2006-1213: QUANTITATIVE ANALYSIS OF PROGRAMS: COMPARING OPEN-SOURCE SOFTWARE WITH STUDENT PROJECTS

Evan Zelkowitz, Purdue University

Mark C Johnson, Purdue University

Yung-hsiang Lu, Purdue University

Quantitative Analysis of Programs: Comparing Open-Source Software with Student Projects

Abstract

The lack of quantitative measures is a common problem in a programming course. Even though most students understand the importance of comments and good program structures, there is no quantitative "rule of thumb" to guide students in determining whether their programs have sufficient comments or are well-structured. For example, an instructor may require one line of comment for every ten lines of codes. These numbers are determined without sufficient scientific support; hence, students may resist the requirements and treat them as burdens.

Open-source programs are widely used today and they can be considered as samples for teaching programming. We analyze 6 open-source software projects with 6233 files and 3.27 million lines of code to discover their commonalities. The projects are python, gdb, emacs, httpd, kde, and doxygen. These open-source programs are used and contributed by many programmers. These particular programs are selected as examples of high quality code by virtue of their extensive and successful use in industry and academia. These programs are used also because it is difficult to obtain large-scale non-trivial programs from companies and sample programs from textbooks are usually very small. Because quality measures are often subjective, we focus on quantitative measures that can be objective and obtained by software tools.

In our analysis of open source software, we find that the average length of codes between comments is fewer than one hundred characters, or only a few lines. Most comments are short, only one or two lines. While global variables are often considered detrimental to program organization by instructors, global variables are actually frequently used in open-source programs maintained by multiple programmers. Hence, instructors should not use the presence of global variables as the sole indication of poor program structures. The 6 projects are written in C or C++ and functions are the fundamental unit of C/C++. In these projects, most functions call only a few other functions. This study shows strong similarities in these different projects and suggests the possibility of using a quantitative approach to teaching programming. We compare the results with the programs written by the students in a senior-level software engineering course. We discover that their programs have similar properties as open-source programs. Hence, we hypothesize that students may benefit by using these quantitative measures from open-source programs as samples and learn better programming skills and styles.

Introduction

Open-source software provides abundant opportunities to study the properties of successful software projects. These projects are considered successful because they enjoy a large population of users and are still being improved. The history of open-source projects can be traced to analyze the organizations and progression of software development. For example, Dinh-Trong et al.[5] analyze the CVS repository and email archive of FreeBSD. Mockus et al.[11] discover that Apache has a small core group of 15 people that contribute more than 80% of the project. Even though most open-source projects have separate documentation, such as `README` and `ChangLog`, the source codes still present the most up-to-date information for a new contributor to understand the projects. Because open-source programs are often developed by volunteers that are geographically distributed, it is imperative that the codes are understandable to new developers.

Lakos[9] suggests that physical properties, in particular the dependency among files, indicate the quality of the software. The physical properties are different from logical properties because the former do not consider the design, performance, or robustness of the software. Physical properties are the first impression a new developer sees, for example, whether the directory and file structures are reasonable or the comments are sufficient. In contrast to logical quality, physical quality can be more easily extracted and analyzed by tools for software metrics. Metrics have not been widely accepted in software engineering, partially because they can be misleading. One example is using thousand lines of codes (KLOC) to quantify the complexity of software. The same functionality can be implemented in different ways and they may change KLOC dramatically. However, quantitative analysis can serve as "rules of thumb" for successful software projects. The literature contains qualitative rules[8, 10] but they do not provide enough quantitative information. One example is adding comments to codes. Every software developer knows the importance of adding comments to source codes. However, few evidences have been provided to suggest *how many* and *where* comments should be added. A manager or an instructor may require one line of comment for every ten lines of codes. These numbers are determined arbitrarily without sufficient scientific support; hence, developers or students may resist the requirements and treat them as burdens.

Analyzing the quality of software is usually considered a difficult task. Many factors can influence the evaluation of quality. These factors can be subjective by individual software developers due to their past experience or domain knowledge of the problems to be solved. Judgment of quality is also affected by developers' familiarity of the tools that can help the developers understand or improve the software. Some rules are advocated to improve software quality; for example, global variables are considered a potential factor to degrade the quality of software. Another rule is to avoid or eliminate cyclic dependence among classes or files. However, these rules are usually presented without sufficient data from real successful software projects. We aim to provide quantitative measures of the easiness for developers to understand, modify, and improve programs. This paper examines several widely used open-source programs to extract the commonalities among these programs.

In this paper, we analyze the physical properties of the source codes from a new developer's view point. The analysis includes popular open-source projects, such as `emacs` and `python`. The physical properties can be obtained using a compiler front end without manually reading and understanding the meanings of the codes. We extract the quantitative measures among these projects; the analysis can provide a foundation to quantify the quality of other software projects. In particular, if a project's physical properties are too far away from the properties presented in this paper, the project may not enjoy the same degree of success of the projects we analyze. Using physical properties to infer quality is still a relatively new approach. We discover some strong similarities among these projects. For example, there is a comment every several lines of codes. Most comments are shorter than one hundred letters. The majority of files are only several hundred lines. Global variables are frequently used so they should not be considered as a primary source of degraded quality. Finally, most functions call only a few other functions; thus, simple dependence exists among most functions and unit tests can be conducted more easily. We extend the study beyond open-source programs and analyze the programs written by students in a senior-level software engineering course. The students wrote programs to compete in a network-based game. Each group included five or six students and they could choose the languages as long as their programs could communicate through the network. Our findings indicate that the students' programs have similar quantitative properties as open-source programs. Hence, we hypothesize that these properties can be taught to improve students' code quality.

