<ch6 toc ch8>

# Chapter 7
# METRICS AND COMPLEXITY

## 1. SYNOPSIS

Measuring software complexity. Halstead’s metrics, token counts, McCabe’s metric, hybrid metrics. Application and implementation.

## 2. METRICS, WHAT AND WHY

### 2.1. Philosophy

One of the characteristics of a maturing discipline is the replacement of art by science. Science is epitomized by quantitative methods. In classical Greece, for example, early physics was dominated by discussions of the “essence” of physical objects, with almost no attempts to quantify such “essences.” They were struggling with what questions should be asked. Quantification was impossible until the right questions were asked. By the sixteenth century, physics began to be quantified—first by Galileo and then by others. In the eighteenth century, we called physical scientists “natural philosophers” recognizing that the right questions hadn’t been asked, and that all was not yet, even in principle, amenable to quantification. Today, physics is as quantified as science can get and far removed from its speculative roots in philosophy.*

*But modern particle physics borders on mysticism and seems poised for a return to speculative philosophy.

Computer science (I think that the term is presumptuous—we’re still more of a craft than a science) is following the same quantification path. Ultimately, we hope to get precise estimates of labor, resources, and reliability from formal specifications by mechanical means—ultimately, that is. If there’s a note of scepticism it’s because so much of what we want to quantify is tied to erratic human behavior. Should we expect greater success over a shorter period of time than economists and sociologists have had? Our objective then is not necessarily the ultimate truth, the “real” model of how programs and systems behave, but a partial, practical, truth that is adequate for the practitioner. Does it matter whether our models of software complexity are right if, right or wrong in some abstract sense, they correlate well with reality? We have programs to write and systems to debug. We can leave the search for truth to the theorists and be content with pragmatism. What works is what counts. For metrics in general, see CURR86, DAV188C, GILB77, GRAD87, HARR82, LEV186, LIHF87 (especially), LIND89, PERL81, POLL87A, RAMA85 and WEYU88B.

### 2.2. Historical Perspective

There’s no record of programming labor estimates on ENIAC, but I’m sure they fell far short of reality. The first programmer, Lady Lovelace, who coded for Charles Babbage’s wonderful but unfinished computer in the nineteenth century, I’m sure often said “Just one more week, Mr. Babbage, and it’ll be done.” Lucky for her the hardware was never finished, so she didn’t have to go beyond desk checking. By the mid-1950s individual programmers had coalesced into programming groups, which meant there was now a new profession—programmer manager. The managers, who were responsible for productivity and quality, began to measure software effort in terms of "number of lines of code." That was used to predict programming costs, testing costs, running time, number of bugs, salaries, the gross national product, the inflation rate of the peso, and who knows what else. It is today still the primary quantitative measure in use: "Count the number of lines of code, and you know all there is to know."

A landmark study by Weinwurm, Zagorski, and Nelson debunked any such simplistic measure (NELS67, WEIN65). They showed that cost was heavily influenced by such things as the number of meetings attended, the number of items in the database, the relative state of hardware development (if new hardware), documentation requirements, and other factors that didn't relate directly to simple measures such as lines of code. The study also debunked popular programmer's myths, such as "real-time software is much more difficult than batch processing." The study is important because it's one of the earliest serious large-scale efforts to quantify the issues in programming. Furthermore, the base for this study was not a bunch of small homework exercises, but a big group of system programs, spanning the range from subroutines to systems. As a basis for cost estimating, these studies have yet to be surpassed by anything that's been published. Despite the fact that the study thoroughly debunked the myths, despite the fact that contemporary estimation programs such as COCOMO (BOEH81) use many factors to predict cost, resources, and schedules—despite these facts, the lines-of-code metric stubbornly remains the most popular (and inaccurate) one in use, much as it was (ab)used a quarter of a century ago.

## 2.3. Objectives

### 2.3.1. How Big Is It?

Science begins with quantification: you can't do physics without a notion of length and time; you can't do thermodynamics until you measure temperature. The most fundamental question you can ask is "How big is it?" Without defining what "big" means, it's obvious that it makes no sense to say "This program will need more testing than that program" unless we know how big they are relative to one another. Comparing two strategies also needs a notion of size. The number of tests required by a strategy should be normalized to size. For example, "Strategy A needs 1.4 tests per unit of size, while strategy B needs 4.3 tests per unit of size."

What is meant by "size" is not obvious in the early phases of science development. Newton's use of mass instead of weight was a breakthrough for physics, and early researchers in thermodynamics had heat, temperature, and entropy hopelessly confused. We seem to be doing about as well (or as badly) as the sixteenth- and seventeenth-century physicists did at a comparable phase of development. "Size" isn't obvious for software. Levitin, in his critique of metrics (LEVI86) discusses fundamental problems in our most popular metric, the seemingly obvious "lines of code." You can't measure "size" if it's based on a subjective ruler—try doing physics or engineering with rubber rulers and spasmodic clocks. Metrics must be objective in the sense that the measurement process is algorithmic and will yield the same result no matter who applies it.

### 2.3.2. The Questions

Our objective in this book is not to quantify all of computer science, but only to explore those **metrics** of complexity that have proved their worth in practice. To see what kinds of metrics we need, let's ask some questions:

1. When can we stop testing?
2. How many bugs can we expect?
3. Which test technique is more effective?

4. Are we testing hard or are we testing smart?
5. Do we have a strong program or a weak test suite?

I won't answer any of these questions in this book because we don't know enough about software and testing to provide answers—at least not with the strength of a physical law. What we can do is take another lesson from physics: measurement leads to empirical "laws," which in turn lead to physical laws. Thus, Kepler's precise measurements* of planetary motion provided the foundations on which Newton could build physics; Mendel's experiments* provided the foundations on which the laws of genetics are based. Quantitative, empirical laws were used by engineers and artisans long before there was a satisfactory theory to explain such laws. Roman aqueducts, for example, can be shown to be structurally and hydraulically almost optimum, based on theory that's less than 100 years old.

*But recent evidence shows that Kepler cooked the data to fit his theory. Ditto for Mendel.

If you want answers to the above questions, then you'll have to do your own measuring and fit your own empirical laws to the measured data. How that's done is discussed in Section 6, below. For now, keep in mind that all the metrics discussed below are aimed at getting empirical laws that relate program size (however it be measured) to expected number of bugs, expected number of tests required to find bugs, test technique effectiveness, etc.

### 2.3.3. Desirable Metric Properties

A useful metric should satisfy several requirements:**

**A formal examination of metrics requirements by Weyuker (WEYU88B) formally extends these concepts and evaluates the extent to which several popular metrics, including statement count, cyclomatic number, Halstead's effort metric, and dataflow complexity do or do not meet them.

1. It can be calculated, uniquely, for all programs to which we apply it.
2. It need not be calculated for programs that change size dynamically or programs that in principle cannot be debugged.
3. Adding something to a program (e.g., instructions, storage, processing time) can never decrease the measured complexity.

The first requirement ensures a usable, objective, measure; the second says that we won't try to apply it to unreasonable programs; and the third is formalized common sense. It's another way of saying that the program is at least as complicated as any of its parts.

### 2.3.4. Metrics Taxonomy

There's no agreement in the literature on how to classify metrics—and there are so many metrics to classify. Here are some broad categories: **linguistic** metrics, **structural** metrics, and **hybrid** metrics. Each can be applied to either programs or specifications; to date, however, application to programs has dominated. The taxonomy is not rigid because a narrowly defined linguistic metric can define a structural property. For example, cyclomatic complexity can be defined in terms of links and nodes in the control flowgraph or alternatively by the number of equivalent branches in the program.

