CHAPTER EIGHT

# Beyond Lines of Code: Do We Need More Complexity Metrics?

*Israel Herraiz*
*Ahmed E. Hassan*

Complexity is everywhere in the software life cycle: requirements, analysis, design, and of course, implementation. Complexity is usually an undesired property of software because complexity makes software harder to read and understand, and therefore harder to change; also, it is believed to be one cause of the presence of defects. Of all the artifacts produced in a software project, source code is the easiest option to measure complexity. However, several decades of software research have failed to produce a consensus about what metrics best reflect the complexity of a given piece of code. It's hard even to compare two pieces of code written in different programming languages and say which code is more complex. Because of this lack of resolution, a myriad of possible metrics are currently offered to measure the complexity of a program. What does the research say are the best metrics for each particular case? Are all these metrics any better than very simple source code metrics, such as lines of code?

In this chapter, we take advantage of the huge amount of open source software available to study the relationships between different size and complexity metrics. To avoid suffocating in the myriads of attributes and metrics, we focus only on one programming language: C, a "classic" in software development that remains one of the most popular programming languages. We measure a grab-bag of metrics ranging from the simplest and most commonly cited (lines of code) to some rather sophisticated syntactic metrics for a set of about 300,000 files. From this we have found out which metrics are independent from a statistical point of

view—that is to say, whether traditional complexity metrics actually provided more information than the simple lines-of-code approach.

## Surveying Software

The first step in this study was to select a representative sample of software, just like surveys in the social sciences. From a known population, statistical methods allow us to obtain the minimum sample size that lets us extract conclusions with a given uncertainty. For instance, the U.S. population is about 300 million people at the moment of writing this text; exit polls can accurately predict the results of elections if the size of the polled sample is large enough, say, 30,000 people.

The problem with this approach when carried over to software engineering is that we do not know the size of the world's software. So we cannot determine a minimum sample that can answer our questions with a given uncertainty, and the classic "survey approach" to the whole population of software becomes unfeasible.

However, even though the whole population of software is indeterminable, a portion of that population is open, accessible for research, and willing to share its source code with the world: open source software. Of course, restricting our population to this kind of software should theoretically bind the answer to our initial question to this kind of software. But when all is said and done, the only difference between open source and closed source software is the license. Although open source software is usually developed using particular practices (projects that are community-driven, source code available, etc.), the open source software population is very heterogeneous, ranging from projects that are completely community-driven to projects that remain under the close control of companies. The only feature held in common by open source software is the set of licenses under which it is released, making its source code publicly available. Therefore, we can assume that the best complexity metrics for source code obtained from open source projects are also the best complexity metrics for any other source code, whether open source or not.

Open source software also presents some other interesting properties for research. Open source software repositories are usually completely available to researchers. Software repositories contain all the artifacts produced by the software project—source code releases, control version systems, issue tracking systems, mailing lists archives, etc.—and often all the previous versions of those artifacts. All that information is available for anyone interested in studying it. Thus, open source not only offers a huge amount of code, allowing us to study samples as large as we might want, but also makes possible repeatable and verifiable studies, which are fundamental and minimal requirements to ensure that the conclusions of empirical studies can be trusted.

There are some perils and pitfalls when using open source software, though. It is hard to obtain data for a large sample of open source software projects because of the heterogeneity in the data sources. Not all projects use the same tools for the different repositories, or even the same kind of repositories, and often those repositories are dispersed across different sites. Some efforts, such as the FLOSSMetrics (*http://flossmetrics.org*) and FLOSSMole (*http://flossmole.org*) projects, deliver databases containing metrics and facts about open source projects, which alleviate the heterogeneity of data when mining open source software repositories. And this problem is partially solved by the open source software community itself in the form of software distributions, such as the well-known Ubuntu and Debian distributions. Distributions gather source code from open source projects, adapt it to the rest of the distribution, and make it available as compiled binary and source code packages. These packages are tagged with meta-information that helps classify them and retrieve the different dependencies needed to install them. Some of these distributions are huge, encompassing thousands of packages and million of lines of source code. So they are ideal for any study that needs to gather a large amount of source code, like the one in this chapter.

