Measuring Software Evolution with Changing Lines of Code

Nikolaus Baer
Research Engineer
Zeidman Consulting
15565 Swiss Creek Lane
Cupertino, CA 95014 USA
phone: 650-832-1797
fax: 408-741-5231
Nik@ZeidmanConsulting.com

Robert Zeidman
President
Zeidman Consulting
15565 Swiss Creek Lane
Cupertino, CA 95014 USA
phone: 408-741-5809
fax: 408-741-5231
Bob@ZeidmanConsulting.com

ABSTRACT

A standard method for quantitatively measuring the evolution of software and the intellectual property it represents is needed. Traditionally, the evolution of software systems has been subjectively measured by counting the addition of new architectural elements or by comparing source code metrics. An architectural analysis is a subjective measurement technique, as each element must be weighed by importance and difficulty of implementation. This method also requires a complete understanding of the architecture, which is not always readily available. Traditional quantitative source code metrics are designed to evaluate static code, and do not properly capture the dynamic changes of code as it evolves. These hurdles in traditional software analysis necessitate the development of a new quantitative method of evolution measurement. This method would also be useful for measuring the evolution of the intellectual property value of the source code. This paper demonstrates a method for measuring the evolution of source code by analyzing the number of lines of code that have been modified, added, or remain through subsequent versions. This new method of measuring the changing lines of code (CLOC) will be demonstrated by examining the evolution of three major open source projects: the Linux Kernel, Apache HTTP Server and Mozilla Firefox.

General Terms

Maintenance, Software Evolution, Source Code Metrics.

Keywords

CodeDiff, CodeSuite, Evolution, Intellectual Property, Metrics, Refactoring.

1. INTRODUCTION

Measuring the evolution of software is necessary in computer science. Every science needs to have metrics upon which to base the research and development. The rate of software evolution is important for: measuring the progress of large long-term software projects, evaluating the remaining value of the initial development as a project grows, improving the maintenances of code through better understanding of the evolutionary activities, performing "due diligence" before acquiring software or software companies, and measuring work done under contract.

There are many valuable source code metrics for measuring the size or complexity of a piece of software. However, there is no standard method for measuring software evolution. The traditional metrics provide valuable insight into the size and complexity of a piece of static code, but comparing these measurements over the course of a software project's life can be inconclusive. They were not developed to measure evolving programs.

The "changing lines of code method" (CLOC) has been designed to measure the kinetics of software evolution. This method analyzes the number of "lines of code" (LOC) that are modified, added, or remain constant through the life of a software project. By concentrating on the kinetic activities that occur between subsequent versions of a software project, instead of examining static snapshots of size or complexity the CLOC method properly captures the evolution of software.

2. MAINTENANCE & DEVELOPMENT

Measuring the evolution of source code is equivalent to measuring the amount of maintenance and development performed on an application. Maintenance and development are the kinetic activities that cause source code evolution. Proper maintenance and development prevents software from wearing out, and should increase the value of the software [1].

The percentage of original code inside the entire software project will diminish as maintenance and development proceeds. Some portions of code will be changed to fix problems, some will become outdated and removed, and some will be refactored to improve the readability or to simplify the structure. A software project will also have enhancements, which will add code. As a rough estimate, the value of a portion of software is proportional to its percentage of the whole project.

3. EXISTING METRICS

There are several common software metrics, including: lines of source code, cyclomatic complexity (also known as the McCabe measure), function point analysis, bugs per line of code, and code coverage [2]. Each method is designed to quantify the amount of work involved in producing or maintaining software. Each method quantifies a single software characteristic in an attempt to measure the software as a whole. Some of these methods such as cyclomatic complexity focus on the complexity of the software, and some such as lines of source code focus on the size of the software. Unfortunately, none of these methods provide a useful measurement of the effort involved in changing the software.

3.1 Cyclomatic Complexity

In 1976 Thomas McCabe proposed a complexity measure known as cyclomatic complexity ("CC") [3]. This method counts the number of individual paths through the code. For instance, a single IF statement would count as two distinct paths through the code: the first path represents the statement being "True" and a second path for "False." CC can be calculated by creating graphs of source code and counting the execution paths. This can be a tedious process, as just a few lines of code can result in many distinct paths. Software tools such as Understand from Scientific Toolworks, Inc. have been developed with the capability to provide CC measurements [4].