Previous Work

Quality Analysis

Whittaker et al[15] review the history of software development in the past 50 years; they explain that 1970s were the turning point of software quality. In 1970s, computers became cheap enough and accessible to more people; meanwhile, more companies started using computers to solve non-numerical problems. Software quality suffered due to the combination of more complex problems and less trained developers. Several studies have demonstrated the feasibility of using simple rules to evaluate the quality of software. Lakos[9] quantifies quality based on the coupling among files using the concept of levelization. If a file is well tested and believed working, it is labeled as level zero. Standard C header or library files are examples of level-0 files. A file is level n if the file needs files of level $n-1$ or smaller for compilation or linking. Cyclic coupling is considered as a sign of inferior design. Arisholm et al.[1] point out that static analysis of codes coupling cannot be directly applied to object-oriented software due to polymorphism and run-time binding. Thus, they analyze dynamic coupling of object-oriented software. They discover that dynamic export coupling can be an effective indicator of change proneness.

Basili et al.[2] classify errors into several types such as incorrect requirements and misunderstanding of the environment. The study discovers that up to 72% errors can be attributed to design errors of single components. There is not clear correlation between the sizes of modules and the error density. Shen et al.[14] analyze software to determine how to allocate resources for testing. Their study compares five products written in Pascal, PL/S, and assembly. They find that smaller modules do not necessarily have lower error density. Error density can be a size-normalized indication of program quality for only the modules with more than 500 lines of codes. Thus, they conclude that error density is not an effective way to measure quality. Withrow[16] analyzes the error reports of an Ada program with 362 modules across 114000 lines. When a module's size is 250 lines the error density is the lowest, approximately 0.5 error per thousand lines of codes. Withrow offers one explanation why a larger module may have a lower error density: the large module is not completely tested.

Yu et al.[18] consider coupling through global variables in Linux. The hypothesis is that global variables can degrade the code quality. Their study classifies global variables into five categories depending on whether a variable is defined or used in kernel codes or not. They find 99 global variables occurring in over 15000 instances. Xie et al.[17] use redundant codes as an indication of severe defects. The premise is that developers do not intentionally write codes that have no effect. One example of redundancy is an assignment whose value is never used. This may occur if the variable's name is mistyped later and it happens to be another valid variable. Another example is dead codes that can never execute; this suggests incorrect control flow in the program. The third instance of redundant codes are conditions that are always true, such as comparing an unsigned integer and a negative constant.

Open-Source Software

Open-source software represents an approach different from traditional ways for building large-scale commercial software. Successful examples such as Linux, Apache, and gcc provide examples for further studies about the characteristics of successful software. Some studies focus on the "macro behavior" of the projects, such as the number of people involved, the volume of email correspondences, the frequencies of version changes, and the number of bug reports. The sources of these analysis often come from CVS repositories, ChangeLog, or the archives of newsgroups and mailing lists. Dinh-Trong et al.[5] perform a case study FreeBSD to reconstruct its 10-year history from the CVS repository, the email archive, and the bug database. Their report highlights some important processes adopted in the FreeBSD project, for instance, the release procedure. The analysis also indicates that many people contribute to the project: top 15 people contribute only 56%; 80% of the contribution comes from 50 people. In contrast, Mockus et al.[11] find that 15 people contribute more than 80% of the codes for Apache. Their study also discover that file ownership is not prevalent; many files are modified by multiple people. Approximately half of problem reports are resolved within a day and 75% are resolved within 42 days. The study is expanded to Mozilla;[12] the development environment of Mozilla is different because it is supported by a company, Netscape. Over four hundred people contribute to the CVS repository of Mozilla and nearly seven thousand people report bugs. In Apache, there is no strict rule about code ownership; in Mozilla, ownership is enforced.

Capiluppi et al.[4] analyze 406 open-source projects, including the numbers of subscribers of the mailing lists, stable and transient developers, and the domains of these projects. Over 66% of the projects build tools for other developers, in particular for the Internet. Documentation is critical for open-source programs in order to help new developers understand the programs. More than one third provide documentation including `README`, UNIX-like manual pages, user manuals and API specifications. Capiluppi et al. discover that among these projects 41.5% use C; 14% use C++ and another 14% use Perl. In another study, Capiluppi[3] analyzes 12 open-source projects from the 406 projects by excluding those that are no longer actively updated. The 12 projects are classified into three categories: large, medium, and small based on the numbers of participants of the projects. The paper presents the relationship among the number of developers, the rate of changes in project sizes, the number of versions, and so on. Based on this analysis, the author proposes an empirical linear formula to characterize the relationships. Ferenc et al.[6] developed a tool to extract information from open-source programs. The information includes the number of methods in each class, the depth of inheritance, number of children in each class, and coupling between classes. Their study finds that Mozilla has an average of 13.4 methods per class; the standard deviation is 14.9. The average depths of inheritance is 1.32 while the longest depth is 9. On average, a class requires 6.8 other classes either through member data or methods. The study also examines several hypotheses about fault proneness from these numbers through 7 different versions of the codes. Godfrey et al.[7] study 96 versions of Linux kernel and discover super-linear growth in code sizes and lines of codes across five years. The amount of codes grows substantially over the five years, at a rate much higher than the other parts of Linux. The median of file sizes for both .c and .h files grow over time but they are confined within narrow ranges. The median sizes of .c files are between 400 and 600 lines after mid 1994. The median sizes of .h files are between 80 and 120 lines after late 1995. Paulson et al.[13] compare three open-source projects and three closed-source projects; evidence shows that open-source projects enjoy better creativity from developers and quicker defect fixes. Meanwhile, the projects have comparable growth rates, simplicity in implementation, and modularity.

Contributions

This paper presents several quantitative measures of the structure of the source codes from 6 open-source projects. We discover strong similarity among these projects; hence, we hypothesize that these quantitative measures form a basis to evaluate the quality of a software project. If the quantitative measures of a project deviates from our discovery, the project may suffer low quality. Our study is based on a developer's viewpoint: whether the source codes are easy to understand, modify, or extract useful portions for further improvement. Our analysis is static and can be easily implemented in a compiler front end; this study does not consider the run-time behavior, such as robustness or correctness. Because of wide use of the software projects studied in this paper, these projects are assumed to be robust and correct.

