What Motivates Programmers to Comment?

David Patrick Marin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2005-18
http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-18.html

November 23, 2005

What Motivates Programmers to Comment?

David Marin
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776 USA
dmarin@cs.berkeley.edu

November 23, 2005

Contents

List of Tables

Abstract

Though programmers are often encouraged to comment their source code more thoroughly, there has been very little scientific investigation into what kinds of situations actually cause programmers to do so. I conducted a statistical study of the CVS repositories of nine Open Source projects, and made four major findings. First, the rate at which programmers comment varies widely from project to project and programmer to programmer; even the same programmer will comment at different rates on different projects. Second, programmers tend to comment larger modifications to source code more thoroughly. Third, more programmers modifying the same file does not, in general, mean more commenting. Finally, programmers tend to comment more when they are modifying code that is thoroughly commented to begin with. I then determined through an experiment with programmers that there is a causal link behind my last finding; that is, the more throughly a source code file is commented, the more thoroughly programmers will comment when they make modifications to it.

Chapter 1

Introduction

It has been long understood that source code comments can make source code easier for programmers to read and modify. Code readability is vitally important for both software maintenance and software reuse.[1]

If we had a better understanding of what motivated programmers to comment, we could probably get programmers to comment more often, and thus produce more readable code. We might also be able to avoid situations that lead to insufficiently commented code.

In order to better understand commenting behavior, I carried out two research investigations. First, I did a statistical study of several real-world software projects, to try to discover "natural laws" of commenting; that is, patterns that one could expect to be exhibited by most programmers in most software projects. That study is described in Chapter 2. Second, I ran an experiment with real programmers in an attempt to validate one such law as a causal relationship. The experiment and its outcome are explained in Chapter 3. Chapter 4 briefly summarizes other research on these topics; Chapter 5 suggests directions for future research.

[1] A more thorough discussion of the relationship between comments and readability can be found in Section 4.1.

Chapter 2

A Statistical Study of Commenting

The Concurrent Version System [CVS], provides an excellent vehicle for studying commenting behavior because it provides a repository to which programmers periodically commit the most recent version of source code files. Thus, by studying a CVS repository, we can look at a file's entire history, rather than just the current version.

My thinking in starting on this project was that by studying the CVS repositories of real projects, I would be able to answer several kinds of quantitative questions about how often programmers comment in certain situations. For example, do programmers comment more...

- in big changes or small ones?
- on big files or small ones?
- on old files or young ones?

I also wanted to ask questions about variance between programmers, and variance from project to project:

- Do all programmers tend to comment about the same amount, or does it vary from programmer to programmer?
- Do all projects tend to get about the same amount of comments, or does it vary from project to project?
- If the same programmer works on different projects, will she tend to comment the same amount on all projects, or will she comment more when working on some projects, and less when working on others?

Finally, I had two specific theories about commenting that I wanted to try to validate. First, I theorized that because one of the main purposes of commenting

was to communicate how source code works, programmers would comment more often in situations involving more collaboration. Second, I theorized that programmers would try to maintain whatever rate of commenting already existed in a file; if the file was thoroughly commented, programmers would comment their changes thoroughly, and the opposite would also be true.

Of course, this raises the question of what it means, quantitatively, to comment "more." Does this mean more total comments? More words in comments? What if someone modifies an existing comment? What if someone comments out a chunk of code; does that count as commenting?

For this study, I used the same measurement of commenting that Shum and Cook employed in an early study about Literate Programming [SC95]; I took the total number of characters that were part of a comment, and divided them by the total number of characters in a file. Hereafter, I refer to this ratio as the "rate of commenting." A higher rate of commenting indicates that a given version of a file is more thoroughly commented; a lower rate indicates that it is less so.

In one of my preliminary studies, [Mar02] I invented a more complicated "scoring" system to attempt to measure how much effort programmers were putting into commenting. For instance, it probably takes less effort to make a comment like `/**********/` than a descriptive comment of the same length. However, in terms of the correlations I found, there was little or no difference between my scoring system and the simpler calculation of commenting rate, so in this study, I opted for the simpler system.

By identifying which characters in a file were added since the previous version of a file, I could determine how thoroughly commented a given modification to a file was by taking the rate of commenting among just the new characters (i.e. new characters in comments, divided by the total number of new characters). This was my fundamental way of gaining insight into what programmers were doing; if a particular modification had a higher rate of commenting, the programmer making that modification was commenting more, and if it had a lower rate of commenting, that programmer was commenting less.

This method of measuring commenting partially solves the problem of commented-out code automatically: if a programmer comments out a section of code, only the characters that begin and end the comment will be considered "added", and thus there will be very little effect on the measurement of how throughly commented that programmer's modifications were. Subsequent versions of the file taken as a whole, however, will be measured, incorrectly, as "more thoroughly commented" due to the commented-out section of code. In any case, commented-out source code does not account for enough of the comments to significantly skew my results. An informal inspection revealed that, for any given project in my study, commented-out source code accounts for less than 20% of the characters in comments; in most projects, the figure is more like 5%.

2.1 Data Collection

In order to collect the data I needed, I wrote a program hereafter referred to as the "Comment Counter." For its technical specifications, and some practical considerations about collecting data on comments, see Appendix A. For the purposes of data collection, the ubiquitous copyright notice comments found in Open Source projects were effectively considered not part of the file.

For each version of each file that the Comment Counter recognized as source code, I collected the following data:

- total number of characters in the file
- number of characters inside comments
- total number of new characters
- number of new characters inside comments
- version number
- revision date (when the version was checked in)
- author (who checked it in)
- commit message (the message that the author submitted to CVS when checking in that version)

Because of the large volume of code, and the automated fashion in which I gathered statistics, I did not make any attempt to categorize comments as to their purpose or usefulness (other than to screen out copyright notices). Comments are meant to be read by human beings, so it is surprisingly difficult to determine what a comment pertains to without actually understanding the code (for instance, does a comment followed by a variable declaration refer to just the variable declaration, or the next section of code?); Van De Vanter [Van02] explains these difficulties in detail, with several real-world code excerpts. Usefulness is of course subjective, though misleading comments are almost never useful. I discuss possible ways of changing my methodology to gather these more subjective kinds of data in Section 5.1.

2.2 Data Sources

Now all I needed were CVS repositories to study. After making some preliminary tests on projects at UC Berkeley, I turned to SourceForge.net [SF], a public repository of open source projects. I chose SourceForge because the CVS repository for each project on SourceForge was easily accessible via an anonymous login. I also chose SourceForge because of its wide variety of projects; in October 2002 when I ran a script on SourceForge's web site to find projects to study, SourceForge had 13208 users working on 3320 projects, 1914 of which had code

in their CVS repositories; today it has nearly 50 times as many users, and 20 times as many projects.

Running the Comment Counter on a single SourceForge project could take several hours, so collecting data for every project on SourceForge (not to mention processing it) was infeasible. Because I was interested in learning whether the same programmer would ever comment at different rates on different projects, I chose the two users[1] who worked on the most projects, and looked at all the projects that they worked on. I found that some projects were too small to yield useful results (one such project had only four source code files that the Comment Counter could read). This left me with nine projects: `aeongdk`, `alleg`, `allegrogl`, `dumb`, `exult`, `fblend`, `fink`, `hexedit`, and `scummvm`, each of which had at least 100 distinct versions of files.

I collected the data for `exult`, `hexedit`, and `scummvm` in October 2002, and the remaining projects in December 2002. The data I collected comprised 20000[2] distinct versions of 2119 distinct files.

Unfortunately, I realized after collecting my data that the `diff` tool on the Solaris box I used had suffered transient, apparently random failures (it was not simply due to large file sizes, for instance). As a result, for some versions I effectively had no information about the number of characters added to a file. It was the output of `diff` that was missing, not the versions of files themselves, so the fact that a data point was missing did not affect the correctness of other data points. These failures occurred so often for `aeongdk` that I re-collected its data in December 2003 (`aeongdk` was last updated in September 2001, so the re-collected data was otherwise identical to the original). I was unable to re-collect data for the other (mostly larger) projects because at the time, SourceForge was experiencing bandwidth problems that made it very difficult to run the Comment Counter.

However, assuming that these failures were indeed unrelated to the data I collected, they should have had very little effect on my final results, other than possibly obscuring a few correlations I would have otherwise found. No more than a quarter of the versions in any project other than `aeongdk` were missing this information, and because of the scoring method I used, such data points were effectively excluded from any results having to do with the commenting rate of a change.

Finally, due to an architectural weakness of the Comment Counter, I was not able to collect data on previous versions of files which had been marked in the CVS repository as "removed," at the time of data collection.

[1] All the data I used in this study is publicly available. However, out of respect for the privacy of the programmers, I do not refer to any individual programmer by his or her SourceForge login in this paper.

[2] Exactly 20000!

2.3 Finding Correlations

The next step was to start looking for interesting correlations in my data set. But first, I had to decide how I wanted to group my data; lumping together all programmers and all projects, and searching for correlations in the aggregate, proved not to be very useful.

What I found is that even on the same project, different programmers would comment their modifications at very different rates. Moreover, even the same programmer would tend to comment at vastly different rates on different projects.[3]

Figures 2.1 and 2.2 show these variations using standard box-and-whisker plots, a way of summarizing a large number of data points without plotting them all. In this case, a data point is the commenting rate of an individual change to an individual file. For each project, the large rectangle (the "box") represents half of the data points, from the first quartile to the third. The median commenting rate for each project is shown as a line across the box (for many projects, the median commenting rate for changes is 0). The dotted lines, or "whiskers", extend beyond the box in each direction by $\frac{3}{2}$ the height of the box (or to the end of the range of values); any data point outside the whiskers is considered an outlier,[4] and is plotted individually as a `+`. Additionally, to give an idea of the number of data points in each project, the width of the boxes is proportional to the square root of the number of data points in each project (including outliers).

What I decided to do was to group my data in three ways:

- by project
- by the programmer making the modification
- by programmer and project

Within each group, for each kind of grouping, I looked for statistical correlations between interesting pairs of variables, usually the commenting rate of the change relative to the previous version of the file, and one other variable (e.g. file size). To determine whether a correlation was significant, I used the arbitrary (but standard) rule that a correlation was significant if it had a less than 5% probability of occurring by chance. That is, if I found a significant correlation between two variables, there was less than a 5% chance that *no relationship* between those two variables existed. Many groups were too small to produce useful data (for instance, within in my data set, some programmers only contributed one or two modifications), so I filtered out any group containing less than 100 data points. I wrote a short program in a statistical package [R] to help count significant correlations.

[3] Interestingly, the fact that different programmers tend to comment at different rates, taken with several other metrics, can actually be used to identify the author of a program from the program's source code [Krs94].

[4] Some projects in Figure 2.1 appear to have a large number of outliers simply because they have so many data points; for example, the data for `exult` comprises 9790 versions.

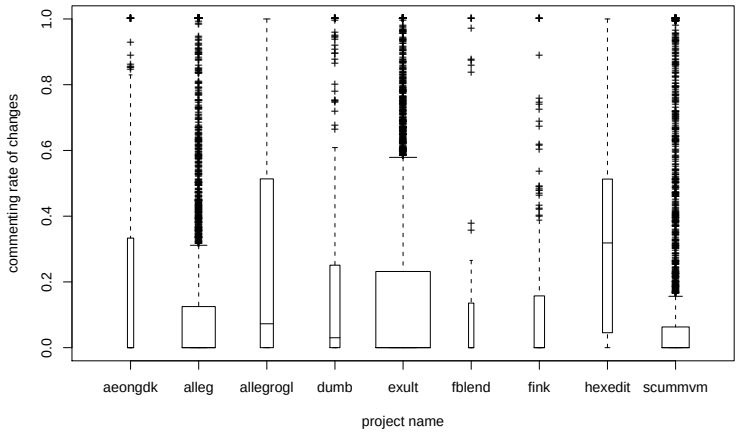

Figure 2.1: Commenting rates of individual changes by all programmers, grouped by project. This is a standard box-and-whisker plot; the width of the boxes is proportional to the square root of the number of data points in each project.