## Measuring the Source Code

We have selected for our case study the ArchLinux software distribution (*http://archlinux.org*), which contains thousands of packages, all open source. ArchLinux is a lightweight GNU/Linux distribution whose maintainers refuse to modify the source code packaged for the distribution, in order to meet the goal of drastically reducing the time that elapses between the official release of a package and its integration into the distribution. There are two ways to install a package in ArchLinux: using the official precompiled packages, or installing from source code using the Arch Build System (ABS).

ABS makes it possible to retrieve the original, pristine source code of all the packages. This is different from other distributions, which make copies of the source code of the packages and often patch it to adapt it to the rest of the distribution. With ABS, we can gather source code from its original location, at the upstream projects' websites and repositories, in an automatic fashion. This ensures that the source code has not been modified, and therefore that the case studies in our sample are independent. As we will show later in the results section, this property of independence is crucial for the validity of the results.

Because of the size of ArchLinux, using it as a case study gives us access to the original source code of thousands of open source projects, through the build scripts used by ABS (see Example 8-1).

*EXAMPLE 8-1. Header of a sample build script in ArchLinux*

```
pkgname=ppl
pkgver=0.10.2 ❶
pkgrel=2 ❷
pkgdesc="A modern library for convex polyhedra and other numerical abstractions."
arch=('i686' 'x86_64')
url="http://www.cs.unipr.it/ppl"
license=('GPL3')
depends=('gmp>=4.1.3')
options=('!docs' '!libtool')
source=(http://www.cs.unipr.it/ppl/Download/ftp/releases/$pkgver/ppl-$pkgver.tar.gz) ❸
md5sums=('e7dd265afdeaea81f7e87a72b182d875') ❹
```

❶ Version of the package. Used to build the download URL.

❷ Minor release version number. Also used to build the download URL.

❸ Source code download URL.

❹ Checksum of the source tarball.

Example 8-1 shows the header of sample build script in ABS. The header contains meta-information that is used to gather the sources, retrieve other dependencies from the package archives, and classify the package in the archives once it is built. We have used the fields highlighted in Example 8-1 to retrieve the source code of all the packages in the ArchLinux archives.

For all the source code gathered, we determined the programming language of every file using the *SlocCount* tool (*http://www.dwheeler.com/sloccount*). Using only C language code for our sample, we measured several size and complexity metrics using the Libresoft tools' *cmetrics* package (*http://tools.libresoft.es/cmetrics*).

## A Sample Measurement

Table 8-1 contains a summary of all the metrics used in this study, and the symbols denoting these metrics in the rest of the tables and figures.

*TABLE 8-1. Selected metrics for the study*

| Variable | Metric (symbol) |
|---|---|
| Size | Source Lines of Code (SLOC), Lines of Code (LOC) |
| | Number of C functions (FUNC) |
| Complexity | McCabe's cyclomatic complexity—maximum of all functions (MCYCLO) |
| | McCabe's cyclomatic complexity—average (ACYCLO) |
| | Halstead's length (HLENG), volume (HVOLUM), level (HLEVE), and mental discriminations (HMD) |

The elements of code that provide input to all these metrics are illutrated in the sample source code file shown in Example 8-2. This file was extracted from the package `urlgfe`, a cross-platform download manager. (`urlgfe` has recently changed its name to `uget`, so it is no longer found in the ArchLinux repositories with its original name.) The file contains preprocessor directives (such as ❶), comments (such as ❸), and only one function (starting at line ❹) containing a `while` loop.

*EXAMPLE 8-2. A sample C source code file*

```
#ifdef HAVE_CONFIG_H ❶
# include <config.h>
#endif
❷
/* Specification.  */ ❸
#include "hash-string.h"


/* Defines the so called `hashpjw' function by P.J. Weinberger
   [see Aho/Sethi/Ullman, COMPILERS: Principles, Techniques and Tools,
   1986, 1987 Bell Telephone Laboratories, Inc.]  */
unsigned long int ❹
__hash_string (const char *str_param)
{
  unsigned long int hval, g;
  const char *str = str_param;

  /* Compute the hash value for the given string.  */
  hval = 0;
  while (*str != '\0')
    {
      hval <<= 4;
      hval += (unsigned char) *str++;
      g = hval & ((unsigned long int) 0xf << (HASHWORDBITS - 4));
      if (g != 0)
        {
          hval ^= g >> (HASHWORDBITS - 8);
      hval ^= g;
        }
    }
  return hval;
}
```

❶ Preprocessor directives, counted both for LOC and SLOC.

❷ Blank lines. Counted for LOC but not for SLOC.