Static Analysis of Open-Source Programs

Selection of Samples

We select 6 open-source projects and analyze them using a compiler front end. These projects include doxygen, emacs, gdb, httpd, kde, and python.

- *doxygen.* Doxygen is a tool to automatically extract documentation from source codes. It analyzes the source codes and creates html pages with hyperlinks among related code segments. For example, if a function calls another function, they are associated by "reference" and "referenced by" links. Users can see both summary pages and colored source codes with hyperlinks. Doxygen can extract the code structure without any modification on the source codes. Programmers can also embed doxygen instructions inside the comments of the source codes. We use the output from doxygen to analyze the source code used in this study.

- *emacs.* Emacs is a text editor that is "extensible, customizable, self-documenting real-time" (from emacs manual). Emacs recognizes many file types, such as C or Latex, and adjusts the features based on the file types. Users can also extend emacs' features through Emacs Lisp; it is a dialect of the Lisp programming language. Emacs can be used for reading email or news. Emacs can collaborate with gdb so that users can use gdb to debug a program while editing the same program in emacs.

- *gdb.* The GNU debugger (gdb) is a source-level debugger for C, C++, and Modula-2. It can set conditional break points and print the values of variables; Gdb can also dereference pointers and traverse the call stack. Gdb can detect whether a variable has changed its value. Gdb is used as the foundation for other debugging interfaces, such as the data display debugger (DDD).

- *httpd.* Httpd is a web server in the Apache project since 1995. It is developed by volunteers; design and implementation decisions are conducted by voting through the Internet. Apache web server supports virtual hosts and database; users can extend the functionality by building modules through an API. In February 2005, the Apache web server is used in over 40 million or 68% sites as the most popular web server (from Netcraft Survey).

- *kde.* Kde is a desktop environment for UNIX-based systems to provide user-friendly easy-to-configure interfaces. Kde starts with the purpose to create an easier environment for writing applications than X11-based windows. Hundreds of applications have been written for kde, ranging from network utilities to games. This study considers the base of the kde environment only.

- *python.* Python is an object-oriented script programming language. Python supports modules, classes, exceptions, and allows users to implement new modules using C or C++. Since 1991, a stable new version of python is released every 6 to 18 months. Python is used by Redhat to write software for installation and system administration.

Table 1 lists the 6 projects studied in this paper. All of them have been updated many versions and are still being improved. The number of files are counted before running the `configure` tool. We only extract the files from the packages as they are provided. Many open-source programs include configure tools to generate platform-specific files or to adjust the compilation and linking options based on the users' inputs or tool availability. When we count the number of lines, we include blank lines because they provide visual effects separating codes and contributing to easier understanding. From Table 1, we can discover several patterns. First, the number of .c files is within 0.58 (emacs) and 2.1 (httpd) of the number of .h files. Second, the ratio of total number of lines of codes (LOC) of .c files and of .h files is between 2.7 (doxygen) and 7.7 (emacs). In this paper, LOC includes both codes and comments. Third, the average LOC per .c file is higher than LOC per .h file but their ratio is between 2.9 (httpd) and 13.3 (emacs). This list intentionally excludes Linux, FreeBSD, or any other open-source operating systems because they contain many hardware-specific files and device drivers; these files can be safely ignored if the developers do not use the particular hardware components. In this paper, we use "`project:file name`" for a particular file in the project.

project	purpose	version	last update	.c files*	LOC	.h files	LOC
doxygen	documentation	1.4.1	01/2005	163	217,877	208	80,091
emacs	editor	21.3	03/2003	182	318,076	313	41,184
gdb	debugger	6.3	11/2004	1473	1,374,740	922	207,773
httpd	web server	2.0.52	09/2004	481	226,114	230	37,400
kde	desktop	3.3.1	08/2004	890	309,210	930	94,104
python	script language	2.4	11/2004	293	313,099	148	51,707
total				3582	2,759,116	2751	512,259

Table 1: The 6 open-source projects studied in this paper. *: both .c and .cpp files; LOC: lines of codes, including blank lines.

Codes and Comments

Adding comments among codes is one of the most advocated practice to improve software understandability. Writing comments also provides an opportunity for a programmer to think about the key information for understanding the implementation. Comments offer another channel to enhancing quality: if the comments and the codes are inconsistent, the problem may be discovered more easily. Some professors in introductory programming courses require that students add comments regularly, for example, every 10 lines of codes. This number would appear to be determined arbitrarily, causing developers or students to resist the requirement as being extra work with little benefit.

We analyze the length of a code between comments and the length of a comment between codes in .h and .c (and .cpp) files. Table 2 shows the results of our analysis. For all projects, the average length of codes is between 200 and 400 letters between comments. The median is significantly shorter, between 33 and 96 letters. Short codes are common for variable declaration. For example, in `doxygen:doxytag.cpp`, many variable declarations are immediately followed by a comment:

```
char *yy_ch_buf;      /* input buffer */
```

Meanwhile, we also discover that the standard deviations are very large because of machine-generated codes. For example, `doxygen:scanner.cpp` contains 823707 letters without comments; however, this file is generated by `flex` (fast lexical analyzer generator). Similarly, in python the longest code segment contains 652866 letters and it is generated by `Qt`. Other long code segments tend to be tables. A table in `emacs:macfns.c` is used for color maps and the code spans nearly 800 lines without any comments. We remove the shortest and the longest 10% codes and recompute the means and the standard deviations. In the table, we also show the middle 80% of lengths. The columns show that the average lengths of codes are very uniform across all 6 projects, from 33 letters (python) to 77 letters (doxygen). The standard deviations are also significantly reduced to smaller than 227. Among all projects, the average number of letters per line for .c and .h files is between 28.8 (emacs) and 32.9 (doxygen). We use the average lengths of codes and divide them by the average numbers of letters per line and obtain that on average a comment occurs every 3.3 (doxygen) to 7.7 (python) lines of code.

