CHAPTER EIGHT

Quality Wars: Open Source vs. Proprietary Software

Diomidis Spinellis

Talk is cheap. Show me the code.

—Linus Torvalds

When developers compare open source with proprietary software, what should be a civilized debate often degenerates into a flame war. This need not be so, because there is plenty of room for a cool-headed objective comparison.

Researchers examine the efficacy of open source development processes through various complementary approaches.

- One method involves looking at the quality of the code, its *internal quality attributes*, such as the density of comments or the use of global variables [Stamelos et al. 2002].
- Another approach involves examining the software's *external quality attributes*, which reflect how the software appears to its end users [Kuan 2003].
- Then, instead of the product one can look at the *process*: examine measures related to the code's construction and maintenance, such as the how much code is being added each week or how swiftly bugs are closed [Paulson et al. 2004].
- Another approach involves discussing specific scenarios. For instance, Hoepman and Jacobs [Hoepman and Jacobs 2007] examine the security of open source software by looking at how leaked source code from Windows NT and Diebold voting machines led to

attacks and how open source practices lead to cleaner code and allow the use of security-verification tools.

- Finally, a number of arguments are based on plain hand waving: more than a decade ago Bob Glass [Glass 1999] identified this trend in the hype associated with the emergence of Linux in the IT industry.

Although many researchers over the years have examined open source artifacts and processes [Fitzgerald and Feller 2002], [Spinellis and Szyperski 2004], [Feller 2005], [Feller et al. 2005], [von Krogh and von Hippel 2006], [Capiluppi and Robles 2007], [Sowe et al. 2007], [Stol et al. 2009], the direct comparison of open source systems with corresponding proprietary products has remained an elusive goal. The reason for this is that it used to be difficult to find a proprietary product comparable to an open source equivalent, and then convince the proprietary product's owner to provide its source code for an objective comparison. However, the open-sourcing of Sun's Solaris kernel and the distribution of large parts of the Windows kernel source code to research institutions provided me with a window of opportunity to perform a comparative evaluation between the open source code and the code of systems developed as proprietary software.

Here I report on code quality metrics (measures) I collected from four large industrial-scale operating systems: FreeBSD, Linux, OpenSolaris, and the Windows Research Kernel (WRK). This chapter is not a crime mystery, so I'm revealing my main finding right up front: there are no significant across-the-board code quality differences between these four systems. Now that you know the ending, let me suggest that you keep on reading, because in the following sections you'll find not only how I arrived at this finding, but also numerous code quality metrics for objectively evaluating software written in C, which you can also apply to your code. Although some of these metrics have not been empirically validated, they are based on generally accepted coding guidelines, and therefore represent the rough consensus of developers concerning desirable code attributes. I first reported these findings at the 2008 International Conference of Software Engineering [Spinellis 2008]; this chapter contains many additional details.

Past Skirmishes

The very ink with which all history is written is merely fluid prejudice.

—Mark Twain

Researchers have been studying the quality attributes of operating system code for more than two decades [Henry and Kafura 1981], [Yu et al. 2004]. Particularly close to the work you're reading here are comparative studies of open source operating systems [Yu et al. 2006], [Izurieta and Bieman 2006], and studies comparing open and closed source systems [Stamelos et al. 2002], [Paulson et al. 2004], [Samoladas et al. 2004].

A comparison of maintainability attributes between the Linux and various Berkeley Software Distribution (BSD) operating systems found that Linux contained more instances of module communication through global variables (known as *common coupling*) than the BSD variants. The results I report here corroborate this finding for file-scoped identifiers, but not for global identifiers (see Figure 8-11). Furthermore, an evaluation of growth dynamics of the FreeBSD and Linux operating systems found that both grow at a linear rate, and that claims of open source systems growing at a faster rate than commercial systems are unfounded [Izurieta and Bieman 2006].

A study by Paulson and his colleagues [Paulson et al. 2004] compares evolutionary patterns between three open source projects (Linux, GCC, and Apache) and three non-disclosed commercial ones. They found a faster rate of bug fixing and feature addition in the open source projects, which is something we would expect for very popular projects like those they examine. In another study focusing on the quality of the code (its internal quality attributes) [Stamelos et al. 2002] the authors used a commercial tool to evaluate 100 open source applications using metrics similar to those reported here, but measured on a scale ranging from *accept* to *rewrite*. They then compared the results against benchmarks supplied by the tool's vendor for commercial projects. The authors found that only half of the modules they examined would be considered acceptable by software organizations applying programming standards based on software metrics. A related study by the same group [Samoladas et al. 2004] examined the evolution of a measure called *maintainability index* [Coleman et al. 1994] between an open source application and its (semi)proprietary forks. They concluded that all projects suffered from a similar deterioration of the maintainability index over time.

The Battlefield

> **You cannot choose your battlefield,**
> **God does that for you;**
> **But you can plant a standard**
> **Where a standard never flew.**
>
> *—Nathalia Crane*

Figure 8-1 shows the history and genealogy of the systems I examine. * All four systems started their independent life in 1991–1993. At that time affordable microprocessor-based computers that supported a 32-bit address space and memory management led to the Cambrian explosion for modern operating systems. Two of the systems, FreeBSD and OpenSolaris, share common ancestry that goes back to the 1978 1BSD version of Unix. FreeBSD is based on BSD/Net2: a distribution of the Berkeley Unix source code that was purged from proprietary AT&T code. Consequently, while both FreeBSD and OpenSolaris contain code written at Berkeley, only OpenSolaris contains AT&T code. Specifically, the code behind OpenSolaris traces its origins

* If you think that the arrows point the wrong way round, you're in good company. Nevertheless, take some time too look them up in your favourite UML reference.

back to the 1973 version of Unix, which was the first written in C [Salus 1994]. In 2005 Sun released most of the Solaris source code under an open-source license.

Linux was developed from scratch in an effort to build a more feature-rich version of Tanenbaum's teaching-oriented, POSIX-compatible Minix operating system [Tanenbaum 1987]. Thus, although Linux borrowed ideas from both Minix and Unix, it did not derive from their code [Torvalds and Diamond 2001].

The intellectual roots of Windows NT go back to DEC's VMS through the common involvement of the lead engineer David Cutler in both projects. Windows NT was developed as Microsoft's answer to Unix, initially as an alternative of IBM's OS/2, and later as a replacement of the 16-bit Windows code base. The Windows Research Kernel (WRK) whose code I examine in this chapter includes major portions of the 64-bit Windows kernel, which Microsoft distributes for research use [Polze and Probert 2006]. The kernel is written in C with some small extensions. Excluded from the kernel code are the device drivers, and the plug-and-play, power management and virtual DOS subsystems. The missing parts explain the large size difference between the WRK and the other three kernels.

Although all four systems I examine are available in source code form, their development methodologies are markedly different. Microsoft and Sun engineers built Windows NT and Solaris within their companies as proprietary systems with minimal if any involvememnt of outsiders in the development process. (OpenSolaris has a very short life as an open-source project, and therefore only minimal code could have been contributed by developers outside Sun in the snapshot I examined.) Furthermore, Solaris has been developed with emphasis on a formal process [Dickinson 1996], while the development of Windows NT employed more lightweight methods [Cusumano and Selby 1995]. FreeBSD and Linux are both developed using open source development methods [Feller and Fitzgerald 2001], but their development processes are also dissimilar. FreeBSD is mainly developed by a non-hierarchical group of about 220 committers who have access to a shared software repository that was initially CVS and currently Subversion [Jrgensen 2001]. In contrast, Linux's developers are organized in a four tier pyramid. At the bottom two levels thousands of developers contribute patches to about 560 subsystem maintainers. At the top of the pyramid Linus Torvalds, assisted by a group of trusted lieutenants, is the sole person responsible for adding the patches to the Linux tree [Rigby and German 2006]. Nowadays, Linux developers coordinate their code changes through git, a purpose-built distributed version control system.

I calculated most of the metrics reported here by issuing SQL queries on a relational database containing the code elements comprising each system: modules, identifiers, tokens, functions, files, comments, and their relationships. The database's schema appears in Figure 8-2. † I constructed the database for each system by running the CScout refactoring browser for C code [Spinellis 2003], [Spinellis 2010] on a number of processor-specific configurations of each

† The databases (141 million records) and all the corresponding queries are available online at *http://www.spinellis.gr/sw/4kernel/*.