❸ Comment lines. Counted for LOC but not for SLOC.

❹ Code, counted both for LOC and SLOC.

The simplest complexity metrics that we can measure are the ones related to lines of code (total lines of code and source lines of code). The number of functions also can be extracted easily from the source code. The rest of the complexity metrics that we measured are slightly more sophisticated: McCabe's cyclomatic complexity and the set of Halstead's Software Science metrics.

Except for McCabe's cyclomatic complexity, all the metrics are defined at the file level. So we measured all of them for all the C files (as identified by *SlocCount*), ignoring header files, which we identified by filename (all files ending in *.h*).

Because of the definition of McCabe's cyclomatic complexity, it must be measured over complete functions or programs because it is defined for entities that have one starting point and one or more exit points. We decided to run the formula over each function and to summarize the cyclomatic complexity of a whole file using two values: the maximum of all the functions included in the file and the average value over all the functions in the file.

Additionally, we also calculated the MD5 hash for every file, so we could discard repeated files from the statistical analysis, as including the same file more than once would introduce a bias in the results.

## Source Lines of Code (SLOC)

For this classic measure, we use the definition given by Conte [Conte 1986]:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

In the case of our sample file in Example 8-2, when we ignore blanks and comments but include preprocessor directives and all the rest of the lines, the file contains 23 SLOC.

## Lines of Code (LOC)

For this we measured the total number of lines in each source code file, including comments, blank lines, etc., using the Unix *wc* utility.

This is straightforward to measure because it counts blanks, comments, etc. Example 8-2 contains 32 LOC (plus 18 LOC of the license comment text, which was removed for clarity purposes).

## Number of C Functions

We counted the number of functions inside each file using the *exuberant-ctags* tool combined with *wc*.

This metric is even easier to measure. Example 8-2 contains only one function, so CFUNC is 1 for this file.

## McCabe's Cyclomatic Complexity

We use the definition given in the original paper by McCabe [McCabe 1976], which indicates the number of regions in a graph representing a source code file. Any program can be represented as a graph. The simplest element is a *flat* series of statements with no conditions, loops, or branches, which is represented by graph (a) in Figure 8-1. An `if` statement is a bifurcation, as shown in graph (b) in Figure 8-1. A loop would be shown through an edge that returns back to an earlier node.

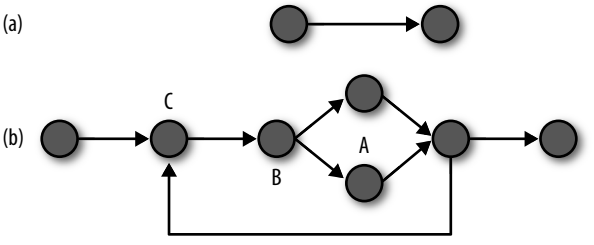


*FIGURE 8-1. Two sample graphs. Graph (a) has a CYCLO of 1, and graph (b) has a CYCLO of 3.*

For a graph $G$ with $n$ vertices, $e$ edges, and $p$ exit points (e.g., function returns in the case of C), the complexity $v$ is defined as follows:

$$v(G) = e - n + 2p$$

The minimum value for the cyclomatic complexity metric (CYCLO) is 1, corresponding to the flat series of statements with no bifurcations or loops. Every additional region in the control flow graph increases the CYCLO by one unit. For instance, a program that contains an `if` statement (with no `else`) has a CYCLO of 2, because it creates a new region in the control flow graph, in addition to the surrounding region.

The program shown in Example 8-2, whose control flow graph is shown in the (b) graph of Figure 8-1, has a CYCLO of 3. The `if` bifurcation creates one new region (A in Figure 8-1), and the `while` loop creates another (B in Figure 8-1). Finally, C is the surrounding region, which always counts as 1 in the cyclomatic complexity.

## Halstead's Software Science Metrics

For this we use the definition given by Kan [Kan 2003]. We measured four metrics: length, volume, level, and mental discriminations. These metrics are based on the redundancy of *operands* and *operators* in a program.

In the C language, operands are string constants and variable names. Operators are symbols (such as `+`, `-`, `++`, and `--`), the * indirection operator, the `sizeof` operator, preprocessor constants, control flow statements (such as `if` and `while`), storage class specifiers (such as `static` and `extern`), type specifiers (such as `int` and `char`), and structure specifiers (`struct` and `union`).