The analysis performed on comments indicates that most comments are also short. The average lengths of comments are 62 letters (python) to 125 letters (httpd). Unlike codes, the standard deviations of comment lengths are significantly smaller: 192 to 506. We interpret this as the lack of "machine-generated comments". If the shortest 10% and the longest 10% comments are removed, the standard deviations are much smaller: between 22 and 66. This suggests that in open-source programs, the lengths of comments are short and highly uniform. If we divide the average lengths of comments by the average lengths per line, most comments are within three lines long.

program	code length					comment length				
	mean	median	STD	mean*	STD*	mean	median	STD	mean*	STD*
doxygen	245	59	6088	77	92	72	42	203	47	24
emacs	214	84	504	77	124	110	61	192	77	48
gdb	219	41	8083	46	120	82	37	257	46	32
httpd	285	96	592	73	158	125	50	215	78	66
kde	342	81	810	47	227	105	43	506	47	27
python	454	33	6224	33	143	62	25	194	33	22

Table 2: Code Length: the lengths (letters) of codes between comments. Comment Length: the lengths (letters) of comments between code. STD: standard deviation. *: the shortest and longest 10% removed.

Figure 1 shows the distributions of code and comment lengths. In all figures, the shortest and the longest 10% lengths have been removed. The curves show that all projects have very similar distributions of code lengths and comments lengths. More than 80% codes contain few than 600 letters; more than 80% comments are shorter than 150 letters. Our analysis

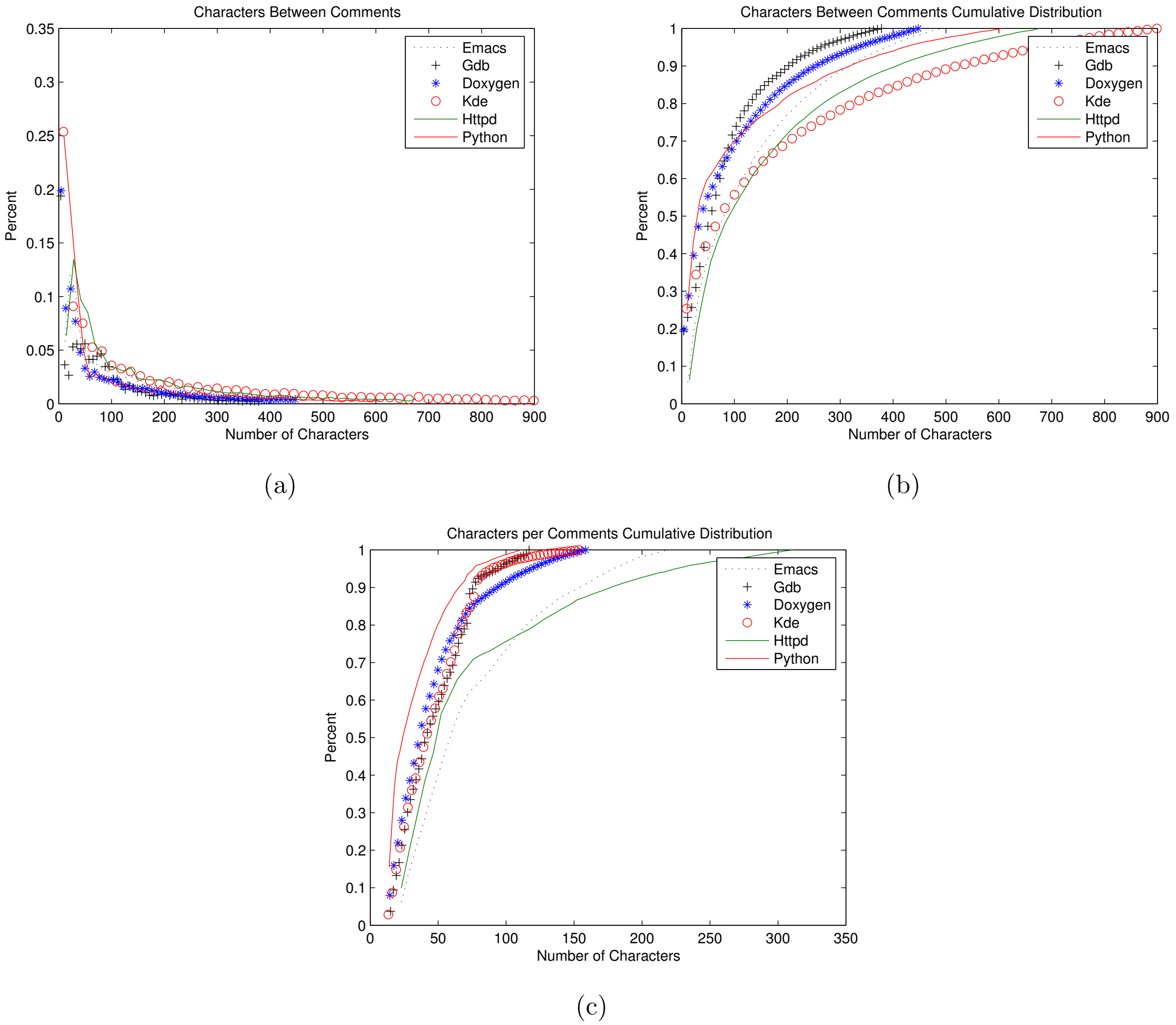
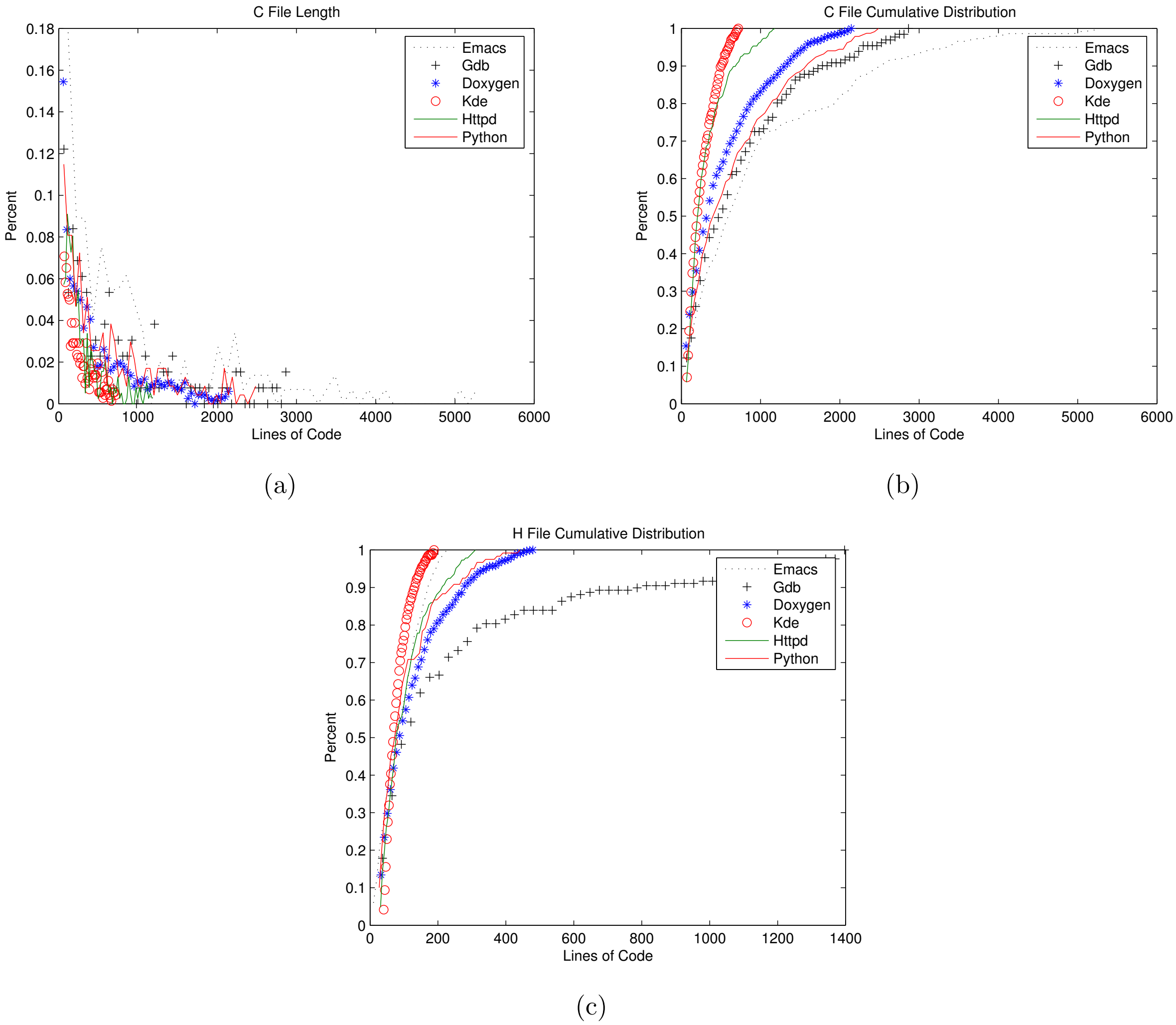


