finding the number of binary decisions to which 1 is added:

Empirical studies have shown that programs with a degree of Cyclomatic Complexity less than 5 are generally considered simple and easy to understand. A degree of Cyclomatic Complexity less than 10 is not considered overly difficult to understand. If the degree of Cyclomatic Complexity exceeds 20, program complexity is considered great. When it exceeds 50, the software becomes practically impossible to test.

There is a direct connection between Cyclomatic Complexity and program maintainability. Related to this is the term "bad fix" - an error made inadvertently in the course of trying to fix a previous error. It turns out that with programs whose degree of Cyclomatic Complexity does not exceed 10, there is a 5% probability of bad fixes; with programs whose degree of Cyclomatic Complexity is somewhere between 20 and 30, this probability is significantly higher - around 20%, while with programs whose degree of Cyclomatic Complexity exceeds 50, the probability of a "bad fix" error is as high as 40%.

**McCabe's Average Cyclomatic Complexity**

The Average Cyclomatic complexity parameter is got as an average value of the McCabe`s Cyclomatic Complexity parameter for the functions within the scope of observation. For the scope of module observation, it is calculated for the functions in the module, and when the scope is a group of modules, it is calculated for all functions in the chosen group of modules.

The parameter offers an insight into the complexity of the code, and it is useful when it comes to planning the process of testing.

**Graphical computation of Cyclomatic complexity**

Result

D2

D1

1. M = E – N + D

M = 8 – 7 + 2

M = 3

1. M = R + 1

M = 3

M -> Cyclomatic complexity

N—nodes

D—decision nodes

R—regions

**Logical Depth of Statement**

It is necessary to determine the maximal logical depth of statement. The logical depth of statement is measured by measuring the logical depth of if, else, for, while, switch, and do-while statements. For example:

// depth 0

if ( a > b ) // depth 0

   while ( d < a ) // depth 1

   { // depth 2

      a--; // depth 2

      for ( i = 0; i < 5; i++ ) // depth 2

      { // depth 3

      } // depth 3

   } // depth 2

The maximal logical depth of statement in this example is 3. Programs with high logical depth of statement can be difficult to handle and maintain.

**Number of Variables Used in the Program**

The number of variables used in a module or function can be an indication of program complexity and maintenance suitability. In software development, minimum use of global variables in the program is often insisted on. The following metrics are measured:

1. Global/Static variables - the number of global variables used in the module or group of modules

2. Local variables - the number of different local variables used in a function. Formal parameters and declarations within the function do not count, only occurrences of variables count

Only variables used in the executable commands of the function count - formal parameters which are not used, as well as declared but not used local variables, do not count. For example:

char Foo( int formPar1, char\* formPar2 )

{

   int localVar1, localVar2 = 5;

   char localVar3;

   localVar3 = 'a';

   if ( formPar1 )

      localVar3 = 'b';

   return localVar3;

}

Only two local variables, formPar1 and localVar3, will count here, because they appear in executable commands. Other variables will be ignored.

**Density of Program Code (Operator Density)**

On the basis of the number of C operators used in the program, it is possible to ascertain code readability. C language, due to the terseness of its syntax, often makes it possible to write cryptic code which is very difficult to maintain, so it is useful to see which parts of the code have high operator density. A large number of operators in a line or statement lessens code readability, and therefore increases software maintenance costs.

**Operator Weight**

As all operators are not of equal readability, we may conclude that seldom-used operators should have greater weight. Default weights have been entered in the following table, and can be configured at need. When calculating operator density, their respective weights are taken into account, as they reflect operator density better than just counting the operators would.

|  |  |  |
| --- | --- | --- |
| Operator | Weight | Comment |
| + - \* / ! | 1 |  |
| [] . | 1 |  |
| < <= > >= | 1 |  |
| != |= | 2 | could be mixed up |
| += -= \*= /= | 2 |  |
| ++ -- | 2 |  |
| && || | 2 |  |
| sizeof | 2 |  |
| (type name) | 2 |  |
| = == | 3 | could be mixed up in conditional statements |
| & | ~ << | 3 |  |
| \* & -> | 3 |  |
| ^ ?: ^= | 3 |  |
| >> >>= % %= | 5 | rarely used |
| , | 5 | rarely used |

**Metrics Based on Counting C Operators**

The following properties are used to describe the number of operators in the program:

1. Average operator weight per expression statement.

This is the weight of all operators in the code divided by the total number of expression statements.

2. Number of operators per token.

This is the total number of operators divided by the number of standard C lexical tokens found in the code.