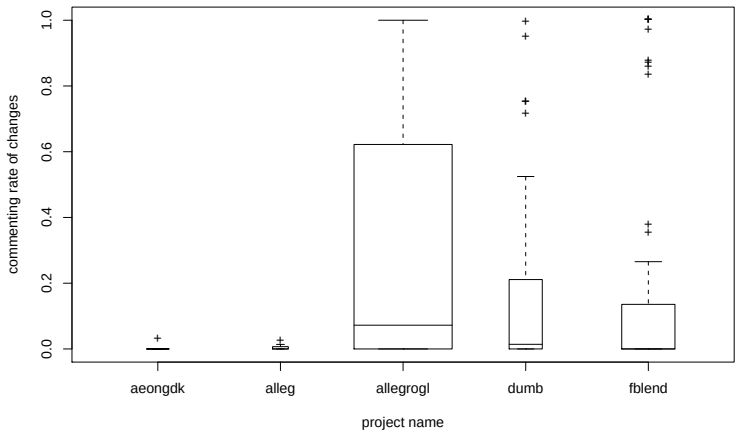

Figure 2.2: Commenting rates of individual changes made by a single programmer, grouped by project. This is a standard box-and-whisker plot; the width of the boxes is proportional to the square root of the number of data points in each project.

My goal was not simply to find correlations, but to find "natural laws" of commenting, that is, correlations related to commenting that I could expect most programmers to exhibit in most software projects. What I sometimes found is that for some pairs of variables, significant correlations appeared in more than 5% of the groups (which is what I would expect if my data were purely random), but they were a mix of positive and negative correlations, so I couldn't make any universal conclusion about the relationship between the two variables.

As an example, in some projects, older files tended to be more thoroughly commented (positive correlation), and in some projects, the opposite was true (negative correlation). Although significant events may have happened in particular projects, such as a programmer joining the project who tended to comment very thoroughly, I wasn't able to make any general claims about the relationship between the age of files and their commenting rates.

What I was looking for were the cases where there was a significant positive correlation between an interesting pair of variables in many or most groups, and very few or no negative correlations (or vice versa). If I could find such a pair of variables, then I was on to a potential "natural law" of commenting.

2.4 Setting the Ground Rules

An important issue I faced in looking for correlations was whether to include the first version of a file. The Comment Counter counts all characters in the first version of a file as "added," but the situation is different. Rather than making a *modification* to an existing file, the programmer is in fact *creating* an entirely new file.

But did this make any difference to programmers? Could I, for the purposes of my study, consider creating a file, and making a modification, to be the same thing? To answer this question, I created a predicate variable that was 1 when programmers were creating the first version of a file, and 0 when they were modifying an existing file, and then looked for correlations between that predicate variable, and the commenting rate of the new code.

In fact, in several cases, programmers did indeed comment files they created more thoroughly than their changes. Refer to Table 2.1 for a tally of the groups where I found significant correlations.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
first version?	c. r. of change/creation	4/0/9	10/0/25	12/0/28

Table 2.1: Is creation equivalent to modification?

I use tables of this format throughout this chapter. For each pair of variables, and each kind of grouping, I give the number of groups where I found a significant positive or negative correlation, and the total number of groups. For instance, Table 2.1 indicates a significant positive correlation in 4 out of 9 projects, and a

significant negative correlation in 0 out of 9 projects. In these tables, predicate variables always end with a "?", and "commenting rate" is abbreviated to "c. r.". I use "by both" as shorthand for "by project and programmer", that is, groups of versions of files which the same programmer checked into the CVS repository for the same project.

So as not to confuse the file creation effect with other effects I tried to observe, I filtered out data about the first version of a file whenever I searched for correlations relating to the commenting rate of modifications. Unfortunately, dropping the first version left one project (`fblend`) with less than 100 data points, causing it to be filtered out.

I must also mention one correlation that, while not directly related to commenting, was not completely obvious. The longer a file has been in the repository, the larger it tends to be, and vice versa. Table 2.2 tallies significant correlations of file size against both version number of a file, and a file's chronological age.[5]

variables		# of correlations (+/-/# of groups)
		by project
version number	file size	9/0/9
age of file (in seconds)	file size	7/0/9

Table 2.2: Do older files tend to be larger?

These results told me that if I observed an effect for large files, I should not be terribly surprised to see the same effect for files with a high version number, and vice versa. Also, these results suggested that version number may be a more meaningful measurement of the age of a file than its actual chronological age, at least as far as CVS repositories were concerned.

2.5 Correlations by Size of Change

Do programmers comment more when they make large changes, or small ones? Table 2.3 shows the correlations between various measurements of the size of a change, and the commenting rate of a change.

There did indeed appear to be a (positive) relationship between the size of a change and its commenting rate; that is, the larger a change, the more thoroughly commented it is. However, a significant correlation to this effect did not appear in all groups of data (e.g. for all projects), or, depending on how size of change was measured, even most groups. More striking was the fact that most correlations were between the commenting rate of the change and the *log* of the number of new characters. This suggested that the size of small changes was particularly important.

[5]Note that when I present data having to do with static versions of a file, rather than modifications, I include data for the first version of files as well, so the project `fblend` is no longer filtered out (thus there are 9 projects instead of 8). In this case, I do not group data by the programmer making changes because it would not be meaningful to do so.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
# new characters	c. r. of change	2/0/8	11/0/25	11/0/27
log of # new characters	c. r. of change	6/0/8	15/1/25	17/1/27
Δ (net change) of file size	c. r. of change	1/0/8	5/0/25	6/0/27
log of Δ of file size	c. r. of change	5/0/8	17/0/25	19/0/27
% change of file size	c. r. of change	4/0/8	9/0/25	10/0/27
log of % change of file size	c. r. of change	4/0/8	9/0/25	10/0/27

Table 2.3: Do programmers comment more when they make large changes, or small ones?

But how small is a small change? To find out, I created a predicate variable that was 0 when the size of the change was less than a given threshold, and 1 otherwise. I used four different numbers for the threshold: 50, 100, 500, and 1000 characters (see Table 2.4). Overall, a threshold of 100 characters seems to be the best predictor (in terms of turning up correlations) for how thoroughly programmers will comment their changes.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
$\geq$ 50 chars in change?	c. r. of change	7/0/8	16/0/25	18/0/27
$\geq$ 100 chars in change?	c. r. of change	6/0/8	18/0/25	20/0/27
$\geq$ 500 chars in change?	c. r. of change	6/0/8	13/0/25	15/0/27
$\geq$ 1000 chars in change?	c. r. of change	3/0/8	8/1/25	8/1/27

Table 2.4: Is there a threshold for size of change that's most significant for predicting rate of commenting?

Thus, I found that in general, the larger a change is, the more thoroughly commented it is, and I discovered two useful ways of measuring size of change (log of size of change, and setting a threshold of 100 characters for size of change) in order to better observe this effect.

I originally theorized that small changes tend to be less thoroughly commented simply because many of them are bug fixes, and bug fixes tend to be less thoroughly commented because they merely make the code do what it was supposed to have done already, so there is no need to add or change comments. Certainly, small changes do tend to be bug fixes; scanning the CVS commit messages accompanying changes revealed that a disproportionate number (about half) of changes of less than 100 characters were bug fixes.

What is less clear is whether bug fixes are actually less thoroughly commented than other changes of the same size. At least, I was not able to come up with a heuristic for identifying bug fixes that yielded interesting results; simply looking for the strings "bug" and"fix" in commit messages doesn't do it. Certainly, programmers do not identify all bug fixes as such in their commit message, but (based on scanning my data by hand) commit messages containing

these strings almost always indicate a bug fix, so if there is something special about bug fixes, such heuristics ought to turn up correlations.

Part of the problem with analyzing CVS commit messages in an automated fashion is that a programmer may check in changes to several different files at the same time, but only make one commit message for the whole batch; thus a commit message that mentions a bug could be referring to another file entirely. However, even when I reduced my search to the 3009 versions of files with a unique commit message (that is, changes that were not checked in as part of a batch), I was still not able to turn up any correlations whatsoever using such string-matching heuristics (see Table 2.5); note that many groups dropped below the threshold of 100 data points and were filtered out, so there are only 5 projects and 8 programmers in the table.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
"bug" in commit message	c. r. of change	0/0/5	0/0/8	0/0/8
"bug" and "fix" in commit message	c. r. of change	0/0/5	0/0/8	0/0/8

Table 2.5: Do bug fixes tend to be less thoroughly commented? (versions with a unique commit message only)

In summary, the connection between change size and rate of commenting has a lot to do with small changes, but little or nothing to do with bug fixes specifically.

2.6 Correlations Based on File Size

Do programmers tend to comment more when modifying large files, or small ones? Table 2.6 shows correlations between the size of a file (before the modification) and the commenting rate of the change.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
# of chars in file	c. r. of change	3/2/8	7/4/25	7/3/27
log of # of chars	c. r. of change	3/3/8	5/3/25	4/3/27

Table 2.6: Do programmers tend to comment larger files more, or less?

Essentially, the results are all over the map. For particular programmers and projects, the size of a file may matter, but not in any generally applicable way. More than anything, this is a demonstration of the fact that in any set of data, you can find all sorts of correlations that are statistically significant, but not meaningful.

However, I found more interesting results when I asked the analogous question about files: do larger files tend to be more or less thoroughly commented?

I correlated the commenting rate of a file with both the number of characters in the file, and the log of the number of characters (see Table 2.7).

variables		# of correlations (+/-/# of groups) by project
# of chars in file	c. r. of file	1/6/9
log of # of chars	c. r. of file	1/7/9

Table 2.7: Do larger files tend to be more or less thoroughly commented?

In the majority of projects I studied, the larger a file was, the less thoroughly commented it was. However, in one project, `alleg`, the reverse was true, and in one project, `aeongdk`, it made no difference. My guess is that it may have to do with a coding standard specific to these projects whereby each file has its own descriptive comment (distinct from a copyright notice), and this comment makes up a larger portion of smaller files. In any case, these results illustrate the fact that the fate of a programming project is determined by more than the tendencies of the individual programmers participating in it.

2.7 Correlations Based on Age

My results based on age were similarly fruitless. To measure age, I used both the version number of a change, and the chronological age of a file at the time of the change, as well as the log of the file's age (see Table 2.8).

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
version number	c. r. of change	3/2/8	6/5/25	6/5/27
age of file	c. r. of change	5/1/8	8/5/25	7/5/27
log of age of file	c. r. of change	4/2/8	8/6/25	7/6/27

Table 2.8: Do programmers tend to comment older files more, or less?

As with file size, correlations are all over the map. Probably these correlations are as a result of events specific to particular projects and programmers; for instance, a programmer who tends to comment significantly more might have checked in changes to a lot of files, making age appear to be a factor in commenting for that project.

I found similarly unimpressive results when I looked for correlations between the age of a file and its overall commenting rate (see Table 2.9). This is particularly remarkable because old files tend to be larger, and larger files tended to be less thoroughly commented in many projects.

In conclusion, the age of a file seems to have very little relationship in general with how much programmers modifying that file will comment, nor on how thoroughly commented the file will be.

variables		# of correlations (+/-/# of groups)
		by project
version number	c. r. of file	1/5/9
age of file	c. r. of file	3/3/9
log of age of file	c. r. of file	4/4/9

Table 2.9: Do older files tend to be more or less thoroughly commented?

2.8 Correlations Having to do with Collaboration

Now we move from the more basic questions to testing the first of my theories about commenting. Because one of the the purposes of commenting is to communicate with other programmers, I theorized that the more collaboration between programmers was going on, the more people would comment.

But how to measure collaboration? I had to pick some method of measuring collaboration that I could measure using the data I had, and that programmers were presumably somewhat aware of. For instance, I could not base my measurements on when programmers were reading other programmers' comments, because I did not have that data, but even if I could, what I would really want to measure is the fact that programmers *knew* that other programmers were reading their comments.

Finally, I decided that I would narrowly define collaboration as the situation of several programmers modifying the same file. For my purposes, a programmer was considered to be involved in collaboration when he or she was modifying a file which other programmers had previously modified. In some ways, this is a weak measurement (two programmers who fix bugs in the same file probably aren't really collaborating), but it is accurate in the negative sense; if a file has only a single author, there is probably no collaboration going on with respect to that file.

Thus, for each change that a programmer made to a file, I measured the number of programmers who had previously modified that file, and attempted to correlate it against the commenting rate of the change. I also speculated that a programmer modifying a file that no other programmer had touched might be a special case, so I also created a predicate variable that was 0 when no other programmers had previously modified a file, and 1 otherwise. My results are in Table 2.10.

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
# of prev other authors	c. r. of change	2/1/8	5/5/25	5/5/27
has other authors?	c. r. of change	1/2/8	2/7/25	1/7/27

Table 2.10: Will programmers comment a file more or less often if it has many previous authors?