Figure 1: (a) Distributions of code lengths. (b) Cumulative distribution of code lengths. (d) Cumulative distribution of comment lengths. The shortest and longest 10% have been removed.

suggest strong similarities among these projects and the similarities can be used as a basis to quantify a software project. If the project's code lengths are substantially longer, more comments should be added to improve the understandability. The figure also shows that there is no strong correlation between the distributions of code lengths and comment lengths.

File Size

The second set of analysis considers the number of lines in each file. Table 3 lists the averages and the standard deviations of file sizes. One question is whether machine generated files should be distributed in the open-source programs. On one hand, these files should



(a)

(b)

(c)

Figure 2: (a) Distributions of C file sizes. (b) Cumulation of C file sizes. (c) Cumulation of H file sizes. The shortest and longest 10% lengths have been removed.

be included because users may not have the tools to re-generate the files. On the other hand, these files should not be included to prevent inconsistency: if the source files have been modified and the machine-generated files are not updated. One solution is not to include these files and to generate them when executing `make`. The purpose of our study is to automatically obtain programs' properties without manually inspecting individual files. Consequently, we include machine-generated files in our analysis because the files are included in the originally downloaded packages.

Emacs has the largest average .c file size, over 1000 LOC per file. Emacs contains 8 .c files with more than 10000 LOC. The longest file is `emacs:xdisp.c` with 15136 lines for updating the display. The second longest file is `emacs:xterm.c` with 14756 lines; the file is also related to display. These two files are written by the same person. Seven of the eight

program	C file length		H file length	
	mean*	STD*	mean*	STD*
doxygen	742	718	269	355
emacs	1024	1081	88	58
gdb	523	500	127	102
httpd	311	257	104	69
kde	260	168	80	34
python	655	611	108	90

Table 3: File sizes (lines). *: the shortest and longest 10% removed.

longest files handle display for different window systems, including W32, X, and Mac. None of them appears to be machine generated. Among all 182 .c files in emacs, 65 have more than 1000 LOC. This may simply be the convention adopted in emacs by using fewer but longer files.

In contrast, the lengths of .h files are much closer in the 6 projects. The average lengths span between 88 lines (emacs) and 269 lines (doxygen). Figure 2 shows the distributions of the file lengths. The data show no strong relationship between the lengths of .c files and .h files. The average length of .c files in emacs is the largest but the average length of .h files is the second smallest. The data indicates that the average file length of a software project should be several hundred lines for .c files and shorter than three hundred lines for .h files. If the average length is too small, the project is scattered into too many files (suppose the total project size is the same). If the average file length is too large, unrelated code may be put into the same file and degrade understandability.