**Linguistic Metrics**—Metrics based on measuring properties of program or specification text without interpreting what that text means or the ordering of components of the text. For example: lines of code, number of statements, number of unique operators, number of unique operands, total number of operators, total number of operands, total number of keyword appearances, total number of tokens.
**Structural Metrics**—Metrics based on structural relations between objects in the program—usually metrics on properties of control flowgraphs or data flowgraphs; for example, number of links, number of nodes, nesting depth.
**Hybrid Metrics**—Metrics based on some combination of structural and linguistic properties of a program or based on a function of both structural and linguistic properties.

## 3. LINGUISTIC METRICS

### 3.1. General

**Linguistic metrics** measure some property of text without interpreting what is measured. A metric is (mainly) linguistic if its value doesn't change when you rearrange the text. Linguistic metrics, to date, have been applied mostly to program text. They can be just as easily applied to formal specifications, but because formal, processable specifications are rare, there is almost no experience with such usage; but see RAMA85.

### 3.2. Lines of Code, Statement Count, and Related Metrics

#### 3.2.1. What Is It?

Count the number of lines of code in a program and use that number as a measure of complexity. If then, bugs appear to occur at 1% per line, a 1000-line program should have 10 bugs and a 10,000 line program should have 100. Going further, if we find that it takes an average of twenty tests to find a bug, we might infer (empirically) the expected number of tests needed per line of code.

I have an easier metric: weigh the listing—or if you don't have a scale, measure its thickness with a ruler. In today's electronic programming world, you could just count the number of k of storage used to store the program text. Why not use listing weight? Why is it that when I propose measuring programs by listing weight people think I'm pulling their leg but they take me seriously and consider it "scientific" when I suggest lines of code? Is listing weight really less scientific? Think about it. There's a high correlation between the weight of the listing or its thickness and the number of statements in it. On a big project, it probably correlates to better than 95%. Within a project with all paper supplied by the same vendor and all listings done under the same printing utility, the formats are regular. I think that if you were to gather statistics over one big project or many small projects (that used the same paper), then the weight of the listings would correlate as well to the bug rate and test efforts as do lines of code. Yet, "lines of code" sounds reasonable and scientific, and "listing weight" seems to be an outrageous puton. Who's putting whom on? The fact is that it makes *exactly* as much sense (or nonsense) to say "This is a 230-gram* program" as it does to say "This is a 500-line program."

*Note my use of "gram" rather than carat, drachm, gera, mina, nsp, ounce, pennyweight, pound, shekel, or talent in order to make my weight metric more "scientific."

#### 3.2.2. What to Count and Not Count

Early users of lines of code did not include data declarations, comments, or any other lines that did not result in object code. Later users decided to include declarations and other unexecutable statements but still excluded comments and blank lines. The reason for this shift is the recognition that contemporary code can have 50% or more data statements and that bugs occur as often in such statements as in “real” code. There is a rationale for including comments. The quality of comments materially affects maintenance costs because the maintenance programmer will depend on the comments more than anything else to do her job. Conversely, too many blank lines and wordy but information-poor comments will increase maintenance effort. The problem with including comments is that we must be able to distinguish between useful and useless comments, and there’s no rigorous way to do that. The same can be said of blank lines and formatting text: a little helps us read the code but too much forces us into excessive page turning.

The lines-of-code metric has obvious difficulties. The count depends on the printing format. For example, it depends on the print-line length. Are we using line lengths of 50, 60, 72, 80, or 128 characters? Programming standards and individual style change lines of code, especially for data declarations. As an example, my data declarations in BASIC usually take more lines of code than needed so that they can look neat, can be ordered by object type (string, real, integer), by semantic type (counter, pointer), and so on. By making the program more legible, I’ve increased the line count, thereby making my version of the program seem more complex when in fact I’ve made it easier to understand. Similarly, for languages that permit multiple statements on each line: putting an entire loop on a line reduces apparent complexity, but overcrowding a line by arbitrarily packing statements makes the program text more obscure. The final weakness of this metric is that if we apply a “pretty printer” format program to source code, inevitably increasing the line count, we get an apparent complexity increase despite the fact that intelligibility has been improved. A rubbery ruler, lines of code.

Lines of code makes more sense for source languages in which there’s a high correlation between statement count and lines of code. For example: assembly languages, old Basic, and FORTRAN.

### 3.2.3. Statement Counts

Some of the difficulties with lines of code can be overcome by using statements instead—but this evokes new problems that are as bad as those of lines of code. The problem, aptly stated by Levitin (LEVI86) is that there’s no unique way to count statements in some languages (e.g., Pascal), and there’s no simple rule for defining “statement” across different languages. Just as subjectivity is involved in applying lines of code, there’s subjectivity in deciding what is and what is not to be called a statement (for some languages).

### 3.2.4. How Good (Bad) Are They?

Thayer, Lipow and Nelson, in their monumental software reliability study (THAY76) showed error rates ranging from 0.04% to 7% when measured against statement counts, with the most reliable routine being one of the largest. The same lack of useful correlation is shown in Rubey (RUBE75). Curtis, Sheppard, and Milliman (CURT79A) show that lines of code is as good as other metrics for small programs, but is optimistic for big programs. Moderate performance is also reported for lines of code by Schneidewind (SCHN79A). The definitive report on the relation between program size and bug rate is Lipow’s (LIPO82). In his study of 115,000 JOVIAL statements, he found a nonlinear relation between bugs per line of code and statements; but also included other factors related to language type and modularization. Small programs had an error rate of 1.3% to 1.8%, with big programs increasing from 2.7% to 3.2%. Lipow’s study, however, and most of the others, only included executable lines and not data declarations. The bottom line is that lines of code is reasonably linear for small programs (under 100 lines) but increases nonlinearly with program size. It seems to correlate with maintenance costs.

Statement count shows the same correlation. These metrics are certainly better than guesses or nothing at all.

## 3.3. Halstead's Metrics

### 3.3.1. What Are They?

Halstead's metrics are based on a combination of arguments derived from common sense, information theory, and psychology. The clearest exposition is still to be found in Halstead's *Elements of Software Science* (HALS77). It's an easy-to-read little book that should be read before applying these metrics. The following exposition is intended only as an introductory overview. The set of metrics are based on two, easily measured, parameters of programs:

$n_1$ = the number of distinct operators in the program (e.g., keywords)

$n_2$ = the number of distinct operands in the program (e.g., data objects)

From these he defines **program length**, which is not to be confused with the number of statements in a program, by the following relation:

$$H = n_1 \log_2 n_1 + n_2 \log_2 n_2 \qquad (1)^*$$

*The late Maurice Halstead used *N* for this metric, as has the rest of the literature. I have several reasons for departing from that practice. The use of *H* for "Halstead length" is a fitting tribute to Halstead's contribution to computer science. The term usually used for this metric is "program length," which is easily confused with statement count or lines-of-code. "Halstead's metric" or "Halstead length" cannot be so confused. Finally, the form of the definition is reminiscent of information theory's "information content," to which it is related. In information theory, the information content is denoted by *h* and is calculated by a similar expression.

In calculating the Halstead length, paired operators such as "BEGIN...END," "DO...UNTIL," "FOR...NEXT," "(...)" are usually treated as a single operator, which is what they actually are. For any given program it's possible to count the actual operator and operand appearances.