The metrics are obtained by counting the number of distinct operators $n_1$, the number of distinct operands $n_2$, along with the total number of operators $N_1$, and the total number of operands $N_2$. The length $L$ of a program is the total number of operators and operands:

$$L = N_1 + N_2$$

The volume $V$ of a program is its physical size, defined as:

$$V = N \cdot log_2(n_1 + n_2)$$

The level *lv* of a program is a parameter with an upper limit of 1; the closer it is to 1, the less complex is the program. The level is defined as the ratio between the volume of the program and its potential volume (the least redundant implementation of the algorithm):

$$lv = \frac{2}{n_1}\frac{n_2}{N_2}$$

The inverse of this metric is sometimes called the code's *difficulty*. The minimum *difficulty* is 1, and it increases without an upper bound.

The effort $E$ that a programmer needs to invest to comprehend a program is defined as:

$$E = \frac{V}{lv}$$

This metric is sometimes called the *number of mental discriminations* that a developer must do to understand a program.

Halstead obtained all these formulas by making an analogy between programming and natural languages. The main idea is that the implementation of an algorithm is an expression written in a language, and this expression will be easier to understand if it is shorter or contains more redundancy of operators and operands because the number of different concepts that a programmer must retain in memory at once will be smaller. For instance, Halstead's level is related to the redundancy of operands. If the redundancy is very high, with lots of repeated operands, the Halstead value will be lower, indicating a less complex program. This approach makes intuitive sense, because redundancy should help one learn and understand a program more quickly.

Halstead's Software Science metrics are similar to McCabe's cyclomatic complexity, being defined over whole, non-modular programs, without imports. But contrary to McCabe's metric, Halstead's metrics do not rely on the structure of the code to measure complexity. This is to say, Halstead's metrics are defined purely for the textual elements making up a program,

not for any kind of semantic unit such as functions, and so we apply Halstead's metrics to whole files.

The sample file shown in Example 8-2 has a Halstead's length of 97, which represents the total number of operators (string constants and variable names) and operands (symbols, statements, etc.). The Halstead's volume is 526, and the Halstead's level is 0.036. The number of mental discriminations is 14,490, the quotient between the volume and level. All those metrics indicate a greater complexity for higher values, except in the case of Halstead's level, where a lower value indicates a more complex program.

## Statistical Analysis

The ArchLinux repositories contained 4,069 packages (as of April 2010), with some of the packages being different versions of the same upstream project. After removing different versions, we obtained a sample of 4,015 packages, containing 1,272,748 source code files. Among all those files, 576,511 were written in C. However, there were some repeated files. In the overall sample, only 776,573 were unique files; in the C subsample, only 338,831 were unique files. From these unique C files, 212,167 were nonheader files and 126,664 were header files.

The same measurements shown in the previous section for the file included in Example 8-2 were repeated in our study for all the files written in C. Thus, we ended up with a set of more than 300,000 measurements. Each element of the set contained a tuple for each file containing the metrics (nine values in each tuple).

### Overall Analysis

The basic analysis on the sample correlated each of the nine metrics defined at the file level with the rest of the metrics. The goal is to extract a set of orthogonal metrics that can characterize software size and complexity. Our goal was to discover, from all the metrics we gathered, which ones do not provide any further information and therefore can be discarded.

For the analysis, we considered each file of the sample as an independent point, which is a fair assumption because we discarded all the repeated files, and because all files came from projects that can be considered independent. For the correlation, we decided to use the logarithm of the value of the metrics. The logarithm was chosen after calculating the ideal Box-Cox power transformation of the data. With the logarithms of all the metrics, we performed a linear regression and calculated the Pearson correlation coefficient. Pearson coefficients close to 1 correspond to highly correlated variables; if they are closer to 0, they indicate independent variables.*

* Wikipedia contains detailed information about correlation analysis and the Box-Cox power transformation: see *http://en.wikipedia.org/wiki/Correlation_coefficient* and *http://en.wikipedia.org/wiki/Box-Cox_transformation* (consulted as of April 3, 2010).

