

Using a Line-of-Code Metric to Understand Software Rework

Edmund P. Morozoff, *Medtronic*

A simple method measuring new effective lines of code showed that between 19 and 40 percent of code written on three projects wasn't in the final release.

Generally, productivity is a function of input effort and output size. A strong understanding of software productivity, coupled with a good estimate of software size, is key to predicting project effort and, ultimately, producing reliable project duration estimates, schedules, and resource needs. Project managers and engineers often measure or predict the size of released software—the volume of software in the marketed product. However, the final release doesn't include reworked code—code that was changed or deleted during development.

Software project cost estimations should consider the potential substantial effort associated with reworked code. Richard E. Fairley and Mary Jane Wilshire describe various types of rework—evolutionary, avoidable retrospective, and avoidable corrective—and how each affects productivity and quality.¹ They state that the quantity of rework can range from 20 to 80 percent of the total effort. Software organizations can measure rework by the effort to alter, revise, or reshape code. Without these effort measurements, they can alternatively gain insight into the size or impact of effort by measuring the volume of reworked product or reworked source code.

Lawrence H. Putnam and Ware Meyers,² Capers Jones,³ and Barbara Kitchenham and Emilia Mendes⁴ indirectly consider reworked code in software cost or size estimation exercises through the concepts of defect removal, maintenance, or reuse. However, they generally apply their input measures toward estimating the released software

size, not the total written during development. In other words, they estimate the delivered software size as a function of added software, initial (that is, reused) software, and software changed owing to maintenance or defect fixes. In the Cocomo II effort estimation model, Barry Boehm and his colleagues describe how to accommodate rework by adjusting both size and productivity measures.⁵

I've developed a simple automated method to measure lines of code and analyze reworked software during a project's development. Using the volume of reworked code as an indicator of total project rework assumes that the effort to revise code (detailed design, detailed design review, coding, code review, unit test, unit test review, integration test, and integration test review) represents a major proportion of the project's total rework effort. I base this assumption on analysis of unplanned tasks during development, a primary source of rework that often leads to code changes. However, a project's total rework also includes effort to change

code that this method doesn't capture—specifically, test harnesses, prototypes, and other infrastructure that isn't included in the build. There's also rework that doesn't result in code change—for example, changes to requirements, architecture, and high-level design done before developers write any code.

Background

Effective lines of code (eLOC) is a simple software size metric that captures the amount of nonblank or noncomment lines of code. Although this metric might not be useful for comparing projects across organizations and industry, the embedded software projects of specific departments tend to follow the same process and produce similar products. Considering these factors and that the scope is limited to a single development group, eLOC is a useful measure of embedded software project size for this analysis.

The size of code developed, known as *new effective lines of code* (nLOC), includes eLOC added, deleted, or changed during the project's development process. Boehm uses a similar definition in his Cocomo II maintenance model where he considers added and modified LOC in estimating software size.⁵ However, he doesn't include deleted lines of code. Ranjith Purushothaman and Dewayne E. Perry consider added, changed, and deleted lines of code in their research associated with the impact of small changes.⁶ I include deleted lines of code in the nLOC definition because the process of removing code is identical to the process of adding code, in that both require detailed design, detailed design review, coding, code review, unit testing, and integration testing.

Putnam and Myers approach the concept of reworked code by considering code reuse.² They describe the change-sizing estimation method, in which new, changed, and deleted source lines of code are a subset of reused code. However, this approach focuses primarily on maintenance activities on an existing code base. Putnam doesn't consider that even when an organization develops an entirely new software product, lines of source code will be added, deleted, and changed—that is, reworked—throughout development. In other words, the change-sizing estimation method focuses primarily on estimating a software size that's a function of changes to the initial code base plus any added code—omitting changes that occur during development.