What I found cast doubt on my theory. In fact, seven programmers tended to do the opposite of what I expected; they commented more on "their own" files, that is, files that no one else had modified.

For completeness, I asked the related question about the state of files in a project; did files that were the result of collaboration tend to be more or less thoroughly commented? For every version of every file, I calculated the total number of programmers who had contributed to that file, up until the current version, and correlated that number against the commenting rate of the file (see Table 2.11).

variables		# of correlations (+/-/# of groups)
		by project
total # of authors	c. r. of file	2/3/9

Table 2.11: Do files with more authors tend to more or less thoroughly commented?

Again, I found no overall pattern. In many cases, collaboration appears to have led to *less* thoroughly commented code.

In summary, I found little evidence to support the theory that having many people working on the same code leads to more commenting. If such an effect does exist, it cannot be measured simply by looking for cases in which two or more people modify the same file.

2.9 Correlations Based on State of Previous Version

If the prospect of other programmers having to read code with few comments did not inspire programmers to comment, what about a different form of social pressure? Perhaps the mere fact that the code a programmer was modifying was thoroughly commented would inspire that programmer to comment more. To find out, all I needed to do was look for correlations between the commenting rate of a modification, and the commenting rate of the previous version of the file modified (see Table 2.12).

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
c. r. of prev version	c. r. of change	6/0/8	20/0/25	20/0/27

Table 2.12: Do programmers tend to comment more or less when modifying a file that is already thoroughly commented?

These were the most striking results I had found so far. Indeed, it appeared that such a correlation existed in nearly all groups.

But was this really because programmers were trying to duplicate what they saw in the file they were modifying? Perhaps some files simply required more comments than others; for instance, some C header files might require extensive comments to explain how some API worked. To test this theory, I correlated the commenting rate of changes with the commenting rate of the *original* version of a file. The idea was to assume that in general, though the contents of a file might change over its lifetime, its role in the project would not, and thus it would require the same rate of commenting. Thus if it was really the role of a file that mattered, not how thoroughly commented it was, I should expect to see at least as many correlations as with the previous test

variables		# of correlations (+/-/# of groups)		
		by project	by programmer	by both
c. r. of version 1	c. r. of change	3/0/8	14/0/25	14/0/27

Table 2.13: Do programmers tend to comment more or less when modifying a file whose original version thoroughly commented?

However, this turned out not to be the case (see Table 2.13); commenting rate of the original version of a file was a weaker predictor than commenting rate of the previous version. Thus my theory about the role of files affecting how thoroughly they were commented was insufficient to explain the correlations I found.

This suggested the question, if thoroughly commented code indeed causes programmers to more thoroughly comment their changes, how powerful is this effect? If the first version of a file is thoroughly commented, will future versions of that file be thoroughly commented as well? To answer this question, I considered only versions of files which had a version number of 20 or greater, and among these datapoints, looked for a correlation between the commenting rate of the original version, and the commenting rate of the later version (version 20 or greater). Table 2.14 has my results; note that filtering out versions less than 20 resulted in some projects dropping below the threshold of 100 data points, which is why there are only 6 projects in the table. Even 20 or more versions later, the original rate of commenting matters, in all projects!

variables		# of correlations
		by project
c. r. of version 1	c. r. of current version (≥ 20)	6/0/6

Table 2.14: Do thoroughly commented files tend to stay thoroughly commented over time?

Thus, in almost all cases, programmers tend to comment changes to a file more thoroughly when the file is itself more thoroughly commented.

2.10 Conclusions

I was indeed successful in discovering some "natural laws" of commenting, in the sense that I found correlations which appeared in nearly all of my groups of data. These laws are summarized in Section 2.10.3.

2.10.1 Caveats: Representativeness

However, there are two major caveats in any study like this. The first has to do with representativeness. In this study, I made several interesting discoveries about nine specific Open Source projects. In order for these discoveries to be generally useful, I need some way of convincing the reader that the patterns I found might apply not only to the nine projects I studied, but to whatever project the reader might be interested in. The question to ask is, was there something special about these particular projects that caused me to find the correlations I did?

For one thing, all the projects I studied were Open Source projects. Though I would guess that any inherent tendencies of programmers would hold true in, say, proprietary commercial projects as well, there are simply so many factors that I cannot speak for these other programming environments without actually studying some of them. For instance, commercial environments might be more rigorous about enforcing certain commenting practices; the increased time pressure in a commercial environment could conceivably alter programmers' commenting habits significantly as well.

All the projects I studied were hosted on SourceForge, but I doubt this in itself makes much difference; the only requirement for a project to be hosted on SourceForge is that it be Open Source.

The more serious problem is that I selected the projects to study arbitrarily. Recall that I selected these nine projects because I wanted to study the commenting habits of two programmers across several projects. The fact that all the projects I studied included these programmers does not skew my results directly, because I observed the same effects when I broke down data by programmer, and these were only two of 25 programmers that I studied. However, sharing a programmer may be part of more subtle similarities between projects. In fact, both programmers had a penchant for working on programming related to games. Of the nine projects I studied, four of them were game libraries (`aeongdk`, `alleg`, `exult`, and `scummvm`), and three of them were extensions to the `alleg` library (`allegrogl`, `dumb`, and `fblend`).

There were also practical considerations that constrained the size of the projects I studied. There was a fairly wide range of number of distinct files (19 in `fblend` to 996 in `alleg`), and number of contributing programmers (1 in `fblend` to 23 in `scummvm`). However, projects any smaller would have had too little data to glean useful correlations from (`fblend` itself was in excluded my measurements having to do with changes within a project because it had too few distinct versions to study). And projects larger than `alleg` would have been cumbersome to collect data for and study; I actually ran the Comment

Counter on `mozilla` (unfortunately, before I realized the importance of excluding copyright comments), and it took over a week to run! It is conceivable that commenting works differently on very small projects or even very large ones; if so, my data says nothing about them.

Finally, all the projects that I studied (and any project I might study with the tools I have built) made use of CVS. This is directly important to commenting, though maybe not in a way that significantly impacts the quantity of comments: as Warren and Nafees [WN02] point out in their study of Nethack, some Open Source projects use comments to track the history of changes in a project. CVS makes such comments unnecessary because programmers can store comments about changes directly in CVS; out of all the source code files I collected data from, I could only find two such comments.

2.10.2 Caveats: Methodology

The second caveat has to do with my methodology. A correlation simply means that two variables in a group of data vary together; statisticians are free to choose or define variables any way they want. Thus, in any body of data, it is possible to find any number of correlations simply by defining variables cleverly; these correlations may or may not be meaningful, but probably would not appear in another dataset of the same type.

These methodological concerns are somewhat mitigated by the fact that I planned out in advance the questions I wanted to ask, and looked for the same correlations in not just one group of data but several, side by side. However, in order to verify my findings, I need to take another, fresh set of data, and find the same correlations (or lack thereof) that I found in this study.

Thus, though my findings may in fact be representative of other Open Source projects, or programming projects in general, it is most scientific to view my statistical study as an exploratory study, used to find what sort of interesting aspects of commenting can be captured statistically, rather than a clear statement about how commenting works in Open Source projects.

2.10.3 "Natural Laws" of Commenting

Keeping these caveats in mind, to the extent that my results *do* apply to commenting in general, I made four major findings about commenting that could be used in future studies with less flawed methodology.

First, commenting rates vary widely from project to project and programmer to programmer; even the same programmer will comment at different rates on different projects. Second, programmers tend to comment more when they make larger modifications to source code. Third, more programmers modifying the same file does not, in general, mean more comments. And finally, programmers tend to comment more when they are modifying code that is thoroughly commented to begin with.

My results suggest (but of course do not prove) that when programmers decide whether or not to comment their modifications, it has little to do with their

concrete expectation of whether other programmers will read their comments. Instead, programmers are attempting to uphold a standard of commenting, based on the perceived social norms of the project they are working on, their own training, and whatever implicit standard they perceive in the code they are modifying.

My results also suggest an easy answer to the big question, how can we get programmers to comment? Simply place code in front of them that is thoroughly commented, and they will try to keep it that way.

However, this assumes a causal relation (thoroughly commented code implies thoroughly commented changes). The danger of relying on statistics is that statistics cannot prove causality; they can only give us correlations. The correlation I found might be for some entirely different reason; for instance, it may be that programmers who like to thoroughly comment their changes simply avoid modifying code that is not thoroughly commented.

If there is indeed a causal relation here, it should be possible to devise an experiment to demonstrate it. This is the second half of my study.

Chapter 3

Testing the Implicit Standards Hypothesis

To determine whether thoroughly commented code could actually be used to induce people to comment more, I devised an experiment where some participants were given thoroughly commented code to modify, and some were given similar code, but with very few comments. My hypothesis was that, all else being equal, programmers modifying more thoroughly commented code would themselves produce changes that were more thoroughly commented than programmers modifying less thoroughly commented code.

My hypothesis fit into a much grander theory that programmers would try to imitate not only the amount of commenting, but also any other standards they perceived in the source code, including indentation, and where comments were placed (for example, after every variable declaration). I did not articulate a formal hypothesis for this theory (indeed, this may be too simple an experiment to produce scientific results about more detailed kinds of standards), but I hoped to use my study as a way of gathering information in order to learn to what extent this grander theory might be true, and to guide future studies.

3.1 Experiment Protocol

I recruited 12 participants by email solicitation of all graduate and undergraduate students at in the Computer Science Department in UC Berkeley. I asked that applicants be comfortable programming in the programming language C. Participants were not told that the study was about commenting; the recruitment message (see Appendix B.1) simply said that it was a study to "help improve the state of the art of software engineering." If asked about the study, I gave the neutral answer that it was "to study the way programmers program."

Participants were given the following scenario: they had been asked to make a set of modifications to an Open Source C program for computing checksums, adding some new printouts and a new kind of checksum which I invented for the

purpose of the study. Once they had made their modifications, participants were supposed to make a patch (a file showing the line-by-line differences between the original program and their modified version) for submission to the authors. (The program, `csum` was in fact based on a real Open Source[1] program called `sum`.) The instructions which participants were told to follow can be found in Appendix B.2.

Unbeknownst to the participants, I had divided them into two groups of six using a random drawing. One group (hereafter called group C) received a version of `csum` that was heavily commented; each function body and variable declaration was commented, and there were several descriptive comments as well. Group C's code can be found in Appendix B.3.1. The other group (UC) received code identical to group C, except that virtually all of the comments had been removed. I left in two comments, a copyright notice and a comment giving credit to the authors, in order to make the scenario of modifying an Open Source program seem more real. Group UC's code can be found in Appendix B.3.2.

I used a program called Camtasia [Cam] to record the participants while they worked. Camtasia essentially makes a video of whatever is on the screen. Because Camtasia is a Windows program, and `csum` is a UNIX program, I simulated a UNIX environment on a Windows 2000 computer using the UNIX-like programming environment Cygwin [Cyg] and the text editor NT Emacs [NT]. Every participant did the study on the same computer (my department-issued laptop).

Once participants indicated to me that they had completed the task (including making the patch), I gave them a short questionnaire to fill out (see Appendix B.4), with questions mostly having to do with commenting.

I asked each participant to schedule at least 90 minutes to do the study; all but one participant finished well within the the time scheduled. (In an attempt to relieve the participants of time pressure, the instructions stated that participants would have "as much time as [they] needed.")

For each participant, I saved his or her entire home directory (which included any scratch files that had been created), as well as the video, generated by Camtasia, of that participant's work.

One participant in Group C ran out of time, leaving the code in an unfinished state (it turned out that although I asked that participants be "reasonably comfortable using C," he had in fact only used it twice before). Because his situation is not easily comparable to other participants', I omit him entirely from my results.

To protect the anonymity of the subjects, I will refer to them by code rather than name. Hereafter the subjects in group C (other than the one who ran out of time) I refer to as are C1 through C5, and the subjects in group UC I refer to as U1 through U6.

[1] The Free Software Foundation, which distributes `sum`, prefers the term "Free Software"

3.2 The Source Code: Counting Comments

The first thing I did to evaluate the results of my study was to run the final source code through the comment counter to see if participants in group C (thoroughly commented original code) did indeed comment their changes more thoroughly. As in the first part of the study, I considered the copyright notice and the comment telling who the authors were, as not part of the file.