Function Calls

Software engineering practice emphasizes modularity. One indication of successful modularity is the degree of coupling among functions. File dependence can be used to quantify program structures.[9] The premise is that high-quality structures should be "levelizable": files are levelizable if there is no circular compile-time or link-time dependence. If two or more files have circular dependencies, unit tests on individual files are impossible. However, files are not an appropriate level to determine dependence— a programmer can put all functionalities inside a single file and eliminate all circular dependence among files. Thus, we study the dependence among functions, instead of files. If a function calls another function, the former is a *caller* of the latter; the latter is a *callee* of the former.

We analyze the number of functions called within other functions. If a function (callee) is called by multiple functions (caller), the callee is counted multiple times. If a callee is called in multiple places inside the same caller, the callee is counted once. For example, consider the call relationship shown in the sample code:

```
A ()                      B()
```

```
{                                {
   B();                             C();
   C();                          }
   B();
}
```

In this example, there are two callers: `A` and `B`. `B` is called only by `A` so `B` is counted once. `C` is called by both `A` and `B` so `C` is counted twice. Table 4 shows our results. We obtain this table by using doxygen to analyze each project. Doxygen works as a compiler front end to analyze the relationship among functions. The table shows that doxygen has 1398 functions that call other functions. Totally, 8801 functions are called and each caller calls 6.3 functions on average. A high percentage of functions have single callees. The table shows that 15.89% of functions in doxygen have only single callees. Similarly, 34.76% of functions in emacs have only single callees. This table considers only functions defined in the project and does not include library functions, such as `printf`. From this table, we can see that there is strong commonality among these six projects. The average number of callees per function spans between 2.62 and 6.30. This result suggests that if a software project has a much higher number of callees per function, the project may be more difficult to maintain due to the complex dependence relationships among functions. This table also shows the maximum number of callees of a function in each project. We do discover, however, a few functions have many callees. For example, the `main` function gdb calls 720 functions. In most of the instances of these projects, the largest caller of functions is the `main` function. In the instance of gdb, its `main` function has to set many variables during usage, open files, and interpret debug information from the files it has opened. This data shows a fairly consistent average of two functions called within each user defined function. Doxygen has the highest callee / caller ratio. This can be explained by multiple machine generated files similar to previous instances. These files use lookup tables and constantly refer to functions to perform lookups on these tables for comparison. This data provides a good basis for programming where only a small set of functions constitute the majority of functions called in lower levels of the code hierarchy. The second largest group of function callees drops by around one third in most cases showing that the largest groups a very much extremes in almost all cases. Figure 3 shows the distribution of callees of each function with the top and bottom ten percent removed.

Global Identifier

Global identifiers, especially global variables, can be detrimental to a successful project depending on the ratio to local variables and frequency of use.[17,18] Within a given scope, a global variable should be either read or written but not both. If a global variable is both read and written in the same scope, it should be a local variable. Yu et al.[18] note that within the Linux kernel more than one tenth of the global variable definitions and instances are "unsafe": If a modification is made to the original code, this can break the code that references the global variable. Global variables can make a program more difficult

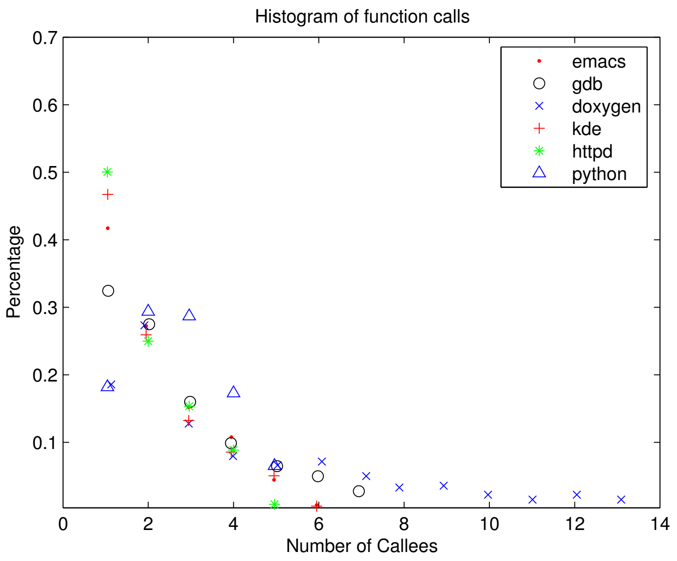


Figure 3: Distribution of function callees

program	Callers	Callees	Ratio	Single Callee	Maximum Callees	Second Maximum
doxygen	1398	8801	6.30	15.89%	152	94
emacs	1441	4148	2.88	34.76%	140	33
gdb	12033	43019	3.58	27.97%	720	95
httpd	1385	3625	2.62	38.23%	139	123
kde	5567	15740	2.83	35.37%	256	102
python	7795	23468	3.01	33.22%	84	47

Table 4: Function callers and respective number of callees.

to understand because different parts of the program become inter-related by writing and reading the variables. Meanwhile, global variables allow different parts of the program to communicate more easily. If a global variable has a well-defined meaning and only a few places may modify the value, reading the variable is a convenient way for communication. This is especially true if a variable represents a status of the program. A C program has several ways to create an identifier that is visible to all files and functions (i.e. global). One example is to define a constant by a macro:

```
#define ACONSTANT  100
```

In our analysis, such a global constant is not counted. However, a constant is counted if it is defined in the following way:

```
const int ACONSTANT = 100;
```

A global constant or variable in this instance is defined as any constant or variable that is defined outside any bracket pair and is used throughout an entire project. This includes the standard integer types as well as characters, arrays, and pointers.

project	Global Variables	Globals*	Numerals	chars	Total LOC	Ratio	Ratio*
doxygen	2032	815	90	6	297968	146	356
emacs	4983	3686	639	32	359260	72	97
gdb	33568	8596	1407	106	1582513	47	184
httpd	3969	2033	155	15	263514	66	129
kde	2086	588	69	3	403314	193	685
python	3752	1378	156	20	364806	97	264
Total	50483	17181	2516	182	3271375	64	190