Jones recognizes the cost of finding and repairing defects as the most expensive and time-consuming work of software development.³ This

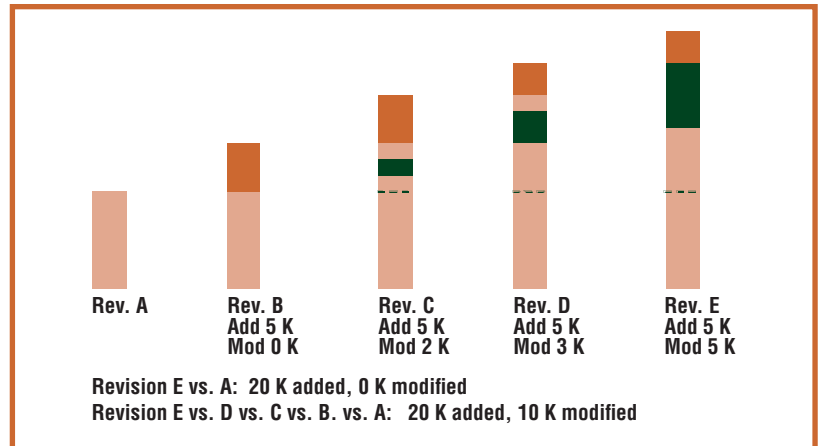


Figure 1. Calculating reworked lines of code in software builds. Frequently sampled builds provide insight into inter-build development. Sampling only between initial and final builds omits reworked code.

activity could be classified as avoidable rework; Jones addresses it by including defect estimation methods coupled with various removal activities. Rather than adjust the size estimate, he adjusts the effort estimates.

Although some of these authors don't directly include reworked code in their sizing estimates, they've accounted for the impact of rework through other productivity-related parameters such as defect volume, defect removal efficiency, and programmer capability. This article investigates software rework at the implementation level as a means of understanding its magnitude and implications on project effort estimation.

Measuring the software's final size doesn't quantify how much code was actually written during development. For example, Figure 1 illustrates the progression of an imaginary project through various releases. The light orange part of each column represents the code base used from the previous release, dark orange represents code added to the previous release, and green represents code from the previous release that was modified (changed or deleted). For simplicity, this example has no modifications to code from the initial release (revision A).

Increasing the frequency of sampling the software revisions provides improved insight into the amount of code the team writes. For example, comparing only the first and final revisions would make it appear that the team added 20 thousand eLOC (KeLOC) to the project with no modification. Alternatively, measuring at a higher frequency, say A to B to C to D to E (see Figure 1), shows that the team added 20 KeLOC and modified 10 KeLOC. So, the team wrote 30 KeLOC but included only 20 KeLOC in the final product. Because there's no

A Note on Lines of Code

Some studies have shown that lines of code (LOC) is an unreliable metric for comparing software projects of varying languages and industries.¹ Joseph Schofield also challenges LOC metrics reliability: even under conditions in which developers create programs given the same set of requirements.² Other researchers, however, have found value in LOC as a size metric when you understand and recognize its limitations.^{3–6} Barbara Kitchenham and Emilia Mendes' work on software productivity illustrates that you can use various measures of size for the same software product for productivity estimates if they're significantly related to effort.⁷

To meet the need for a robust, comparable size metric, Allan J. Albrecht developed another measure of program size, the *function point*, which has gained acceptance.⁸ He calculates it using a standardized method based on the number and complexity of external inputs, external outputs, external queries, internal logical files, and external interface files (see www.ifpug.org). Various researchers have developed similar sizing methods; Jones presents an excellent summary of these techniques.¹ The function-point method includes a technique to accommodate change by calculating enhancement function points. You could use this method after each build to account for deleted, modified, and added functionality. Because this methodology has limited automation, it might prove impractical for frequently determining the enhancement function points. Furthermore, if the changes are more related to implementation—for example, performance improvements or refactoring⁹—there might not be a corresponding difference in the total number of function points. Alternatively, using lines of code as a measure decisively determines which lines developers have added, deleted, and changed, providing further insight into the volume of rework.