The results were striking. Changes made by the participants in group C were, in every case, much more thoroughly commented than by participants in group UC. Table 3.1 shows the commenting rate of the original version, the new version, and the commenting rate of the change.

participant	commenting rate		
	original	final	change
C1	42.66	38.18	18.54
C2	42.66	35.55	18.18
C3	42.66	37.79	22.68
C4	42.66	41.28	35.12
C5	42.66	38.55	25.04
U1	0.00	3.08	7.51
U2	0.00	2.46	5.99
U3	0.00	0.00	0.00
U4	0.00	0.00	0.00
U5	0.00	0.00	0.00
U6	0.00	0.57	1.98

Table 3.1: Did participants who received more thoroughly commented code make more thoroughly commented changes?

In fact, the results were almost suspiciously good. Running the results through R gave a 91.34% correlation between the commenting rate of the original file, and the commenting rate of the change, much higher than what I observed for any programmer or any project; the largest correlation I recorded (for one of the programmers working on `allegrogl`) for any programmer working on any project was 34.09%; more typical were correlations in the 10s and 20s.

In fact, there was a significant problem with these results. I recommend trying to figure out what the problem was as you learn more about the other aspects of the experiment. However, if you don't want to wait, you can learn what the problem was, and how I fixed it, by skipping ahead to Section 3.7.

3.3 The Source Code: Solution Structure

Out of curiosity, I looked over the final source code produced by each participant to see how they solved the task I had given them.

The instructions asked participants to add a new kind of checksum, called PPAIRS, to the code, and to add two new printing options; one that prints the checksum in hexadecimal, and one that keeps the program from printing the size of the file(s) it performs a checksum on.

The original source code has two important functions, `sysv_sum_file()` and `bsd_sum_file()`, which calculate SysV and BSD-style checksums, respectively. Not surprisingly, all participants created another, analogous function to perform the PPAIRS checksum. In fact, they all gave it the exact same name: `ppairs_sum_file()`.

`sysv_sum_file()` and `bsd_sum_file()` contain a lot of similar code; both open and close a file, and both are responsible for printing their own checksum. (For whatever reason, the methods they use to read the file are different.) Several of the programmers took this as an opportunity to perform refactorings.[2] Seven programmers in all refactored the code in some way. Of these, five (C2, C5, U2, U3, and U6) created a separate function for printing (including printing the checksum in hex) to replace the printing code in the checksum functions.

The other two programmers performed more radical transformations. U5 changed the signature of the checksum functions so that rather than just taking a filename and returning an error code, they returned the size of the file and the checksum as well (via pointers). An intermediate function would call the appropriate checksum function and then print the results in the proper format. U4 did the same thing only his intermediate function handled opening and closing the files to be checksummed as well.

It is interesting to note that only 2 out 5 participants from group C performed refactorings, whereas 5 out of 6 participants from group UC did the same (though as a correlation, this has about a 17% chance of occurring by chance, so it fails the test for statistical "significance"). More striking is the fact that the two most intensive refactorings occurred in group UC. This is probably for good reason; changing the signature of the checksum functions, as U4 and U5 did, would involve changing a rather lengthy comment as well in the commented version of the source code. Alternatively, U4 and U5 may have used refactoring as a way of making the code easier to understand, to make up for the lack of comments.

3.4 The Source Code: Style and Placement of Comments

I also looked over participants' source code to see to what extent they followed existing commenting standards. I designed the original commented version of the source code (see Appendix B.3.1) to follow three commenting standards:

- All function declarations are commented, above the function header.

[2] One programmer, C4, neglected to implement the new printing options at all; I include his results in the study because he left his code in a finished state, and it worked in all other respects.

- All variable declarations are commented.
- Only C-style comments are used.

What I hoped to see is that participants in group C would follow these commenting standards in their final versions.

Only three members of group C followed the standard of commenting every function declaration: C1, C3, and C4. These were the members of group C who performed no refactorings, so they only had one new function to comment. C3, in fact, simply copied the comment from `bsd_sum_file()`, neglecting even to update it in its new location.

Only one of the members of group C (C4) created code where all variable declarations are commented, though all members except C5 made some effort, albeit an inconsistent one, to comment new variable declarations they created. C3 actually said in his survey that he commented all variable declarations, but neglected to comment two variables he added to `main()`.

However, the results for what style of comment participants used were much less ambiguous (see Table 3.2). As of the C99 standard [JTC99], both C-style comments (`/* ... */`) and C++-style comments (`// ...`) can be used in C. All five participants in group C produced code with C-style comments only, including C4, who indicated in his questionnaire that he had only actually ever used C++.

Compare this to the results for group UC; of the 3 participants who added any comments, two of them, U1 and U6, used C++-style comments exclusively. U2 used C-style comments exclusively, but indicated in his questionnaire that he always uses C-style comments in C because he is never sure if the compiler implements the new standard.

participant	style of comments used	
	`/* C-style */`	`// C++-style`
C1	√	
C2	√	
C3	√	
C4	√	
C5	√	
U1		√
U2	√	
U3		
U4		
U5		
U6		√

Table 3.2: What style of comments did participants use?

In conclusion, the results for when participants commented function and variable declarations were fairly disappointing, and in any case are not useful to prove anything because I don't have anything meaningful to compare them against.

However, receiving thoroughly, consistently commented code does seem to have had a measurable effect on the syntactic style of the comments people chose. If we express the correlation between what group people were in, and whether they chose to use C-style comments as a correlation, it is statistically significant, even excluding the participants who added no comments at all!

3.5 The Questionnaire: Quantitative Questions

Next, I looked at participants' completed questionnaires. Many of the questions on the survey (see Appendix B.4) were quantitative in nature, so I hoped that even if patterns in the questionnaire weren't apparent at first glance, I might be able to notice otherwise invisible patterns by analyzing them with R (as I did in the first half of the study).

The Questionnaire had seven quantitative questions. Below, I give each of these questions a code number so I can refer to them side-by-side in a table.

Four questions asked about the participants' prior experience with C:

> What is your programming experience in C? (fill in/check all that apply)
>
> E1: Years of experience (give a number)
>
> E2: Use C on a regular basis (yes/no)
>
> E3: Have worked on a large project in C (yes/no)
>
> E4: Have worked on a project with other programmers in C (yes/no)

One question uses a Likert-style scale to ask about readability.

> LR: How easy did you find it to read and understand the original program? (1 to 7, with 1 being "very easy", and 7 being "very difficult")

Two questions use a Likert-style scale to ask about how much the participants commented.

> LC1: Do you think you wrote more or less comments (measured as a percentage of lines of code written) than you usually do when you work on software projects of comparable size? (1 to 7, with 1 being "much more", and 7 being "much less")
>
> LC2: Did you put more or less effort into writing comments than you usually do when you work on software projects of comparable size? (1 to 7, with 1 being "much more", and 7 being "much less")

What I hoped to see was that participants in group C (thoroughly commented code) would report putting commenting more often and putting more effort into commenting, to a greater extent than participants in group UC.

This tended to be the case, but not strongly enough to be significant. R reported a 38.16% correlation between which group participants were in, and their response about how much effort they put into commenting; in a group this small, a correlation this big has a nearly a 1 in 4 probability of occurring by chance! (The correlation between group and how much more people commented was even weaker.)

Disappointingly, the quantitative questions on the questionnaire gave me virtually nothing in the way of interesting correlations. In fact, the *only* statistically significant correlation was between participants' response on LC1 (how much they commented) and LC2 (how much effort they put into commenting), which is not at all surprising.

Having comments to read, at least in this study, did not make the source code significantly easier for participants to read and understand. The small size of the program and the descriptive names may have made up for the lack of comments in the code that group UC received; some participants remarked that the code was too small to need comments (see section 3.6.1)

Both groups were fairly equal in terms of how often their participants used C on a regular basis (E2) and as to whether their participants had worked with other programmers in C (E4). All participants reported working on a large project in C.

However, in terms of years of experience (E1) programmers in group UC tended to be somewhat more experienced than those in group C. This may provide an alternate explanation for why some programmers refactored and some didn't: the seven programmers who performed refactorings were the also seven programmers with the most experience.

Table 3.3 contains the survey results in their entirety. Note that participant C4 indicated that he had actually never programmed in C before, but had used C++ for 2.5 years.

participant	question						
	E1	E2	E3	E4	LR	LC1	LC2
C1	6	N	Y	N	2	4	4
C2	9	Y	Y	Y	1	2	4
C3	6	Y	Y	Y	1	3	4
C4	2.5*	N	Y	Y	6	6	6
C5	10	Y	Y	Y	2	7	7
U1	3	N	Y	N	2	4	7
U2	12	N	Y	Y	3	6	6
U4	8	N	Y	Y	3	7	7
U3	15	Y	Y	Y	2	4	4
U5	12	Y	Y	Y	1	7	7
U6	8	N	Y	N	1	4	5

Table 3.3: How did participants respond to quantitative survey questions?

Why wasn't I able to get more meaningful results from the quantitative

parts of the questionnaire? Part of the problem is that asking programmers to compare against what they "usually do" (in question LC1 and LC2); asking programmers to include their habits in their response may add more variables and thus more noise, which is bad for a sample size this small.

Also, the "years of experience" question yielded somewhat deceptive results. Out of 11 participants, only 5 were using C regularly at the time of the study; the other 6 had picked up C at some time in the past, and then stopped using it. Better questions might have been "in what year did you first use C?" and "in what year was the last time you used C for a large project?". Alternatively, participants could have been given a task in a more currently popular language, such as Java.

3.6 The Questionnaire: Qualitative Results

Fortunately, the questionnaire also included short-answer questions, and the responses for these were much more interesting. After reading through the questionnaires, I noted that different participants tended to made the same types of responses. In this section, I attempt to categorize participants' response in order to give better insight into what motivates programmers to comment.

3.6.1 Quantity of Comments

Questions LC1 ("Do you think you wrote more or less comments...") and LC2 ("Did you put more or less effort into writing comments...") were both followed by a short-answer question asking participants to explain their reasons for doing so ("If more or less, briefly explain why").

LC1 and LC2 were very similar in nature, and participants tended to give similar short-answer responses to both, so I treat them in the same section. In fact, three participants (C4, U1, and U2) explicitly linked the two short-answer questions with an arrow or a remark like "same as above".

Six participants gave perceived commenting standards as a reason for how much they commented, or how much effort they put into commenting. Here are their responses:

> "To conform with comments already there." (C2)
>
> "I tried to maintain a comment level that was roughly equivalent to that of the surrounding code." (C3)
>
> "There were very few helpful comments. To stick with the spirit of the code, I omitted comments as well." (U1)
>
> "I wrote less comments because I saw the rest of the code did not have the many comments either." (U2)
>
> "There were no comments in the original, and I was trying to match its style." (U3)

"There were no comments in the original code, so I felt like this was a no-comments kind of project." (U4)

Three participants gave the inherent understandability of the code as a reason that comments were unnecessary.

"Code speaks for itself. Variable & function names are clear." (C5)

"Once I understood the code, I decided, 'Why bother?"' (U2)

"This program was very simple, and I don't think it (or most programs of comparable size) need any comments." (U3)

Two participants gave the similarity of their new code to the old code as a reason not to comment. I am not entirely sure of their reasoning; they might have been only thinking about explaining their changes to the maintainer of `csum`, or perhaps they might have been trying to make an argument based on commenting standards: anyone who can understand the original version can presumably understand the participants' changes, so there is no reason to comment any *more* than the original version is commented (even if it is not commented at all). In any case, here are their responses:

"The code mostly followed the existing version." (U5)

"Most of the new code was very similar (cut + paste) to existing code." (U6)

(Ironically, U5 made one of the more extensive changes to the code; see Section 3.3)

Only one participant mentioned explicitly the fact that other programmers might read her comments:

"Amount of comment depends on whether or not I expect other people to be reading the code." (C1)

Finally, three participants gave a reason relating to the perceived importance of the final product, all of them a reason not to comment. In some ways, this shows a weakness in my study (and most studies of this type): although I presented a scenario where their final product was important, participants knew that in fact their code was never going to be used. Here are their responses:

"I didn't think of this as production code." (C4)

"I wanted to complete the program as quickly as possible." (C5)

"I just focused on completing the task at hand. I know I'm not maintaining this." (U5)

The fact that so many participants mentioned following existing standards as a reason to comment more or less strongly supports my hypothesis. In fact, it suggests that following existing standards is the *most* important factor in determining when people comment more often than usual. However, the perceived inherent understandability of the code, the perceived importance of the task, and an explicit desire to communicate with other programmers all appear to play some role as well.