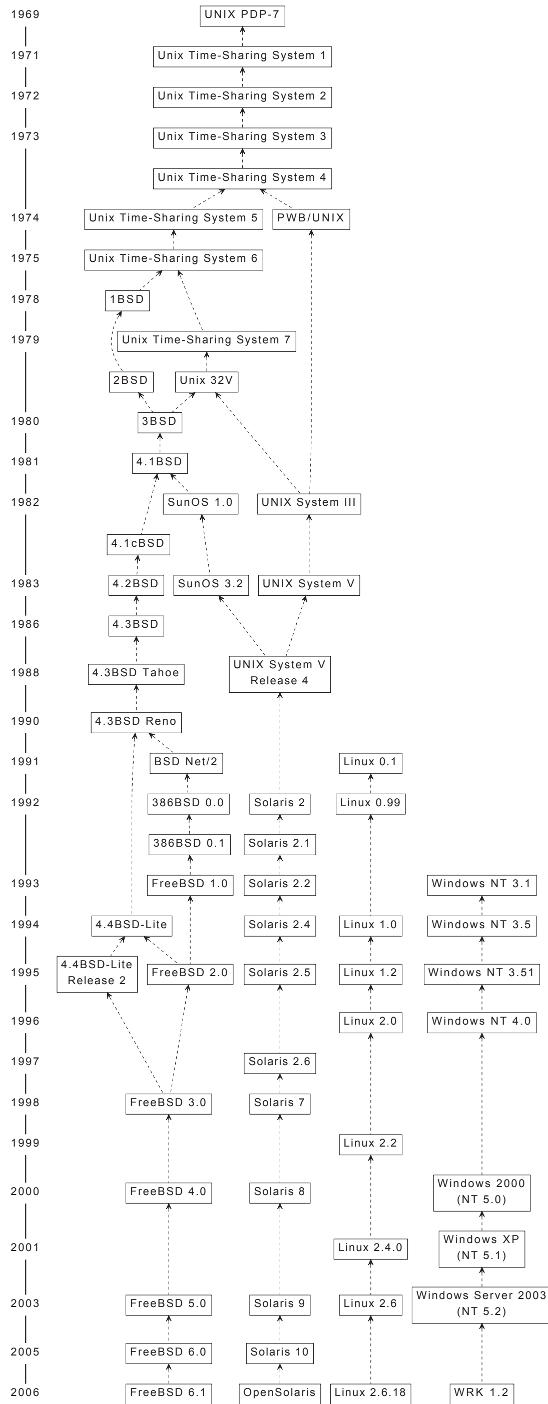

FIGURE 8-1. History and geneology of the four systems

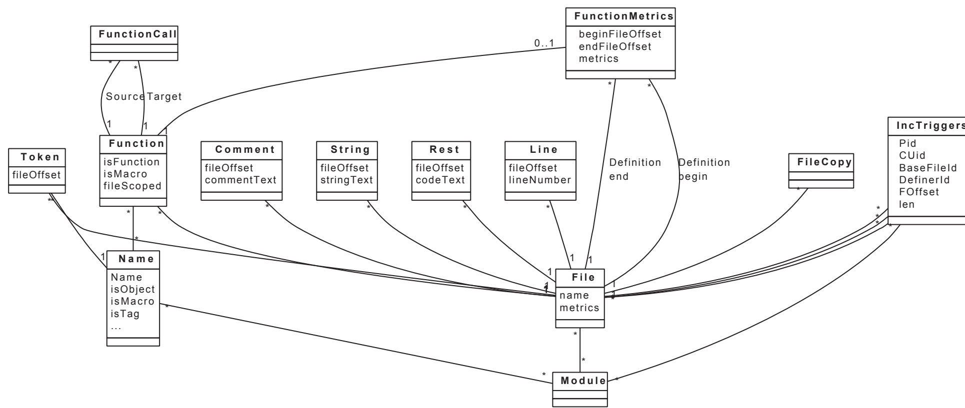

FIGURE 8-2. Schema of the database containing the code's analysis

operating system. (A processor-specific configuration comprises different macro definitions and files, and will therefore process code in a different way.) To process the source code of a complete system CScout must be given a configuration file that will specify the precise environment used for processing each compilation unit (C file). For the FreeBSD and the Linux kernels I constructed this configuration file by instrumenting proxies for the GNU C compiler, the linker, and some shell commands. These recorded their arguments (mainly the include file path and macro definitions) in a format that could then be used to construct a CScout configuration file. For OpenSolaris and the WRK I simply performed a full build for the configurations I investigated, recorded the commands that were executed in a log file, and then processed the compilation and linking commands appearing in the build's log.

In order to limit bias introduced in the selection of metrics, I chose and defined the metrics I would collect before setting up the mechanisms to measure them. This helped me avoid the biased selection of metrics based on results I obtained along the way. However, this *ex ante* selection also resulted in many metrics—like the number of characters per line—that did not supply any interesting information, failing to provide a clear winner or loser. On the other hand my selection of metrics was not completely blind, because at the time I designed the experiment I was already familiar with the source code of the FreeBSD kernel and had seen source code from Linux, the 9th Research Edition Unix, and some Windows device drivers.

Other methodological limitations of this study are the small number of (admittedly large and important) systems studied, the language specificity of the employed metrics, and the coverage of only maintainability and portability from the space of all software quality attributes. This last limitation means that the study fails to take into account the large and important set of quality attributes that are typically determined at runtime: functionality, reliability, usability, and efficiency. However, these missing attributes often depend on factors that are beyond the control of the system's developers: configuration, tuning, and workload selection. Studying them would introduce additional subjective biases, such as configurations that were unsuitable

for some workloads or operating environments. The controversy surrounding studies comparing competing operating systems in areas like security or performance demonstrates the difficulty of such approaches.

The large size difference between the WRK source code and the other systems, is not as problematic as it may initially appear. An earlier study on the distribution of the maintainability index [Coleman et al. 1994] across various FreeBSD modules showed that their maintainability was evenly distributed, with few outliers at each end [Spinellis 2006]. This means that we can form an opinion about a large software system by looking at a small sample of it. Therefore, the WRK code I examine can be treated as a representative subset of the complete Windows operating system kernel.

TABLE 8-1. Key metrics of the four systems

Metric	FreeBSD	Linux	Solaris	WRK
Version	HEAD 2006-09-18	2.6.18.8-0.5	2007-08-28	1.2
Configuration	i386 AMD64 SPARC64	AMD64	Sun4v Sun4u SPARC	i386 AMD64
Lines (thousands)	2,599	4,150	3,000	829
Comments (thousands)	232	377	299	190
Statements (thousands)	948	1,772	1,042	192
Source files	4,479	8,372	3,851	653
Linked modules	1,224	1,563	561	3
C functions	38,371	86,245	39,966	4,820
Macro definitions	727,410	703,940	136,953	31,908

Into the Battle

No battle plan survives contact with the enemy.

—Helmuth von Moltke the Elder

The key properties of the systems I examine appear in Table 8-1. The quality metrics I collected can be roughly categorized into the areas of file organization, code structure, code style, preprocessing, and data organization. When it is easy to represent a metric with a single number, I list its values for each of the four systems in a table and on the left I indicate whether ideally that number should be high (↑), low (↓), or near a particular value (e.g. ≅ 1). In other cases we must look at the distribution of the various values, and for this I use so-called candlestick figures, like Figure 8-3. Each element in such a figure depicts five values:

- the minimum, at the bottom of the line,
- the lower (25%) quartile, at the bottom of the box,

- the median (the numeric value separating the higher half of the values from the lower half), as a horizontal line within the box,
- the upper (75%) quartile, at the top of the box,
- the maximum value, at the top of the line, and
- the arithmetic mean, as a diamond.

Minima and maxima lying outside the graph's range are indicated with a dashed line along with a figure of their actual value.

File Organization

TABLE 8-2. File organization metrics

Metric	Ideal	FreeBSD	Linux	Solaris	WRK
Files per directory	↓	6.8	20.4	8.9	15.9
Header files per C source file	≅ 1	1.05	1.96	1.09	1.92
Average structure complexity in files	↓	2.2×10^{14}	1.3×10^{13}	5.4×10^{12}	2.6×10^{13}

In the C programming language source code files play a significant role in structuring a system. A file forms a scope boundary, while the directory it is located may determine the search path for included header files [Harbison and Steele Jr. 1991]. Thus, the appropriate organization of definitions and declarations into files, and files into directories is a measure of the system's modularity [Parnas 1972].

Figure 8-3 shows the length of C and header files. Most files are less than 2000 lines long. Overly long files (such as the C files in OpenSolaris and the WRK) are often problematic, because they can be difficult to manage, they may create many dependencies, and they may violate modularity. Indeed the longest header file (WRK's *winerror.h*) at 27,000 lines lumps together error messages from 30 different areas, most of which are not related to the Windows kernel.