Table 8-2 shows the Pearson correlation coefficient among the logarithms of all the metrics. This matrix is symmetrical; that is to say, values above and below the diagonal are the same, because the correlation coefficient between two variables is the same, regardless of which variable depends on the other one. So we have removed half of the values from Table 8-2 for clarity's sake. For most of the metrics, the coefficients are very high, indicating a strong correlation between the pair of metrics. We have highlighted the higher correlation coefficients of LOC and SLOC with the rest of metrics; those values show that most of the complexity metrics are correlated with these two simple size metrics. A high correlation means that either of the metrics provides as much information about complexity as the other metric. For instance, Halstead's length and SLOC are correlated with a coefficient of 0.97, so it doesn't matter which one we use to determine a file's complexity.

*TABLE 8-2. Correlation coefficients among all the metrics*

| | SLOC | LOC | NFUNC | MCYCLO | ACYCLO | HLENG | HVOLU | HLEVE | HMD |
|---|---|---|---|---|---|---|---|---|---|
| SLOC | 1.00 | **0.97** | 0.68 | **0.77** | 0.63 | **0.97** | **0.97** | 0.88 | **0.96** |
| LOC | | 1.00 | 0.67 | **0.75** | 0.60 | **0.94** | **0.94** | 0.84 | **0.92** |
| NFUNC | | | 1.00 | 0.63 | 0.32 | 0.64 | 0.63 | 0.67 | 0.66 |
| MCYCLO | | | | 1.00 | 0.91 | 0.76 | 0.75 | 0.82 | 0.80 |
| ACYCLO | | | | | 1.00 | 0.63 | 0.62 | 0.72 | 0.68 |
| HLENG | | | | | | 1.00 | 0.99 | 0.90 | 0.99 |
| HVOLU | | | | | | | 1.00 | 0.90 | 0.98 |
| HLEVE | | | | | | | | 1.00 | 0.96 |
| HMD | | | | | | | | | 1.00 |

Some of the complexity metrics are not very well correlated. For instance, both the maximum McCabe's complexity (MCYCLO in Table 8-2) and the average McCabe's complexity (ACYCLO) are poorly correlated with SLOC and LOC. Can we find a reason for this?

MCYCLO and ACYCLO are related to the structure of the code: the more bifurcations and loops, the higher their values will be. This produces a possible source of inconsistencies between McCabe's metrics and lines of code: the difference between header and nonheader files.

Our analysis studied both sets of files. In C, header files include mostly specifications about the different functions and entities. But they may also contain conditional preprocessor directives, which are resolved at compilation time, not at runtime. Although conditional preprocessor directives are bifurcations of a sort, our tools do not consider them to be new regions in the control flow graph of the program. Therefore, header files will probably be quite flat in terms of McCabe's cyclomatic complexity, regardless of their size in lines of code. Consequently, to find out whether there is indeed a correlation between size and cyclomatic complexity, we should remove header files from the sample.

## Differences Between Header and Nonheader Files

If we divide the sample into two subsamples, one corresponding to header files and the other one to non-header files, the values of the correlation coefficients change.

We include here only a subset of the metrics. The rest of the metrics presented very high correlation coefficients both in the original overall analysis and in the two subsamples. So, for more clarity, we have removed those metrics from the results tables.

Table 8-3 shows the results for header files. We have highlighted two correlation coefficients, showing the low dependence between cyclomatic complexity and SLOC for header files. The other complexity metric, Halstead's level (HLEVE in the table), is highly correlated with SLOC. This result is expected because header files are flat, regardless of their size. Therefore, the cyclomatic complexity of these files will always be low, and the correlation with size will be poor. For instance, for the maximum cyclomatic complexity (MCYCLO), the mean value over all the files is 2, and the median is 1, indicating that most of the files have a MCYCLO of 2 or less. However, the independence between size and cyclomatic complexity does not indicate that we can use the cyclomatic complexity of a header file to measure its complexity. Moreover, interestingly, all other complexity metrics are highly correlated with size.

*TABLE 8-3. Correlation coefficients for header files*

| | SLOC | MCYCLO | ACYCLO | HLEVE |
|---|---|---|---|---|
| SLOC | 1.00 | **0.37** | **0.34** | 0.72 |
| MCYCLO | | 1.00 | 0.97 | 0.49 |
| ACYCLO | | | 1.00 | 0.46 |
| HLEVE | | | | 1.00 |

The results for nonheader files are shown in Table 8-4. In this case, all correlation coefficients are high. The only metric that is not so highly correlated with size is ACYCLO, the average cyclomatic complexity. However, it is highly correlated with the maximum cyclomatic complexity, MCYCLO, which is highly correlated with size itself. Halstead's level (HLEVE) is also highly correlated with size for non-header files.