3.6.2 Style and Placement of Comments

The last two questions on the survey dealt with not the amount of commenting, but the style and placement of comments:

> Q3: Please list any patterns you tried to follow when commenting (for example, placing a comment before every function, starting [functions inside comments][3] with a "//", etc.)

and

> Q4: If [there are] any of the above patterns that you don't normally follow when you program in C, circle them, and explain briefly why you chose to follow these patterns during this study.

In looking at the surveys, I wasn't so much interested in what commenting conventions people followed (which I had already seen from looking at the source code), but what caused them to alter their preexisting commenting habits, and why.

Seven out of eleven programmers (C1, C2, C3, C4, U1, U5, and U6) indicated in some way through their responses that they changed the way they commented for this study.

Q4 asks programmers to circle what they did differently; nobody actually did this. Q3 asks programmers to name patterns that they followed; only two of the programmers in group C (C2 and C3) named any patterns explicitly. (Group UC, receiving uncommented code, had very little to work with; one respondent (U1) talked about following "the pattern of not commenting at all".) However, though the participants weren't in general very specific about explaining what they did, they did give interesting reasons.

Five of the participants mentioned trying to follow existing commenting standards in their responses:

> "I copied the style of commenting that already exists in the code." (C1)

> "I placed a comment before the function I added (or copy-pasted, rather). I commented every variable declaration, saying what it's for. I placed a comment before every major block, stating its purpose. I followed all of the above comment styles to maintain consistency with the existing code." (C3)

[3] I had a couple of typos in the last two questions, see Appendix B.4 for the actual version. Only one participant noticed.

"I have my own coding conventions that I use. Here I tried to follow existing conventions." (C4)

"I followed the pattern of not commenting at all." (U1)

"I tried to make my code be consistent with the original file." (U6)

Three of the participants gave reasons for where they placed comments based on the understandability of the code:

"I only commented the checksum PPairs algorithm, since I thought that would be the trickiest to understand." (U2)

"Normally I carefully comment out function prologs. I only comment details in functions when they are complicated and/or tricky and/or big. In this case I added just one function I didn't really have to explain much." (U5)

"Since this code was clear to me, I didn't try to improve the code documentation." (U6)

One participant said that he put less effort into commenting because he was maintaining (i.e. modifying) someone else's code:

"Usually I'm not maintaining code, so I put more into comments." (U6)

If other programmers have the same tendency, it may explain why I found that the first version of a file (where the entire file is the "change") tends to have a higher rate of commenting than subsequent changes to the file.

One participant mentioned the strange development environment as a reason for introducing extraneous comments:

"In this case I commented '//Me' just so I could know where I was making changes. In a real situation, I would be better versed in the source code & would be using CVS, so I wouldn't need to do this." (U1)

Finally, one participant mentions cutting and pasting source code as a reason for following that code's commenting standards:

"Copied comment style when copied code." (C4)

3.7 Videos, and the Copying Problem

I had a nagging suspicion that my quantitative results for the commenting rate of people's changes was too good to be true, (or at least, too good to be meaningful). Eventually, I realized what the problem was.

A large part of the final code participants produced was `ppairs_sum_file()`, the function that computes the PPAIRS checksum. In all cases, `ppairs_sum_file()` performed a task analogous to the other two checksum functions, `bsd_sum_file()` and `sysv_sum_file()`. What I wondered was, might participants have copied and pasted one of the existing checksum functions to use as a model, and (in the case where the original code was commented), copied the comments along with it? `diff`, which I used to determine which characters were added, cannot determine if the "added" characters were actually written by the programmer, or merely copied from somewhere else.

To find out what programmers were doing, I looked at the screenshot videos generated by Camtasia. It turns out that programmers did indeed make heavy use of copying. Eight of the eleven participants (C3, C4, C5, U2, U3, U4, U5, U6) copied one of the two original checksum functions wholesale to use as a starting point for `ppairs_sum_file()`; those who received commented code copied the function header comment (which is quite large) as well as all comments in the checksum function.

Of the remaining three participants, C2 created the function header and braces for `ppairs_sum_file()`, but copied its body (including comments) from `sysv_sum_file()`. U1 essentially created a copy of `sysv_sum_file()` by copying pieces of both checksum functions at different times. Only C1 typed in `ppairs_sum_file()` by hand but this is the exception that proves the rule; she used `bsd_sum_file()`, as a model, retyping relevant code almost verbatim, but interestingly, excluding the comments!

The problem here is not simply that programmers copied code; this is a commonly used technique in real programs as well; for example, a study by Antoniol et al. estimates that the Linux kernel is 15–25% duplicated code [APMV00]. In fact, the problem with with my experiment is not merely that programmers copied code more often than they might in most software projects.[4]

The problem here is that the large degree of copying affected my method of measurement in a large, but meaningless way. It is not particularly interesting that programmers in group C "produced" comments by cutting and pasting them, nor is this effect likely to happen in projects that involve less copying. If I wanted my results to be useful, what I really needed was a way of measuring how thoroughly commented programmers' changes were, that was unlikely to be affected by copying and pasting code.

Ideally, I would use a tool similar to `diff` that could also determine when chunks of code were copied and pasted. Then I could modify the Comment Counter to consider characters "added" only if they weren't part of a copy-paste. This way, I could meaningfully determine which characters had been added by participants, and would have a way of measuring commenting that would not be affected by copying and pasting. There are several tools that find copies in source code, such as Copy-Paste Detector [CPD]. However, such tools

[4]The task in this study had a lot more to do with checksums than most software projects, but most people would not argue that my results would only apply to software projects involving checksums, because the fact that the task was about checksums is unlikely to significantly affect the amount of commenting that people produced.

are not directly analogous to `diff`; they are more aimed at finding copies within a body of code (to suggest refactorings) rather than between versions.

What I realized is that I didn't actually need to obtain such a tool, or rewrite Comment Counter. I could achieve the same effect by simply undoing the copying. I knew from the videos which of the two functions participants had based their `ppairs_sum_file()` function on. If I could remove the function that participants had copied from, and move `ppairs_sum_file()` into its place, then I could run the Comment Counter as normally. `diff` would now only consider characters in `ppairs_sum_file()` "added" if they were added after the programmer had copied-and-pasted the original checksum function to use as a template. Thus, if a participant copied, say `sysv_sum_file()`, comments and all, the copied comments would not show up in my measurements because `diff` would consider them made of the same characters that made up `sysv_sum_file()` in the original version of the source code.

I performed this transformation by hand on the participants' source code, and re-ran the Comment Counter (manually checking `diff`'s output to ensure it was behaving as expected). Table 3.4 contains the revised results, using the new, meaningful definition of "change" (compare to Table 3.1). (The "final" rate of commenting is still that of the file as participants left it, not the way it was after my edits.)

participant	commenting rate		
	original	final	change
C1	42.66	38.18	13.47
C2	42.66	35.55	9.16
C3	42.66	37.79	10.58
C4	42.66	41.28	15.13
C5	42.66	38.55	5.07
U1	0.00	3.08	3.06
U2	0.00	2.46	6.98
U3	0.00	0.00	0.00
U4	0.00	0.00	0.00
U5	0.00	0.00	0.00
U6	0.00	0.57	3.18

Table 3.4: Did participants who received more thoroughly commented code make more thoroughly commented changes? (revised)

Even after removing the effect of copying and pasting, there is still an impressively strong correlation (81.30%) between the commenting rate of the original file, and the commenting rate of people's changes. This correlation has an 0.23% chance of occurring by chance. The fact that I observed a higher correlation than in my statistical studies probably has to do with the controlled nature of my experiment, and the extremes of commenting in the code that I gave to participants (either no comments, or as many comments as are reasonably possible).

Some participants (C2 and U1) created an usually large amount of commented-

out source code that the Comment Counter misinterpreted as comments (i.e. they wrote code, and then commented it out). Also, as U1 mentioned in her questionnaire, some participants (C2, C5, and U1) left comments that said nothing about how the source code worked, but were simply there to mark where they had made changes (such as `/* new !! */`). To get a less noisy picture of what was going on, I removed all of these comments as well, and re-ran the Comment Counter. See Table 3.5 for those results.

participant	commenting rate		
	original	final	change
C1	42.66	38.18	13.47
C2	42.66	35.55	5.88
C3	42.66	37.79	10.58
C4	42.66	41.28	15.13
C5	42.66	38.55	1.87
U1	0.00	3.08	2.63
U2	0.00	2.46	6.98
U3	0.00	0.00	0.00
U4	0.00	0.00	0.00
U5	0.00	0.00	0.00
U6	0.00	0.57	3.18

Table 3.5: Did participants who received more thoroughly commented code make more thoroughly commented changes? (descriptive comments only)

With these non-descriptive comments removed, the correlation between the original rate of commenting and the commenting rate of the change drops to 68.95% (the noise actually favored my hypothesis), but is still well within the bounds of statistical significance (a correlation this large has only a 1.89% probability of occurring by chance in a sample this large).

3.8 Conclusions

There is ample evidence in this study to show that programmers given more thoroughly commented code produced more thoroughly commented changes as a result. Both my quantitative measurements, and participants' survey responses, support my hypothesis.

3.8.1 Caveats: Representativeness

However, as with my statistical study, there is the question of representativeness. How much bearing does the effect I observed in this one study have on programmers in general? Or to put it another way, is there something special about my study that caused some people to comment more, that isn't there in most situations?

For a while, I was worried that the high degree of copying might seriously impact the representativeness of this study (in which case it would at best apply to situations in which comments were likely to be copied and pasted), but I was able to isolate and remove this effect.

I don't think the language used for this study has any more bearing on its representativeness than the fact that it's about checksums; commenting in C is not much different from commenting in other languages. (Organized systems of commenting such as Javadoc [Kra99] that are meant to be read by an automated tool may have an effect, but this is an orthogonal issue; such tools, such as Doxygen [Dox], exist for C as well.)

The code in this study was by necessity much smaller than most software projects (though it was based on a real program), and simple to the point than many of the participants indicated that they didn't believe the code needed comments. However, in some ways, the simplicity of the code only strengthens my hypothesis; I was able to induce participants to comment even when it was largely unnecessary. In theory, given a larger, more complex program that required more comments to be understood, participants might simply comment their changes at the necessary level, regardless of whether the original code was throughly commented. However, in practice, programmers routinely produce insufficiently commented code, so there is good reason to believe that thoroughly commented code would induce programmers to comment more often on large, complex projects, just as on small, simple ones.

In keeping with my statistical study, the scenario I gave to participants in this study is that they were submitting a patch for an Open Source project. While I doubt that participants only try to comply with existing standards in Open Source projects, work environments are complex enough that there could easily be other factors in proprietary projects that I have not considered, so it would not be entirely proper to assert that the results of this study apply to non-Open Source projects, without further investigation.

It is probably significant that in the scenario, participants were explicitly asked to modify someone else's source code, presumably for release to the outside world. Though only one participant talked explicitly about other people reading her comments, many participants spoke of attempting to comply with existing standards; the scenario implies a certain degree of social pressure. It is unclear from this study what effect, if any, the rate of commenting of a piece of source code would have on programmers who modify it solely for their own use.

Finally, there are two caveats that apply to most studies of this kind. The first is that I used students, not professional programmers. The common complaint about using students is that they tend to be less skilled or experienced than professionals. In this case, all but two of the participants whose results I used had more than 5 years of C experience (albeit not always active), so this is probably less of a problem for my study than for most. Another problem is that students at the same university tend to have taken the same classes, and thus may have acquired similar biases not representative of programmers in general. However, in this case, nine of the eleven participants were graduate students, and thus had actually received their basic programming training from a variety

of sources.

The other caveat about my study is that it's a study. It is by nature, short, simplified, and about doing something that participants know to be contrived. For example, it is conceivable that when programmers work on a project for a long time, they become more comfortable following their own commenting tendencies, and feel less obligation to try to comply with commenting standards they perceive in the code they modify. The opposite could be true as well; programmers might become habitualized to the commenting standards in a project, and follow them more strongly over time. If either of these are true, there's simply no way to determine this scientifically in a 90-minute study.

3.8.2 Making Use of My Results

To the extent that you believe my study applies to your situation, my results suggest that it is indeed worthwhile to comment code before asking other people to modify it if you want their changes to be thoroughly commented.

