Software Complexity Metrics*

Many measures of software complexity have been proposed. Many of these, although yielding a good representation of complexity, do not lend themselves to easy measurement. Some of the more commonly used metrics are

- McCabe's cyclomatic complexity metric
- Halsteads software science metrics
- Henry and Kafura introduced Software Structure Metrics Based on Information Flow in 1981[2] which measures complexity as a function of fan in and fan out. They define fan-in of a procedure as the number of local flows into that procedure plus the number of data structures from which that procedure retrieves information. Fan-out is defined as the number of local flows out of that procedure plus the number of data structures that the procedure updates. Local flows relate to data passed to and from procedures that call or are called by, the procedure in question. Henry and Kafura's complexity value is defined as "the procedure length multiplied by the square of fan-in multiplied by fan-out" (Length ×(fan-in × fan-out)^2).

- A Metrics Suite for Object Oriented Design[3] was introduced by Chidamber and Kemerer in 1994 focusing, as the title suggests, on metrics specifically for object oriented code. They introduce six OO complexity metrics; weighted methods per class, coupling between object classes, response for a class, number of children, depth of inheritance tree and lack of cohesion of methods

Associated with, and dependent on the complexity of an existing program, is the complexity associated with changing the program. The complexity of a problem can be divided into two parts:[4]

1. Accidental complexity: Relates to difficulties a programmer faces due to the chosen software engineering tools. A better fitting set of tools or a more high-level programming language may reduce it.
2. Essential complexity: Is caused by the characteristics of the problem to be solved and cannot be reduced.

- McCabe’s measure (Cyclomatic complexity)
  - measures the number of linearly independent paths through a program's source code
- Halstead’s measures (computed statically)
  - Program length: N = N1 + N2

- Program vocabulary: n = n1 + n2
- Volume:
- Difficulty : D = (n1/2) × (N2/n2)
- Effort: E = D × V
-

Halstead complexity measures are software metrics introduced by Maurice Howard Halstead in 1977[1] as part of his treatise on establishing an empirical science of software development. Halstead made the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code.

Halstead's goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (analogous to the gas equation). Thus his metrics are actually not just complexity metrics.

Calculation[edit]

For a given problem, Let:

- η_1 = the number of distinct operators
- η_2 = the number of distinct operands
- N_1 = the total number of operators
- N_2 = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary: $\eta = \eta_1 + \eta_2$
- Program length: $N = N_1 + N_2$
- Calculated program length: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume: $V = N \times \log_2 \eta$
- Difficulty : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort: $E = D \times V$

The difficulty measure is related to the difficulty of the program to write or understand, e.g. when doing code review.

The effort measure translates into actual coding time using the following relation,

- Time required to program: $T = \frac{E}{18}$ seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs : $B = \frac{E^{\frac{2}{3}}}{3000}$ or, more recently, $B = \frac{V}{3000}$ is accepted[citation needed].

Example[edit]

Let us consider the following C program:

```
main()
{
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

The unique operators are: `main`, `()`, `{}`, `int`, `scanf`, `&`, `=`, `+`, `/`, `printf`

The unique operands are: `a`, `b`, `c`, `avg`, `"%d %d %d"`, `3`, `"avg = %d"`

- $\eta_1 = 10, \eta_2 = 7, \eta = 17$
- $N_1 = 16, N_2 = 15, N = 31$
- Calculated Program Length: $\hat{N} = 10 \times log_2 10 + 7 \times log_2 7 = 52.9$
- Volume: $V = 31 \times log_2 17 = 126.7$
- Difficulty: $D = \frac{10}{2} \times \frac{15}{7} = 10.7$
- Effort: $E = 10.7 \times 126.7 = 1,355.7$
- Time required to program: $T = \frac{1,355.7}{18} = 75.3$ seconds
- Number of delivered bugs: $B = \frac{1,355.7^{\frac{2}{3}}}{3000} = 0.04$

Cyclomatic complexity is a software metric (measurement). It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program. It is a quantitative measure of the complexity of programming instructions. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

One testing strategy, called basis path testing by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.

Correlation to number of defects

A number of studies have investigated cyclomatic complexity's correlation to the number of defects contained in a function or method. Some studies find a positive correlation between cyclomatic complexity and defects: functions and methods that have the highest complexity tend to also contain the most defects, however the correlation between cyclomatic complexity and program size has been demonstrated many times and since program size is not a controllable feature of commercial software the usefulness of McCabes's number has been called to question. The essence of this observation is that larger programs (more complex programs as defined by McCabe's metric) tend to have more defects. Although this relation is probably true, it isn't commercially useful . As a result the metric has not been accepted by commercial software development organizations.

Studies that controlled for program size (i.e., comparing modules that have different complexities but similar size, typically measured in lines of code) are generally less conclusive, with many finding no significant correlation, while others do find correlation. Some researchers who have studied the area question the validity of the methods used by the studies finding no correlation

(nist) There are many good reasons to limit cyclomatic complexity. Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify. Deliberately limiting complexity at all stages of software development, for example as a departmental standard, helps avoid the pitfalls associated with high complexity software. Many organizations have successfully implemented complexity limits as part of their software programs. The precise number to use as a limit, however, remains somewhat controversial. The original limit of 10 as proposed by McCabe has significant supporting evidence, but limits

as high as 15 have been used successfully as well. Limits over 10 should be reserved for projects that have several operational advantages over typical projects, for example experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan. In other words, an organization can pick a complexity
limit greater than 10, but only if it is sure it knows what it is doing and is willing to devote the additional testing effort required by more complex modules.

Somewhat more interesting than the exact complexity limit are the exceptions to that limit. For example, McCabe originally recommended exempting modules consisting of single multiway
decision (“switch” or “case”) statements from the complexity limit. The multiway decision issue has been interpreted in many ways over the years, sometimes with disastrous results. Some naive developers wondered whether, since multiway decisions qualify for exemption from the complexity limit, the complexity measure should just be altered to ignore them. The result would be that modules containing several multiway decisions would not be identified as overly complex. One developer started reporting a “modified” complexity in which cyclomatic complexity was divided by the number of multiway decision branches. The stated intent of this metric was that multiway decisions would be treated uniformly by having them contribute the average value of each case branch. The actual result was that the developer could take a module with complexity 90 and reduce it to “modified” complexity 10 simply by adding a ten-branch multiway decision statement to it that did nothing.