References

1. C. Jones, *Estimating Software Costs: Bringing Realism to Estimating*, McGraw-Hill, 2007.
2. J. Schofield, "The Statistically Unreliable Nature of Lines of Code," *CrossTalk*, Apr. 2005, pp. 29–33.
3. L.H. Putnam and W. Myers, *Measures for Excellence: Reliable Software on Time, within Budget*, Prentice-Hall, 1992.
4. B. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice-Hall, 2000.
5. J. Dolando, "A Validation of the Component-Based Method for Software Size Estimation," *IEEE Trans. Software Eng.*, vol. 26, no. 10, 2000, pp. 1006–1021.
6. J. Verner and G. Tate, "A Software Size Model," *IEEE Trans. Software Eng.*, vol. 18, no. 4, 1992, pp. 265–278.
7. B. Kitchenham and E. Mendes, "Software Productivity Measurement Using Multiple Size Measures," *IEEE Trans. Software Eng.*, vol. 30, no. 12, 2004, pp. 1023–1035.
8. A.J. Albrecht and J.E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Eng.*, vol. 9, no. 6, 1983, pp. 639–648.
9. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

modified code between the first and last release, the reworked code size is 10 KeLOC. Thus, of the 30 KeLOC that the team wrote, 33 percent was rework.

In this article, I measure and accumulate the volume of new lines of code, on average, from bi-weekly builds over the development life cycle of three embedded software projects to understand each project's volume of rework on the basis of six factors (see the sidebar "A Note on Lines of Code").

First, most of the team members worked on the various projects. So, each project team's skills, capabilities, and experience remained relatively constant. Second, the projects included the same domain and types of products, so they weren't adopting new technologies. Third, limited memory space placed an emphasis on code size constraints, so team members targeted dead code identified for removal. Fourth, team members derived the projects from similar platforms and used the same programming language. Fifth, the projects used similar development processes so activities such as reviews and unit testing were relatively similar. Finally, team members used standardized file and directory names to support automated data collection.

Methods

I created a tool that counted the eLOC in a C source file. It recognized both the standard C++ comments—that is, those starting with `"/"`—and traditional C comments—that is, those bracketed by `"/"` and `"/"`. The tool can compare two versions of a file and count nLOC.

The tool's comparison engine, a third-party file difference tool, can output the lines it identifies as different—that is, modified—into an ASCII file containing the original lines and the modified lines to let reviewers check the tool's validity. We applied two versions of a build containing 177 headers and 116 source files to the tool and manually reviewed the output.

They identified three limitations with the tool. First, the tool doesn't support file name changes because it wouldn't find a matching file for comparison. The development team didn't change file names and thus avoided this issue. Second, large changes to a file caused the comparison feature to lose its place. This issue occurred when developers made large changes in the middle of a file. To accommodate this limitation, the tool output an error message associated with the files being compared. Because developers placed most additions at the end of the files, few files (5 in 250) lost synchronization. Manual review of the files indicated that the modifications represented a small volume. Finally, moving code can create problems. Simply moving a function gives the appearance of modification, yet no functional change took place. Code reviews regulated this behavior.

Results

All three projects produced software for medical devices and were built from previous platforms, leveraging an existing code base. Team sizes peaked at about 100 developers and testers for the largest

Table 1**Volume of reworked code in three projects (KeLOC)**

	Project A	Project B	Project C
1. Initial size—thousands of effective lines of code (KeLOC)	49.2	51.6	38.6
2. Total added—KeLOC	43.3	8.4	12.9
3. Total modified—KeLOC	56.9	14.5	19.4
4. Total new—thousands of new lines of code (KnLOC)	100.2	22.9	32.3
5. Final size—KeLOC	92.5	60.0	51.5
6. Final vs. initial modified—KeLOC	16.5	6.5	13.3
7. Reworked code as net modified—KeLOC	40.4	8.0	6.1
8. Reworked code as a percentage of new	40	35	19

project and about 40 for each of the two smaller projects. Each project lasted from three to five years.

Table 1 lists the accumulated added, modified, and new lines of code for each project as measured biweekly. The tool subtracted the volume of modified code between the first and last release (row 6) from the total modified code (row 3) to create the net modified KeLOC (row 7). Then it calculated the percentage of reworked code (row 8) by dividing this net modified KeLOC (row 7) by the total KnLOC (row 4). So, the percentage of reworked code represents code written during the project and not included in the final release.

