Proceedings of the Annual Oregon Workshop on Software Metrics, 1992

Investigation of Coding Effort During the Testing Phase of Several Software Projects

Kathleen A. Iberle
Cardiology Business Unit
Hewlett-Packard
February 7, 1992

Introduction

A considerable amount of useful data can be gathered from a post-release analysis of coding effort in each project phase. This particular investigation focused on coding effort during the testing phase on three projects. The results were used to evaluate the accuracy of scheduling estimates and to identify potential process improvements that would lead to shorter time-to-market and/or increased quality. The investigation is based on breaking total coding effort (measured in non-commented lines of code or LOC) into the design, coding, and testing phases. The projects studied were two original software projects of around 100 KLOC of C code each, and one smaller follow-on or maintenance release. The projects are named in chronological order.

Projects B and C were typical new products produced by a medium-sized organization. Both were implemented primarily in the C language, involved large amounts of data exchange with digital instruments, used little or no code from previous projects, and were built on top of off-the-shelf operating systems and computers. Each project consisted of eight to ten developers. The products were both "organic" products as described by Boehm's COCOMO model: each involved a substantial number of concurrently interacting parts.[1]

The follow-on project was what is known in the trade as a "maintenance release" to an original project similar to what is described above. Follow-ons in this organization usually consist primarily of new features rather than bug fixes. These projects are usually staffed with three to five developers. Project A was a follow-on to a project that was not included in this study.

The analysis of coding effort during the testing phase is shown to be a simple study which yields substantial information that can be used to improve software development processes, particularly estimation and scheduling. This study can be performed after product release by any project that consistently uses a revision control system. Investigating the causes of the additional coding during the testing phase requires some additional record-keeping that must be done during the testing phase.

Investigation

Measuring code size at numerous points in time during a project provides a more accurate picture of coding effort than the final code size because a sizable percentage of the code is

rewritten during the project. The coding effort for the projects studied was divided into design, coding, and testing phases. A set of files to be compared consisted of one from the start of the design phase, one from the design to coding transition, one from the coding to testing transition, and the most recent revision. The files were then compared in pairs to look for differences. The file from the start of the design phase generally did not exist unless the file was reused code from a previous project.

Table 1.

Project name	Type	Predecessor	Final size in LOC
Project A	follow-on	not included	9689
Project B	original	none	116519
Project C	original	none	141807

The code was measured in non-commented lines of code (LOC). A non-commented line of code was defined as any line of text that did not consist entirely of comments and/or whitespace.

The pairs of files for comparison were prepared by checking out the most recent revision of a particular file before the first cut-off date, and the most recent revision before the second cut-off date. The two files were stripped of comments and empty lines using a homegrown comment-stripping program for C, and an awk script for OS/2 or DOS batch files. The two files were then compared using PVCS vdiff, and the number of lines marked with "+" were counted. Thus both modified and entirely new lines are counted as new. If the earlier file did not exist, all lines are assumed to be new. Deleted lines are not counted. The contents of include files are counted once. The #include lines in the referencing files are counted.

The raw data consisted of numbers of new LOC for each file for each phase. They were summed to obtain numbers of new LOC for each directory for each phase, since each directory represented a conceptual design unit.

The accuracy of these measurements depend on the consistent use of a revision control system. Changing file names during the course of the project makes the number of lines written appear higher than it was. Revisions can be lost entirely by deleting them from the logfile. They can also be lost by maintaining more than one copy of the logfile during part of the project and subsequently merging the logfiles into one. Lost revisions will make the coding effort appear lower than it was. Checking in work in progress very infrequently may shift coding effort across time-points and thus into a later phase.

Project A, which was the earliest project in the group studied, did not institute revision control on most files until the end of the project. The project was actually considerably larger than the 10 KLOC shown, but only two sets of files had been under revision control long enough to contain the historical data needed to be included in this study.

Project B, the next project to be done, was fairly consistent in its revision control practices. Some coding effort during the design phase was shifted into the coding phase due to insufficiently

frequent check-ins. In a few cases involving reused code, the final revision of the files from the previous project had to be retrieved to use as the starting point for the design phase, because the initial revision for Project B was not checked in until substantial work had been done on it. There were a small number of changed file names during the project as well.

Project C used exemplary revision control practices, so the data collection process was simplified to the point of automation.

For all projects, 5-10% of the code was not included in the study for various reasons, such as use of a language for which a comment-stripper did not exist.

The study was implemented in Thompson Toolkit shell scripts (a Unix-like shell language for OS/2 or DOS). It took about 2 weeks to write the tools and complete the first part of the investigation for Projects A and B. Project C took the tools and were ready to run their investigation in about a day. The actual study runs automatically for around 10 hours, using about 60 MB of disk space, for a 100 KLOC project.

Analysis of Results

Several different and potentially useful measurements can be made on the raw data. If the project is considered to be representative, some of the measurements can be used to predict progress and schedules on future projects. Other measurements can be used to pinpoint problem areas.