A related measure examines the contents of files, not in terms of lines, but in terms of defined entities. In C source files I count global functions. In header files an important entity is a structure; the closest abstraction to a class that is available in C. Figure 8-4 shows the number of global functions that are declared in each C file and the number of aggregates (structures or unions) that are defined in each header file. Ideally, both numbers should be small, indicating an appropriate separation of concerns. The C files of OpenSolaris and WRK come out worse than the other systems, while a significant number of WRK's header files look bad, because they define more than 10 structures each.

The four systems I've examined have interesting directory structures. As you can see in Figure 8-5 to Figure 8-8, three of the four systems have similarly wide structures. The small size and complexity of the WRK reflects the fact that Microsoft has excluded from it many large

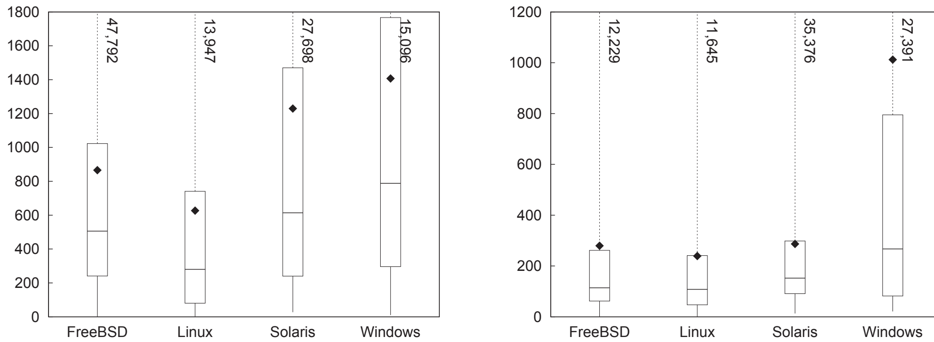

FIGURE 8-3. File length in lines of C files (left) and headers (right)

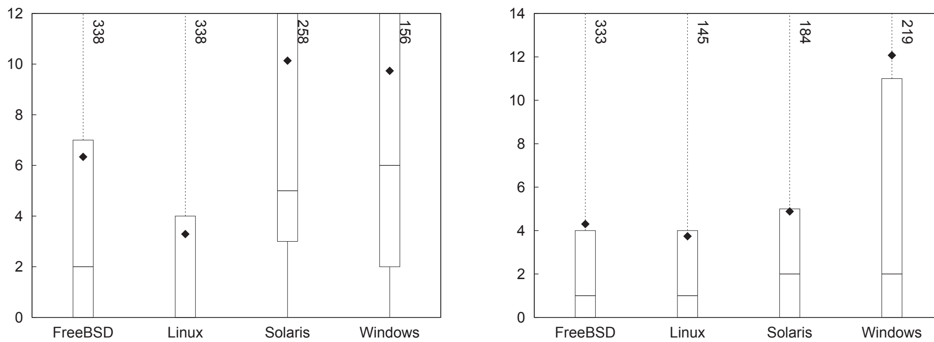

FIGURE 8-4. Defined global functions (left) and structures (right)

parts of the Windows kernel. We see that the directories in Linux are relatively evenly distributed across the whole source code tree, whereas in FreeBSD and OpenSolaris some directories lump together in clusters. This can be the result of organic growth over a longer period of time, because both systems have twenty more years of history on their back (Figure 8-1). The more even distribution of Linux's directories may also reflect the decentralized nature of its development.

At a higher level of granularity, I examine the number of files located in a single directory. Again, putting many files in a directory is like having many elements in a module. A large number of files can confuse developers, who often search through these files as a group with tools like *grep*, and lead to accidental identifier collisions through shared header files. The

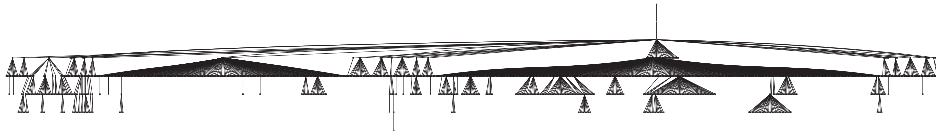

FIGURE 8-5. The directory structure of FreeBSD

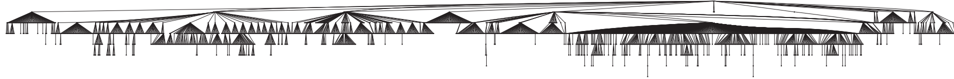

FIGURE 8-6. The directory structure of Linux

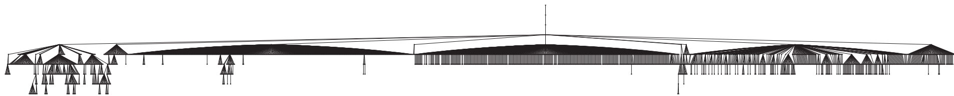

FIGURE 8-7. The directory structure of OpenSolaris

FIGURE 8-8. The directory structure of the Windows Research Kernel

numbers I found in the examined systems can be seen in Table 8-2, and show Linux lagging the other systems.

The next line in the table describes the ratio between header files and C files. I used the following SQL query to calculate these numbers.

```
select (select count(*) from FILES where name like '%.c') /
 (select count(*) from FILES where name like '%.h')
```

A common style guideline for C code involves putting each module's interface in a separate header file and its implementation in a corresponding C file. Thus a ratio of header to C files around 1 is the optimum; numbers significantly diverging from one may indicate an unclear

distinction between interface and implementation. This can be acceptable for a small system (the ratio in the implementation of the *awk* programming language is 3/11), but will be a problem in a system consisting of thousands of files. All the systems score acceptably in this metric.

Finally, the last line in Table 8-2 provides a metric related to the complexity of file relationships. I study this by looking at files as nodes in a directed graph. I define a file's *fan-out* as the number of efferent (outgoing) references it makes to elements declared or defined in other files. For instance, a C file that uses the symbols `FILE`, `putc`, and `malloc` (defined outside the C file in the *stdio.h* and *stdlib.h* header files) has a fan-out of 3. Correspondingly, I define as a file's *fan-in* the number of afferent (incoming) references made by other files. Thus, in the previous example, the fan-in of *stdio.h* would be 2. I used Henry and Kafura's information flow metric [Henry and Kafura 1981] to look at the corresponding relationships between files.

The value I report is

$$(\text{fanIn} \times \text{fanOut})^2$$

I calculated the value based on the contents of the CScout database table `INCTRIGGERS`, which stores data about symbols in each file that are linked with other files.

```
select avg(pow(fanout.c * fanin.c, 2)) from
 (select basefileid fid, count(definerid) c from
   (select distinct BASEFILEID, DEFINERID, FOFFSET from INCTRIGGERS) i2 group by basefileid) fanout
 inner join
 (select definerid fid, count(basefileid) c from
   (select distinct BASEFILEID, DEFINERID, FOFFSET from INCTRIGGERS) i2 group by definerid) fanin
 on fanout.fid = fanin.fid
```

The calculation works as follows. For the connections each file makes the innermost `select` statements derive a set of unique identifiers and the files they are defined or referenced. Then, the middle `select` statements count the number of identifiers per file, and the outermost `select` statement joins each file's definitions with its references and calculates the corresponding information flow metric. A large value for this metric has been associated with the occurrence of changes and structural flaws.

Code Structure

TABLE 8-3. Code structure metrics

Metric	Ideal	FreeBSD	Linux	Solaris	WRK
% global functions	↓	36.7	21.2	45.9	99.8
% strictly structured functions	↑	27.1	68.4	65.8	72.1

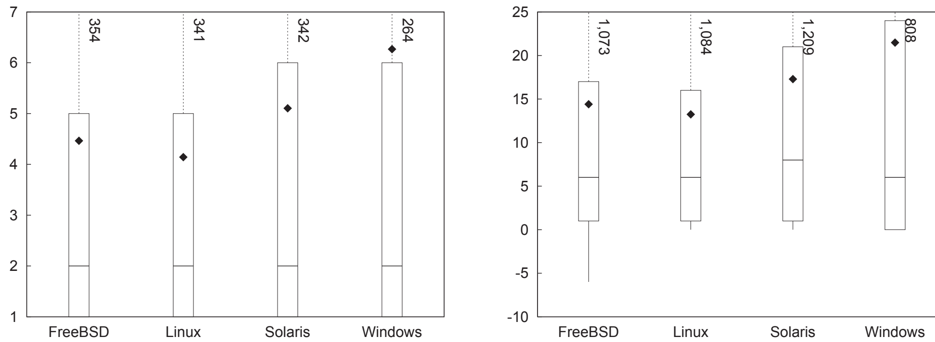

FIGURE 8-9. Extended cyclomatic complexity (left) and number of statements per function (right)

Metric	**Ideal**	**FreeBSD**	**Linux**	**Solaris**	**WRK**
% labeled statements	↓	0.64	0.93	0.44	0.28
Average number of parameters to functions	↓	2.08	1.97	2.20	2.13
Average depth of maximum nesting	↓	0.86	0.88	1.06	1.16
Tokens per statement	↓	9.14	9.07	9.19	8.44
% of tokens in replicated code	↓	4.68	4.60	3.00	3.81
Average structure complexity in functions	↓	7.1×10^4	1.3×10^8	3.0×10^6	6.6×10^5

The code structures of the four systems illustrates how similar problems can be tackled through different control structures and separation of concerns. It also allows us to peer into the design of each system.