For example, comparing the results between project A's first and last release, I determined that 43.3 KeLOC were added (row 2) and 16.5 KeLOC were modified (row 6). This implied that the original release (that is, the starting code base) had 16.5 KeLOC modified during the project. Sampling the builds during project A's development showed that developers modified 56.9 KeLOC (row 3). Row 4 shows that the lines of code added and modified (that is, nLOC) during the project totals 100.2 KeLOC. In other words, at project A's completion, the code base had increased by 43.3 KeLOC. However, developers wrote 100.2 KeLOC (added or modified). Of this total code, they reworked 40.4 KeLOC (row 7).

Figure 2 illustrates the accumulation of added, modified, and total new code throughout project A. I normalized the time axis to the percentage of project duration. Figures 3 and 4 represent the same data for projects B and C.

Discussion

Data from the three projects indicate that more code is modified than added. This makes sense because each project's software was built on an existing

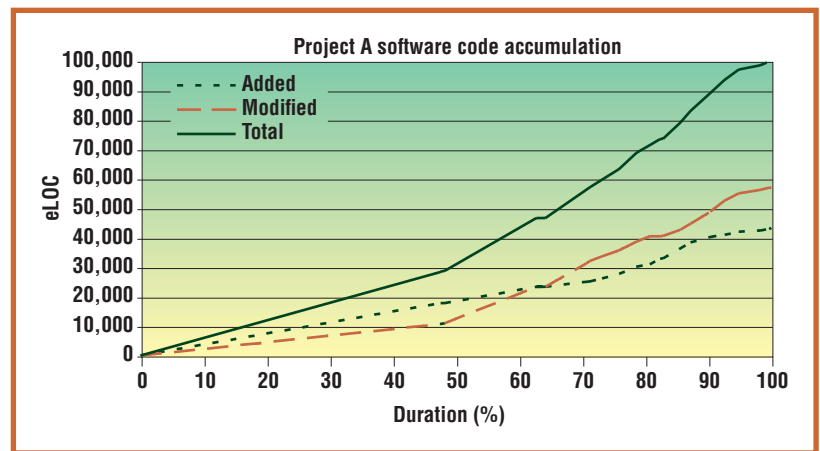


Figure 2. Project A effective lines of code (eLOC) accumulation. The largest of the three projects illustrates more code is modified than added.

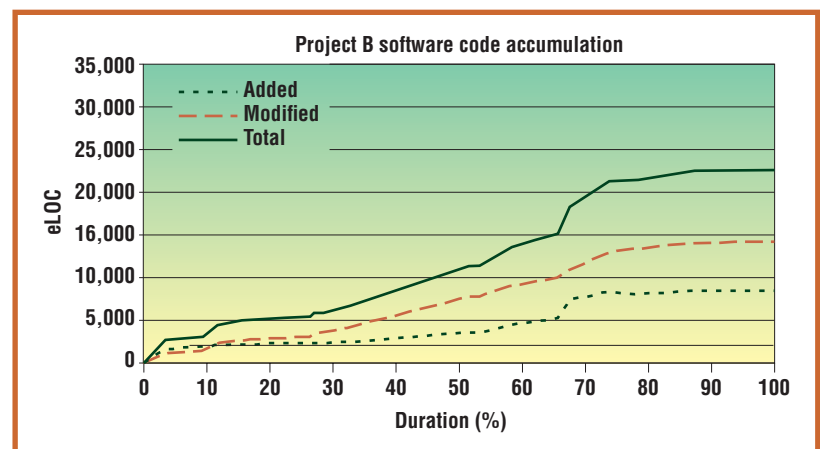


Figure 3. Project B effective lines of code (eLOC) accumulation. The smallest of the three projects has nearly the same percentage of reworked code as the largest—Project A.

platform and a previous product's code base. For the largest project (A), developers made most of the modifications on code that they added during the