*TABLE 8-4. Correlation coefficients for non-header files*

| | SLOC | MCYCLO | ACYCLO | HLEVE |
|---|---|---|---|---|
| SLOC | 1.00 | 0.83 | 0.60 | 0.91 |
| MCYCLO | | 1.00 | 0.86 | 0.82 |
| ACYCLO | | | 1.00 | 0.65 |
| HLEVE | | | | 1.00 |

What results, then, can we draw from the high correlations between metrics in our study? Are lines of code just as good as the fancy metrics defined by McCabe and Halstead? Let's look at some of the literature on software metrics before jumping to conclusions.

## The Confounding Effect: Influence of File Size in the Intensity of Correlation

The empirical validation of metrics with samples of source code is a popular topic in the research community. Most of these studies are similar to the one presented in this chapter. [El Emam et al. 2001] includes a detailed review of many of these validation works, and raises a concern about the validity of this kind of statistical study. In particular, the authors show how class size threatens the validity of object-oriented metrics for fault prediction; when repeating the same validation studies and controlling for size, some of the conclusions do not hold.

Although the methodology cannot be directly applied to this case, because we are dealing with files and not classes, and the rationale behind the metrics is also different, we should consider the risk that our metrics are affected by file size. To test for this confounding effect, we broke down all the results shown here by file size, to test how the correlation was affected by the size of files, and find out whether some of the conclusions could not be validated for all the size ranges.

### Effects of size on correlations for header files

One of the results previously shown is that McCabe's cyclomatic complexity is very poorly correlated with size, and we suggested that this metric should not be used with header files, because of their ambiguous relationship to bifurcation. Figure 8-2 shows the variation of the value of the correlation coefficient of cyclomatic complexity (vertical axis) versus SLOC (horizontal axis) when file size changes. It was obtained by dividing the files into 20 intervals, each one containing the same amount of files, and calculating the correlation coefficient using only those files. The horizontal axis is in logarithmic scale because of the wide range of file sizes. There are two curves, one corresponding to the maximum cyclomatic complexity of the file (MCYCLO) and the other to the average cyclomatic complexity (ACYCLO).

At first glance, there is no correlation at all for very small files (a correlation coefficient close to zero), and the correlation coefficient grows until it stabilizes at approximately 500 SLOC. The coefficient does not change much for very large files.

We can extract two conclusions from Figure 8-2. First of all, there are huge files that should probably be removed from the sample. However, those files affect the value of the correlation coefficient only slightly, as evidenced by the small increase in that part of the plot. The second conclusion is that there is indeed an influence of file size on the value of the correlation. That influence does not affect our results for header files, because the correlation coefficient remains very low regardless of file size. But how will the rest of correlations be affected by this issue?

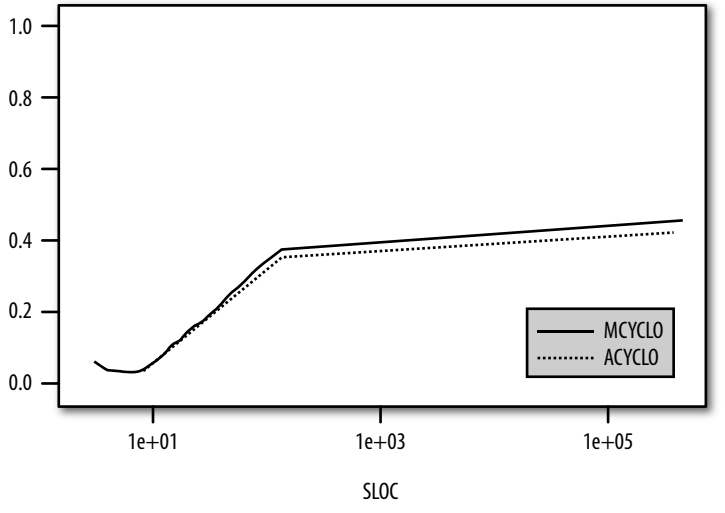


*FIGURE 8-2. Effect of file size on the correlation between cyclomatic complexity and SLOC in header files*

### Effects of size on correlations for nonheader files

Figure 8-3 shows the influence of file size on the correlation coefficient between cyclomatic complexity and SLOC for the case of nonheader files. The vertical axis shows the value of the correlation coefficient, and the horizontal axis, in logarithmic scale, shows the value of the file size in SLOC. There are two curves: the maximum cyclomatic complexity (MCYCLO) and the average cyclomatic complexity (ACYCLO).