$N_1$ = program operator count
$N_2$ = program operand count

The actual Halstead length is evidently

$$N = N_1 + N_2 \qquad (2)$$

The actual length is effectively a static count of the number of tokens in the program (see Section 3.4 below).

Halstead also defines a program's **vocabulary** as the sum of the number of distinct operators and operands. That is:

$$n = \text{vocabulary} = n_1 + n_2 \qquad (3)$$

The central claim to Halstead's conjecture (equation 1) is that the program's actual Halstead length ($N$) can be calculated from the program's vocabulary even though the program hasn't been written. It does not tell us how long the program will be in terms of statements, but it is an amazing claim nevertheless. Its immediate importance is that it's often possible to get an operand count and to estimate operator counts before a program is written. This is especially true when the program is written after a data dictionary already exists. If a program is written using 20 keywords out of a total of 200 in the language, and it references 30 data objects, its Halstead length should be $20 \log_2 20 + 30 \log_2 30 = 233.6$. How well does $H$ (the predicted Halstead length) compare with $N$, (the actual Halstead length measured on the program)? The answer is: closely. The validity of the relation has been experimentally confirmed, many times, independently over a wide range of programs and languages. Further-more, the relation appears to hold when a program is subdivided into modules.

The bug prediction formula is based on the four values: $n_1$, $n_2$, $N_1$, and $N_2$:

$$B = \frac{(N_1 + N_2)\log_2(n_1 + n_2)}{3000} \qquad (4)$$

All of which are easily measured parameters of a program. A program then that accesses 75 data objects a total of 1300 times and uses 150 operators a total of 1200 times, should be expected to have $(1300 + 1200)\log_2(75 + 150)/3000 = 6.5$ bugs.

In addition to the above equations, Halstead also derived relations that predict programming effort and time, and they also seem to correlate with experience. The time prediction is expectably nonlinear, showing 2.4 hours for a 120-statement program, 230 hours for a 1000-statement program, and 1023 hours for a 2000-statement program. Although the correlation is not perfect, it is at least of the right order of magnitude. Over a set of 24,000 statements, the predicted 31,600 work hours agreed with the actual 36,000 within 12%. Though not perfect, it is better than most simple metrics, but is not better than multifactor models such as COCOMO.

### 3.3.2. How Good?

Confirmation of these metrics has been extensively published by Halstead and others. The most solid confirmation of the bug prediction equation is by Lipow (LIPO82), who compared actual to predicted bug counts to within 8% over a range of programs sizes from 300 to 12,000 executable statements. The analysis was of postcompilation bugs, so that syntax errors caught by the compiler are properly excluded. However, of the 115,000 statements in the experiment, only the 75,000 executable statements were examined. It would be interesting to see whether better accuracy would have been obtained had all declarations been included in the analysis. Ottenstein, in an earlier report (OTTE79), showed similar good correlation. Curtis (CURT79A) shows that Halstead's metrics are at least twice as good as lines of code and are not improved by augmenting them with lines of code or with McCabe's metric.

There has been extensive experimental confirmation of the utility of Halstead's metrics, to the point that they have been firmly established as the principal linguistic metric. It is likely that if ever a true software science emerges, it will be based in part on Halstead's work. Other confirming studies include FEUE79A, FEUE79B, FITS80, FUNA76, GAFF84, LIPO86, and OTTE79. Critiques are to be found in CURT79A, DEYO79, DUNN82, LEVI86, LEVI87, LASS79, SHEP79C, and ZWEB79.

### 3.3.3. The Hidden Assumptions and Weaknesses

Because Halstead’s metric is the best established (serious) linguistic metric we have, it’s fitting that it be subjected to the harshest criticism. There are fewer hidden assumptions in Halstead’s work than in other comparable efforts, but some assumptions and weaknesses still remain. Some of the following weaknesses apply to all linguistic metrics.

1. *Modularity*—Modularity is not ignored because each call, together with the parameters of the call sequence, will contribute to the values of $n_1$, $n_2$, $N_1$, and $N_2$, and therefore to the predicted bug count. Note that Halstead treats each distinct subroutine call as a unique operator and not just as the one keyword “CALL.” In this respect, the impact of hypermodularity on bug rate is not ignored. The criticism is that Halstead does not distinguish between a programmer’s “own” subfunctions and a subfunction provided by another programmer. Each is given equal weight. The metric might be improved by fudging the weight given to external calls by multiplying it with a constant $k_f$, greater than 1, which is a measure of the thickness of the semantic fog that exists whenever two programmers attempt to communicate. For own subroutines and modules, the index is equal to 1, and Halstead’s criterion relating to optimum module size (see HALS77) holds. For external calls, the fog index might be as high as 5 (depending on the documentation clarity) and would result in bigger components produced by a single programmer. Common subroutines and macros in a library are treated as language extensions and have unity fog index as for built-in language features. If this isn’t valid, then documentation must be improved to the point where such calls are no more difficult than built-in keywords.
2. *Database Impact and Declarations*—Halstead isn’t to be faulted on this one. It’s just that most evaluators and confirmers have continued the erroneous practice of ignoring unexecutable statements such as declarations and data statements—which is to say that most initialization and data structure bugs are ignored. They can’t be faulted for this entirely, because in many cases errors in data declarations and initializing data statements are not even counted as bugs. If a true bug count is used, one that includes declaration errors and data statement bugs, then declarations and data statements, equates, constant declarations, and any other statements that affect not just what code is produced, but what data are loaded, should also be counted in determining Halstead’s metrics. If such bugs are ignored, the corresponding declarations should not be counted.
3. *Opera/Operand Ambiguity*—There is a hidden assumption that code is code and data are data, and never the twain should meet. If Halstead’s metrics are rigidly applied to an assembly language program that does extensive modification of its own code (ugh!) it will predict the same bug count as a routine that performs the same operations on a fixed data area. Trying to expand the bug prediction or program length equations to such dynamically varying calls would make it impossible to do a static calculation of the metrics. Only a dynamic count would do. In principle, the method still works. Each modified instruction, after each modification, would increase the operator count. Say that a programmer implemented a flag by changing a NOOP instruction to an unconditional branch (a common practice at one time), that instruction, in addition to its appearance as a unique operand, would also contribute two unique operators. The bug prediction would be clearly raised. Whether it would be increased to the point where it would correctly predict the catastrophic impact of instruction modification remains to be seen.
   The issue in code-data ambiguity is not with instruction modification, but with code masquerading as data. That is, with data whose actual function is control—as in jump tables, finite-state machines, undeclared internal languages and their interpreters, and similar, effective and useful programming tactics. There is nothing inherently weak in Halstead’s approach if we count such things correctly. Control and instructions in the guise of data are operators and not operands. State tables are tables of operators and operands. In the same vein, in languages that permit subroutine names in a call, each value of a subroutine name parameter should be treated as if it were different

operator. Similarly for calls that permit a variable number of operands—each value constitutes a different operator. Both of these are instances of dynamically bound objects, which no static metric can handle. See Section 3.4 below.

**4.** *Data-Type Distinctions*—In strongly typed languages that force the explicit declaration of all types and prohibit mixed-type operations unless a type conversion statement has been inserted, the weakness does not exist if we count all type conversation statements, whether or not they result in object code. If we count all declarations, including user-defined type declarations, there is no weakness in Halstead's metrics. The problem arises in the more common languages that permit mixed operations (e.g., integer floating-point addition) and that have no user-defined types. In principle, each use of an operator must be linked with type of data over which it operates. For example, if a language has integers and short and long operands, and mixed-mode operations are allowed for any combination, then each of the basic arithmetic operators (add, subtract, multiply, divide) actually represents six operators each, one for each combination of operand types. If a language has four types of arithmetic operands, then each of the basic operators correspond to ten operators, and so on for all combinations of data types and all arithmetic operators.