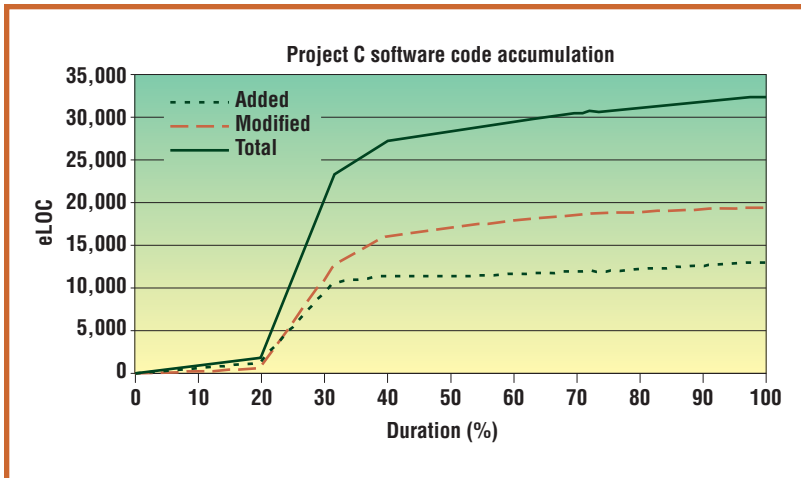


Figure 4. Project C effective lines of code (eLOC) accumulation. Most code was developed in a relatively short period of time and more was modified than added. It had about half the percentage of reworked code as the other two projects.

project—not on the initial platform. Project A had a total of 56.9 KeLOC modified code; 40.4 KeLOC of it (that is, 71 percent) was done on the code added to the project. The smallest project (B), had a total of 14.5 KeLOC modified code; 8.0 KeLOC (that is, 55 percent) was done on code added to the project.

Figure 2 shows that developers added and modified about 100 KeLOC during project A's development. It also demonstrates that they added the code at a relatively constant rate; however, the modification rate doubled halfway through development. Figure 3 shows similar development behavior for project B, in the middle of which the rate of code modification was about twice the rate at which code was added. Developers wrote most of Project C's software in a relatively short time, and the rate of code addition is generally the same as the rate of code modification. So, although Table 1 illustrates different volumes of reworked code, the plots can provide some insight into the team's software development behavior.

Projects A and B had a volume of discarded code roughly equal to the added code (See Table 1, rows 2 and 7). However, I also observed that the percent of reworked code differed in all three projects. Further analysis, though not published in this study, also demonstrated which subsystems had the most reworked code. Although I measured these differences, I haven't yet explored the differences' root causes.

If the percentage of reworked code is constant from project to project—say, a percentage of final code size—then productivity would be independent of reworked code. However, if this percentage varies by project, then effort estimates

based on historical data would be inconsistent. For example, at a general level, *estimated effort* = *estimated final size*/*productivity*, and, *productivity* = *historical final size*/*historical effort*. However, historical effort could vary as the amount of reworked code per project varied. In other words, with 80 percent reworked code, creating a final target size of 100 KeLOC would take more effort than creating the same target size if the reworked code was 20 percent. Inconsistent volumes of reworked code between projects may also indicate the software's extensibility. For example, using historical reworked code data, developers might establish thresholds that indicate they shouldn't reuse the baseline code on the next project—a redesign might prove more cost effective.

What can we learn from the reworked code data? Each line of code added or modified to a release has a cost associated with the effort for its design, design review, code, code review, unit test, and integration test. So, rework should be a concern if it represents wasted effort. However, reworked code and its associated effort can also be a function of a particular development process. For example, an iterative process in which the knowledge gained during an iteration can support a decision to revise and improve the existing design and code in the next iteration is an acceptable development process.⁷ Furthermore, refactoring, which focuses on improving code structure without changing functionality, is an acceptable and promoted development activity.⁸ The projects in this article had a minor amount of planned rework and refactoring, but the observed total volume of reworked code, planned and unplanned, might be acceptable. Fairley and Wilshire state that projects typically have 40 percent rework; others might have as much as 80 percent rework.¹ They further state that as a rule of thumb, 10 to 20 percent rework is acceptable. However, independent of an organization's rework goals, measuring and understanding rework is the first step toward managing it.