Possible measurements:

- Percentage of code reused without change from prior product.
- Average code output during coding phase (productivity).
- Percentage of coding effort performed during testing phase in individual directories.
- Percentage of overall coding effort performed during testing phase (rework).
- Percentage of code in C, as opposed to other languages.

Productivity

A productivity study was performed on Project B. The coding phase required 226 person-weeks. The average output of C code per engineer during coding phase was 65 lines/day. The average output of all types of code was about 75 lines/day. Output varied from person to person depending on other duties, style, experience, and skill. However, the average code output could be used to estimate the length of the coding phase on a following project. Since some individuals submitted time estimates for coding phase based on estimates of code size and code output, the individual average output can be used to improve the future estimates of those individuals.

Rework

Rework is defined as the percentage of coding effort performed during the testing phase. The coding effort is measured as the total LOC written, which is more than the final LOC written (since some lines are rewritten.)

It is instructive to consider the allocation of coding effort between the coding and testing phases. The testing phase begins after unit testing and system integration is complete. The system is believed to work at this point. The ideal would be 0% of effort during testing phase, implying no feature changes and no defects found during testing phase.

For Project B, Table 2 shows that at least 19,600 lines of code were written during the testing phase, about 17% of the total lines of code written during the entire project. This represents a minimum of 52 engineer-weeks of effort, if coded at 75 lines per day, which is very optimistic. Project B was attempting 1800 testing hours. At 6 hours per day, that's 60 weeks of effort. The time spent fixing bugs and finishing features was nearly equal to the actual testing hours. This does not count diagnosing errors (often very time-consuming), supporting beta sites, writing test procedures, patching systems that broke under testing, etc. These results were surprising, as the organization had been assuming that the ratio of testing hours to defect-fixing time was much larger than it actually was.

Partial data recovered from the Project A files indicate that about 14% of the Project A coding was performed during the testing phase. The same study performed on Project C, which was roughly the same size as Project B, showed that about 12% of the coding effort was performed during the testing phase. Within the uncertainty of the experimental method, all three projects had the same amount of rework during the testing phase.

Table 2.

Project name	LOC written during design	LOC written during coding	LOC written during testing	Total LOC	% rework (effort)
Project A	1505	6858	1326	9689	13.68%
Project B	9617	87315	19587	16519	16.81%
Project C	NA	NA	17328	141807	12.22%

Fifteen percent rework during testing is consistent with observed code growth in the industry. "Even after the program is implemented and in test, code expansion of 10 percent to 25 percent is likely." [2]

If it is assumed that 15% rework is average, this can be used at two points in scheduling. When entering design phase, final code size is estimated roughly. This estimate is used to predict the number of testing hours that will be needed. Coding time is also estimated. Both these values can be used to estimate time needed to fix bugs during the testing phase. The bug fixing time will be 15/85 or 17% of the coding time, according to the results of this study. As seen above, the rework time will be a very substantial part of the testing phase. At the end of coding phase, the actual amount of code can be measured, and the testing phase estimate refined by using actual LOC and measured code output.

A closer look at the distribution of rework among various design areas in Project B proved interesting. Figure 1 shows the LOC produced during each phase of the project in each "module". A module was a single directory containing multiple files which implemented a

specific design area. Generally the files were all linked into a single library, dynamic link library, or executable. Each module was the responsibility of a specific engineer. Each engineer was responsible for a number of modules. A similar arrangement was used on Project C, and Figure 2 shows the LOC produced during each phase for Project C.

There was a great deal of variability in the amount of rework in different modules. Each engineer had some modules that required a great deal of rework, and others that required very little. Upon investigation, the modules that were 100% written during the testing phase were either test harnesses or substantive additional product features. The distribution of effort on Project B during testing was as follows:

- Feature creep: 16% of the testing phase coding effort (2536 LOC) went to writing two major features that were added to the product specification during the testing phase. (Modules 39, 40, 41, 47)
- Test harnesses: 5% of the effort (888 LOC) was accounted for by unanticipated test harnesses. (Modules 4, 32)
- System integration bugs: 9% of the effort during testing went into an area that can best be described as managing all the other processes. 75% of the effort in this area was done during the testing phase. (Modules 44, 45, 46)
- Functionality bug fixes: 74% of the effort can be attributed to fixing functionality which was included in the original specifications and was not system integration.

Root Cause Analysis

Clearly there was far more rework in some modules than others. Since the individual responsible for the module did not correlate with the percentage of rework, it can be assumed that some modules were more difficult than others. It would be very interesting to know if the rework was primarily due to problems in a particular development phase. In theory, this could point out whether development practices in specification, design, coding, or testing needed the most improvement.

However, attributing the causes of the defects requires that some records be kept during the testing phase. The ideal would be to be able to extract the number of lines written for each defect in that area, and then group the defects according to cause, such as wrong/missing/incomplete specification, design, or coding. A less rigorous approach is to retrieve the number of defects for each area, and then group those defects. This assumes that the effort is proportional to the number of defects, which may not be the case.