Again, we can observe that file size does influence the value of the correlation coefficient. However, the values stabilize beyond a certain file size (around 1,000 SLOC for MCYCLO and 500 SLOC for ACYCLO). For very small files, low correlation coefficients are reasonable, because the variability of files at that end is very high. For medium file sizes, the values of the correlation coefficients are similar to those shown in previous sections.

### Effect on the Halstead's Software Science metrics

Figure 8-4 shows the influence of file size on the correlation coefficient of Halstead's metrics versus SLOC. Again, the vertical axis shows the correlation coefficient, the horizontal axis shows file size in logarithmic scale, and there is a curve for each one of the metrics (four in total).

The pattern is similar to the previous cases, albeit even more uniform across all the metrics. Even though file size influences the value of the correlation coefficient, all the coefficients remain high, regardless of file size.

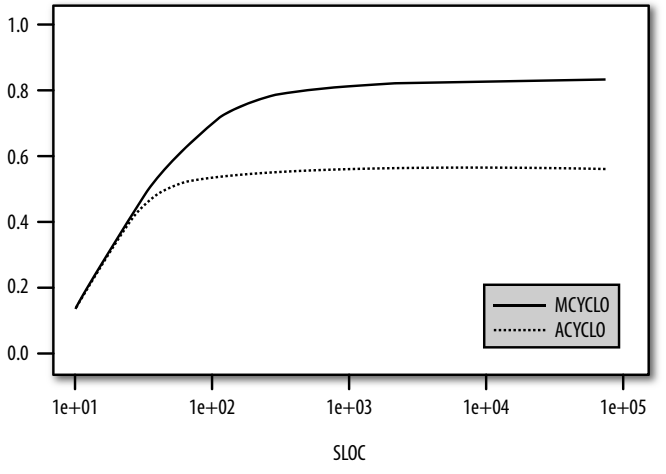


*FIGURE 8-3. Effect of file size on the correlation between cyclomatic complexity and SLOC in nonheader files*

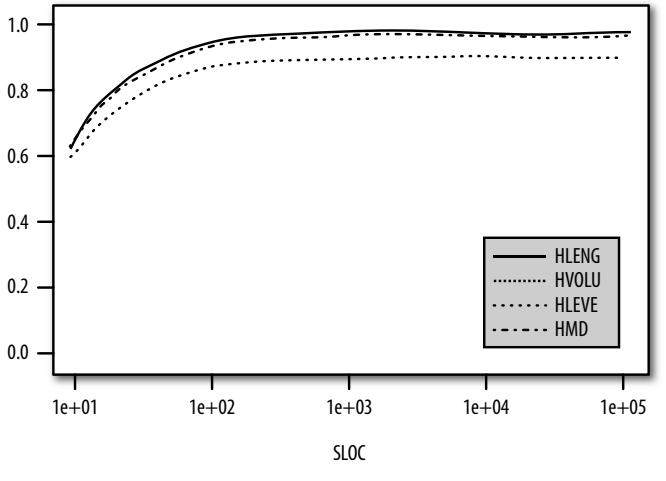


*FIGURE 8-4. Effect of file size on the correlation between Halstead's Software Science metrics and SLOC in non-header files*

### Summary of the confounding effect of file size

Although file size certainly influences the values of correlations, this influence does not substantially change the conclusions extracted from those correlations in any case. And indeed, the argument about the confounding effect of class size on the correlation of object-oriented metrics has faced some criticism [Evanco 2003].

In the case of our sample, we should probably remove very large files from the analysis because those probably introduce bias in the results. We randomly inspected some of these files. Some seem to have an automatic origin, and therefore should not be taken into account for an analysis of the complexity of source code. That is, no programmer has to read and edit those files, because they are automatically derived from the source files written by programmers. In any case, this bias is not significant, as shown in the previous figures.

As some practical advice, when applying the methodology shown in this chapter to other projects, we recommend that researchers break down the analysis by file size, to test whether the correlation coefficient changes with the size of files.