**5.** *Call Depth*—No notice is made of call depth. A routine that calls ten different subroutines as a sequence of successive calls would actually be considered more complex than a routine that had ten nested calls. If the totality of the routines and all that it calls were considered, the bug prediction would be the same for both because the total operator and operand counts would be the same. A nonunity fog index associated with each call, which would be multiplicative with depth would raise the predicted bug rate for deeply nested calls. This would be more realistic. Call depth is a structural property and therefore this criticism applies to all linguistic metrics.

**6.** *Operator Types*—An IF-THEN-ELSE statement is given the same weight as a FOR-UNTIL, even though it's known that loops are more troublesome. This is an example of a broader criticism—that operators and operands are treated equally, with no correlation to the bug incidence associated with specific operators or with operand types.

**7.** *General Structure Issues*—Of these, the fact that nesting is ignored (nested loops, nested IF-THEN-ELSE, and so on) is probably the most serious critique. A Basic statement such as

```
100 A = B + C @ D = SIN(A)                N = 9
```

is less bug-prone than

```
100 D = SIN(B + C)                        N = 6
```

but is claimed to have a higher bug potential. This claim is true if it is assumed that bugs are dominated by typographical errors, but untrue in reality. The first structure is more prone to typographical errors, but is clearer. The second structure is less prone to typographical errors, but is more prone to nesting errors. If we counted SIN(B + C) as a different operator than SIN(A), this problem is taken care of, but this may be an overly harsh use of operator counts. In general, then, a nested sequence of operators, be they loops or logic, is more complicated than an unnested sequence with the same operator and operand count. The possible solution might be a nesting fog index that increases by multiplication with increasing depth. Again, the weakness is generic to all linguistic metrics that ignore structure.

The gist of the above criticism is not so much with Halstead's metrics, but with the way we should apply them. They have proved their worth, but that doesn't mean we should accept and apply them blindly and in a doctrinaire, orthodox, immutable, and rigid manner. It's too early for that. The above critique implies methods of improvement; although some of those suggestions may in practice prove effective, most will probably prove to make little difference or to be outright wrong. It is only by trying, by making variations, by meticulous and honest recording of data, that we will find what components and what interpretations are really useful and the context to which those interpretations apply. The main value of Halstead's work is not in the "theorems"—because they're not really theorems in the sense that they can be proved deductively from axioms—but that he provided strong, cogent, albeit intuitive, arguments for the general functional form that our empirical "laws" should take and what the significant parameters of such "laws" should be.

### 3.4. Token Count

Levitin, in his incisive critique of linguistic metrics (LEVI86), makes a strong argument for the use of program **token** count rather than lines of code, statement count, or Halstead's length. A **token** in a programming language is the basic syntactic unit from which programs are constructed. Tokens include keywords, labels, constants, strings, and variable names. Token counting is easy because the first stage of compilation is to translate the source code into equivalent numerical tokens. The number of tokens is easily obtained as a by-product of compilation. Token count gets around the subjectivity problems we have with statement counts and lines of code.

There are two possible ambiguities: paired delimiters and dynamically bound objects. Paired delimiters are usually converted by the compiler into two tokens and then eliminated by the (typical) subsequent translation to parenthesis-free form (i.e., polish prefix or suffix form). Although syntactically, paired delimiters function as a single object, there is a strong case for treating them as two objects, in that a paired delimiter is more complex than a single operator, and bugs can occur with mismatched paired delimiters if both a begin and end delimiter are dropped or added. In some languages it is possible to add or drop a single half of the pair and still have a syntactically correct program because the second half is optional—for example, if a single END is used to service a bunch of BEGINs or a single ENDIF concludes a nesting of IF. . . THENs. The pragmatic approach is not to agonize over the point but to simply use the compiler's token count as the metric, however it treats paired delimiters. The issue is not important because at worst it introduces a statistically small error.

Levitin's argument (LEVI89) with respect to dynamically bound objects such as a variable number of objects in a call or a variable subroutine name in a call is that you cannot define a static metric over such objects—the size is dynamic and only a metric defined over the program at run-time will do. Dynamic binding violates our second desirable metric property. The observation is important because it clarifies the issue—the fact that the use of dynamically bound objects is much more complex than statically bound objects. A static count can still be used, but the metric should be augmented with a count of the instances of dynamically bound objects as determined at compile time, which could result in additional tokens when bound. That is, the metric consists of two parts: the static token count and unbound token count. This approach gets around the problems of how to count dynamic subroutine names in calls, pointer variables, variable number of objects in a call, etc. We would expect the unbound variable count to have a much higher coefficient in a bug prediction formula, say.

Why this simple, obvious, linguistic metric has been all but ignored in the literature is as mysterious to me as it is to Levitin. Its only significant weaknesses are those that apply to all purely linguistic metrics: e.g., ignoring nesting depth and treating all operators as equivalent. We should progress from simpler to more complicated metrics only to the extent that we have to in order to get good empirical laws. Most metrics, as actually used, are applied to source code, so that some of the utility of Halstead's conjecture is eroded—why estimate $N$ (effectively the token count) from $n_1$ and $n_2$ using Halstead's conjecture when the compiler can give you the actual value directly?

## 4. STRUCTURAL METRICS

### 4.1. General

Structural metrics take the opposite viewpoint of linguistic metrics. Linguistic complexity is ignored while attention is focused on control-flow or data-flow complexity—metrics based on the properties of flowgraph models of programs. Graph theory is more than a half-century old but the study of graph-theoretic problems and metrics goes back three centuries: see MAYE72. If it's a graph (i.e., it consists of links and nodes) then there are hundreds of interesting metrics that can be applied to it: metrics whose mathematical properties have been studied in minute detail. The thrust of structural metrics research has not been so much the invention of new graph-theoretic metrics but the investigation of the utility of known graph-theoretic metrics. McCabe's use of the cyclomatic number (MCCA76) is an archetypal example.

## 4.2. Cyclomatic Complexity (McCabe's Metric)

### 4.2.1. Definition

McCabe's cyclornatic complexity metric (MCCA76) is defined as:

$$M = L — N + 2P$$

where

$L$ = the number of links in the graph

$N$ = the number of nodes in the graph

$P$ = the number of disconnected parts of the graph (e.g., a calling program and a subroutine)

The number $M$ that appeared alongside the flowgraphs in Chapter 3 was McCabe's metric for that flowgraph. In all the examples, except for the one on page 73, there was only one connected part, and consequently the value of $P$ was 1. Figure 7.1 shows some more examples and their associated $M$ values.

This metric has the intuitively satisfying property that the complexity of several graphs considered as a set is equal to the sum of the individual graphs' complexities. You can see this by analyzing Figure 7.2. Two disconnected graphs having $N_1$, $L_1$ and $N_2$, $L_2$ nodes and links respectively, are combined and treated as a single entity with $N_1 + N_2$ nodes and $L_1 + L_2$ links. The arithmetic shows directly that the complexity of the sum is equal to the sum of the complexities.

This metric can also be calculated by adding one to the number of binary decisions in a structured flowgraph with only one entry and one exit. If all decisions are not binary, count a three-way decision as two binary decisions and $N$-way case statements as $N – 1$ binary decisions. Similarly, the iteration test in a DO or other looping statement is counted as a binary decision. The rationale behind this counting of $N$-way decisions is that it would take a string of $N – 1$ binary decisions to implement an $N$-way case statement.