Figure 8-9 shows the distribution across functions of the extended cyclomatic complexity metric [McCabe 1976]. This is a measure of the number of independent paths contained in each function. The number shown takes into account Boolean and conditional evaluation operators (because these introduce additional paths), but not multi-way switch statements, because these would disproportionally affect the result for code that is typically cookie-cutter similar. The metric was designed to measure a program's testability, understandability, and maintainability [Gill and Kemerer 1991]. In this regard Linux scores better than the other systems and the WRK worse. The same figure also shows the number of C statements per function. Ideally, this should be a small number (e.g. around 20) allowing the function's complete code to fit on the developer's screen. Linux again scores better than the other systems.

In Figure 8-10 we can see the distribution of the Halstead volume complexity [Halstead 1977]. For a given piece of code this is based on four numbers.

n1

Number of distinct operators

n2

Number of distinct operands

N1

Total number of operators

N2

Total number of operands

Given these four numbers we calculate the program's so-called volume as

$$(N1 + N2) \times \log_2(n1 + n2)$$

For instance, for the expression

```
op = &(!x ? (!y ? upleft : (y == bottom ? lowleft : left)) :
(x == last ? (!y ? upright : (y == bottom ? lowright : right)) :
(!y ? upper : (y == bottom ? lower : normal))))[w->orientation];
```

the four variables have the following values.

n1

`= & () ! ?: == [] ->` (8)

n2

`bottom last left lower lowleft lowright normal op orientation right upleft upper upright w x y` (16)

N1

27

N2

24

The theory behind calculating the Halstead volume complexity number is that it should be low, reflecting code that doesn't require a lot of mental effort to comprehend. This metric, however, has often been criticized. As was the case with the cyclomatic complexity, Linux scores better and the WRK worse than the other systems.

Taking a step back to look at interactions between functions, Figure 8-11 depicts common coupling in functions by showing the percentage of the unique identifiers appearing in a function's body that come either from the scope of the compilation unit (file-scoped identifiers declared as `static`) or from the project scope (global objects). Both forms of coupling are undesirable, with the global identifiers considered worse than the file-scoped ones. Linux scores better than the other systems in the case of common coupling at the global scope, but

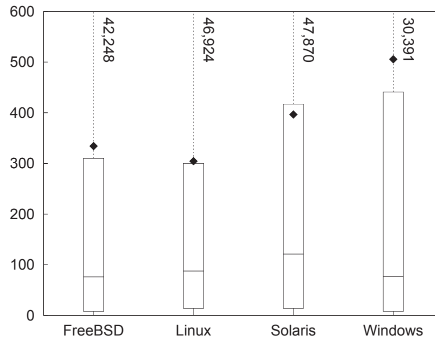

FIGURE 8-10. Halstead complexity per function

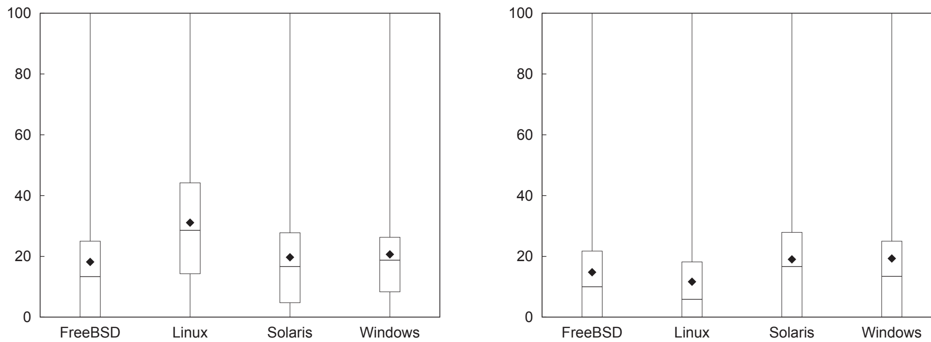

FIGURE 8-11. Common coupling at file (left) and global (right) scope

(probably because of this) scores worse than them when looking at the file scope. All other systems are more evenly balanced.

Other metrics associated with code structure appear in Table 8-3. The percentage of global functions indicates the functions visible throughout the system. The number of such functions in the WRK (nearly 100%; also verified by hand) is shockingly high. It may however reflect Microsoft's use of different techniques—such as linking into shared libraries (DLLs) with explicitly exported symbols—for avoiding identifier clashes.

Strictly structured functions are those following the rules of structured programming: a single point of exit and no `goto` statements. The simplicity of such functions makes it easier to reason about them. Their percentage is calculated by looking at the number of keywords within each function through the following SQL query.

```
select 100 -
(select count(*) from FUNCTIONMETRICS where nreturn > 1 or ngoto > 0) /
(select count(*) from FUNCTIONMETRICS) * 100
```

Along the same lines, the percentage of labeled statements indicates `goto` targets: a severe violation of structured programming principles. I measured labeled statements rather than `goto` statements, because many branch targets are a lot more confusing than many branch sources. Often multiple `goto` statements to a single label are used to exit from a function while performing some cleanup—the equivalent of an exception's `finally` clause.

The number of arguments to a function is an indicator of the interface's quality: when many arguments must be passed, packaging them into a single structure reduces clutter and opens up opportunities for optimization in style and performance.

Two metrics tracking the code's understandability are the average depth of maximum nesting and the number of tokens per statement. These metrics are based on the theories that both deeply nested structures and long statements are difficult to comprehend [Cant et al. 1995].

Replicated code has been associated with bugs [Li et al. 2006] and maintainability problems [Spinellis 2006]. The corresponding metric (% of tokens in replicated code) shows the percentage of the code's tokens that participate in at least one clone set. To obtain this metric I used the *CCFinderX*‡ tool to locate the duplicated code lines and a script (Example 8-1) to measure the ratio of such lines.

EXAMPLE 8-1. Determining the percentage of code duplication from the CCFinderX report

```
# Process CCFinderX results
open(IN, "ccfx.exe P $ARGV[0].ccfxd|") || die;
while (<IN>) {
    chop;
    if (/^source_files/ .. /^\}/) {
        # Process file definition lines like the following:
        # 611     /src/sys/amd64/pci/pci_bus.c        1041
        ($id, $name, $tok) = split;
        $file[$id][$tok - 1] = 0 if ($tok > 0);
        $nfile++;
    } elsif (/^clone_pairs/ .. /^\}/) {
        # Process pair identification lines like the following for files 14 and 363:
        # 1908    14.1753-1832    363.1909-1988
        ($id, $c1, $c2) = split;
        mapfile($c1);
        mapfile($c2);
```

‡ *http://www.ccfinder.net/*

```
    }
}

# Add up and report tokens and cloned tokens
for ($fid = 0; $fid <= $#file; $fid++) {
    for ($tokid = 0; $tokid <= $#{$file[$fid]}; $tokid++) {
        $ntok++;
        $nclone += $file[$fid][$tokid];
    }
}
print "$ARGV[0] nfiles=$nfile ntok=$ntok nclone=$nclone ", $nclone / $ntok * 100, "\n";

# Set the file's cloned lines to 1
sub mapfile
{
    my($clone) = @_;
    my ($fid, $start, $end) = ($clone =~ m/^(\d+)\.(\d+)\-(\d+)$/);
    for ($i = $start; $i <= $end; $i++) {
        $file[$id][$i] = 1;
    }
}
```

Finally, the average structure complexity in functions uses Henry and Kafura's information flow metric [Henry and Kafura 1981] again to look at the relationships between functions. Ideally we would want this number to be low, indicating an appropriate separation between suppliers and consumers of functionality.

Code Style

TABLE 8-4. Code style metrics

Metric	Ideal	FreeBSD	Linux	Solaris	WRK
% style conforming lines	↑	77.27	77.96	84.32	33.30
% style conforming typedef identifiers	↑	57.1	59.2	86.9	100.0
% style conforming aggregate tags	↑	0.0	0.0	20.7	98.2
Characters per line	↓	30.8	29.4	27.2	28.6
% of numeric constants in operands	↓	10.6	13.3	7.7	7.7
% unsafe function-like macros	↓	3.99	4.44	9.79	4.04
% misspelled comment words	↓	33.0	31.5	46.4	10.1
% unique misspelled comment words	↓	6.33	6.16	5.76	3.23

Various choices of indentation, spacing, identifier names, representations for constants, and naming conventions can distinguish sets of code that functionally do exactly the same thing. [Kernighan and Plauger 1978], [The FreeBSD Project 1995], [Cannon et al.], [Stallman et al. 2005]. In most sane cases, consistency is more important than the specific code style convention that was chosen.