Table 5: Number of global variables in the 6 open source projects. Globals* is with static, constant, and enum variables removed. Ratio: LOC/Globals

Table 5 shows the analysis of global variables. From this table, we can see that global variables are widely used in these projects. In fact, gdb contains over thirty thousand global variables. Most of these global variables are used for handling many different architectures and has many global tables depending on the architecture used. This greatly increases the total number of global variables compared to other projects. From this table, we hypothesize that global variables in a software project, even thousands of global variables, do not provide a clear indication of the project's success. However, nearly two thirds of global variables are limited to file scope (static) or constants (`const` or `enum`). The majority of global variables used are not of the standard integer, float, long, or char types. Many global variables are user-defined `struct`. In doxygen, "dictionary" types are used as a pre-defined library for their own data types. While the majority of variables are not of standard types, those that are standard are usually of a numerical type. This is most likely used for debug values to change a behavior at runtime, or in the case of gdb, to access specific memory locations such as the stack pointer. Our study does not support the common belief that global variables are harmful to successful software projects.

Analysis of Students' Programs

We analyze the programs from five groups of students in a senior-level course on software engineering. In this course, the students were divided into five teams, each with five or six students. The project was to build a "Yali" game that could compete through a network. Yali is a two-player board game. Each player has 8 marbles; the winner is the first one that moves all marbles to the other side of the board. The game uses the board's center of gravity to determine which player to move. Because the teams competed through a network, they could choose any programming language as long as their programs could communicate. Among the five teams, four chose C/C++ and one chose Java. The first four teams also used Phython as the "glue" language. One team used OpenGL to create 3-dimensional user interface and another team used Qt library. Because most code in the OpenGL and Qt libraries are computer-generated or written by programmers outside the class, we analyze only C/C++ and Java programs. Even though the five teams used different languages, the code sizes were close, between 10.8 and 21.8 KLOC. During the semester, the instructor and the two teaching assistants frequently inspect students' code and encouraged the students to write more comments and to improve their code structures.

Team	code length			comment length		
	mean	median	mean*	mean	median	mean*
team 1	257	89	121	51	47	45
team 2	177	81	108	74	26	38
team 3	224	79	137	83	28	47
team 4	255	83	136	57	62	52
team 5	154	73	108	43	34	37

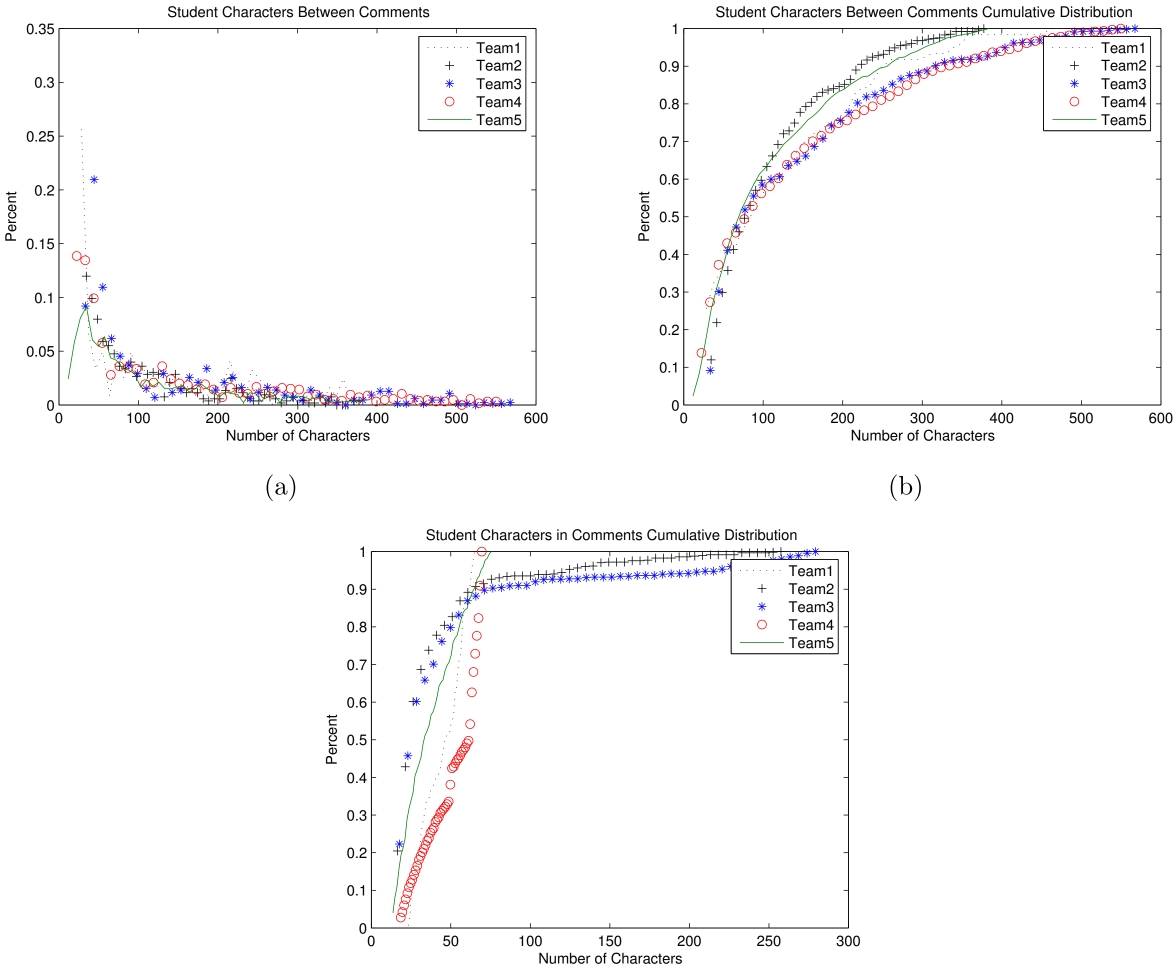
Table 6: Code and Comment Lengths of the Teams

Table 6 shows that most of the groups have very close medians to each other. Team4 is slightly different from the other teams because team 4 used existing FL and OpenGL libraries to create a 3-dimensional environment. Since these were not written by the teams themselves the result might be skewed. These standard libraries include many comments in order to make them easily understood and modified. Even without FL and OpenGL libraries, the median and mean lengths of code between comments are still longer than the means and medians found for the open source projects. Similarly, comment lengths were generally shorter for student code. This suggests that many student projects should use more commenting to be more easily understood by both future students and teachers.