The complexity of the code is not a clear indicator of source code evolution. There are several forms of source code maintenance that may simplify the execution paths of a program. The fixing of bugs, deletion of outdated code, and the refactoring of code can all take a significant amount of work and have a significant impact on the functionality of a program, but may decrease or have no effect on the CC measurement.

3.2 Lines of Code

The simplest way to measure software is to count the source lines of code to determine the size of the software ("SLOC"). Although the SLOC metric does not take into account the complexity of the code, it is a good metric for measuring the effort involved in the production of code. Typically a larger SLOC measurement means there was more effort involved.

The SLOC metric is simple to measure; it is less subjective and it correlates well with effort and programming productivity. It has a stronger correlation to the work involved with building software than other measurement techniques [5 p. 119]. However, comparing SLOC values of subsequent versions is not a perfect measurement of the evolution of source code. It does not properly measure the efforts involved in refactoring, debugging or trimming existing source code. SLOC equates productivity to the development of more code and is inaccurate when the activities involve deletion or alteration of code [6 p. 18]. As stated by Bill Gates, "Measuring programming progress by lines of code is like measuring aircraft building progress by weight" [7]. The SLOC metric is valuable, but comparing SLOC measurements between versions does not properly measure the progress of normal software maintenance and development.

4. CHANGING LINES OF CODE

This paper demonstrates the measurement of the changing lines of code (CLOC) metric. A useful software evolution measurement must include an analysis of the lines that were changed or removed. It examines the changes involved instead of the static size or complexity. The CLOC method counts the number of lines of code that have been modified, added, or remain constant between subsequent versions of a software project. These measurements are then analyzed to determine the percentage growth of the software. Software evolution and the CLOC method can be shown as either the growth of the software or the decay of the original source code. The CLOC method properly measures the intricacies of source code maintenance and provides a quantitative metric for software evolution.

5. SETUP

We chose to demonstrate the CLOC technique through the investigation of open-source projects. We chose three different projects, to have a wider data set and to not be restricted to anomalies in a single project. In order for us to use a project it had to be primarily written in C and/or C++, have at least 4 major versions over at least 5 years, and be sufficiently complex. Three well known open-source projects that fit these qualifications were the Linux Kernel, the Apache HTTP Server and the Mozilla Firefox browser.

The major releases of each open source project were downloaded and analyzed. For the Linux Kernel it was determined that the major releases spanning from 1994 until 2003 are: 1.0, 1.2, 2.0, 2.2, 2.4, and 2.6 [8]. These versions were downloaded from www.kernel.org . The major Apache Http Server versions are: 1.3.0, 1.3.41, 2.0.35, 2.0.63, 2.2.9 [9]. The Apache source was downloaded from http://httpd.apache.org/download.cgi. The Mozilla Firefox project started in 2002 and the major versions that we chose to examine are: 0.1, 0.8, 1.0, 1.5, 2.0, and 3.0 [10]. The source code was downloaded from ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/.

5.1 Expected Growth

For comparison, the CLOC measurements will be compared to a calculated expected growth rate. We decided to use the expected growth rate suggested by David Roux, who said that each version of a software project (“V_n”) grows by the size of the initial software release (“V_0”) [11 p. 13].

$$\mathbf{V_n = V_{n-1} + V_0}$$

5.2 CodeSuite®/CodeDiff®

The CLOC method needs a way to measure the differences between files. We selected the CodeDiff® tool, which is part of the CodeSuite® program from Software Analysis & Forensic Engineering Corporation (S.A.F.E.), to perform the measurements. This application provides the ability to quickly compare the lines of code between different directories. It produces detailed reports on the differences between each version of each file for an entire project. We placed the different versions of each project into individual directories and then compared the directories. CodeDiff produced HTML reports of the differences as well as statistics on the changes between the project versions.

The CLOC method is distinguished from other examinations by framing the analysis around the differences between versions. CodeDiff allows us to distinguish the changes in non-blank LOC between versions. Non-blank LOC are those lines that are either statements or comments, but not blank lines. This eliminates the inaccuracies of counting empty lines while retaining the value of comments.