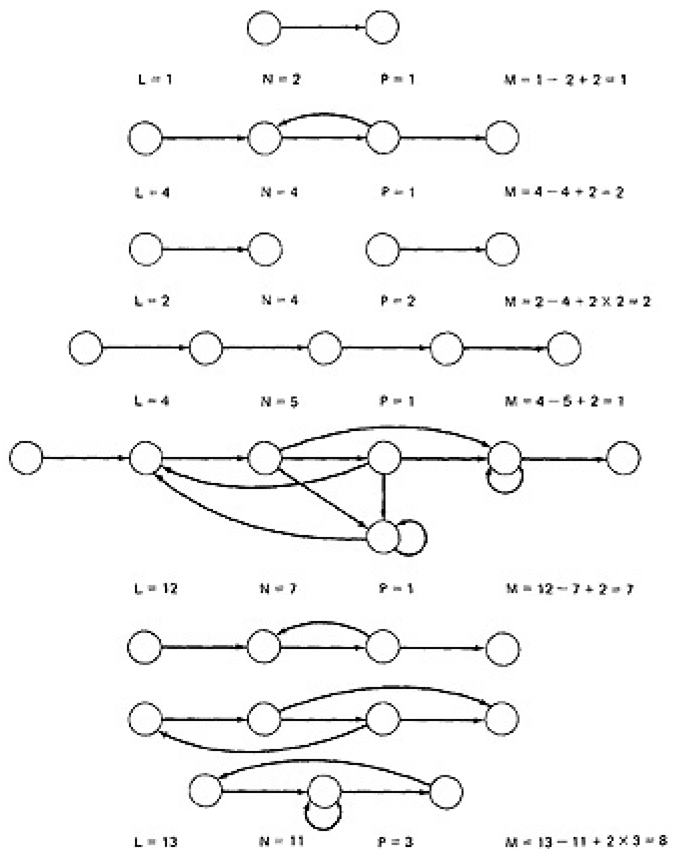

**Figure 7.1.** Examples of Graphs and Calculation of McCabe's Complexity Metric.

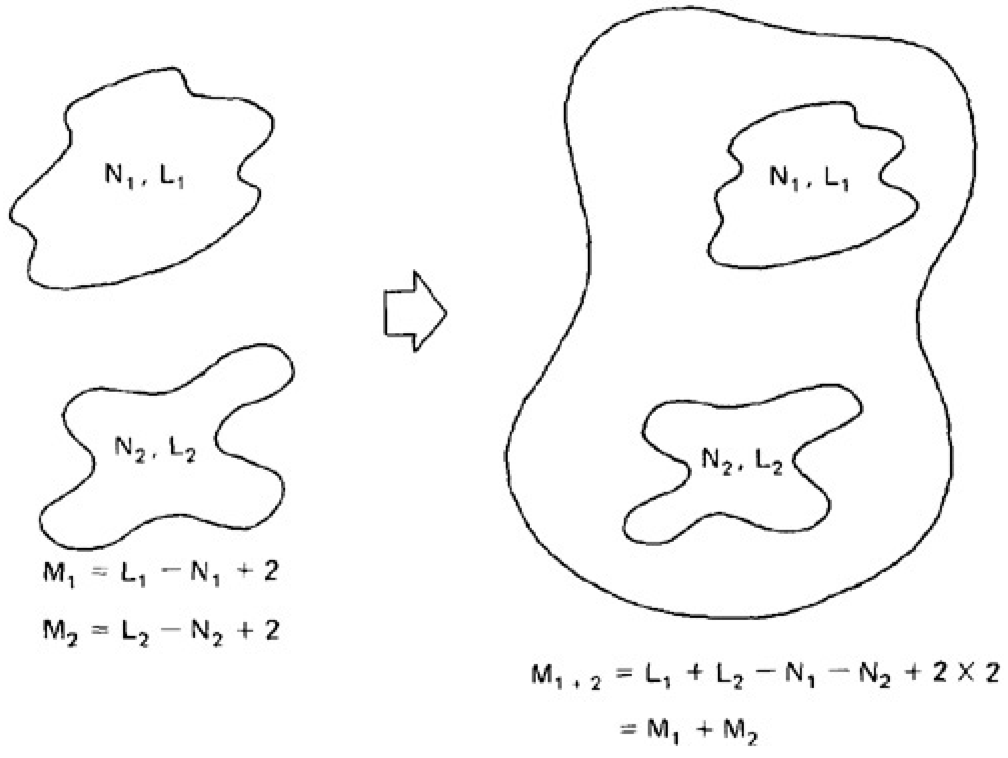

**Figure 7.2.** Complexity Sum and Sum of Complexities.

### 4.2.2. Applications to Test Plan Completeness and Inspections

Evaluate the cyclomatic complexity of the program's design (e.g., from the design control flowgraph). As part of self-inspection, reevaluate the complexity by counting decisions in the code. Any significant difference should be explained, because it's more likely that the difference is due to a missing path, an extra path, or an unplanned deviation from the design than to something else. Having verified the code's cyclomatic complexity, compare the number of planned test cases to the code's complexity. In particular, count how many test cases are intended to provide coverage. If the number of covering test cases is less than the cyclomatic complexity, there is reason for caution, because one of the following may be true:

1. You haven't calculated the complexity correctly. Did you miss a decision?
2. Coverage is not really complete; there's a link that hasn't been covered.
3. Coverage is complete, but it can be done with a few more but simpler paths.
4. It might be possible to simplify the routine.

**Warning:** Don't be rigid in applying the above because the relation between cyclomatic complexity and the number of tests needed to achieve branch coverage is circumstantial. Use it as a guideline, not as an immutable fact.

### 4.2.3. When to Subroutine

McCabe's metric can be used to help decide whether it pays to make a piece of code which is common to two or more links into a subroutine. Consider the graph of Figure 7.3. The program has a common part that consists of $N_c$ nodes and $L_c$ links. This is the part being considered for conversion to a subroutine. This common part recurs *k* times in the body of the main program. The main program has $N_m$ nodes and $L_m$ links over and above the common part. The total number of links and nodes for the main program, therefore, is $L_m + kL_c$ and $N_m + kN_c$. When the common parts are removed, an additional link must be added to the main program to replace the code by a subroutine call. The subroutine's code must be augmented with an additional entry node and exit node. The following table summarizes the transformation:

| | **EMBEDDED COMMON PART** | **SUBROUTINE FOR COMMON PART** |
|---|---|---|
| Main nodes | $N_m + kN_c$ | $N_m$ |
| Main links | $L_m + kL_c$ | $L_m + k$ |
| Subnodes | 0 | $N_c + 2$ |
| Sublinks | 0 | $L_c$ |
| Main complexity | $L_m + kL_c - N_m + kN_c + 2$ | $L_m + k$ |
| Subcomplexity | 0 | $L_c - N_c - 2 + 2 = L_c - N_c = M$ |
| Total complexity + 2 | $L_m + kL_c - N_m + kN_c + 2$ | $L_m + L_c - N_m - N_c + k + 2$ |

The break-even point occurs when the total complexities are equal. A little algebra shows that this is independent of the main routine's complexity and is equal to: $M_c = k/(k - 1)$. For one call ($k = 1$), the total complexity must increase no matter how complex the subroutine itself is. For two calls, the crossover occurs at a complexity of 2 for the subroutine. For more calls, the crossover complexity decreases and is asymptotic to 1. In general, then, creating subroutines out of straight-line code (complexity of 1) tends to increase net complexity rather than reduce it, if one takes into account the complexity of the calls and the fact that there are separate routines. Of course, you would not use this analysis as the sole criterion for deciding whether or not to make a subroutine out of common code.