For this study, I measured each system's consistency of style by applying the formatting program *indent*§ on the complete source code of each system, and counting the lines that *indent* modified. The result appears on the first line of Table 8-4. The behavior of *indent* can be modified using various options in order to match a formatting style's guidelines. For instance, one can specify the amount of indentation and the placement of braces. In order to determine each system's formatting style and use the appropriate formatting options, I first run *indent* on each system with various values of the 15 numerical flags, and turning on or off each one of the 55 Boolean flags (see Example 8-2 and Example 8-3). I then chose the set of flags that produced the largest number of conforming lines. For example, on the OpenSolaris source code *indent* with its default flags would reformat 74% of the lines. This number shrank to 16% once the appropriate flags were determined (*-i8 -bli0 -cbi0 -ci4 -ip0 -bad -bbb -br -brs -ce -nbbo -ncs -nlp -npcs*).

EXAMPLE 8-2. Determining a system's indent formatting options (Unix variants)

```
DIR=$1
NFILES=0
RNFILES=0

# Determine the files that are OK for indent
for f in `find $DIR -name '*.c'`
do
    # The error code is not always correct, so we have to grep for errors
    if indent -st $f 2>&1 >/dev/null | grep -q Error:
    then
        REJECTED="$REJECTED $f"
        RNFILES=`expr $RNFILES + 1`
        echo -n "Rejecting $f - number of lines: "
        wc -l <$f
    else
        FILES="$FILES $f"
        NFILES=`expr $NFILES + 1`
    fi
done

LINES=`echo $FILES | xargs cat | wc -l`
RLINES=`echo $REJECTED | xargs cat | wc -l`

# Format the files with the specified options
# Return the number of mismatched lines
try()
{
    for f in $FILES
    do
        indent -st $IOPT $1 $f |
        diff $f -
    done |
```

§ *http://www.gnu.org/software/indent/*

```
    grep '^<' |
    wc -l
}

# Report the results in a format suitable for further processing
status()
{
    echo "$IOPT: $VIOLATIONS violations in $LINES lines of $NFILES files ($RLINES of $RNFILES files not process
}

# Determine base case
VIOLATIONS=`try`
status
```

EXAMPLE 8-3. Determining a system's indent formatting options (Windows)

```
# Try various numerical options with values 0-8
for try_opt in i ts bli c cbi cd ci cli cp d di ip l lc pi
do
    BEST=$VIOLATIONS
    for n in 0 1 2 3 4 5 6 7 8
    do
        NEW=`try -$try_opt$n`
        if [ $NEW -lt $BEST ]
        then
            BNUM=$n
            BEST=$NEW
        fi
    done
    if [ $BEST -lt $VIOLATIONS ]
    then
        IOPT="$IOPT -$try_opt$BNUM"
        VIOLATIONS=$BEST
        status
    fi
done

# Try the various Boolean options
for try_opt in bad bap bbb bbo bc bl bls br brs bs cdb cdw ce cs bfda \
    bfde fc1 fca hnl lp lps nbad nbap nbbo nbc nbfda ncdb ncdw nce \
    ncs nfc1 nfca nhnl nip nlp npcs nprs npsl nsaf nsai nsaw nsc nsob \
    nss nut pcs prs psl saf sai saw sc sob ss ut
do
    NEW=`try -$try_opt`
    if [ $NEW -lt $VIOLATIONS ]
    then
        IOPT="$IOPT -$try_opt"
        VIOLATIONS=$NEW
    fi
    status
done
```

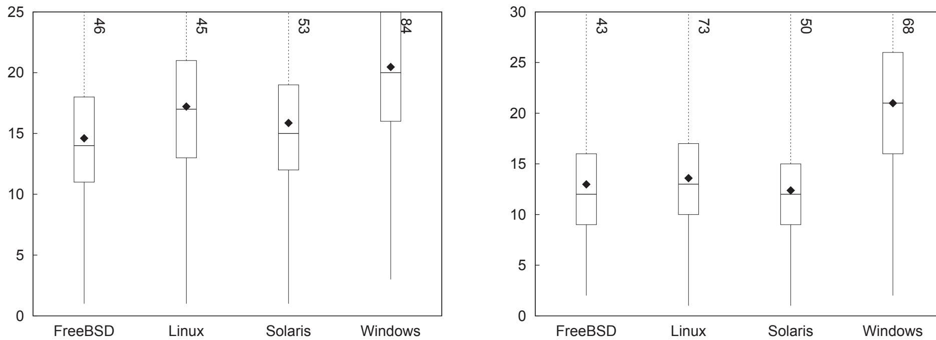

FIGURE 8-12. Length of global (left) and aggregate (right) identifiers

Figure 8-12 depicts the length distribution of two important classes of C identifiers: those of globally visible objects (variables and functions) and the tags used for identifying aggregates (structures and unions). Because each class typically uses a single name space, it is important to choose distinct and recognizable names (see chapter 31 of reference [McConnell 2004]). For these classes of identifiers, longer names are preferable, and the WRK excels in both cases, as anyone who has programmed using the Windows API could easily guess.

Some other metrics related to code style appear in Table 8-4. To measure consistency, I also determined through code inspection the convention used for naming `typedef`s and aggregate tags, and then counted the identifiers of those classes that did not match the convention. Here are the two SQL queries I ran, one on the Unix-like systems and the other on the WRK.

```
select 100 * (select count(*) from IDS where typedef and name like '%_t') /
  (select count(*) from IDS where typedef)

select 100 * (select count(*) from IDS where typedef and name regexp '^[A-Z0-9_]*$') /
  (select count(*) from IDS where typedef)
```

Three other metrics aimed at identifying programming practices that style guidelines typically discourage:

- Overly long lines of code (characters per line metric)
- The direct use of “magic” numbers in the code (% of numeric constants in operands),
- The definition of function-like macros that can misbehave when placed after an `if` statement (% unsafe function-like macros)‖

The following SQL query roughly calculates the percentage of unsafe function-like macros by looking for bodies of such macros that contain more than one statement, but no `do` keywords. The result represents a lower bound, because the query can miss other unsafe macros, such as those consisting of an `if` statement.

```
select 100.0 * (select count(*) from FUNCTIONMETRICS left join
  FUNCTIONS on functionid = id where defined and ismacro and ndo = 0 and nstmt > 1) /
(select count(*) from FUNCTIONS where defined and ismacro)
```

Another important element of style involves commenting. It is difficult to judge objectively the quality of code comments. Comments can be superfluous or even wrong. We can't programmatically judge quality on that level, but we can easily measure the comment density. So Figure 8-13 shows the the comment density in C files as the ratio of comment characters to statements. In header files I measured it as the ratio of defined elements that typically require an explanatory comment (enumerations, aggregates and their members, variable declarations, and function-like macros) to the number of comments. In both cases I excluded files with trivially little content. With remarkable consistency, the WRK scores better than the other systems in this regard and Linux worse. Interestingly, the mean value of comment density is a lot higher than the median, indicating that some files require substantially more commenting than others.

```
select nccomment / nstatement from FILES where name like '%.c' and nstatement > 0

select (nlcomment + nbcomment) / (naggregate + namember + nppfmacro + nppomacro + nenum +
  npfunction + nffunction + npvar + nfvar)
  from FILES
  where name like '%.h' and naggregate + namember + nppfmacro + nppomacro > 0 and nuline / nline < .2
```

I also measured the number of spelling errors in the comments as a proxy for their quality. For this I ran the text of the comments through the *aspell* spelling checker with a custom dictionary consisting of all the system's identifier and file names (see Example 8-4). The low number of errors in the WRK reflects the explicit spell-checking that according to accompanying documentation, was performed before the code was released.

EXAMPLE 8-4. Counting comment words and misspelings in a CScout database

```
# Create personal dictionary of correct words
# from identifier names appearing in the code
```

‖ Function-like macros containing more than one statement should have their body enclosed in a dummy `do ... while(0)` block in order to make them behave like a call to a real function.