## Some Comments on the Statistical Methodology

Statistical analyses are subject to threats to the validity of the conclusions extracted from the data. Returning to the exit poll example given in the introduction to this chapter, the socioeconomic profile of the interviewees might bias the results of the exit poll. A robust exit poll requires a random sample to be selected; otherwise, the poll will not accurately predict the actual results of the elections. Along the same lines, the results shown in this chapter might be biased because of some threats to validity, and the conclusions might not hold for other software projects. For the sake of the completeness of our analysis, we discuss the threats to validity:

- The first problem you might have spotted is regarding the level of significance of the correlations shown in the previous sections. Because of the statistical properties and the size of the samples (in the range of hundred of thousands), this level of significance will always be very high (in the order of 99.99%).
- From a software development point of view, this study should be extended to other programming languages. The conclusions shown here for C might not hold for other languages. Although C is currently the most popular language in the open source community (in terms of available code), other languages are also very popular, growing much faster than C, and with vast amounts of code available.
- Finally, all the source code used for this study was released as open source. Although we believe there are no technical differences between open source and other forms of software releases, the selection of the sample might affect the validity of the conclusions. In particular, the conclusions might not hold for software developed under different conditions. To solve this threat to validity, this study should be repeated with code obtained from other domains (e.g., industry, scientific software).

## So Do We Need More Complexity Metrics?

The results shown in this chapter suggest that for non-header files written in C language, all the complexity metrics are highly correlated with lines of code, and therefore the more complex metrics provide no further information that could not be measured simply with lines of code.

However, these results must be accepted with some caution. Header files show poor correlation between cyclomatic complexity and the rest of metrics. We argue that this is because of the nature of this kind of file. In other words, header files do not contain implementations, only specifications. We are trying to measure the complexity of source code in terms of program comprehension. Programmers must of course read and comprehend header files, which means that header files can contribute to complexity to a certain extent. However, even though cyclomatic complexity is poorly correlated with lines of code in this case, that does not mean that it is a good complexity metric for header files. On the contrary, the poor correlation is due only to the lack of control structures in header files. These files do not contain loops, bifurcations, etc., so their cyclomatic complexity will always be minimal, regardless of their size.

For nonheader files, all the metrics show a high degree of correlation with lines of code. We accounted for the confounding effect of size, showing that the high correlation coefficients remain for different size ranges.

In our opinion, there is a clear lesson from this study: syntactic complexity metrics cannot capture the whole picture of software complexity. Complexity metrics that are exclusively based on the structure of the program or the properties of the text (for example, redundancy, as Halstead's metrics do), do not provide information on the amount of effort that is needed to comprehend a piece of code—or, at least, no more information than lines of code do. This has implications for how these metrics are used. In particular, defect prediction, development and maintenance effort models, and statistical models in general cannot benefit from these metrics, and lines of code should be considered always as the first and only metric for these models.

The problem of code complexity versus comprehension complexity has been faced in the research community before. In particular, a *semantic entropy* metric has been proposed, based on how obscure the identifiers used in a program are (for instance, names of variables). Interestingly, those kind of measurements are good defect predictors [Etzkorn et al. 2002].

This does not mean there are no useful lessons to take from traditional complexity metrics. First, cyclomatic complexity is a great indicator for the amount of paths that need to be tested in a program. Halstead's Software Science metrics also provide an interesting lesson: there are always several ways of doing the same thing in a program. So if you choose one way and use it in many parts of the program, you'll make your code more redundant, in turn making it more readable and less complex—in spite of what other statistics might say.

## References

[Conte 1986] S. D. Conte. 1986. *Software Engineering Metrics and Models*. San Francisco: Benjamin Cummings.

[El Emam et al. 2001] El Emam, K., S. Benlarbi, N. Goel, and S. Rai. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 27(7): 630–650.

[Etzkorn et al. 2002] Etzkorn, L. H., S. Gholston, and . W. E. Hughes, Jr. 2002. A semantic entropy metric. *Journal of Software Maintenance and Evolution: Research and Practice* 14(4), 293–310.

[Evanco 2003] Evanco, W. 2003. Comments on "The confounding effect of class size on the validity of object-oriented metrics". *IEEE Transactions on Software Engineering* 29(7): 670–672.

[Kan 2003] Kan, S. H. 2003. *Metrics and Models in Software Quality Engineering* (2nd Edition). Boston: Addison-Wesley Professional.

[McCabe 1976] McCabe, T.J. 1976. A complexity measure. *IEEE Transactions on Software Engineering* SE-2(4): 308–320.