One of the popular fads of the 1970s was blind modularization. Rules such as “No subroutine shall contain more than a hundred statements” were used in the hope of reducing bug frequency. Presumably, if programmers followed such rules, the programs would be simpler, easier to debug, more reliable, and so on. This rule was, unfortunately, put into several government specifications. The statistics (discussed below) show that McCabe’s metric is a better measure of complexity than lines of code. It’s intuitively better because it takes into account the increase in complexity resulting from subdividing a routine—something which lines of code does not do. If anything, McCabe’s metric underestimates the impact of subdividing code, because it says that the complexity of the whole is equal to the sum of the complexities of its parts—which is neat, but it is an underestimate because complexity increases nonlinearly with more parts. The above analysis warns us that complexity can increase, rather than decrease, with modularization—as the following experience with my favorite bad project shows:[*]

[*]It was a really bad project, but it had its good sides. I will never again witness so many bad design practices and such a lack of testing discipline (which supplied ample material for this book) under one roof. It was a real project, and it was rotten from keelson to hounds. Obviously, I can’t identify it without risking a libel suit. As for discussing bad projects and what appears to be an uncommon frequency of them in my experience—it’s not that I’ve had bad luck, but that consultants aren’t usually called in for good projects. As for discussing bad projects at all, there’s more to be learned from our mistakes than from our triumphs.

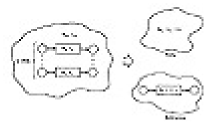
**Figure 7.3.** Subroutine Complexity.

The system was a multicomputer, distributed-processing, distributed-database kind of thing—a tough proposition at best. Of all the silly rules that this benighted project adopted in the interest of improving programmer productivity and software reliability, the silliest was an absolute, unappealable restriction of all modules to less than fifty *object-code* instructions.

Because the program was written in a higher-order language, and because the call/return sequence took up about half of the object code, what resulted were thousands of tiny subroutines and modules hardly more than ten source statements long. The typical number of calls per module was 1.1 ($k = 1.1$), which meant that the subroutines would have had to have a complexity of 10 or greater, if net complexity was to be reduced. But the fifty-object-statement rule kept the average complexity down to about 1.01. This hypermodularity increased the predicted test and debug effort to several times as much as had been estimated for unit design, test, and debugging. Because the hyperfine modularity and the other rules adopted were supposed to eliminate *all* bugs, the labor estimated for system integration and test was an order of magnitude less than it should have been. My estimates of the labor needed to integrate this vast horde of tiny, gnat-like subroutines was about 30 or 40 times higher than the project’s management expected. My predictions were not politely received, to say the least, and my association with the project ended soon after. They quietly dropped that rule (so I heard) when they redesigned the system for the version that did (after a fashion) work.

I don’t advocate great complicated unwashed masses of overly verbose code that rambles on and on and on and on, page after page after page, beyond the scope of human ken. Programs like that are also untestable. But consider this when setting modularity rules: in addition to the net increase in complexity due to breaking a function up, there is a further increase attributable to additional code in a calling sequence and possibly additional declarations. All of these increases add to the bug rate and testing labor. Furthermore, each subdivision creates a new interface between the calling routine and called routine, and all interfaces breed bugs. Partition cannot be dictated by an arbitrary rule but must result from a trade analysis whose resulting optimum value is unlikely to exist at either extreme of the range.

### 4.2.4. A Refinement

Myers (MYER77) points out a weakness of McCabe's metric and suggests the use of a refinement thereto. A decision statement in languages such as FORTRAN can contain a compound predicate of the form: IF A & B & C THEN . . . . A statement such as this could be implemented as a string of IFs, resulting in a different complexity measure. Figure 7.4 shows three alternate, equivalent, representations of the same IF-THEN-ELSE statement. If the compound predicate is used in a single statement as in the first case, the complexity of the construct is only two, but if it is broken down into its constituent parts, the complexity increases to four. However, intuitively, all three constructs should have the same complexity. The refinement consists of accounting for each term of the predicate expression separately. For a predicate expression of the form A&B&C . . . , each predicate should be counted as if it was a decision. In more complicated expressions, such as A&B&C OR D&E . . . , again count each predicate. If a predicate appears more than once in an expression, you can take a pessimistic point of view and count each appearance or a slightly optimistic point of view and count only the first appearance.

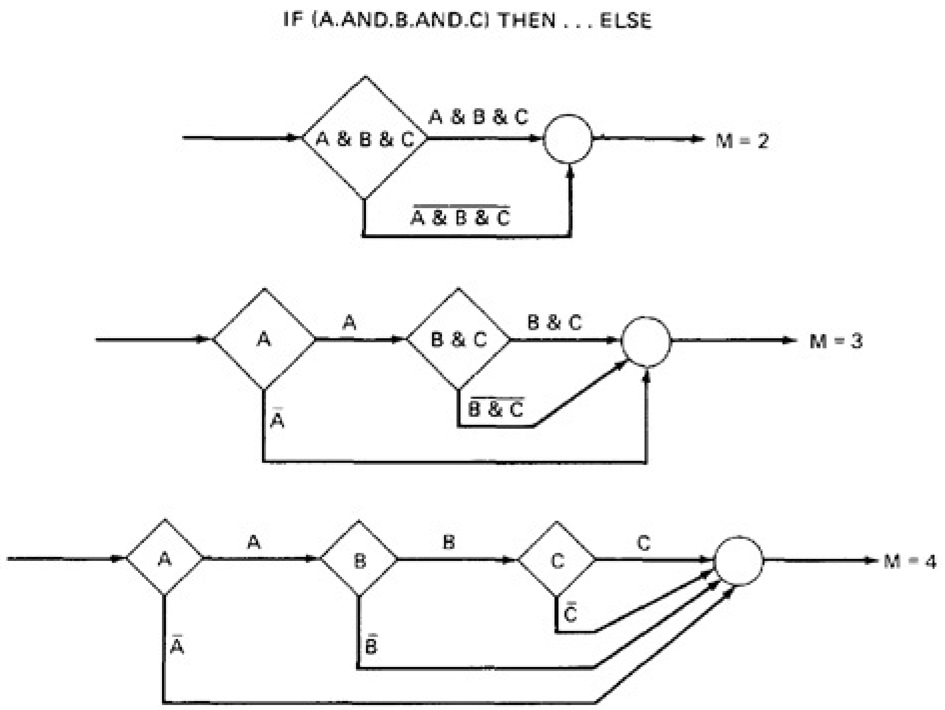

**Figure 7.4.** Cyclomatic Complexity and Compound Predicates.

### 4.2.5. How Good a Measure; A Critique

Statistics on how well McCabe's metric correlates with design, test, and debugging difficulty are encouraging (BELF79, CHEN78A, CURT79A, ENDR75, FEUE79A, FEUE79B, LIPO77, SCHN79A, SCHN79B, SCHN79D, SHEP79C, THAY76, WALS79, ZOLN77).* The reported results confirm the utility of McCabe's metric as a convenient rule of thumb that is significantly superior to statement count.

*Not all of these references provide direct information for McCabe's metric. Some (ENDR75, THAY76) provide correlations to decision counts and similar metrics that can be converted to McCabe's metric. Note also that if we forget the structural basis of cyclomatic complexity and merely use equivalent binary decision count or simple predicate count, then this metric can be obtained by purely linguistic means.