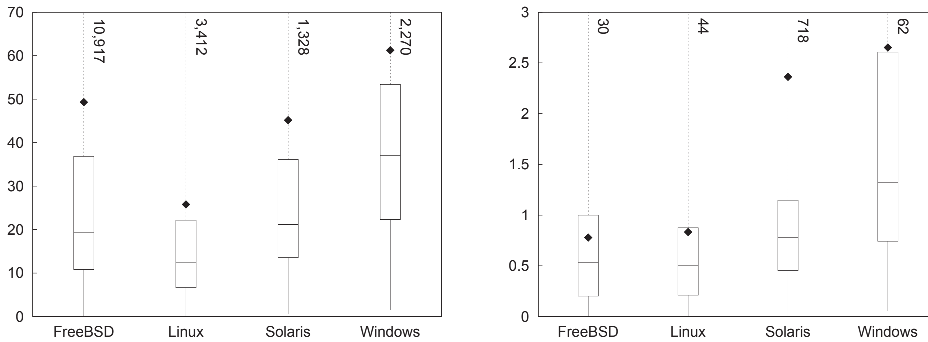

FIGURE 8-13. Comment density in C (left) and header (right) files

```
PERS=$1.en.pws
(
echo personal_ws-1.1 en 0
    (
    mysql -e 'select name from IDS union select name from FUNCTIONS union select name from FILES' $1 |
    tr /._ \\n |
    sed 's/\([a-z]\)\([A-Z]\)/\1\
\2/g'
    mysql -e 'select name from IDS union select name from FUNCTIONS union select name from FILES' $1 |
    tr /. \\n
    ) |
sort -u
) >$PERS

# Get comments from source code files and spell check them
mysql -e 'select comment from COMMENTS left join FILES on COMMENTS.FID = FILES.FID where not name like "%.cs
sed 's/\\[ntrb]//g' |
tee $1.comments |
aspell --lang=en --personal=$PERS -C --ignore=3 --ignore-case=true  --run-together-limit=10 list >$1.err
wc -w $1.comments    # Number of words
wc -l $1.err        # Number of errors
```

Although I did not measure portability objectively, the work involved in processing the source code with CScout allowed me to get a feeling of the portability of each system's source code between different compilers. The code of Linux and WRK appears to be the one most tightly bound to a specific compiler. Linux uses numerous language extensions provided by the GNU C compiler, sometimes including assembly code thinly disguised in what passes as C syntax in *gcc* (see Example 8-5). The WRK uses considerably fewer language extensions, but relies significantly on the `try` `catch` extension to C that the Microsoft compiler supports. The FreeBSD kernel uses only a few *gcc* extensions, and these are often isolated inside wrapping macros. The

OpenSolaris kernel was a welcome surprise: it was the only body of source code that did not require any extensions to CScout in order to compile.

EXAMPLE 8-5. The definition of `memmove` *in the Linux kernel*

```
void *memmove(void *dest, const void *src, size_t n)
{
        int d0, d1, d2;

        if (dest < src) {
                memcpy(dest,src,n);
        } else {
                __asm__ __volatile__(
                        "std\n\t"
                        "rep\n\t"
                        "movsb\n\t"
                        "cld"
                        : "=&c" (d0), "=&S" (d1), "=&D" (d2)
                        :"0" (n),
                         "1" (n-1+(const char *)src),
                         "2" (n-1+(char *)dest)
                        :"memory");
        }
        return dest;
}
```

Preprocessing

TABLE 8-5. Preprocessing metrics

Metric	Ideal	FreeBSD	Linux	Solaris	WRK
% of preprocessor directives in header files	↓	22.4	21.9	21.6	10.8
% of non-#include directives in C files	↓	2.2	1.9	1.2	1.7
% of preprocessor directives in functions	↓	1.56	0.85	0.75	1.07
% of preprocessor conditionals in functions	↓	0.68	0.38	0.34	0.48
% of function-like macros in defined functions	↓	26	20	25	64
% of macros in unique identifiers	↓	66	50	24	25
% of macros in identifiers	↓	32.5	26.7	22.0	27.1

The relationship between the C language proper and its (integral) preprocessor can at best be described as uneasy. Although C and real-life programs rely significantly on the preprocessor, its features often create portability, maintainability, and reliability problems. The preprocessor, as a powerful but blunt instrument, wrecks havoc with identifier scopes, the ability to parse and refactor unpreprocessed code, and the way code is compiled on different platforms. Thus most C programming guidelines recommend moderation in the use of preprocessor constructs.

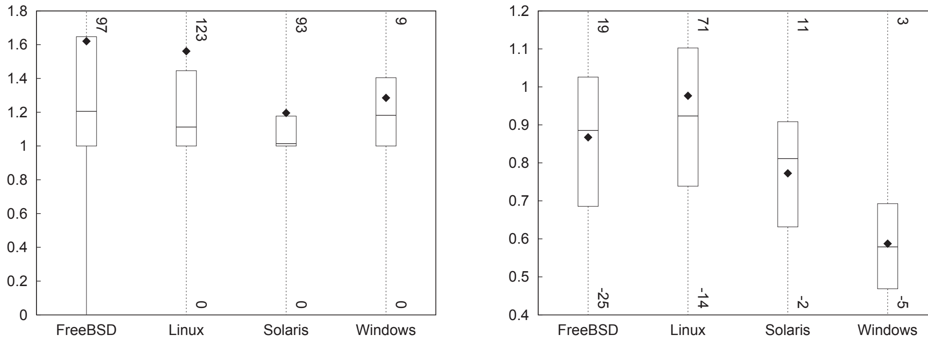

FIGURE 8-14. Preprocessing expansion in functions (left) and files (right)

Also for this reason modern languages based on C have tried to replace features provided by the C preprocessor with more disciplined alternatives. For instance, C++ provides constants and powerful templates as alternatives to the C macros, while C# provides preprocessor-like functionality only to aid conditional compilation and code generators.

The use of preprocessor features can be measured by the amount of expansion or contraction that occurs when the preprocessor runs over the code. Figure 8-14 contains two such measures: one for the body of functions (representing expansion of code), and one for elements outside the body of functions (representing data definitions and declarations). The two measurements were made by calculating the ratio of tokens arriving into the preprocessor to those coming out of it. Here is the SQL query I used for calculating the expansion of code inside functions.

```
select nctoken / npptoken from FUNCTIONS
inner join FUNCTIONMETRICS on id = functionid
where defined and not ismacro and npptoken > 0
```

Both expansion and contraction are worrisome: expansion signifies the occurrence of complex macros, while contraction is a sign of conditional compilation, which is also considered harmful [Spencer and Collyer 1992]. Therefore, the values of these metrics should hover around 1. In the case of functions OpenSolaris scores better than the other systems and FreeBSD worse, while in the case of files the WRK scores substantially worse than all other systems.

Four further metrics listed in Table 8-5 measure increasingly unsafe uses of the preprocessor:

- Directives in header files (often required)
- Non-`#include` directives in C files (rarely needed)

- Preprocessor directives in functions (of dubious value)
- Preprocessor conditionals in functions (a portability risk)

Preprocessor macros are typically used instead of variables (where we call these macros *object-like macros*) and functions (where we call them *function-like macros*). In modern C, object-like macros can often be replaced through enumeration members and function-like macros through inline functions. Both alternatives adhere to the scoping rules of C blocks and are therefore considerably safer than macros, whose scope typically spans a whole compilation unit. The last three metrics of preprocessor use in Table 8-5 measure the occurrence of function-like and object-like macros. Given the availability of viable alternatives and the dangers associated with macros, all should ideally have low values.

Data Organization

TABLE 8-6. Data organization metrics

Metric	Ideal	FreeBSD	Linux	Solaris	WRK
% of variable declarations with global scope	↓	0.36	0.19	1.02	1.86
% of variable operands with global scope	↓	3.3	0.5	1.3	2.3
% of identifiers with wrongly global scope	↓	0.28	0.17	1.51	3.53
% of variable declarations with file scope	↓	2.4	4.0	4.5	6.4
% of variable operands with file scope	↓	10.0	6.1	12.7	16.7
Variables per typedef or aggregate	↓	15.13	25.90	15.49	7.70
Data elements per aggregate or enumeration	↓	8.5	10.0	8.6	7.3

The final set of measurements concerns the organization of each kernel's (in-memory) data. A measure of the quality of this organization in C code can be determined by the scoping of identifiers and the use of structures.

In contrast to many modern languages, C provides few mechanisms for controlling namespace pollution. Functions can be defined in only two possible scopes (file and global), macros are visible throughout the compilation unit in which they are defined, and aggregate tags typically live all together in the global namespace. For the sake of maintainability, it's important for large-scale systems such as the four examined in this chapter to judiciously use the few mechanisms available to control the large number of identifiers that can clash.

Figure 8-15 shows the level of namespace pollution in C files by averaging the number of identifiers and macros that are visible at the start of each function. With roughly 10,000 identifiers visible on average at any given point across the systems I examine, it is obvious that namespace pollution is a problem in C code. Nevertheless, FreeBSD fares better than the other systems and the WRK worse.