We did several CodeDiff analyses to generate the datasets for further statistical examination. We limited the CodeDiff comparison to only compare the lines of files with the same name. Typically file names are not changed from version to version, and a movement of source code between files or a file name change represents work being performed. CodeDiff was configured to compare the projects as C and C++ source code.

The Firefox comparisons each involved well over 200MB of source code. Even with the speed of the fast CodeDiff algorithms, these comparisons would have took a considerable amount of time. To efficiently compare such a large amount of code we used the CodeGrid® computer grid from S.A.F.E. Corp. to split the CodeDiff processing of the Firefox versions across four machines simultaneously.

5.2.1 Intellectual Property

CLOC can provide insight into how the original intellectual property (IP) continues through subsequent versions. We obtained the earliest version of source code available from each software project and deemed it to be the “original version.” We used CodeDiff to perform the comparisons between the original and each subsequent version of the particular project. This enabled us to determine how many files and LOC in the original version remain in each subsequent version. Although IP value is not directly related to measuring lines of code, the CLOC measurement can be useful for calculating the value of the original IP relative to value of the IP in the current software version.

5.2.2 Non-Header files

Header files typically contain definitions, declarations, and simple macros, but not a significant amount of functional code. Header files can be added to a project to provide declarations for third-party library functions that are not part of the protectable intellectual property of a project. We generated a second set of results based upon non-header files (source code files other than header files), which we expected to have a higher rate of change.

5.2.3 Removing Duplicates

One artifact of CodeDiff that had to be compensated for was the existence of duplicate matches. For our purposes, file A “matches” file B if they have the same name and no other file has a higher percentage of LOC in common. CodeDiff reports every file that can be considered a match, meaning that it will report that file A matches file B and file C matches file B, so file B is considered as matching two different files. We decided that each file in the initial version should have at most one true match in a subsequent version. We developed a post-process utility that searched the database created by CodeDiff, found matching file pairs, eliminated the lowest percentage duplicates, and then re-matched files. This process continued until all files were uniquely matched or could not be matched.

6. RESULTS

After each comparison was processed, the results were entered into spreadsheets that contained formulas for the CLOC calculations. We generated the number of new LOC, counting changed lines as new lines. The process also provided the total and percentage of original lines and files that continued in each subsequent version.

We also generated the traditional metrics for comparison. We used Understand from SciTools, Inc. to measure the total cyclomatic complexity (TCC) of each project. The SLOC of each version was provided by the CodeSuite analysis. These measurements were compared against the new CLOC method.

6.1 Average Results

The results from each of the three projects were combined into average percentages as shown in Figure 1. We averaged the percentages calculated for each software project instead of the actual number of lines in order to compensate for the varied sizes and characteristics of each project. The observed rate of evolution starts lower than the calculated expected value and then increases to a higher rate than calculated. The average observed rate seems to follow an exponential growth trend, whereas the calculated expected rate is linear. There are many models of software growth rate that are based upon exponential formulas, which our observed growth rate indicates may be more accurate [11].

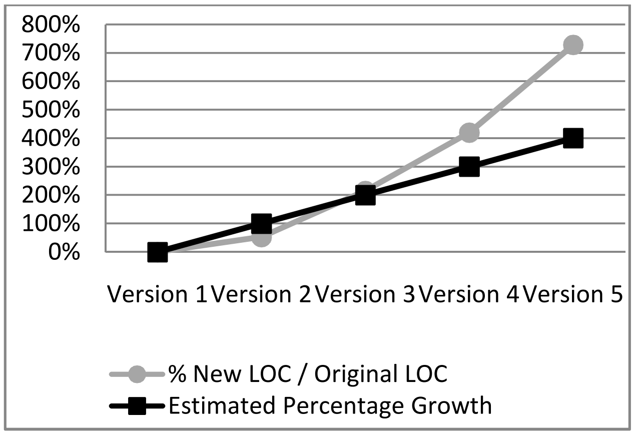

Figure 1: Average CLOC and Estimated Average Growth from Linux, Apache and Firefox

The CLOC measurements can also be shown as an average percentage of continuing code through each subsequent version. The decay of original code is shown in Figure 2. Both the percentage of remaining LOC and the percentage of remaining files are displayed. The two data series are very close, which shows that on average the majority of new content was added into new files.