However, it is worth noting that although participants who received thoroughly commented code tended to comment more than other participants, they commented at a much lower rate than the original code was commented (see Table 3.5). Also, as in my statistical study, there was quite a bit of variance from programmer to programmer; two of the participants who received thoroughly commented code actually commented their changes at a lower rate than one of the participants who received uncommented code. Thoroughly commenting code is *not a guarantee* that other programmers will follow in your footsteps. Aside from programmers' own personal tendencies, the perceived importance of their contribution can be a factor as well. And getting programmers to make comments that they see as unnecessary may be a lost cause!

There also seems to be very little assurance that programmers will follow, or even notice, your coding conventions. At least in this study, it seems to be easier to get programmers to follow syntactic standards for commenting than to get them to comment variable declarations, and even more difficult to get them to comment what functions do (probably because this takes the most work). However, to be fair, this was not a properly constructed study about commenting style or placement; I discuss possible ways of constructing such a study in Section 5.2.

Chapter 4

Related Work

4.1 Comments and Readability

In this paper, I have operated under the simplifying assumption that comments are almost always desirable, in the sense that they make programs easier for programmers to read and understand.[1] In fact, the reality is a bit more subtle. As Sheppard et al. found in one of the earliest empirical studies of commenting and readability [SBC78][2], comments do not harm readability, but they are not necessarily always helpful either. In many cases (as in my own empirical study, see Section 3.5), source code is so simple and straightforward that comments provide no additional benefit.

Exactly when comments are helpful depends largely on context. Woodfield et al. [WDS81] conducted an empirical study dealing specifically with short comments inserted just before a logical module in a program, and found that they aided program comprehension. Tenny [Ten88] conducted an empirical study of how comments and procedures affect program readability, and found that comments were only helpful in the absence of procedures (ironically, Woodfield et al. found that dividing the program in their study into procedures actually made it *less* readable).

Descriptive names (for variables, procedures, classes, etc.) can also aid readability, and in some cases can make up for a lack of comments. In fact, Detienne, in her book on cognitive aspects of programming [Det02], lumps both commenting and descriptive naming into "documentation" and treats them similarly.

4.2 Affecting Commenting Behavior

Two general facts are known about changes in programmers' commenting behavior. First, the degree of expertise that programmers have will affect what

[1] Though if someone wanted to induce programmers to produce *fewer* comments, they could conceivably use my findings to do that as well.

[2] Cited through Woodfield et al. [WDS81].

kinds of comments they create. Riecken et al. found that novices' comments tend to convey syntactic knowledge (how the programming language works), while experts, considering such knowledge obvious, tend to focus on semantic knowledge (how the program works) [RKBR91].

Second, expert programmers tend to make more high-level comments when designing their own program than when modifying somebody else's code; Rouet et al. [RDDB95][3] found that when modifying someone else's code, experts tend to produce only comments that reflect reasoning at the textbase level (how the program is structured, rather than what it does, or the rationale behind structuring it that way). This difference may partially explain the difference in rate of commenting that I found between the first version of a file and subsequent versions (see Section 2.4).

4.3 Tools to Help Programmers Comment

As early as 1982, researchers attempted to help programmers to produce more thoroughly commented code by designing tools which help them comment. Erickson [Eri82] wrote a tool for FORTRAN that creates comments by interactively querying the user about parts of the program. Roach et al. wrote a similar tool for Prolog [RBT90]. Shum and Cook developed an entire language and programming system, the Abstraction-Oriented Programming System (AOPS), in order to encourage programmers to produce better-commented code [SC93]. Interestingly, students tended to produce lengthier comments (more characters per comment) when using AOPS than when using a conventional programming system, but produced about the same number of comments overall [SC95].

More recently, a number of investigations have been made into commenting by voice. Chiueh et. al [CWL00] conducted a survey of programmers to help them design their own voice commenting tool, Variorum, and found that programmers prefer speech as a medium of documentation over drawing and typing. Voice commenting systems have a significant affect on the quantity and nature of comments that programmers produce. Soudian and Fels [SF02], in a study of their voice commenting tool, the Verbal Source Code Descriptor (VSCD), found that users produced about twice as many comments as a control group using a conventional editor; VSCD users also tended to produce a greater proportion of variable definition and inline comments. Zhao [Zha04], in a study of her own VoiceNotes system, did not perform a direct experimental comparison with conventional commenting, but did find an important correlation between the purpose of voice comments and the time when they are made, which is not present in conventional commenting. Finally, Begel, who created a system called Commenting by Voice, gives a good summary of the motivations behind voice commenting, and the practical issues behind building a voice commenting tool [Beg02].

[3] Cited through Detienne et al. [DRBDD96].

4.4 Mining Software Repositories

My study was not the first to investigate comments by running an automated tool on existing software repositories. Stamelos et al. ran an automated code quality tool on 100 Open Source projects, and found that, on the average, 31% of the components in any given project needed to be more thoroughly commented (according to the code quality tool) [SAOB02]. Matwin et al. used noun phrases extracted from comments to build a model of a program's problem domain in an automated fashion. Etzkorn et al. used an ad-hoc technique meant to simulate random sampling, to collect statistics on comments' verb tense and content, and then used that knowledge to modify a natural lanaguage parser so that it was able to correctly parse comments [EBD99].

Finally, there are a very large number of studies that mine software repositories for purposes other than studying comments. At the time of writing, the first International Workshop on Mining Software Repositories is scheduled to be held later this month. The one software repository mining technique which may be particularly useful for studying commenting is *origin analysis*, that is, determining when a piece of code has been moved or copied into another file in the same project. Godfrey et al. [GDKZ04] describe two case studies where they used their tool Beagle to conduct origin analysis on the source code for GCC and PostgreSQL.

Chapter 5

Future Work

There is still quite a lot of work to be done to understand scientifically programmers' commenting habits, and the motivations behind commenting. In this section, I explain how studies like my statistical study and my experiment could be improved, and then more general ways that this field could be explored.

5.1 Improving on the Statistical Study

It would be very worthwhile to re-do my statistical study without the methodological flaws of the first one. Projects should be selected randomly from among the projects on SourceForge rather than arbitrarily, and the correlations I expect to find should be chosen and defined in advance (including the way that variables are defined; for instance, is "age" age in seconds or version number?). It would also be instructive to study more than nine projects (though gathering and processing such data could be time-consuming). It would also be helpful to study some very large Open Source project (such as Mozilla), and some proprietary commercial projects.

My data collection methods could also be extended. So far, the Comment Counter only collects data for the main branch or "trunk" of a project; perhaps something interesting happens in the branch versions, especially before a release. It might also be possible to integrate clone-detection software into the Comment Counter in order to guess when code has been copied and pasted, rather than originally conceived. One final thing that could be added is a heuristic for determining when a comment is in fact commented-out code, making use of Zemankova and Eastman's finding that comments tend to be lexically more similar to English than source code [ZE80].

But it would also be possible to collect more subjective kinds of data, if a different method were used. Though in a limited sense it may be possible to infer some things using simple heuristics (such as my "bug fix" heuristic), it is in general difficult to do this in a totally automated fashion.

Instead, it would make more sense to follow the lead that Etzkorn et al.

[EBD99] took in their comment parsing study, and use random sampling: have a program randomly select some comments from a code base, and then have human beings categorize them as to their purpose and usefulness. This method would be particularly useful to identify comments which have become out-of-date and are no longer correct.

5.2 Improving on the Experiment

Though my experiment was quite successful in supporting its hypothesis, there are several ways it could be improved if someone wanted to try to duplicate my results.

The most serious problem with this study was the problem of participants copying and pasting code. Though I was able to identify and counteract this problem by closely studying how participants created their source code, it would be better to avoid this problem entirely. It should be possible to construct a scenario where all the work that programmers need to do is orthogonal to what the code already does; for example, if the code is designed to perform a sort, the task should focus on anything but sorting.

It could be helpful to try think of programming tasks more "worthy" of commenting; many participants in my study indicated that they saw no need for commenting, and more comments would probably produce more striking results (or maybe not; maybe if comments are truly necessary, perceived commenting standards would have less effect). However, using such comment-worthy tasks in a study is tricky; code that requires explanation through comments is probably also more difficult for participants to construct.

It would be worthwhile to give participants a scenario that does not involve Open Source, in order to better show that the effect I observed does not apply exclusively to Open Source programs. (My guess is that this would make no difference, as long as the scenario indicates to participants that their work is important.) It also might be helpful to use a language that participants are more likely to be currently using (such as Java).

Though my study was able to incidentally turn up statistically significant results about the style of comments that programmers chose (C- or C++-style comments), it was not really set up to be a study about style. A study that gave two groups of programmers identical code to modify, only one used all C-style comments, and one used all C++-style comments, would probably turn up similarly significant results (ideally, such a study should also have a control group, that received code with no comments). It would also be best to conduct such a task in a language where programmers are aware of both styles of comments, and both are commonly used (again, such as Java).

5.3 Other Future Work

Though I was ultimately able to show that the commenting rate of code affects how much people will comment modifications to that code, I was not able to give a clear answer about how big this effect is. Ideally, one should be able to look at the commenting tendencies in a given project, and the commenting tendencies of a given programmer, and be able to predict, within some degree of error, what that programmer will do (at least with respect to commenting) upon joining that project. Statistical studies like the one I conducted would be the basis of forming such a model, though it is possible that other sorts of data about commenting (such as the purpose of comments) may turn out to be important in predicting how people will comment.

It would be useful to consider commenting in conjunction with other documentation. For example, do programmers tend to comment a program more thoroughly when it has other supporting documentation (such as manuals)?

Another piece in constructing a model about commenting is to learn more about what programmers are thinking when they choose whether to make comments. Though the short-answer responses on the questionnaire in my experiment gave some insight into what programmers were thinking, both my statistical study and my experimental study were primarily about what programmers did, not what they were thinking, or why they did it.

Finally, further investigations should be made into the relationship between comments and refactoring. In my experiment, it appeared that programmers were more likely to refactor uncommented code than heavily commented code. Do comments indeed discourage refactoring, and if so, is it the mere mechanical barrier of having to rewrite comments as well as code, or are programmers more likely to re-arrange code that they perceive to be stylistically "deficient" in some way?

Bibliography

[APMV00] G. Antoniol, M. Di Penta, E. Merlo, and U. Villano. Analyzing cloning evolution in the linux kernel. *Journal of Information and Software Technology*, 44(13):755–765, 2000.

[Beg02] Andrew Begel. Program commenting by voice. http://www.cs.berkeley.edu/∼abegel/cs294-1/voice-comments.pdf, 2002.

[Cam] Techsmith Camtasia. http://www.techsmith.com/.

[CPD] Copy-Paste Detector. http://pmd.sourceforge.net/cpd.html.

[CVS] Concurrent Versions System. http://www.cvshome.org/.

[CWL00] Tzi-Cker Chiueh, Wei Wu, and Lap-Chung Lam. Variorum: A multimedia-based program documentation system. In *IEEE International Conference on Multimedia and Expo (I)*, pages 155–158, 2000.

[Cyg] Cygwin. http://www.cygwin.com.

[Det02] Franciose Detienne. *Software Design – Cognitive Aspects*. Springer, 2002.

[Dox] Doxygen. http://www.stack.nl/∼dimitri/doxygen/.

[DRBDD96] Francoise Detienne, Jean-Francois Rouet, Jean-Marie Burkhardt, and Catherine Deleuze-Dordron. Reusing processes and documenting processes: toward an integrated framework. In *Proceedings of the Eight Conference on Cognitive Ergonomics*, pages 139–144, 1996.

[EBD99] Letha H. Etzkorn, Lisa L. Bowen, and Carl G. Davis. An approach to program understanding by natural language understanding. *Nat. Lang. Eng.*, 5(3):219–236, 1999.

[Eri82] Timothy E. Erickson. An automated fortran documenter. In *Proceedings of the 1st annual international conference on Systems documentation*, pages 40–45. ACM Press, 1982.

[GDKZ04] Michael Godfrey, Xinyi Dong, Cory Kapser, and Lijie Zou. Four interesting ways in which history can teach us about software. In *International Workshop on Mining Software Repositories*, 2004.

[GNUa] GNU diffutils. ftp://ftp.gnu.org/pub/gnu/diffutils.

[GNUb] GNU diffutils manual. http://www.gnu.org/software/diffutils/manual/diff.html.