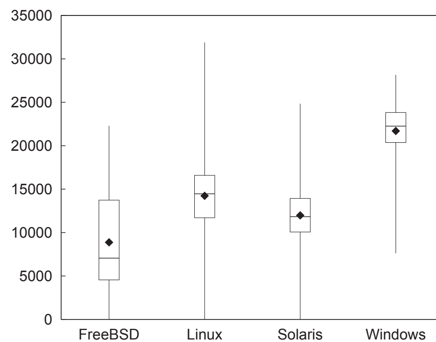

FIGURE 8-15. Average level of namespace pollution in C files

The first three measures in Table 8-6 examine how each system deals with its scarcest naming resource, global variable identifiers. One would like to minimize the number of variable declarations that take place at the global scope in order to minimize namespace pollution. Furthermore, minimizing the percentage of operands that refer to global variables reduces coupling and lessens the cognitive load on the reader of the code (global identifiers can be declared anywhere in the millions of lines comprising the system). The last metric concerning global objects counts identifiers that are declared as global, but could have been declared with a static scope, because they are accessed only within a single file. The corresponding SQL query calculates the percentage of identifiers with global (linkage unit) scope that exist only in a single file.

```
select 100.0 * (select count(*) from
  (select  TOKENS.eid from TOKENS
    left join IDS on TOKENS.eid = IDS.eid
    where ordinary and lscope group by eid having min(fid) = max(fid) ) static) /
  (select count(*) from IDS)
```

The next two metrics look at variable declarations and operands with file scope. These are more benign than global variables, but still worse than variables scoped at a block level.

The last two metrics concerning the organization of data provide a crude measure of the abstraction mechanisms used in the code. Type and aggregate definitions are the two main data abstraction mechanisms available to C programs. Therefore, counting the number of variable declarations that correspond to each type or aggregate definition provides an indication of how much these abstraction mechanisms have been employed.

```
select ((select count(*) from IDS where ordinary and not fun) /
  (select count(*) from IDS where suetag or typedef))

select ((select count(*) from IDS where sumember or enum)  /
  (select count(*) from IDS where suetag))
```

These statements measure the number of data elements per aggregate or enumeration in relation to data elements as a whole. This is similar to the relation that Chidamber and Kemerer's object-oriented weighted methods per class (WMC—[Chidamber and Kemerer 1994])) metric has to code. A high value could indicate that a structure tries to store too many disparate elements.

Outcome and Aftermath

There are two kinds of statistics, the kind you look up and the kind you make up.

—Archie Goodwin

Table 8-7 summarizes my results. I have marked cells where an operating system excels with a + and corresponding laggards with a –. For a number of reasons it would be a mistake to read too much from this table. First of all, the weights of the table's metrics are not calibrated according to their importance. In addition, it is far from clear that the metrics I used are functionally independent, and that they provide a complete or even representative picture of the quality of C code. Finally, I entered the +/– markings subjectively, trying to identify clear cases of differentiation in particular metrics.

TABLE 8-7. Result summary

Metric	Free BSD	Linux	Solaris	WRK
File Organization				
Length of C files			–	–
Length of header files		+		–
Defined global functions in C files			–	–
Defined structures in header files				–
Directory organization		+		
Files per directory		–		
Header files per C source file				
Average structure complexity in files	–		+	
Code Structure				
Extended cyclomatic complexity		+		–

Metric	Free BSD	Linux	Solaris	WRK
Statements per function		+		
Halstead complexity		+		–
Common coupling at file scope		–		
Common coupling at global scope		+		
% global functions		+		–
% strictly structured functions	–			+
% labeled statements		–		+
Average number of parameters to functions				
Average depth of maximum nesting			–	–
Tokens per statement				
% of tokens in replicated code	–	–	+	
Average structure complexity in functions	+	–		
Code Style				
Length of global identifiers				+
Length of aggregate identifiers				+
% style conforming lines			+	–
% style conforming typedef identifiers	–	–		+
% style conforming aggregate tags	–	–	–	+
Characters per line				
% of numeric constants in operands		–	+	+
% unsafe function–like macros			–	
Comment density in C files		–		+
Comment density in header files		–		+
% misspelled comment words				+
% unique misspelled comment words				+
Preprocessing				
Preprocessing expansion in functions	–		+	
Preprocessing expansion in files				–
% of preprocessor directives in header files		–	–	+
% of non–#include directives in C files	–		+	
% of preprocessor directives in functions	–		+	
% of preprocessor conditionals in functions	–	+	+	
% of function–like macros in defined functions		+		–
% of macros in unique identifiers	–		+	+
% of macros in identifiers	–		+	
Data Organization				
Average level of namespace pollution in C files	+			–

Metric	Free BSD	Linux	Solaris	WRK
% of variable declarations with global scope		+		–
% of variable operands with global scope	–	+		
% of identifiers with wrongly global scope		+		–
% of variable declarations with file scope	+			–
% of variable operands with file scope		+		–
Variables per typedef or aggregate		–		+
Data elements per aggregate or enumeration		–		+

Nevertheless, by looking at the distribution and clustering of markings, we can arrive at some important plausible conclusions. The most interesting result, which I drew from both the detailed results listed in the previous sections and the summary in Table 8-7, is the similarity of the values among the systems. Across various areas and many different metrics, four systems developed using wildly different processes score comparably. At the very least, the results indicate that the structure and internal quality attributes of a large and complex working software artifact, will represent first and foremost the formidable engineering requirements of its construction, with the influence of process being marginal, if any. If you're building a real-world operating system, a car's electronic control units, an air traffic control system, or the software for landing a probe on Mars it doesn't matter if you're managing a proprietary software development team or running an open source project: you can't skimp on quality. This does not mean that process is irrelevant, but that processes compatible with the artifact's requirements lead to roughly similar results. In the field of architecture this phenomenon has been popularized under the motto “form follows function” [Small 1947].

One can also draw interesting conclusions from the clustering of marks in particular areas. Linux excels in various code structure metrics, but lags in code style. This could be attributed to the work of brilliant motivated programmers who aren't, however, effectively managed to pay attention to the details of style. In contrast, the high marks of WRK in code style and low marks in code structure could be attributed to the opposite effect: programmers who are effectively micro-managed to care about the details of style, but are not given sufficient creative freedom to develop techniques, design patterns, and tools that would allow them to conquer large-scale complexity.

The high marks of OpenSolaris in preprocessing could also be attributed to programming discipline. The problems from the use of the preprocessor are well-known, but its allure is seductive. It is often tempting to use the preprocessor to create elaborate domain-specific programming constructs. It is also often easy to fix a portability problem by means of conditional compilation directives. However, both approaches can be problematic in the long run, and we can hypothesize that in an organization like Sun programmers are discouraged from relying on the preprocessor.

A final interesting cluster appears in the low marks for preprocessor use in the FreeBSD kernel. This could be attributed to the age of the code base in conjunction with a gung-ho programming attitude that assumes code will be read by developers at least as smart as the one who wrote it. However, a particularly low level of namespace pollution across the FreeBSD source code could be a result of using the preprocessor to set up and access conservatively scoped data structures.

Despite various claims regarding the efficacy of particular open or close-source development methods, we can see from the results that there is no clear winner (or loser). One system with a commercial pedigree (OpenSolaris) has the highest balance between positive than negative marks. On the other hand, WRK has the largest number of negative marks, while OpenSolaris has the second lowest number of positive marks. Looking at the open source systems, although FreeBSD has the highest number of negative marks and the lowest number of positive marks, Linux has the second highest number of positive marks. Therefore, the most we can read from the overall balance of marks is that open source development approaches do not produce software of markedly higher quality than proprietary software development.

Acknowledgments and Disclosure of Interest

I wish to thank Microsoft, Sun, and the members of the FreeBSD and Linux communities for making their source code available in a form that allows analysis and experimentation. I also thank Fotis Draganidis, Robert L. Glass, Markos Gogoulos, Georgios Gousios, Panos Louridas, and Konstantinos Stroggylos for their help, comments, and advice on earlier drafts of this work. This work was partially funded by the European Community's Sixth Framework Programme under the contract IST-2005-033331 “Software Quality Observatory for Open Source Software (SQO-OSS)”.

I've been a source code committer in the FreeBSD project since 2003, I have participated as an invited guest in three Microsoft-sponsored academic initiatives, and I've been using all four systems for more than a decade.

Bibliography

[Cannon et al.] , Cannon, L. W., and . others. Recommended C style and coding standards. http://sunland.gsfc.nasa.gov/info/cstyle.html.

[Cant et al. 1995] Cant, S. N., D. R. Jeffery, and B. L. Henderson-Sellers (1995, June). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology* 37(7), 351–362.

[Capiluppi and Robles 2007] , Capiluppi, A., and G. Robles (Eds.) (2007, May). FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development. IEEE Computer Society.