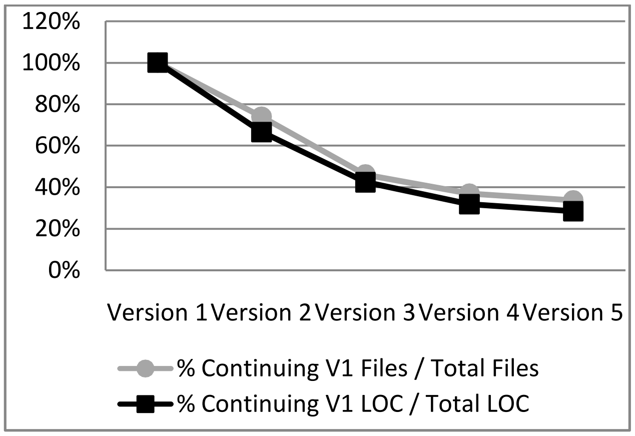

Figure 2: Average percent continuing

6.2 Apache Server Results

The Apache HTTP Server analysis involved comparing version 1.3.0 against versions 1.3.41, 2.0.35, 2.0.63, and 2.2.9. The CLOC growth measurements can be seen in Figure 3. The project grew from 150 files to 809 files. Figure 4 shows that eight percent of the original files were found in the final version, and six percent of the original LOC continued throughout all of the versions. This project's growth closely followed the calculated expected growth rate.

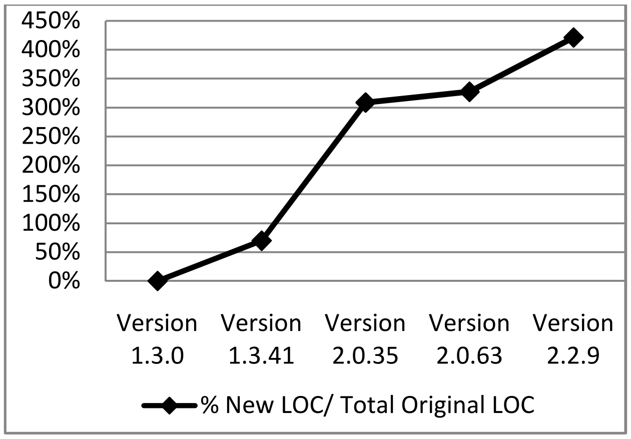

Figure 3: Apache new LOC

The data series in Figure 4 show the decay of original code as measured by the CLOC method as well as the traditional software metrics. The percentage of remaining code is shown to be higher than the percentage of continuing code when the simpler SLOC is used. This is consistent with the short-coming of the SLOC comparison method discussed earlier; the method does not adjust for the change or removal of original code.

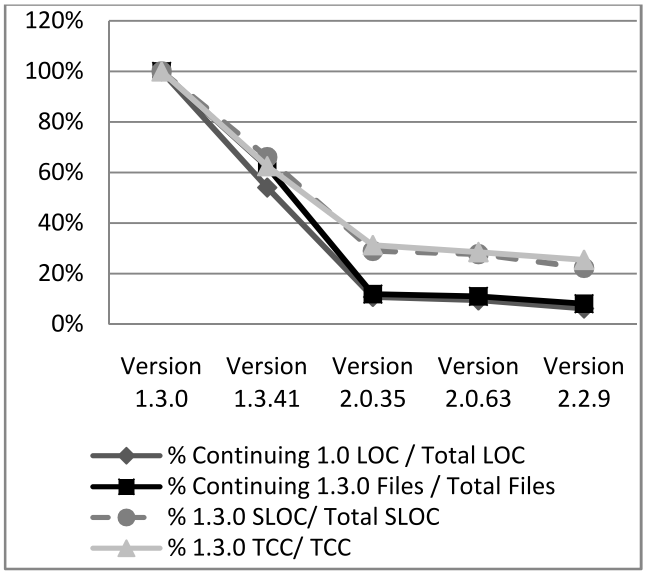

Figure 4: The percentage of continuing LOC, files, SLOC and TCC in subsequent Apache releases

The change in the total cyclomatic complexity is also shown in Figure 4. The TCC evolution measurement shows less decay and thereby less evolution than when measured with the CLOC method, but it is a small difference.