McCabe advises partitioning routines whose metric exceeds 10. Walsh confirms this advice (WALS79) by citing a military software project to which the metric was applied. They found that 23% of the routines with a metric value greater than 10 accounted for 53% of the bugs. Walsh further states that in the same study of 276 procedures, the routines with *M* greater than 10 had 21% more errors per line of code than those with metric values below 10. Feuer (FEUE79A, FEUE79B) cites mixed results but provides strong correlation between the metric (actually, decision counts) and error rates for big programs. Curtis (CURT79A) also shows fair predictability. For other references related to structural complexity, McCabe's metric, and related subjects see BAKE79A, BAKE80, PAIG75A, PAIG77, PAIG78, SCHN79B, and WOOD78.

It's too early to tell whether this metric will eventually serve as a true bug predictor. It's not likely. The main appeal of *M* is its simplicity. People have long used program length as the main criterion for judging software complexity. Simple measures are inherently inaccurate but they can provide useful rules of thumb. McCabe's metric should be used in combination with token count to get a quick measure of complexity.

McCabe's metric has some additional weaknesses. It makes no real distinction as to the direction of flow. For example, an IF-THEN-ELSE statement has the same complexity as a single loop. This situation is not intuitively satisfying because we know that loops are more troublesome than simple decision sequences. Case statements, which provide a nice, regular structure, are given a high complexity value, which is also counterintuitive. It also tends to judge highly regular, repetitive control structures unfairly. Conversely, if these weaknesses were to be rectified, it could only be done by increasing the complexity of the metric itself and taking it out of the realm of an easy-to-use measure of complexity.

The strongest criticism of this metric is its shaky theoretical foundation (EVAN84). McCabe wanted to relate the cyclomatic complexity to the number of test cases needed—that is, to use cyclomatic complexity as a coverage metric. His testing strategy ("structured testing," MCCA82) is based on selecting control-flow paths that satisfy graph-theoretic properties on which this metric is based. The cyclomatic number of a graph is the number of paths in the base set of paths for strongly connected, undirected, graphs. That is, for graphs whose exit is connected to the entry **(strongly connected)** and for which all links are bidirectional **(undirected)**, all paths can be built out of a combination of paths in the base set. McCabe's conjecture was that adequate coverage could be achieved by using a base set of test paths. Evangelist (EVAN84) says no and provides counterexamples. Furthermore, research has failed to provide either a theoretical basis for this conjecture or to relate McCabe's heuristic testing strategies to any other structural testing strategy even though the strategy appears to be pragmatically effective. Despite these criticisms, whether McCabe's reasoning is valid or invalid, cyclomatic complexity (or more often, equivalent binary decision count) is a firmly established, handy rule of thumb for structural complexity.

### 4.2.6. Utility Summary

Cyclomatic complexity provides some useful rules of thumb:

**1.** Bugs per line of code increase discontinuously for *M* greater than 10.
**2.** Arbitrary modularity rules based on length, when applied to straight-line code that has few calls or only one call, increase rather than reduce complexity.
**3.** The amount of design, code, test design, and test effort is better judged by cyclomatic complexity than by lines of code.
**4.** Routines with high complexity, say 40 or more, should be given close scrutiny, especially if that complexity is not due to the use of case statements or other regular control structures. If the complexity is due to loops and raw logic, consideration should be given to subdividing the routine into smaller, less complex segments in order to avoid the nonlinear increase in effort associated with high complexity.
**5.** Cyclomatic complexity establishes a useful lower-bound rule of thumb for the number of cases required to achieve branch coverage—especially when based on simple predicate counts rather than raw decision counts (i.e., use the number of simple predicates plus 1). If fewer test cases than *M* are proposed, look for missing cases or the possibility of simplifying the logic or using tests that are less complicated.

### 4.3. Other Structural Metrics

Many other structural metrics have been proposed and investigated. Chen (CHEN78A) combines structural properties with information theoretic concepts. Gong (GONG85) combines decisions and nesting depth. Rodriguez and Tsai (RODR86) apply structural concepts to data flowgraphs, as do Tai (TAIK84) and Tsai et al. (TSAI86). Van Verth (VANV87) proposes using a combination of control flow and data flow, as does Whitworth (WHIT80B). This list is incomplete as the number of papers on alternate complexity metrics is as great as those on alternate testing strategies. The problem with these metrics is that confirming statistics on their usefulness is lacking. It is intuitively clear that cyclomatic complexity over data flowgraphs should be as useful a metric as cyclomatic complexity over control flowgraphs but corroboration is still lacking (OVIE80).

## 5. HYBRID METRICS

The appearance of McCabe's and Halstead's metrics spurred the proposal, development, refinement, and validation of a host of similar and related metrics, or totally different alternatives based on different assumptions. Some of those cited below preceded McCabe's and Halstead's work and some followed. It's too early to tell which will be experimentally confirmed independently over a range of projects and applications, no matter how rational and sensible their basis might be. Some of the more interesting and promising alternatives are presented in BAKE79A, BAKE80, BELF79, CHAP79, CHEN78A, DEYO79, EVAN84, FEUE79B, GILB77, LIPO77, MCCL78B, PAIG80, RAMA88, SCHN79B, VANV87, WHIT80B, and ZONL77. Most of these metrics and their variations recognize one or more weaknesses in the popular metrics and seek to measure reliability and/or predict bug counts through refinement or alternative formulation of the problem. It is inevitable that increased predictability and fidelity can only be achieved at a cost of increased sophistication, complexity of evaluation, and difficulty of use.

## 6. METRICS IMPLEMENTATION

### 6.1. The Pattern of Metrics Research

You may well ask why you, who may be concerned with getting next week's release working, should care about how metrics research is done. There are several reasons:

**1.** Like it nor not, sooner or later, what you do will be measured by metrics. As a programmer or tester, the quality of your work and your personal productivity will be tied (rightly or wrongly) to metrics whose validity ranges from statistically sound, to theoretically sound but not germane, to downright stupid. Understanding the research issues may help you to tell them apart.
**2.** Metrics *are* useful—to programmers, testers, and managers, and almost any complexity metric (even listing weight) is better than no metric at all.
**3.** If you're using any complexity metric at all, you're either doing metrics research yourself or you're a subject in a metrics research experiment. Some experiments are better (worse) than others. I don't really mind being run like a rat through a maze but I'd like to know about it.

Let's go back to how a (good) metrics research project is done:

**1.** A new metric is proposed, based on previous metrics, a functional combination of previous metrics, wholly new principles, intuition, valid or invalid reasoning, or whatever. The metric's definition is refined to the point where a tool that measures it can be written.
**2.** The metric is compared by correlation analysis to previous metrics: lines of code, Halstead's length, Halstead's volume, and cyclomatic complexity are favorites. The correlation analysis is done over a set of known programs such as a set of programs for which metric values were previously published. The new metric shows a high correlation to the old metrics. Metrics that don't correlate well usually die a quiet death. Correlation with previous metrics is important because we know how well (or how poorly) such metrics correlate with things we're interested in, such as number of tests required, expected number of remaining bugs, programming effort, and so on.
**3.** The metric is used to predict interesting characteristics of programming and software for new code or for maintenance. Statistics are gathered and the predictions are compared with actual results. Good predictors are retained, bad predictors remain unpublished.
**4.** The metric is "tweaked" (i.e., refined) to provide better predictions and step 3 above is repeated. This process continues until the predictor stabilizes.
**5.** A more sophisticated alternative is to throw a whole bunch of possible metrics into a pot and do the above steps with the additional refinement of using regression analysis (DRAP81) to obtain a weighted multifactor metric.