[Chidamber and Kemerer 1994] Chidamber, S. R., and C. F. Kemerer (1994). A metrics suite for object oriented design, http://dx.doi.org/10.1109/32.295895. *IEEE Transactions on Software Engineering* 20(6), 476–493.

[Coleman et al. 1994] Coleman, D., D. Ash, B. Lowther, and P. W. Oman (1994). Using metrics to evaluate software system maintainability, http://dx.doi.org/10.1109/2.303623. *Computer* 27(8), 44–49.

[Cusumano and Selby 1995] , Cusumano, M. A., and R. W. Selby (1995). Microsoft Secrets. New York: The Free Press.

[Dickinson 1996] K. Dickinson (1996). Software process framework at Sun, 10.1145/240819.240830. *StandardView* 4(3), 161–165.

[Feller 2005] , FellerJ. (Ed.) (2005). 5-WOSSE: Proceedings of the Fifth Workshop on Open Source Software Engineering. ACM Press.

[Feller and Fitzgerald 2001] , Feller, J., and B. Fitzgerald (2001). Understanding Open Source Software Development. Reading, MA: Addison-Wesley.

[Feller et al. 2005] , Feller, J., B. Fitzgerald, S. Hissam, and K. Lakhani (Eds.) (2005). Perspectives on Free and Open Source Software. Boston: MIT Press.

[Fitzgerald and Feller 2002] Fitzgerald, B., and J. Feller (2002). A further investigation of open source software: Community, co-ordination, code quality and security issues, 10.1046/j.1365-2575.2002.00125.x. *Information Systems Journal* 12(1), 3–5.

[Gill and Kemerer 1991] Gill, G. K., and C. F. Kemerer (1991). Cyclomatic complexity density and software maintenance productivity, http://dx.doi.org/10.1109/32.106988. *IEEE Transactions on Software Engineering* 17(12), 1284–1288.

[Glass 1999] R. L. Glass (1999, January/February). Of open source, Linux ... and hype. *IEEE Software* 16(1), 126–128.

[Halstead 1977] , M. H. Halstead (1977). Elements of Software Science. New York: Elsevier New Holland.

[Harbison and Steele Jr. 1991] , Harbison, S. P., and G. L. Steele Jr. (1991). C: A Reference Manual Third ed.. Englewood Cliffs, NJ: Prentice Hall.

[Henry and Kafura 1981] Henry, S. M., and D. Kafura (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering* SE-7(5), 510–518.

[Hoepman and Jacobs 2007] Hoepman, J. H., and B. Jacobs (2007). Increased security through open source, http://doi.acm.org/10.1145/1188913.1188921. *Communications of the ACM* 50(1), 79–83.

[Izurieta and Bieman 2006] Izurieta, C., and J. Bieman (2006). The evolution of FreeBSD and Linux. In *ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pp. 204–211. ACM Press.

[Jrgensen 2001] N. Jrgensen (2001, October). Putting it all in the trunk: Incremental software development in the FreeBSD open source project, 10.1046/j.1365-2575.2001.00113.x. *Information Systems Journal* 11(4), 321–336.

[Kernighan and Plauger 1978] , Kernighan, B. W., and P. J. Plauger (1978). The Elements of Programming Style Second ed. New York: McGraw-Hill.

[Kuan 2003] J. Kuan (2003, January). Open source software as lead user's make or buy decision: A study of open and closed source quality. In *Second Conference on The Economics of the Software and Internet Industries*.

[Li et al. 2006] Li, Z., S. Lu, S. Myagmar, and Y. Zhou (2006). CP-miner: Finding copy-paste and related bugs in large-scale software code, 10.1109/TSE.2006.28. *IEEE Transactions on Software Engineering* 32(3), 176–192.

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

[McConnell 2004] , S. C. McConnell (2004). Code Complete: A Practical Handbook of Software Construction second ed. Redmond, WA: Microsoft Press.

[Parnas 1972] D. L. Parnas (1972, December). On the criteria to be used for decomposing systems into modules. *Communications of the ACM* 15(12), 1053–1058.

[Paulson et al. 2004] Paulson, J. W., G. Succi, and A. Eberlein (2004, April). An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* 30(4), 246–256.

[Polze and Probert 2006] Polze, A., and D. Probert (2006). Teaching operating systems: The Windows case. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pp. 298–302. ACM Press.

[Rigby and German 2006] , Rigby, P. C., and D. M. German (2006, January). A preliminary examination of code review processes in open source projects. University of Victoria. http://helium.cs.uvic.ca/thread.html/Rigby2006TechReport.pdf.

[Salus 1994] , P. H. Salus (1994). A Quarter Century of UNIX. Boston, MA: Addison-Wesley.

[Samoladas et al. 2004] Samoladas, I., I. Stamelos, L. Angelis, and A. Oikonomou (2004). Open source software development should strive for even greater code maintainability, http://doi.acm.org/10.1145/1022594.1022598. *Communications of the ACM* 47(10), 83–87.

[Small 1947] , SmallH.A. (Ed.) (1947). Form and Function: Remarks on Art by Horatio Greenough. Berkeley and Los Angeles: University of California Press.

[Sowe et al. 2007] , Sowe, S. K., I. G. Stamelos, and I. Samoladas (Eds.) (2007). Emerging Free and Open Source Software Practices. Hershey, PA: IGI Publishing.

[Spencer and Collyer 1992] Spencer, H., and G. Collyer (1992, June). #ifdef considered harmful or portability experience with C news. In AdamsR. (Ed.), *Proceedings of the Summer 1992 USENIX Conference*, Berkeley, CA, pp. 185–198. USENIX Association.

[Spinellis 2003] D. Spinellis (2003, November). Global analysis and transformations in preprocessed languages, doi:10.1109/TSE.2003.1245303. *IEEE Transactions on Software Engineering* 29(11), 1019–1030.

[Spinellis 2006] , D. Spinellis (2006). Code Quality: The Open Source Perspective. Boston, MA: Addison-Wesley.

[Spinellis 2008] D. Spinellis (2008, May). A tale of four kernels. In Schäfer, W., M. B. Dwyer, and V. Gruhn (Eds.), *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, New York, pp. 381–390. Association for Computing Machinery.

[Spinellis 2010] D. Spinellis (2010, April). CScout: A refactoring browser for C, 10.1016/j.scico.2009.09.003. *Science of Computer Programming* 75(4), 216–231.

[Spinellis and Szyperski 2004] Spinellis, D., and C. Szyperski (2004, January/February). How is open source affecting software development?, doi:10.1109/MS.2004.1259204. *IEEE Software* 21(1), 28–33. Guest Editors' Introduction: Developing with Open Source Software.

[Stallman et al. 2005] , Stallman, R., and . others (2005, December). GNU coding standards. http://www.gnu.org/prep/standardstoc.html.

[Stamelos et al. 2002] Stamelos, I., L. Angelis, A. Oikonomou, and G. L. Bleris (2002). Code quality analysis in open source software development, 10.1046/j.1365-2575.2002.00117.x. *Information Systems Journal* 12(1), 43–60.

[Stol et al. 2009] Stol, K. J., M. A. Babar, B. Russo, and B. Fitzgerald (2009). The use of empirical methods in open source software research: Facts, trends and future directions. In *FLOSS '09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, Washington, DC, USA, pp. 19–24. IEEE Computer Society.

[Tanenbaum 1987] , A. S. Tanenbaum (1987). Operating Systems: Design and Implementation. Englewood Cliffs, NJ: Prentice Hall.

[The FreeBSD Project 1995] , The FreeBSD Project (1995, December). Style---Kernel Source File Style Guide. The FreeBSD Project. FreeBSD Kernel Developer's Manual: style(9). Available online http://www.freebsd.org/docs.html (January 2006).

[Torvalds and Diamond 2001] , Torvalds, L., and D. Diamond (2001). Just for Fun: The Story of an Accidental Revolutionary. New York: HarperInformation.

[von Krogh and von Hippel 2006] vKrogh, G., and E. vHippel (2006, July). The promise of research on open source software, 10.1287/mnsc.1060.0560. *Management Science* 52(7), 975–983.

[Yu et al. 2006] Yu, L., S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt (2006). Maintainability of the kernels of open source operating systems: A comparison of Linux with FreeBSD, NetBSD and OpenBSD, 10.1016/j.jss.2005.08.014. *Journal of Systems and Software* 79(6), 807–815.

[Yu et al. 2004] Yu, L., S. R. Schach, K. Chen, and J. Offutt (2004). Categorization of common coupling and its application to the maintainability of the Linux kernel, 10.1109/TSE.2004.58. *IEEE Transactions on Software Engineering* 30(10), 694–706.