The measurements were similar when header files were excluded. The average difference between the data for all files and the data for non-header files exclusively was only 1%. The Apache project did have a larger rate of original code decay when header files were excluded, which indicates that the LOC and original code changed more in the non-header files than in the header files.

6.3 Mozilla Firefox Results

The data for the Firefox browser was composed of comparisons between versions 0.1 and the subsequent versions: 0.8, 1.0, 1.5, 2.0, and 3.0. A large 75% of the original 0.1 files remained in the last version, as well as 53% of the original LOC. Although decay in the original code was detected, it was inconsistent. Version 1.0 actually had 3% more of the original LOC than the earlier version 0.8. The growth was also inconsistent; the total SLOC fluctuated instead of consistently increase as expected.

The percentage of SLOC in Figure 5 is a drastic demonstration of the possible idiosyncrasies of using SLOC to measure source code evolution. The graph in Figure 5 would indicate that version 3.0 of the Firefox browser was 96% continuing code, because the size had not changed much. It is unlikely that the maintenance and development of the popular Internet browser over five major versions would only result in 4% new code. The 47% new code measured by the CLOC method is clearly a more accurate measure of the source code evolution.

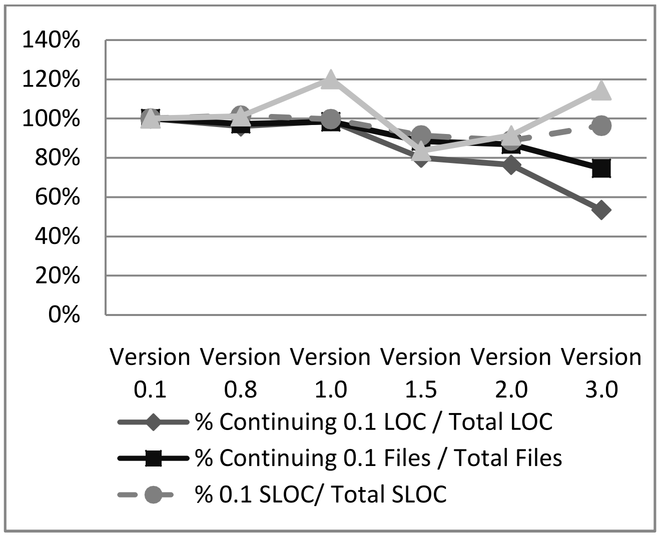

Figure 5: The percentage of continuing LOC, files, SLOC and TCC in subsequent Firefox releases

Figure 5 also demonstrates the problem with using complexity as a measure of software evolution. The subsequent versions 0.8 and 1.0 have lower levels of TCC than the original version. Although there are many parts of source code maintenance that can cause this, a comparison of these TCC measurements incorrectly indicates that reverse evolution occurred. This inconsistency shows that comparing complexities is not a reliable method for measuring software evolution.

The CLOC method is the only measurement that shows a consistent evolution of the Firefox project, as well as a consistent decay of original code. The other metrics produced inaccurate results.

6.4 Linux Kernel Results

The data on the Linux Kernel was compiled by comparing version 1.0 against 1.2, 2.01, 2.2.0, 2.4.0 and 2.6.0. The project grew from 487 files to 12,412 files. As shown in Figure 7 below, only 3% of the final files were continuing files from version 1.0, and 1% of the LOC in version 2.6.0 were from version 1.0.

This project showed the greatest growth and the most consistent decay of original code. Figure 6 shows that by release 2.2.0 the growth of the Kernel began to outpace the calculated estimated software growth. By version 2.6.0 the size of the code was over four times that of the estimated size.