Most published metrics research doesn't get beyond step 2, and little has been published that gets past the iteration between steps 3 and 4. We seem to have more papers that critique old metrics and that propose new metrics (to be subsequently critiqued by others) than papers that show how useful a specific metric is. Useful comparisons and criticisms of various metrics can be found in BAKE79A, CURR86, CURT79A, DAVI88C, EVAN84, FEUE79A, FEUE79B, GRAD87, HARR82, LEVI86, LIHF87, LIND89, NEJM88, PAIG80, RODR86, RODR87, VANV87, and WEYU88B. The most useful information on metrics research, though, is never published because the sponsors of that research consider the information too valuable and too proprietary to divulge it to their competitors.

## 6.2. A Pattern for Successful Applications

Successful applications of metrics abound but aren't much talked about in the public literature (but see GRAD87). Mature metrics (emphasis on mature—see below) can help us predict expected number of latent bugs, help us decide how much testing is enough and how much design effort, cost, elapsed time, and all the rest we expect from metrics. Here's what it takes to have a success.

**1.** *Any Metric Is Better Than None*—A satnav system or loran is better than a compass, but even a compass beats navigating with your head stuck in the bilges. Correlate what you're interested in with listing weight if that's the best you can do. Implement token counting and cyclomatic complexity as the next step. Worry about the fancy metrics later.

**2.** *Automation Is Essential*—Any metrics project that relies on having the programmers fill out long questionnaires or manually calculate metric values is doomed. They never work. If they work once, they won't the second time. If we've learned anything, it's that a metric whose calculation isn't fully automated isn't worth doing: it probably harms productivity and quality more than any benefit we can expect from metrics.

**3.** *Empiricism Is Better Than Theory*—Theory is at best a guide to what makes sense to include in a metrics project—there's no theory sufficiently sound today to warrant the use of a single, specific metric above others. Theory tells you what to put into the empirical pot. It's the empirical, statistical data that you must use for guidance.

**4.** *Use Multifactor Rather Than Single Metrics*—All successful metrics programs use a combination (typically linear) of several different metrics with weights calculated by regression analysis. Some organizations measure more than 200 factors but only a few of these end up with significant coefficients in prediction formulas.

**5.** *Avoid Naive Statistics*—Getting empirical laws from measured data is old hat to statisticians and they know how to avoid the statistical gaffes that trap the unwary amateur. I saw one big project go down the tubes because of naive statistics. They predicted the safe release point with exquisite precision but no accuracy—"We can ship the release on June 25 at 2:04:22.71 P.M. (plus or minus three centuries)." Bring a professional statistician in to assure yourself that the metrics are significant when compared to random events. Use a statistician to tell you how much confidence is or isn't warranted in the predictions.

**6.** *Don't Confuse Productivity Metrics with Complexity Metrics*—Productivity is a characteristic of programmers and testers. Complexity is a characteristic of programs. It's not always easy to tell them apart. Examples of productivity metrics incorrectly used in lieu of complexity metrics are: number of shots it takes to get a clean compilation, percentage of modules that passed testing on the first attempt, number of test cases required to find the first bug. There's nothing wrong with using productivity metrics as an aid to project management, but that's a whole different story. Automated or not, a successful metrics program needs programmer cooperation. If complexity and productivity are mixed up, be prepared for either or both to be sabotaged to uselessness.

**7.** *Let Them Mature*—It takes a lot of projects and a long time for metrics to mature to the point where they're trustworthy. If the typical project takes 12 months, then it takes ten to fifteen projects over a 2- to 3-year period before the metrics are any use at all.

**8.** *Maintain Them*—As design methods change, as testing gets better, and as QA functions to remove the old kind of bugs, the metrics based on that past history loses its utility as a predictor of anything. The weights given to metrics in predictor equations have to be revised and continually reevaluated to ensure that they continue to predict what they are intended to predict.

**9.** *Let Them Die*—Metrics wear out just like test suites do, for the same reason—the pesticide paradox. Actually, it's not that the metric itself wears out but that the importance we assign to the metric changes with time. I've seen projects go down the tubes because of wornout metrics and the predictions based on them.

## 7. TESTABILITY TIPS

We don't have testability metrics yet—but we have complexity metrics and we know that more complexity means more bugs and more testing to find those bugs. It's not that you shouldn't design complicated software, but that you should know your design's complexity.

**1.** Use Halstead's conjecture, Eq. (1), to estimate the token count before coding. You can estimate the number of operands ($n_2$) from the data dictionary and the number of operators ($n_1$) from your experience with the language and the number of subroutine and function calls you expect to make.

**2.** Use the number of equivalent binary decisions in your PDL or design control–flowgraph as an estimator of control-flow complexity. Remember that there's a jump in the bug potential for cyclomatic complexity at around 10 to 15. Remember that cyclomatic complexities of more than 40 (excluding neat case statements) are probably over your (and anyone's) head. Learn your own limits and your own ability to handle control-flow complexity.
**3.** If you use a design methodology or data-structuring tools that allow you to find the total data-flow complexity, use that as a guide to how much data complexity you can tolerate.
**4.** Once you have compiled code, use token counts, and if you have the tools for it, also control-flow and data-flow cyclomatic complexity. Compare these metrics to the precoding estimates to be sure that you haven't inadvertently buried yourself in complexity. Break it up before testing and debugging if you're in over your head.
**5.** As testers and designers, support your QA group and agitate for corporate, project, and personal statistics that relate the expected number of bugs and the expected number of tests needed to shake out those bugs to the primary complexity metrics (tokens, control flow, data flow). Agitate for tools that will pull this stuff out for you directly as a by-product of compilation. Don't stop using your intuition and common sense—use the statistics to sharpen them.

We all have personal complexity barriers. The barriers we can safely hurdle get bigger with experience, stabilize to some maximum personal level, and then get smaller as we age and lose vitality. A by-product of experience is that we have a better understanding of our personal barriers and are less likely to get in over our heads—that's called wisdom. Because there's less wasted effort banging our heads against barriers we can't conquer, net productivity (that is, the production of fully tested, reliable software as distinct from raw code) generally increases with experience. Exploit the complexity metrics and their relation (however fuzzy that relation may be) to testing requirements and bug potentials to learn about yourself, in weeks and months, what others have taken years and decades to learn.

## 8. SUMMARY

**1.** Computer science is still far from a quantitative science. Because of the high human content, the model is more likely to be that of economics than of physics.
**2.** Three complexity metrics appear to be basic: token count, control-flow cyclomatic complexity, and data-flow cyclomatic complexity. Learn to use these instead of lines of code.
**3.** Don't bother with metrics that aren't automated: the payoff is probably less than the effort, and the accuracy is dubious.
**4.** As testers and designers we want to relate the metric value to a prediction of the expected number of bugs in a piece of code and the expected number of tests needed to find those bugs. That relation, today, is best obtained empirically from data gathered over a long period of time and many different pieces of software. The specific values are personal; that is, they differ widely between applications, organizations, projects, and individuals. Until we understand the underlying cause of bugs we'll have to make do with such personal, empirical "rules."
**5.** All metrics today, and the relations derived from them, are at best quantified rules of thumb dressed up in statistical finery—they provide guidance. Don't use them as rigid rules.
**6.** Metrics are based on statistics. It's as important to know how much statistical confidence is warranted in a metric as it is to know the results that the metric claims to predict. Don't get trapped by naive statistics.

<ch6 toc ch8>