[JTC99] JTC1/SC22/WG14. *International Standard 9899: Programming languages — C*. ISO/IEC, 1999.

[Kra99] Douglas Kramer. API documentation from source code comments: A case study of javadoc. In *Proceedings of the 7th Annual International Conference of Computer Documentation (SIGDOC-99)*, pages 147–153, N.Y., September 12–14 1999. ACM Press.

[Krs94] Ivan Krsul. Authorship analysis: Identifying the author of a program. Technical Report Purdue Technical Report CSD-TR-94-030, Purdue University, 1994.

[Mar02] David Marin. Collaboration and commenting behavior in the harmonia project. http://www.cs.berkeley.edu/∼dmarin/papers/ccbhp.ps, 2002.

[NT] NT emacs. http://www.gnu.org/software/emacs/windows/ntemacs.html.

[R] The R project. http://www.r-project.org/.

[RBT90] David Roach, Hal Berghel, and John R. Talburt. An interactive source commenter for prolog programs. In *Proceedings of the conference on SIGDOC '90*, pages 141–145. ACM Press, 1990.

[RDDB95] Jean-Francois Rouet, Catherine Deleuze-Dordron, and Andre Bisseret. Documentation skills in novice and expert programmers: an empirical comparison. In *Proceedings of the seventh workshop of the Psychology of Programming Interest Group*, January 1995.

[RKBR91] R. Douglas Riecken, Jurgen Koenemann-Belliveau, and Scott P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 177–195, 1991.

[SAOB02] I. Stamelos, L. Angelis, A. Oikonomu, and Georgios L. Bleris. Code quality analysis in Open-Source software development. *Information Systems Journal*, 12(1):43–60, 2002.

[SBC78] S. B. Sheppard, M. A. Borst, and B. Curtis. Predicting programmers' ability to understand and modify software. In *Symposium Proceedings: Human Factors and Computer Science*, pages 115–135, June 1978.

[SC93] Stephen Shum and Curtis Cook. AOPS: an abstraction-oriented programming system for literate programming. *Software Engineering Journal*, 8(3):113–120, May 1993.

[SC95] Stephen Shum and Curtis Cook. Using literate programming to teach good programming practices. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 26(1):66–70, May 1995.

[SF] Sourceforge.net. http://www.sf.net.

[SF02] S. Soudian and D. L. Fels. Verbal source code descriptor. http://www.info.uqam.ca/∼lounis/wess2002papers/Soudian_Fels.pdf, 2002.

[SQL] PostgreSQL. http://www.postgresql.org/.

[Ten88] T. Tenny. Program readability: Procedures versus comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.

[Van02] Michael L. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, October 2002.

[WDS81] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223. IEEE Press, 1981.

[WN02] Robert Warren and Omar M. Nafees. Understanding software evolution through comment analysis, 2002.

[ZE80] Marie Zemankova and Caroline M. Eastman. Comparative lexical analysis of fortran code, code comments and english text. In *Proceedings of the 18th annual ACM Southeast regional conference*, pages 193–197. ACM Press, 1980.

[Zha04] Jinger Yu Zhao. Voicenotes: A discourse analysis of programmer voice annotations. http://www.people.fas.harvard.edu/∼jyzhao/Final.pdf, 2004.

Appendix A

The Comment Counter

In true UNIX fashion, the Comment Counter works by gluing together several small utility programs to accomplish a larger task.

The Comment Counter examines each file in the repository. If the filename has an extension that the Comment Counter recognizes as belonging to a source code file (e.g. ".c"), the Comment Counter downloads the CVS log for that file (by invoking the `cvs` command) to determine how many revisions were made, when, and by whom.

Next, the Comment Counter checks out each version of the file (again by invoking `cvs`).[1] Files are checked out with the `-kk` option, so that `cvs` does not create spurious changes by expanding CVS keywords inside files.

The Comment Counter then compares each version of the file (other than the first) against its previous version with GNU `diff` [GNUa, GNUb], to determine which characters were added (for this study, I did not use data about which characters were removed). `diff` normally works line-by-line, but the Comment Counter forces it to compare files character-by-character by feeding it each character on a separate line. I used the "`-d`" option to diff, which is somewhat slower, but yields the most accurate results. The Comment Counter automatically considers all characters in the first version of a file to be "added" (though I later found this to be inappropriate; see Section 2.4).

For each version of the file, the Comment Counter chooses and runs one of several lexers, built using `flex`, in order to determine which characters are in comments. I ultimately built lexers to deal with C, C++, Java, Objective C, ML, shell script, Tcl, Perl, Lisp, and SQL. Many of the lexers were by necessity approximate; for instance, the Perl lexer assumes that anything starting with a hash (`#`) and ending with a newline is a comment, even though there are special

[1] Each version on the main branch. CVS repositories can contain more than a linear sequence of versions; in some cases, programmers create a "branch" in the sequence of versions, for example, in order to simultaneously work on a stable release, and to add new features to a less stable version. For simplicity, I chose only to analyze the main branch. Besides making the code of the Comment Counter simpler, it also allowed me to interpret version number as an integer (for example, version 1.5 becomes simply 5), so that I could use it in correlations.

cases in Perl for which this is not true.

Finally, the Comment Counter combines this information to make a tally of how many characters were in comments, how many characters were added, how many added characters were in comments, and the total number of characters in the file.[2] The Comment Counter enters this information into a PostgreSQL database [SQL], along with descriptive information about the change, such as name of the file, which programmer made the change, CVS version number, and the message that the programmer gave to the CVS repository when checking the change into CVS.

After some early attempts at analyzing statistics collected with the Comment Counter, I realized that I needed to make a special case for copyright notices. Common in open source projects, copyright notices are a source code comment which explains how the source code is licensed (under the GPL or a BSD-style license, for instance). Typically, identical or nearly identical copyright notices appear at the beginning of every source code file in an open source project. Because they are simply copy-and-pasted, copyright notices do not represent significant effort by programmers, nor, unlike most other comments, do they explain how the code works.

Unfortunately, copyright notices are usually fairly large (about 300 characters), and often end up representing a significant portion of the characters in comments for a given file (if not a significant portion of the total characters in the file). To keep copyright notices from skewing my results, I modified the Comment Counter to treat copyright comments as if they were not part of the file at all, counting characters in a copyright comment neither as characters in comments, nor towards the total character count for a file.[3]

[2]The Comment Counter can also, for each of these categories, collect information about which characters were non-whitespace, and which were alphanumeric, to use in a more complicated form of measuring how much programmers were commenting. In the end, the complicated measurement turned out not to be significantly different than the "rate of commenting" method I use in this paper.

[3]Copyright notices could usually be easily identified because they contained the string "copyright", though I could also instruct the Comment Counter to screen out particular comments that were known to be part of a copyright notice.

Appendix B

Experimental Materials

B.1 Recruitment Message

Are you interested in participating in a study to help improve the state of the art of software engineering? My name is David Marin; I'm a grad student in Computer Science, and I'm going to be conducting a study for the Harmonia group, the premiere software engineering research group at UC Berkeley. The Harmonia group is part of the Computer Science department, and is headed by Prof. Susan L. Graham.

The study will be conducted over the next month. It involves a short programming task in C, so you should be reasonably comfortable with this language (you need not be an expert). The study can be conducted at a time and place that is convenient for you. I ask that you set aside at least 90 minutes for this study, though it is possible that you may finish early.

As a thank you, participants will receive a 14 oz. package of Trader Joe's Triple Ginger Snaps. To satisfy university rules, you must be over 18 to participate.

For more information, including how to participate, contact David Marin by email at XXX@XXX.XXX, or by phone at (XXX) XXX-XXXX.

B.2 Instructions

csum is a simple Open Source program that can print out two kinds of checksums for one or more files. You are asked to add a third kind of checksum, the PPAIRS checksum, and add some printout options. You are submitting a patch to the authors in the hopes that your changes will be added to the main distribution.

To calculate the PPAIRS checksum of a file, interpret every byte as an (8-bit) integer, and take the product of each byte with the byte following it. The PPAIRS checksum is the sum of all the products, modulo 2^16. For the last byte in the file, the "following" byte is the *first* byte in the file.

If called with the `--ppairs` option, your final program should print out on a separate line, for each file it is passed, the PPAIRS checksum as a decimal number, followed by the filename (i.e. like the default behavior, only with a different kind of checksum).

Furthermore, if called with the `-x` or `--hex` options, the program will print out the checksum as an 8-digit hexadecimal number, and if called with the `--no-byte-count` option, it will not print out the file size. These options should work regardless of which checksum algorithm is specified.

A Makefile is provided for your convenience; type `make` on the command line to compile your program. You should not need to create or modify any files other than csum.c. Makefile.ppairs contains a PPAIRS checksum of the Makefile. To make a patch, type `make patch` on the command line.

If any part of the problem statement is unclear to you, please ask me.

You will have as much time as you need to complete this task.

B.3 Source Code

The source code I used in my experiment is based on a real Open Source program called `sum`, which is part of the GNU `coreutils` package, available here: `ftp://ftp.gnu.org/pub/gnu/coreutils`. Because the original version was licensed under the GNU General Public License (see `http://www.gnu.org/licenses/gpl.html`), my modified versions may be distributed under this license as well.