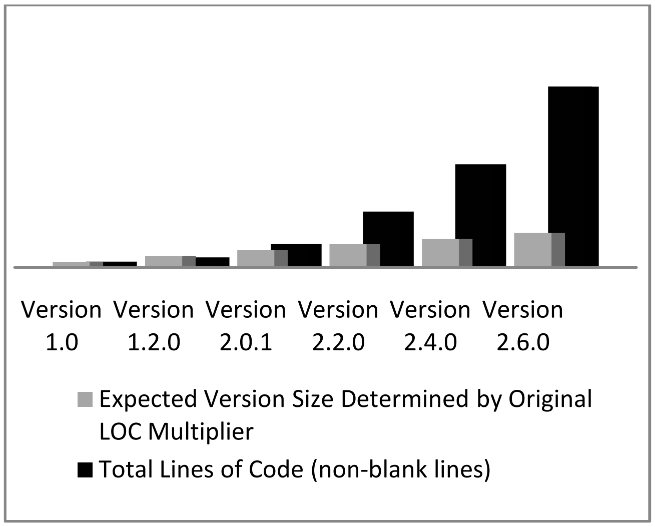

Figure 6: The calculated vs. actual growth of the Linux Kernel

The average difference between the data for all files and the data for non-header files only was only 1%. This is a small difference, but it does confirm that the source code in the non-header files did change more than in the header files.

The data series in Figure 7 also show the change in SLOC and TCC for each version. We believe the rapid rate of code growth in the Linux Kernel causes these metrics to actually follow the trend of the more accurate CLOC measurements and masks the inaccuracies of the traditional metrics that were apparent in the other project comparisons.

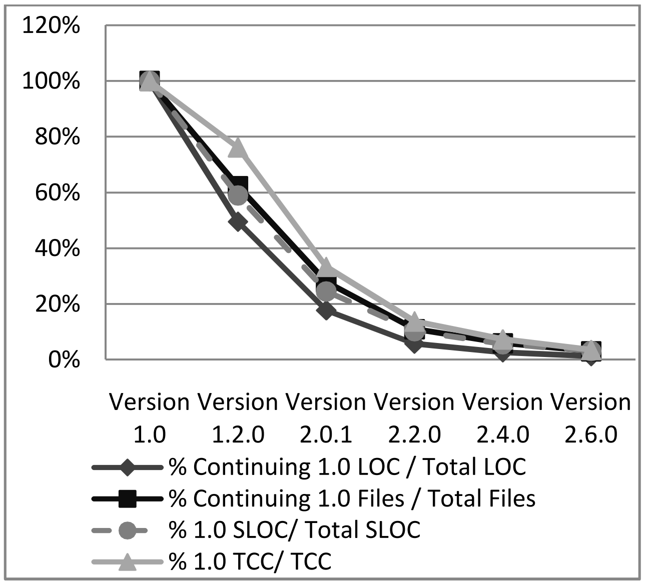

Figure 7: The percentage of continuing LOC, files, SLOC and TCC in subsequent Linux Kernel releases

7. AREAS FOR FUTURE WORK

There are a number of interesting areas for future work in this field. Another program from S.A.F.E. called CodeMatch® could be used to exclude comments from the CLOC method to determine if there is a difference in results when only functional statements are examined.

The CLOC method has been shown to be an accurate quantitative measurement of software evolution, so it would be very interesting to test the different rules that have been proposed regarding sustainable software development and expected software growth. The test could be set-up in a manner similar to this paper, in that additional open-source projects could be evaluated and the average results compared to the various rules. Once software growth and sustainability rules are vetted by a large scale CLOC test, individual projects could be analyzed against the newly verified rules.

It would also be interesting to use the CLOC method to study the value of software and software growth. Further studies could investigate how the monetary value of the original software and intellectual property changes as the software project evolves. The CLOC method could also be used to examine possible correlations between software growth and market value growth.

8. CONCLUSION

The measurement of source code evolution by analyzing the number of LOC that have been modified, added, or remain through different versions of a software project has been demonstrated through the CLOC method. The use of CodeDiff allows for the precise measurement of how the source code has changed. By examining the differences instead of the SLOC, the evolution of the source code is better understood. The first main advantage is the ability to measure how many changed LOC exist. The CLOC metric is more representative of the growth because it takes refactoring, and deletion of code into account. Through the CLOC method, the percent of continuing files and LOC in each subsequent release can be clearly measured. This measurement of remaining source code represents the original code that continues throughout the project as the project evolves. The final advantage of this method is that it is simple and quantitative. By measuring the changes in LOC between versions, it does not rely on any subjective metrics. The CLOC method's advantages of precision and objectivity set a new standard for quantitatively measuring source code evolution.