Rework's impact on project effort estimates is also a function of the estimation model used. For example, *effort* = *size*/*productivity* is a simple linear model in which factoring rework into size or productivity would result in the same effort estimation. In other words, if a team expected 20 percent reworked code, they could increase the size by 20 percent or reduce the productivity by the same amount. Another equation, *effort* = *size^N*/*productivity*, where *N* = scaling factor and *N* > 1, is a simplified nonlinear model based on Boehm's early work.⁹ In this equation, increasing

the size by 20 percent isn't equivalent to reducing the productivity by 20 percent.

A software project team can account for rework in project estimates using various methods:

- Use historical reworked code data to inflate the estimated size to account for the rework and omit rework from productivity variables. This simple method ignores non-code-related rework.
- Ignore reworked code in the size estimates (that is, estimate the final size only) and factor rework into the productivity variables. This also accommodates non-code-related rework.
- Factor necessary source code rework into the size calculations using historical data. Also factor non-code-related rework and unnecessary rework into the productivity. Although Boehm doesn't specifically focus on rework, his model manages it by adjusting size using a requirement evolution and volatility factor (REVL), a maintenance adjustment factor (MAF), and a maintenance change factor (MCF).³ He also adjusts productivity measures, taking into account effort multipliers that can contribute to increased rework, including factors such as platform volatility, complexity, and programmer capability.

The data clearly demonstrate that by measuring both code addition and modification and sampling these values at appropriate intervals during development, you can more accurately assess the actual amount of code developed during a project. The incremental increase in code size between a project's start and end is a poor indicator of size and might not be useful for estimating project effort if rework isn't factored into productivity metrics.

When estimating project effort, size matters. Research demonstrates a nonlinear relationship between size and effort; underestimating size by ignoring reworked code can result in underestimating project effort. In other words, unless you account for the effort associated with rework, your total project effort estimation will be low.


Collecting historical effort data associated with rework can be difficult if the original developers are no longer available or can't provide sufficient detail in their recollections. If you don't have historical effort metrics to gauge rework, a mea-

About the Author



Edmund P. Morozoff is an engineering manager at Medtronic. His primary areas of interest include biomedical instrumentation and embedded software engineering. Morozoff has an MASC in biomedical engineering from Simon Fraser University. He's a member of the IEEE Computer Society and the IEEE Engineering in Medicine and Biology Society. Contact him at paul.p.morozoff@medtronic.com.

sure such as the size of reworked code might be applicable.

I explored the utility of using lines of code as a sizing metric and found that it's relevant under limited criteria such as similar projects, processes, development teams, and coding constraints. Frequent sampling of software size during development provides improved insight into rework and produces a quantitative baseline from which you can measure process and tool improvements related to reducing reworked code. Areas of future research include discriminating between necessary and unnecessary rework, correlating the effort to rework source code to the size of reworked code, and investigating the correlation between defects and reworked code. 

Acknowledgments

I thank Charles Meyer, Paul Wassmund, Lauren Frear, and the reviewers for their constructive suggestions.

References

1. R.E. Fairley and M.J. Wilshire, "Iterative Rework: The Good, The Bad, and The Ugly," *Computer*, vol. 38, no. 9, pp. 34–41.
2. L.H. Putnam and W. Myers, *Measures for Excellence: Reliable Software on Time, within Budget*, Prentice-Hall, 1992.
3. C. Jones, *Estimating Software Costs: Bringing Realism to Estimating*, McGraw-Hill, 2007.
4. B. Kitchenham and E. Mendes, "Systematic Review of Software Productivity Measurement," Univ. of Auckland Dept. Computer Science, 2004; www.cs.auckland.ac.nz/emilia/srsp.pdf.
5. B. Boehm et al., *Software Cost Estimation with Cocomo II*, Prentice-Hall, 2000.
6. R. Purushothaman and D.E. Perry, "Toward Understanding the Rhetoric of Small Source Code Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, 2005, pp. 511–526.
7. C. Larman and V.R. Basili, "Iterative and Incremental Development: A Brief History," *Computer*, vol. 36, no. 6, 2003, pp. 47–56.
8. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
9. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.