B.3.1 Group C (commented)

```
/* csum -- checksum and count the blocks in a file
   Copyright (C) 86, 89, 91, 1995-2001 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.  */

/* Similar to BSD sum or SysV sum -r,
   except like SysV sum if -s option is given. */

/* Written by Kayvan Aghaiepour and David MacKenzie. */

#include <stdio.h>
#include <sys/types.h>
#include <getopt.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

/* Dummy character values corresponding to
   options, for use in getopt_long() */
/* --help */
#define GETOPT_HELP_CHAR -2
/* --version */
#define GETOPT_VERSION_CHAR -3

/* What error code to return */
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1

/* Check if two strings are equal */
#define STREQ(str1, str2) (strcmp(str1, str2) == 0)
```

```
/* A pointer type for a checksum function that
   takes a filename (or "-" for standard input) */
typedef int (*sum_func_ptr) (const char *);


/* The official name of this program (e.g., no ‘g’ prefix).  */
#define PROGRAM_NAME "csum"

/* The authors; printed out by usage() */
#define AUTHORS "Kayvan Aghaiepour and David MacKenzie"

/* The name this program was run with. (set in main()) */
char *program_name;

/* Nonzero if any of the files read were the standard input. */
static int have_read_stdin;

/* Struct to make getopt_long work
   the fields are:
   - option name
   - how many args it takes
   - how to return the argument (in this case,
     just return a character from getopt_long())
   - the character to return (this is why we need
     the dummy character values, above) */
static struct option const longopts[] =
{
  {"sysv", no_argument, NULL, 's'},
  {"help", no_argument , NULL, GETOPT_HELP_CHAR },
  {"version", no_argument , NULL, GETOPT_VERSION_CHAR },
  {NULL, 0, NULL, 0}
};

/* Print an error message for an error that happened
   when trying to do something with the given file.
   (The function gets the type of error from errno.)*/
static void
print_error (const char* file)
{
  fprintf(stderr, "%s: %s\n", strerror(errno), file);
}

/* Utility function that works exactly like read(), except
   that it can't be interrupted by a system call.

   Read LEN bytes at PTR from descriptor DESC, retrying if interrupted.
   Return the actual number of bytes read, zero for EOF, or negative
   for an error.  */
ssize_t
safe_read (int desc, void *ptr, size_t len)
{
  /* Return value, usually the number of characters read */
  ssize_t n_chars;

  if (len <= 0)
    return len;
```

```
  do
    {
      n_chars = read (desc, ptr, len);
    }
  while (n_chars < 0 && errno == EINTR);

  return n_chars;
}

/* Print how to use the program, and exit.

   If status is 0, the user used --help (so print out all the usage info)

   If status is nonzero, the user gave the program a bad switch,
   so just tell the user to use --help */
void
usage (int status)
{
  if (status != 0)
    {
      /* Just tell the user to use --help */
      fprintf (stderr, "Try `%s --help' for more information.\n",
     program_name);
    }
  else
    {
      /* Print out all the options */

      printf ("\
Usage: %s [OPTION]... [FILE]...\n\
",
      program_name);
      fputs ("\
Print checksum and byte counts for each FILE.\n\
\n\
  -r              defeat -s, use BSD sum algorithm\n\
  -s, --sysv      use System V sum algorithm\n\
", stdout);
      fputs ("      --help     display this help and exit\n", stdout);
      fputs ("      --version  output version information and exit\n", stdout);
      fputs ("\
\n\
With no FILE, or when FILE is -, read standard input.\n\
", stdout);
      printf ("\nReport bugs to <%s>.\n", "bug-textutils@gnu.org");
    }

  /* If the user screwed up, report a failure, otherwise,
     exit cleanly */
  exit (status == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}

/* Calculate and print the rotated checksum and the size in bytes
   of file FILE, or of the standard input if FILE is "-".
   print FILE to the left of the checksum and size.
   The checksum may vary depending on sizeof(int).
```

```
   Return 0 if successful, -1 if an error occurs. */
static int
bsd_sum_file (const char *file)
{
  register FILE *fp;            /* The file to read from */
  register int checksum = 0; /* The checksum mod 2^16. */
  register int total_bytes = 0; /* The number of bytes. */
  register int ch; /* Each character read. */

  /* Open the file, or use standard input */
  if (STREQ (file, "-"))
    {
      fp = stdin;
      have_read_stdin = 1;
    }
  else
    {
      fp = fopen (file, "r");
      if (fp == NULL)
{
  print_error (file);
  return -1;
}
    }

  /* Process input character-by-character,
     doing shifts to make the checksum more interesting */
  while ((ch = getc (fp)) != EOF)
    {
      total_bytes++;
      checksum = (checksum >> 1) + ((checksum & 1) << 15);
      checksum += ch;
      checksum &= 0xffff; /* Keep it within bounds. */
    }

  /* Report errors */
  if (ferror (fp))
    {
      print_error (file);
      if (!STREQ (file, "-"))
fclose (fp);
      return -1;
    }

  /* Close the file (unless it's stdin, so we can reuse it) */
  if (!STREQ (file, "-") && fclose (fp) == EOF)
    {
      print_error (file);
      return -1;
    }

  /* Print out the filename and checksum */
  printf ("%s %d %d", file, checksum, total_bytes);
  putchar ('\n');

  return 0;
}
```

```
/* Calculate and print the checksum and the size
   of file FILE, or of the standard input if FILE is "-".
   Print FILE to the left of the checksum and size.
   Return 0 if successful, -1 if an error occurs. */
static int
sysv_sum_file (const char *file)
{
  /* Uses safe_read() instead of getc(), so different fields from
     bsd_sum_file, even though they're basically both just walking
     through the file. */

  int fd;                   /* The file descriptor of a file to use */
  unsigned char buf[8192];  /* A buffer to read chars into */
  register int bytes_read;  /* Number of bytes read into the buffer */
  int total_bytes = 0;      /* Number of bytes in the file */
  int r;                    /* Temp variable for checksum algorithm */
  int checksum;             /* The checksum mod 2^16 */

  /* The sum of all the input bytes, modulo (UINT_MAX + 1).  */
  register unsigned int s = 0;

  /* Figure out which file to open. */
  if (STREQ (file, "-"))
    {
      fd = 0;
      have_read_stdin = 1;
    }
  else
    {
      fd = open (file, O_RDONLY);
      if (fd == -1)
{
  print_error (file);
  return -1;
}
    }

  /* Read all the bytes in the file, adding each to the checksum */
  while ((bytes_read = safe_read (fd, buf, sizeof buf)) > 0)
    {
      register int i;

      for (i = 0; i < bytes_read; i++)
s += buf[i];
      total_bytes += bytes_read;
    }

  /* Check for an error */
  if (bytes_read < 0)
    {
      print_error (file);
      if (!STREQ (file, "-"))
close (fd);
      return -1;
    }
```

```
  /* Close the file (if it wasn't STDIN) and
     report any errors. */
  if (!STREQ (file, "-") && close (fd) == -1)
    {
      print_error (file);
      return -1;
    }

  /* Manipulate the checksum itself to make things more interesting. */
  r = (s & 0xffff) + ((s & 0xffffffff) >> 16);
  checksum = (r & 0xffff) + (r >> 16);

  /* Print out the checksum. */
  printf ("%s %d %d\n", file, checksum, total_bytes);

  return 0;
}

/* Read in the options, and then calculate the checksum for each file
   accordingly */
int
main (int argc, char **argv)
{
  int errors = 0;  /* Set this to 0 if there were errors */
  int optc;        /* Return value from getopt_long() */
  int files_given; /* Number of files given. If this is zero, use stdin! */
  /* The default checksum function */
  sum_func_ptr sum_func = bsd_sum_file;

  /* Set program name (used elsewhere) */
  program_name = argv[0];

  /* Set this if we've read from standard input,
     so we can try closing it, and maybe find an error */
  have_read_stdin = 0;

  /* Read in options */
  while ((optc = getopt_long (argc, argv, "rs", longopts, NULL)) != -1)
    {
      switch (optc)
{
case 0:
  break;

case 'r': /* For SysV compatibility. */
  sum_func = bsd_sum_file;
  break;

case 's':
  sum_func = sysv_sum_file;
  break;

case GETOPT_HELP_CHAR: usage(0); break;

case GETOPT_VERSION_CHAR: printf("sum (textutils) 2.1\n\
            Written by Kayvan Aghaiepour and David MacKenzie.\n");
```

```
default:
  usage (1);
}
    }

  /* Compute checksum for each file */
  files_given = argc - optind;
  if (files_given == 0)
    {
      if ((*sum_func) ("-") < 0)
errors = 1;
    }
  else
    for (; optind < argc; optind++)
      if ((*sum_func) (argv[optind]) < 0)
errors = 1;

  /* Check if there is something wrong with stdin
     by trying to close it (EOF means fclose() had an error) */
  if (have_read_stdin && fclose (stdin) == EOF)
    print_error ("-");

  exit (errors == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

B.3.2 Group UC (uncommented)

```
/* csum -- checksum and count the blocks in a file
   Copyright (C) 86, 89, 91, 1995-2001 Free Software Foundation, Inc.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software Foundation,
   Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.  */

/* Written by Kayvan Aghaiepour and David MacKenzie. */

#include <stdio.h>
#include <sys/types.h>
#include <getopt.h>
#include <errno.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define GETOPT_HELP_CHAR -2

#define GETOPT_VERSION_CHAR -3


#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1


#define STREQ(str1, str2) (strcmp(str1, str2) == 0)

typedef int (*sum_func_ptr) (const char *);



#define PROGRAM_NAME "csum"


#define AUTHORS "Kayvan Aghaiepour and David MacKenzie"


char *program_name;


static int have_read_stdin;


static struct option const longopts[] =
{
  {"sysv", no_argument, NULL, 's'},
  {"help", no_argument , NULL, GETOPT_HELP_CHAR },
  {"version", no_argument , NULL, GETOPT_VERSION_CHAR },
  {NULL, 0, NULL, 0}
};


static void
print_error (const char* file)
{
  fprintf(stderr, "%s: %s\n", strerror(errno), file);
}


ssize_t
safe_read (int desc, void *ptr, size_t len)
{

  ssize_t n_chars;

  if (len <= 0)
    return len;

  do
    {
      n_chars = read (desc, ptr, len);
    }
```

```
  while (n_chars < 0 && errno == EINTR);

  return n_chars;
}


void
usage (int status)
{
  if (status != 0)
    {

      fprintf (stderr, "Try `%s --help' for more information.\n",
     program_name);
    }
  else
    {


      printf ("\
Usage: %s [OPTION]... [FILE]...\n\
",
      program_name);
      fputs ("\
Print checksum and byte counts for each FILE.\n\
\n\
  -r              defeat -s, use BSD sum algorithm\n\
  -s, --sysv      use System V sum algorithm\n\
", stdout);
      fputs ("      --help     display this help and exit\n", stdout);
      fputs ("      --version  output version information and exit\n", stdout);
      fputs ("\
\n\
With no FILE, or when FILE is -, read standard input.\n\
", stdout);
      printf ("\nReport bugs to <%s>.\n", "bug-textutils@gnu.org");
    }

  exit (status == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}


static int
bsd_sum_file (const char *file)
{
  register FILE *fp;
  register int checksum = 0;
  register int total_bytes = 0;
  register int ch;


  if (STREQ (file, "-"))
    {
      fp = stdin;
      have_read_stdin = 1;
    }
  else
```

```
    {
      fp = fopen (file, "r");
      if (fp == NULL)
{
  print_error (file);
  return -1;
}
    }


  while ((ch = getc (fp)) != EOF)
    {
      total_bytes++;
      checksum = (checksum >> 1) + ((checksum & 1) << 15);
      checksum += ch;
      checksum &= 0xffff;
    }


  if (ferror (fp))
    {
      print_error (file);
      if (!STREQ (file, "-"))
fclose (fp);
      return -1;
    }


  if (!STREQ (file, "-") && fclose (fp) == EOF)
    {
      print_error (file);
      return -1;
    }


  printf ("%s %d %d", file, checksum, total_bytes);
  putchar ('\n');

  return 0;
}


static int
sysv_sum_file (const char *file)
{
  int fd;
  unsigned char buf[8192];
  register int bytes_read;
  int total_bytes = 0;
  int r;
  int checksum;


  register unsigned int s = 0;


  if (STREQ (file, "-"))
```

```
    {
      fd = 0;
      have_read_stdin = 1;
    }
  else
    {
      fd = open (file, O_RDONLY);
      if (fd == -1)
{
  print_error (file);
  return -1;
}
    }


  while ((bytes_read = safe_read (fd, buf, sizeof buf)) > 0)
    {
      register int i;

      for (i = 0; i < bytes_read; i++)
s += buf[i];
      total_bytes += bytes_read;
    }


  if (bytes_read < 0)
    {
      print_error (file);
      if (!STREQ (file, "-"))
close (fd);
      return -1;
    }

  if (!STREQ (file, "-") && close (fd) == -1)
    {
      print_error (file);
      return -1;
    }


  r = (s & 0xffff) + ((s & 0xffffffff) >> 16);
  checksum = (r & 0xffff) + (r >> 16);


  printf ("%s %d %d\n", file, checksum, total_bytes);

  return 0;
}


int
main (int argc, char **argv)
{
  int errors = 0;
  int optc;
  int files_given;
```

```
  sum_func_ptr sum_func = bsd_sum_file;


  program_name = argv[0];

  have_read_stdin = 0;


  while ((optc = getopt_long (argc, argv, "rs", longopts, NULL)) != -1)
    {
      switch (optc)
{
case 0:
  break;

case 'r':
  sum_func = bsd_sum_file;
  break;

case 's':
  sum_func = sysv_sum_file;
  break;

case GETOPT_HELP_CHAR: usage(0); break;

case GETOPT_VERSION_CHAR: printf("sum (textutils) 2.1\n\
            Written by Kayvan Aghaiepour and David MacKenzie.\n");


default:
  usage (1);
}
    }


  files_given = argc - optind;
  if (files_given == 0)
    {
      if ((*sum_func) ("-") < 0)
errors = 1;
    }
  else
    for (; optind < argc; optind++)
      if ((*sum_func) (argv[optind]) < 0)
errors = 1;

  if (have_read_stdin && fclose (stdin) == EOF)
    print_error ("-");

  exit (errors == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

B.4 Questionnaire

What is your programming experience in C? (fill in/check all that apply)

_____ years of experience

___ use C on a regular basis
___ have worked on a large project in C
___ have worked on a project with other programmers in C

How easy did you find it to read and understand the original program?

1	***2***	***3***	***4***	***5***	***6***	***7***
Very Easy	Moderately Easy	Somewhat Easy	Normal	Somewhat Difficult	Moderately Difficult	Very Difficult

Do you think you wrote more or less comments (measured as a percentage of lines of code written) than you usually do when you work on software projects of comparable size?

1	***2***	***3***	***4***	***5***	***6***	***7***
Much More	Moderately More	Somewhat More	About the Same	Somewhat Less	Moderately Less	Much Less

If more or less, briefly explain why:

Did you put more or less effort into writing comments than you usually do when you work on software projects of comparable size?

1	***2***	***3***	***4***	***5***	***6***	***7***
Much More	Moderately More	Somewhat More	About the Same	Somewhat Less	Moderately Less	Much Less

If more or less, briefly explain why:

Please list any patterns you tried to follow when commenting (for example, placing a comment before every function, starting functions inside comments with a "//", etc.):

If any of the above are patterns that you don't normally follow when you program in C, circle them, and explain briefly below why you chose to follow these patterns during this study.