ACKNOWLEDGEMENTS

The authors would like to thank Tim Hoehn of Software Analysis & Forensic Engineering Corporation who ran large CodeDiff analyses on the SAFE CodeGrid system.

REFERENCES

[1] **Spolsky, Joel.** *Joel on Software.* New York : Apress, 2004.

[2] Software Metrics. *SQL Software Quality Assurance.* [Online] [Cited: 10 02, 2008.] http://www.sqa.net/softwarequalitymetrics.html.

[3] **McCabe, Thomas J.** *A Complexity Measure.* December 1976, IEEE Transactions on Software Engineering, Vols. SE-2, No. 4, pp. 308-320.

[4] **KevinG.** Documentation: How to get metrics with Understand 2.0. *SciTools.* [Online] July 11, 2008. [Cited: October 2, 2008.] http://www.scitools.com/blog/2008/07/metrics-galore.html.

[5] **Park, Robert E.** *Software Size Measurement: A Framework for Counting Source Statements.* Pittsburgh : Software Engineering Institute: Carnegie Mellon University, 1992.

[6] **Jones, Capers.** *Programming Productivity.* San Francisco : McGraw-Hill Publishing Company, 1986.

[7] Source Lines of Code. *Wikipedia.* [Online] [Cited: October 2, 2008.] http://en.wikipedia.org/wiki/Source_lines_of_code.

[8] Linux kernel. *Wikipedia.* [Online] [Cited: October 3, 2008.] http://en.wikipedia.org/wiki/Freax#Stable_version_history.

[9] **Boyd, Summer, et al.** The Apache Web Server: an Open Source Project.

[10] Mozilla Firefox. *Wikipedia.* [Online] [Cited: July 25, 2008.] http://en.wikipedia.org/wiki/Firefox.

[11] **Wiederhold, Gio.** What is Your Software Worth? [Online] June 19, 2006. [Cited: August 12, 2008.] http://infolab.stanford.edu/pub/gio/2006/worth30.pdf.

[12] **Howard, Daniel.** Source code directories overview. *Mozilla.org.* [Online] June 2, 2005. [Cited: August 12, 2008.] http://www.mozilla.org/docs/source-directories-overview.html.

[13] Project metrics Help. *Aivosto.com.* [Online] [Cited: 11 14, 2008.] http://www.aivosto.com/project/help/pm-complexity.html.

[14] **Greg Kroah-Hartman, Jonathan Corbet, Amanda McPherson.** Linux Kernel Development How Fast it is Going, Who is Doing It, What They are Doing, and Who. *www.novell.com.* [Online] March 2008. [Cited: August 12, 2008.] http://www.novell.com/rc/docrepository/public/37/basedocument.2008-05-13.7066315223/Who_Writes_Linux_en.pdf.

[15] Index of ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/. *Mozilla.org.* [Online] [Cited: July 25, 2008.] ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/.

[16] Index of /pub/linux/kernel. *Kernel.org.* [Online] [Cited: July 24, 2008.] http://www.kernel.org/pub/linux/kernel/.

[17] Downloading the Apache HTTP Server. *Apache HTTP Server Project.* [Online] [Cited: July 24, 2008.] http://httpd.apache.org/download.cgi.

AUTHOR BIOS

Nikolaus Baer is a research engineer at Zeidman Consulting. He has developed software for marine research, optical testing equipment, military terrain databases, mobile applications, and medical devices. He has written about software trade secret theft and is certified in the use of CodeSuite. He holds a BS degree in computer engineering from UC Santa Barbara, where he attended on a Regents Scholarship. He placed first in the Start Cup 2004 business plan competition.

Robert Zeidman is a Senior Member of the IEEE and President of Zeidman Consulting (www.ZeidmanConsulting.com) a contract research and development firm. Since 1983, he has designed ASICs, FPGAs, and PC boards for RISC-based parallel processor systems, laser printers, network switches and routers, and other real time systems. Among his publications are technical papers on hardware and software design methods as well as three textbooks -- *Designing with FPGAs and CPLDs*, *Verilog Designer's Library*, and *Introduction to Verilog*. He has taught courses at engineering conferences throughout the world. Bob holds three patents and earned an MS degree in electrical engineering at Stanford University and BS degrees in physics and electrical engineering at Cornell University.