Team	Global Variables	Globals*	Numerals	chars	Total LOC	Ratio	Ratio*
team 1	5	3	0	0	10857	2171	3691
team 2	37	36	17	2	18700	505	519
team 3					10692		
team 4	210	176	22	0	21807	103	123
team 5	152	91	20	0	15782	103	173

Table 7: Global variable usage across teams

Table 7 shows the usage of global variables. Four teams use global variables except the third team, which used Java. Java does not support global variables. The fourth team used GL libraries and they contained 49 global variables, including 24 constants. Compared with Table 5, the students' programs have comparable usages of global variables. Similar results were also found about function caller-callee relationship, as indicated in Table 8.



(a)

(b)

(c)

Figure 4: Student's programs: (a) Distributions of code lengths. (b) Cumulative distribution of code lengths. (c) Cumulative distribution of comment lengths. The shortest and longest 10% have been removed.

Limitations of This Study

This study selects 6 widely used open-source projects as the samples of high-quality software. A widely used program does not necessarily have high quality even though it is open-source. All projects studied in this paper can be classified as "tools for other programmers", not programs for end users. Many of them include sample inputs for configuration or testing. For example, emacs has 759 Lisp files (with .el suffix) for user configuration. Our study does not include them because the programming language is substantially different from C/C++ studied here. Similarly, there are 1606 python files (`.py` suffix) in the python package and these python files are not studied. Because these 6 projects are considered as tools, many

program	Callers	Callees	Ratio	Single Callee	Maximum Callees	Second Maximum
team1	12	39	3.25	30.7%	11	9
team2	61	187	3.07	32.6%	26	12
team3	54	193	3.57	27.9%	22	19
team4	58	144	2.48	40.2%	30	9
team5	42	85	2.02	49.4%	11	11

Table 8: Function callers and respective number of callees.

users may not modify the C/C++ source code. A user may download python, compile it, and then start writing python code without reading the implementation of the python language. We believe this is not a serious problem in our study because all projects are updated sufficiently often by multiple developers. We rely on doxygen to determine the caller/callee relationship. Doxygen, however, cannot handle function pointers. Hence, some calling relationships are not captured because the callers use function pointers. One direction of our future work is to handle function pointers correctly. Due to the large number of files we study, we cannot manually select remove every machine-generated files so most of them are included in this study. Some machine-generated files have been discovered because they have exceptional characteristics; hence, these remaining files should not significantly distort the data.

Conclusion

This paper provides quantitative analysis of 6 popular open-source programs of over 3.27 million lines to discover their commonalities. We find strong similarities in the sizes of codes between comments and the sizes of comments between codes. Most files are only a few hundred lines. Most functions call only a few callees and global variables are widely used in these projects. This study provides a set of characteristics that can be common in successful software. The data provide a basis for future quantitative study of software quality.

Acknowledgements

Evan Zelkowitz was supported by Purdue SURF (Summer Undergraduate Research Fellowship) in summer 2005. Prof. Lu is supported in part by National Science Foundation CAREER CNS-0347466. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors."

References

[1] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8):491–506, August 2004.

[2] V. R. Basili and B. T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[3] A. Capiluppi. Models for the Evolution of OS Projects. In *International Conference on Software Maintenance*, pages 65–74, 2003.

[4] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of Open Source Projects. In *European Conference on Software Maintenance and Reengineering*, pages 317–327, 2003.

[5] T. Dinh-Trong and J. M. Bieman. Open Source Software Development: A Case Study of FreeBSD. In *International Symposium on Software Metrics*, pages 96–105, 2004.

[6] R. Ferenc, I. Siket, and T. Gyimothy. Extracting Facts from Open Source Software. In *IEEE International Conference on Software Maintenance*, pages 60–69, 2004.

[7] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *International Conference on Software Maintenance*, pages 131–142, 2000.

[8] D. Hamlet and J. Maybee. *The Engineering of Software* . Addison Wesley, 2001.

[9] J. Lakos. *Large Scale C++ Software Design*. Addison Wesley, 1996.

[10] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[11] A. Mockus, R. T. Fielding, and J. Herbsleb. A Case Study of Open Source Software Development: the Apache Server. In *International Conference on Software Engineering*, pages 263–272, 2000.

[12] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two Case Studies of Open Source Software Develpment: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[13] J. W. Paulson, G. Succi, and A. Eberlein. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering*, 30(4):246–256, April 2004.

[14] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying Error-Prone Software - An Empirical Study. *IEEE Transactions on Software Engineering*, 11(4):317–324, April 1985.

[15] J. A. Whittaker and J. M. Voas. 50 Years of Software: Key Principles for Quality. *IT Professional*, 4(6):28–35, November 2002.

[16] C. Withrow. Error Density and Size in Ada Software. *IEEE Software*, 7(1):26–30, January 1990.

[17] Y. Xie and D. Engler. Using Redundancies to Find Errors. *IEEE Transactions on Software Engineering*, 29(10):915–928, October 2003.

[18] L. Yu, S. R. Schach, K. Chen, and J. Offutt. Categorization of Common Coupling and Its Application to the Maintainability of the Linux Kernel. *IEEE Transactions on Software Engineering*, 30(10):694–706, October 2004.