3. Number of operators per operand.

This is the total number of operators divided by the total number of operands found in the code. Identifiers, numeric constants, strings, and typedef type names are considered operands.

4. Average operator weight per operand.

This is the weight of all the operators in the code divided by the total number of operands.

5. Average number of operators per line.

This is the total number of operators divided by the total number of lines in the code.

6. Total weight of all operators (sum of weights).

This is the sum of weights of all the operators found in the code.

**Halstead's Software Science (Error and Effort in Program Maintenance)**

The software science developed by M. H. Halstead principally attempts to estimate the rate of program errors and the effort invested in program maintenance. Halstead Metrics are used in project scheduling and reporting, in that they measure the overall quality of the program and rate the effort invested in its development. They are easy to calculate and do not require in-depth analysis of program structure.

Halstead Metrics are based on the measurement and interpretation of tokens. A token is the smallest unit of text recognized by the compiler. The metrics analyzer considers the following tokens as operators of Halstead Metrics:

**Keywords**

break case continue default do else for goto if return sizeof switch while

**Operators**

( [ . -> ++ -- , sizeof & \* + - ~ ! / % << >> < > <= >= == != ^ | && || ? : = \*= /= %= += -= <<= >>= &&= ^= |= ; { ...

Basically, all operators except "}", ")", and "]" since paired tokens such as "(" and ")" are considered a single operator).

The following tokens are considered Halstead Operands:

1. Identifiers,

2. Typedef name types,

3. Numerical constants,

4. Strings.

A label and its terminating colon do not count, as they are comments according to Halstead. In addition, function headings, including the initializations included in them, do not count.

Solely executable commands, that is, the very program logic, and not declarations, are taken into account when calculating Halstead Metrics. The exceptions to this rule are the initializers within compound commands. They are considered a logic which appears in the declaration coincidentally - and are equivalent to assigning statements. Therefore, the name of the object, the operator of assigning ( = ), and the initialization expression go towards the total number of Halstead Metrics. Operators which appear as command terminators (comma or semicolon) are not taken into account. It is assumed that these lexemes belong to the declaration, and not to the logic. For example, the two following functions are equivalent:

foo()

{

   int a = 3;

}

foo()

{

   int a;

   a = 3;

}

Yet, Halstead Metrics are slightly different for these two functions, as a result of the second function having an additional semicolon operator.

Note that a single token can be considered to belong to more than one category. For example, a procedure or function call will involve one operator count (the function call), and one operand count (the name of the function being called).

All tokens which do not belong to the function body, that is, not enclosed in the "{ }" parentheses which make up the function body, are not counted. Specifically, tokens in the declaration part of the program are not counted.

All Halstead metrics are derived from basic properties. Emphasis is placed here on the difference between the total number of operators or operands (every occurrence counts), and the number of unique operators or operands (which designate the number of different occurrences, for example, if there are four occurrences of the operator "=" in a function, they are counted as a single unique occurrence).

The basic properties are:

1. Unique operators (n1) the number of unique occurrences of Halstead Operators in the program,

2. Unique operands (n2) the number of unique occurrences of Halstead Operands in the program,

3. Total operators (N1) the total number of Halstead Operators,

4. Total operands (N2) the total number of Halstead Operands.

The derived properties of Halstead Metrics are of great importance to the interpretation of code complexity. Basic properties are used to calculate them.

**Halstead Program Length**

The total number of operator occurrences and the total number of operand occurrences.

N = N1 + N2

**Halstead Vocabulary**

The total number of unique operator and unique operand occurrences.

n = n1 + n2

**Program Volume**

Proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The properties V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.

V = N \* log2(n)

**Program Difficulty**

This parameter shows how difficult to handle the program is.

D = (n1 / 2) \* (N2 / n2)

**Programming Effort**

Measures the amount of mental activity needed to translate the existing algorithm into implementation in the specified program language.

E = V \* D

**Language Level**

Shows the algorithm implementation program language level. The same algorithm demands additional effort if it is written in a low level program language. For example, it is easier to program in Pascal than in Assembler.

L' = V / D / D

**Intelligence Content**

Determines the amount of intelligence presented (stated) in the program This parameter provides a measurement of program complexity, independently of the program language in which it was implemented.

I = V / D

**Programming Time**

Shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language.

T = E / (f \* S)

The concept of the processing rate of the human brain, developed by the psychologist John Stroud, is also used. Stroud defined a moment as the time required by the human brain requires to carry out the most elementary decision. The Stroud number S is therefore Stroud’s moments per second with:  
5 <= S <= 20. Halstead uses 18.

Stroud number S = 18 moments / second

seconds-to-minutes factor f = 60