In principle, the number of defects and their causes could have been retrieved from our system in two ways. The first method depended on recording the Service Request identifier in the logfile as defect fixes are checked in, and maintaining cause attribution in the Service Request records. The second depended on recording the cause attribution and module in which the defect was found in the Service Request records.

Unfortunately, attribution of the cause of defects in the Service Request records was so incomplete in both Projects B and C that meaningful results are not possible. (Project A did not

keep Service Request records in any organized fashion.) The Service Requests have a field for cause attribution of the defect, but this was filled in incorrectly or not at all by more than half of the project staff on both projects. The most damaging practice was that of recording every defect as "wrong code", thereby invalidating any analysis of the group of defects. Therefore, there is not sufficient data from which to draw useful conclusions. Apparently prior education of the project staff is critical in attempting to analyze defect causes.

Several staff members on Project B did record the Service Request identifiers in the logfiles. Two of these individuals went through the logfiles after product release and attributed causes for each defect from memory. The results are shown in Figure 3. The defects per KLOC did not correlate to the individuals in question. It was not possible to measure the LOC per defect, so it is not clear whether the most frequently occurring defects actually accounted for the bulk of the rework.

The defects were attributed to problems occurring in various phases of the project, in an attempt to identify the phase where improvements would be most beneficial. The category "OEM Software" was used where the defect was in off-the-shelf software purchased from another vendor (such as the operating system). In some cases, such defects resulted in very substantial recoding to work around the problem. The two obvious areas of improvement are in Missing Specifications and Wrong Code. The project staff was not particularly surprised by the first but found the second to be a bit alarming.

A necessary adjunct to collecting and analyzing metrics is talking to the people who did the work. After pinpointing problem modules (and sometimes problem files within modules), discussion with the individuals showed that they usually had ideas as to why module X was so much more difficult than module Y. The metrics could be surprising, as developer's perception of which modules took the most time and the difference between the highest and lowest were sometimes not accurate.

Sample Process Improvements

The analysis of data collected on Projects A, B, and C resulted in a number of specific process improvements recommended for the particular software production organization where the studies were done. These are included as representative examples of what can be done with the rework study results. The specific improvements and strong points will, of course, vary depending on the organization.

Process Improvements for Cardiology Business Unit

Improve estimate of test phase duration at Design->Coding checkpoint by including total bug fixing time.

> Estimate total bug fixing time as 17% of engineer-hours estimated for coding phase. (The bug fixing time will be increased if some code was written during design phase.) Multiply by undetermined fudge factor for diagnosis time. Fixing time is a very large contributor to total test phase duration, and is usually neglected by current practices.

Recalculate expected test phase duration at the beginning of test phase.

Get total LOC written using methods described above. Compare to original LOC estimate. Total fixing time and total test hours can be re-estimated more accurately.

Emphasize correct and thorough requirements specifications.

Missing or incorrect requirements specifications were the single largest cause of rework where data could be gathered.

Increase coding time estimate by about 400% for system integration applications implemented in batch files.

The large amount of rework done in the system management area shows that this type of coding is particularly time-consuming. (This area would also be a candidate for careful consideration of ways to make it easier to do!).

De-emphasize complete unit testing.

On the basis of admittedly limited data, the expected benefit of unit testing did not appear. The fragmentary data from Project A showed a rework percentage roughly equal to that of Projects B and C, despite the use of unit testing for Projects B and C. The modules that were more thoroughly unit tested (according to the authors) did not have consistently lower rework percentages. (However, several authors commented that sticking to a limit of 10 or 15 for complexity was good.)

Strong Points

Late addition of features was well controlled.

New features added during the testing phase of Project B accounted for a reasonable 16% of the testing phase coding effort, or 3% of the entire coding effort.

Metrics Improvements

Simplify cause attribution of defects in the Service Request.

The Service Request records for cause of defects was unusable for this study because some people entered all defects as coding error, and others neglected the field entirely. A much smaller list of choices would encourage proper use of the field. Education of the staff prior to testing phase on proper entry of Service Requests is critical.

The Service Request should include the directory in which the fix was made.

It would be useful to be able to measure number of defects and causes of defects by conceptual area. Conceptual areas are commonly mapped to directories. The author was often able to state that a particular directory had been unusually easy or hard for a particular reason. This would be very helpful in identifying potential process improvements and measuring the effect of improvements.

Conclusions

A considerable amount of useful data can be gathered from a post-release analysis of the coding effort in each project phase. The necessary preconditions are use of a revision control system, and an initial revision that represents the start of the project or beginning of coding. A hierarchical directory structure is very helpful to group the data into meaningful sets.

The rework study is reasonably easy to implement. The tools used in this study are generally available for both PC and Unix platforms. The study can identify potential process improvements and strong points in the current processes, particularly in project estimation and scheduling. The study is surprisingly effective in suggesting process improvements considering its simplicity.

References

1. Boehm, Barry. *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall, 1981.
2. Humphrey, Watts. *Managing the Software Process.* Addison-Wesley Publishing Company, 1989. pp 95-96

Acknowledgements

Thanks to Dave Houston, Rommel Raj, and John Kent for their assistance in data collection.